SPARQL query processing with conventional relational database systems

Steve Harris

IAM, University of Southampton, UK
swh@ecs.soton.ac.uk

Abstract. This paper describes an evolution of the 3store RDF storage system, extended to provide a SPARQL query interface and informed by lessons learned in the area of scalable RDF storage.

1 Introduction

The previous versions of the 3store [1] RDF triplestore were optimised to provide RDQL [2] (and to a lesser extent OKBC [3]) query services, and had no efficient internal representation of the language and datatype extensions from RDF 2004 [4].

This revision of the 3store software adds support for the SPARQL [5] query language, RDF datatype and language support. Some of the representation choices made in the implementation of the previous versions of 3store made efficient support for SPARQL queries difficult. Therefore, a new underlying representation for the RDF statements was required, as well as a new query engine capable of supporting the additional features of SPARQL over RDQL and OKBC.

2 Related Work

The SPARQL query language is still at an early stage of standardisation, but there are already other implementations.

2.1 *Federate*

Federate [6] is a relational database gateway, which maps RDF queries to existing database structures. Its goal is for compatibility with existing relational database systems, rather than as a native RDF storage and querying engine. However, it shares the approach of pushing as much processing down into the database engine as possible, though the algorithms employed are necessarily different. A SPARQL front-end is under development.

2.2 Jena

Jena [7] is a Java toolkit for manipulating RDF models which has been developed by Hewlett-Packard Labs. It has largely complete support for SPARQL features over the in-memory store provided by the Jena engine.

There is also a SQL back-end available, sparql2sql, though it is not as complete an implementation of the SPARQL specification and it does not perform as much processing in the database engine as the technique described in this paper.

2.3 Redland

The Redland Framework [8] also has substantial support for SPARQL queries over its in-memory and on-disk stores, but does not have a query engine that translates into relational expressions in the way that 3store and sparql2sql do. 3store uses the RDF parser and SPARQL query parser components from Redland, which were a great help in reducing the implementation effort.

3 The Design of the 3store RDF knowledge base

The overall architecture of this version of 3store (version 3) is broadly based on the previous versions. The basic design, in which the RDF query expressions are transformed to an SQL query expression containing a large number of simple joins across a small number of tables has been shown to perform and scale well [9].

3.1 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 license chosen was the GNU General Public License [10]. A Free Software license, that allows free distribution under certain conditions. Although a significant portion of 3store consists of libraries, the decision was made to license 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 and storage engine.

As in previous versions of 3store, MySQL was chosen as the sole persistent back-end. Although the MySQL optimiser is not particularly sophisticated it has already be shown in 3store versions 1 and 2 to be adequate for optimising RDF graph matching queries of the form produced by the 3store query translator.

3.2 Three layer model

The existing RDF engines that support medium sized knowledge bases and database back-ends (such as Jena and Redland, discussed in Sections 2.2 and 2.3 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 versions of 3store indicated that this can quantitatively reduce query execution times.

The 3-layer model of HYWIBAS[11] and SOPHIA[12] 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.

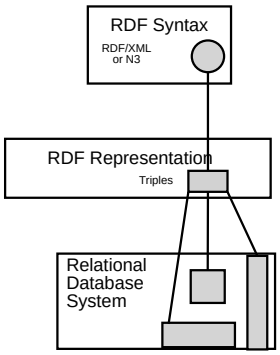

Fig. 1. 3store's RDF adaptation of the 3 layer model, after Norrie et al. [11]

4 RDF–SQL Mapping

The mapping between RDF and SQL has been changed since the previous version of 3store. In the previous version resources (URIs) and literals were kept in separate tables, requiring the use of outer joins to return values. Resources and literals are now kept in a single table, which allows inner joins to be used to return values, at the cost of a larger table.

4.1 Hashing function

Resources and literals are internally identified by a 64 bit hash function. This is used so that string comparisons can be kept to a minimum and to reduce the storage requirements per statement.

Discrimination As before a portion of the MD5 function [13] is used for it's efficiency and universality (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 [14]). However recent concerns over MD5's security [15] suggest that it may be better to seek out a hash function that provides greater protection from the possibility of hash collisions where security is a concern.

In order to allow the URI `<http://aktors.org>` to be easily distinguished from the literal string `"http://aktors.org"` the hash space is divided into two segments. Literals occupy the lower half of the 64 bit hash space and URIs occupy the upper half, such that $hash(literal(x))\ \&\ 2^{63} = hash(uri(x))$.

Additionally the literal hash values are modified by applying the bitwise exclusive-or operator to bit ranges of the literal hash and the integer identifier of the datatype and language. This ensures that that hash value for an integer literal `"100"` is distinct from the float, string etc. literals with the same lexical form, `"100"`. This stage is not strictly necessary, but it facilitates certain comparisons and joins we wish do do in the query language, and enables some optimisations.

Collisions The system detects colliding hashes for differing RDF nodes at assertion time. Though there is no obvious way to recover from this state, it is possible to warn the user and reject the RDF file containing the colliding hash. This could potentially happen though a pair of URIs engineered to collide. With the current state of the art in MD5 collisions they would both have to be at least 1024 bytes long [15], and it would take an extensive search to find a colliding pair that contained only characters that were legal in URIs.

The "Birthday Paradox"[1] gives the probability of an accidental hash collision occurring anywhere in a knowledge base of 10^9 distinct nodes to be around $1 : 10^{-10}$ when using an effective 63 bit hash. The situation for literal values is complicated by the exclusive-or operator, but the probability will be of a similar order of magnitude.

4.2 SQL tabular storage

Due to the change in the hashing function used to represent RDF nodes the SQL schema must also be changed. An overview of the tables and the relations is shown in figure 5.

[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. However this assumes a perfectly even hashing function.

model	subject	predicate	object
int64	int64	int64	int64

Fig. 2. Structure of the `triples` table

Triples The central table is the `triples` table (figure 2) which holds the hashes for the subject, predicate, object and SPARQL `GRAPH` [5] identifier. The 64 bit integer columns hold hashes of the RDF nodes that occupy those slots in the triple. The `model` column is used to hold the `GRAPH` identifier, or zero if the triple resides in the anonymous background graph.

hash	lexical	integer	floating	datetime	datatype	language
int64	text	int64	real	datetime	int32	int32

Fig. 3. Structure of the `symbols` table

Symbols The `symbols` table (figure 3) allows reverse lookups from the hash to the hashed value, for example, to return results. Furthermore it allows SQL operations to be performed on pre-computed values in the `integer`, `floating` or `datetime` value columns without the use of casts.

`hash` holds the value of the hash the represents this symbol (URI or literal). URIs and literals may be discriminated by checking the value of the MSB of the integer value, see section 4.1.

`lexical` holds the UTF-8 representation of the textual form of the symbol as it appeared in the RDF document. This allow reconstruction of the node in output documents or query results, and is used for the storage of URIs and strings.

`integer`, `floating` **and** `datetime` are computed at assertion time according to the RDF datatype of the literal to be stored. The `floating` column has multiple uses and is used to store values in the XSD decimal, float and double datatypes [16]. `NULL` (ω) is stored in the value columns for which there is no value of that type.

`datatype` **and** `language` columns hold foreign keys to the `datatypes` and `languages` tables respectively. Where the is no known RDF datatype or language for a particular literal the appropriate columns are set to ω.

Datatypes and languages The `datatypes` and `languages` tables are joined to the `symbols` table by the `id` columns. Datatype could have been stored in the symbols table and joined back onto itself to recover the URI. However, the author felt that the approach of keeping it in a separate table would be more efficient overall. No benchmarking has been done to justify this decision.

The `id` column is an auto incrementing field in this implementation, though it could equally be a hash of the value as in the `triples` table.

id	uri
int32	text

id	lang
int32	char(5)

Fig. 4. Structure of the `datatypes` and `languages` tables

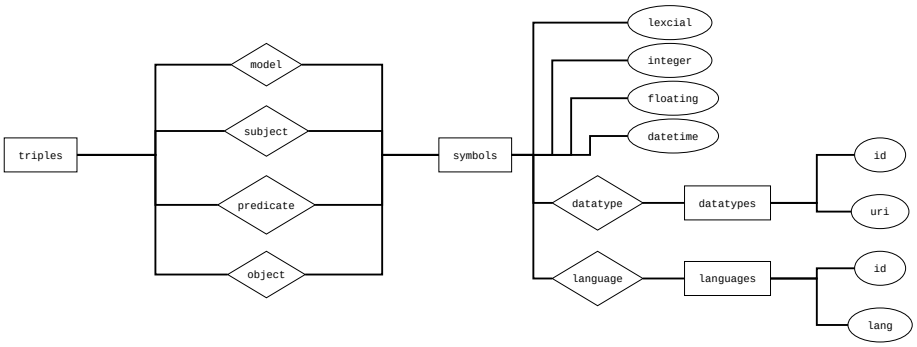

Fig. 5. ER diagram showing relationships between tables in the 3store schema

5 Query execution

As far as is possible the task of executing the query is passed down to the database engine. This allows the database optimiser to operate on the query, hopefully spot optimisations that are not clear from the external view of the tables and benefit from decades of research into relational database optimisers.

5.1 Simple graph expression mapping

To illustrate how the mapping to relational database expressions is performed for simple SPARQL queries imagine a query to discover the label associated the URI `mailto:alice@example`. In SPARQL this query could be written as shown in figure 6, where `?x` is a variable to be bound by the results of the query.

```
SELECT ?x
WHERE { <mailto:alice@example.com>
          <http://www.w3.org/2000/01/rdf-schema#label> ?x . }
```

Fig. 6. Graph showing RDF structure and equivalent SPARQL query

This query may simply be translated into relational algebra[2] as:

$$\text{tmp(x)} \leftarrow \pi_{\text{object}}(\sigma_{\text{predicate}=hash(p)\wedge\text{subject}=hash(s)\wedge\text{model}=0}(\text{triples})$$

$$\pi_{\text{lexical,datatype,language}}(\text{tmp} \bowtie_{\text{x=hash}} \text{symbols})$$

where *hash(s)* and *hash(p)* are the 64 bit hash values for the URIs `mailto:alice@-example.com` and `http://www.w3.org/2000/01/rdf-schema#label` respectively. Equivalently the SPARQL query may be translated into MySQL's dialect of SQL as shown in figure 7.

```
CREATE TABLE tmp (
  x bigint
)

INSERT INTO tmp
SELECT object
FROM triples
WHERE predicate=0xcf75ce12336ab89a
AND subject=0xbb37ab7444225966
AND model=0

SELECT v0.lexical, v0.datatype, v0.language
FROM tmp, symbols
WHERE tmp.x=symbols.hash
```

Fig. 7. Translation of query in figure 6 into SQL

Where the graph pattern is more complex and contains multiple triples, the `triples` table must be joined to itself, as in:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex: <http://example.com/schema#>
SELECT ?uri ?homepage
WHERE { ?uri ex:homepage ?homepage .
        ?uri rdfs:label "Alice" . }
```

which becomes

[2] In the paper describing the translation algorithm of the previous version of this software [1] the authors used relational calculus. However, some of the expressions used in this paper are more clearly expressed in relational algebra, which regrettably makes comparisons difficult

$$\text{tmp}(\text{homepage}, \text{uri}) \leftarrow \pi_{\text{t0.object,t0.subject}} \sigma_{\text{t1.object=0x64489c85dc2fe078} \wedge \text{t1.predicate=0xcf75ce12336ab89a} \wedge \text{t1.model=0} \wedge \text{t0.predicate=0xd290a78cfc5c100b} \wedge \text{t0.model=0}} (\rho_{\text{t0}}(\text{triples}) \bowtie_{\text{t0.subject=t1.subject}} \rho_{\text{t1}}(\text{triples}))$$

$$\pi_{\text{v0.lexical,v0.datatype,v0.language,v1.lexical,v1.datatype,v1.language}} ((\text{tmp} \bowtie_{\text{uri=hash}} \rho_{\text{v0}}(\text{symbols})) \bowtie_{\text{homepage=hash}} \rho_{\text{v1}}(\text{symbols}))$$

The algorithm employed to implement the transformation of these simple expressions is straightforward. First the `triples` table is joined once for each triple in the pattern and renamed as tn ($\rho_{\text{tn}}(\text{triples})$). Then the set of triples in the graph pattern is traversed and triple slots (subject, predicate or object) with constant values are constrained to their hash values. For example t1.predicate is constrained to the hash for `rdfs:label` with $\sigma_{t1.predicate=0xcf75ce12336ab89a}$. Concurrently variables are bound; when first encountered the triple and slot in which the variable appears is recorded, and subsequent occurrences in the graph pattern are used to constrain any appropriate joins with their initial binding, as in $\bowtie_{t0.subject=t1.subject}$ in the example.

To produce the intermediate results table (tmp), the hashes of any SPARQL variables required to be returned in the results set are projected, as in $\pi_{t0.object,t0.subject}$.

Finally, the hashes from the intermediate results table are joined to the `symbols` table to provide the textual representation of the results. The datatype and language columns are also projected to allow them to be returned for RDF serialisation for example, or as data in the SPARQL XML results serialisation.

Clearly, the intermediate table could be projected within the final results expression (see section 5.4). Keeping it separate at this stage of the processing has advantages when dealing with more complex expressions. It allows a simple representation of optional match expressions and can be used to circumvent the join size optimiser restrictions on some database engines (including MySQL), by breaking down the query into multiple sub expressions.

5.2 Optional match implementation

SPARQL's OPTIONAL operator is used to signify a subset of the query that should not cause the result to fail if it cannot be satisfied. As such it is roughly analogous to the left outer join of relational algebra.

Again, the algorithm for the transformation from SPARQL to relational algebra is quite straightforward. The triples are grouped according to what block they appear in (the single "required" block, or one of the optional blocks). Intermediate tables are produced for each block as before, but in the case of optional blocks, columns that allow joining onto the required block must also be projected. In the example below `uri` is the only column which is present in both intermediate tables.

The intermediate tables are then joined to the `symbols` table in the results phase. The `symbols` tables for optional intermediate tables must be outer joined, as the bindings may be ω from failed matches in the optional blocks.

Using this approach, the SPARQL query

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex: <http://example.com/schema#>
SELECT ?uri ?homepage
WHERE { ?uri rdfs:label "Alice" .
        OPTIONAL { ?uri ex:homepage ?homepage . }
}
```

becomes[3]

$$\text{tmp(uri)} \leftarrow \pi_{\text{subject}}\sigma_{\text{object=0x64489c85dc2fe078} \wedge \text{predicate=0xcf75ce12336ab89a} \wedge \text{model=0}}(\text{triples})$$

$$\text{opt1(homepage, uri)} \leftarrow \pi_{\text{object,uri}}\sigma_{\text{predicate=0xd290a78cfc5c100b} \wedge \text{model=0}}(\text{tmp} \bowtie_{\text{uri=subject}} \text{triples})$$

$$\pi_{\text{v0.lexical,v1.lexcial}}((\text{tmp} \bowtie_{\text{uri=hash}} \rho_{\text{v0}}(\text{symbols})) \bowtie_{\text{uri=uri}} (\text{opt1} \bowtie_{\text{homepage=hash}} \rho_{\text{v1}}(\text{symbols})))$$

All simple, legal optional expressions may be transformed in this way. Though a more sophisticated algorithm is required to express nested optional graph patterns.

5.3 Value constraints

The application of value constraints can be more complex than the transformation of graph patterns. There is a simple case, where the value constraint refers only to variables that are only bound in the current block and the constraint can be mapped into an equivalent relational expression. In this case the constraint may be applied simply by joining the `symbols` table and selecting on the appropriate column. For example, where we have `?x ex:age ?y . FILTER(?y > 30)` we can produce $\sigma_{integer>30}(triples \bowtie_{object=hash} symbols)$. However, there are many cases where this simple transformation cannot be applied, including:

Non-relational expressions These may either be constraints that are beyond the expressive power of the relational engine (such as certain regular expressions) or extension functions that are implemented in the application layer. These must be handled by externally processing the intermediate tables or the final results table in the application layer.

[3] In this example the projections of the datatype and language of the returned bindings have been omitted for brevity.

Late bound expressions Sometimes it is desirable to place constraints in optional blocks with variables that do not appear in that block. This is problematic as the bindings for that variable are not available at the time the intermediate table is selected. In this case the constraint can be transferred to the results processing step. Alternatively, if available the bindings for the variables in question can be joined to the intermediate blocks in which they appear. Delaying the selection of the FILTER constraints is undesirable as it increases the size of the intermediate tables.

Required constraints on optionally bound variables In SPARQL it is syntactically legal to place constraints in the required block on variables that are bound only in an optional block. This has the effect of demoting the optional block to a required block, so the blocks must either be merged or the constraint must be delayed until the results phase of the query.

Additionally, there are complexities around the type constraints and promotion rules of SPARQL. On the whole these can be expressed in relational algebra, though a description of the algorithm is beyond the scope of this paper.

5.4 Optimisation

There are a number of layer two optimisations that can be performed on the query. The purpose of these is to perform transformations on the relational expressions that the relational engine would not be aware are valid (things that are specific to RDF) or to aid the relational engine in further optimisations.

Expressions such as the one in figure 6 may be optimised by simply substituting the intermediate table into the results expression, with appropriate renaming operations where necessary.

This operation may also be performed on certain simple optional match graph patterns. The query in section 5.2 is an example. As there are no constraints or dependencies between the blocks and each optional match consists of a single triple, the intermediate tables may be substituted into the results expression to form a single expression.

The benefit of this optimisation is in allowing the relational database's optimiser to work over the whole query and to allow it to bypass the construction of intermediate tables where possible. The total number of joins will be reduced as the required intermediate table does not have to be joined a second time.

Another area to which optimisation can be applied is the projection of the datatype and language attributes of a variable. Where the variable is known to be only able to bind to URI values then there is no need for these values to be projected. Equally, where the language or datatype attributes are constrained in FILTER expressions in certain parts of the query then their values are known to be constant.

6 Future work

Substantial development has been undertaken on this version of 3store. However, there is still a considerable amount of work to be done before it reaches the level of usability and performance of the previous versions.

6.1 Complete implementation

Not all the SPARQL features have been implemented at the time of writing. In particular, the UNION operator and nested optional blocks have yet to be supported. They pose considerable complexities to an efficient query translation. It is hoped that the intermediate table building approach will be advantageous in optimising queries using these features.

6.2 Reasoning

As yet there is no support for RDFS reasoning in this version. The current plan is to forward port the 3store version 2 reasoner. This reasoner is sound but incomplete [1], and has performed well in the past.

6.3 Optimisation

The biggest remaining implemented, unoptimised feature is the handling of SPARQL graphs. The current GRAPH support is a more-or-less naïve implementation of the specification and is a considerable overhead compared to the equivalent in previous versions of 3store. It is not yet clear what form a valid and practical optimisation of this feature could take.

Additionally there are number of other expression optimisations that could be developed.

7 Acknowledgments

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

References

1. Harris, S., Gibbins, N.: 3store: Efficient bulk RDF storage. In: Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03). (2003) 1–20 http://eprints.aktors.org/archive/00000273/.
2. Hewlett-Packard Labs: RDQL - RDF data query language. http://www.hpl.hp.com/semweb/rdql.htm (2003)

3. Chaudhri, V., Farquhar, A., Fikes, R., Karp, P.D., Rice, J.P.: Open knowledge base connectivity. Technical report, OKBC Working Group (1998) http://www.ai.sri.com/~okbc/spec.html.
4. Beckett, D.: RDF/XML syntax specification (revised). Technical report, World Wide Web Consortium (2004)
5. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. Technical report, World Wide Web Consortium (2005) http://www.w3.org/TR/rdf-sparql-query/.
6. Prud'hommeaux, E.: Optimal RDF access to relational databases. Technical report, Wold Wide Web Consortium (2004) http://www.w3.org/2004/04/30-RDF-RDB-access/.
7. Hewlett-Packard Labs: The Jena Semantic Web Toolkit. Technical report, Hewlett-Packard Labs (2003) http://www.hpl.hp.com/semweb/jena.htm.
8. Beckett, D.: Redland RDF Application Framework. http://librdf.org/ (2005)
9. Lee, R.: Scalability report on triple store applications. MIT tech report (2004) http://simile.mit.edu/reports/stores/.
10. Free Software Foundation: GNU General Public License. http://www.gnu.org/licenses/gpl.txt (1991)
11. Norrie, M., Reimer, U., Lippuner, P., Rhys, M., Schek, H.: 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
12. Abernethy, N., Altman, R.: 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)
13. Rivest, R.: 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.
14. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algortihms. MIT Press, Cambridge, MA (1989)
15. Klima, V.: Finding MD5 collisions on a notebook PC using multi-message modifications. In: Proceedings of the 3rd International Conference Security and Protection of Information. (2005)
16. Biron, P.V., Malhotra, A.: XML schema part 2: Datatypes second edition. W3C recommendation, World Wide Web Consortium (2004) http://www.w3.org/TR/xmlschema-2/.