Note on database layouts for SPARQL datastores

Richard Cyganiak
Digital Media Systems Laboratory
HP Laboratories Bristol
HPL-2005-171
September 28, 2005*

RDF, semantic web, databases, query

The SPARQL query language for RDF provides a standardized way to access Semantic Web data. This report summarizes some lessons learnt while implementing a SPARQL datastore on top of ModelRDB, the database backend of the Jena Semantic Web Framework, and puts forward recommendations for the database layout of a future, dedicated SPARQL datastore.

* Internal Accession Date Only
Approved for External Publication

Note on database layouts for SPARQL datastores

Richard Cyganiak
richard@cyganiak.de

HP Labs, Bristol, UK

Abstract. Abstract: The SPARQL query language for RDF provides a standardized way to access Semantic Web data. This report summarizes some lessons learnt while implementing a SPARQL datastore on top of ModelRDB, the database backend of the Jena Semantic Web Framework, and puts forward recommendations for the database layout of a future, dedicated SPARQL datastore.

This report summarizes some lessons the author has learnt while implementing the `sparql2sql` triple store [2]. `sparql2sql` is based on the database backend of the Jena Semantic Web Framework [1], which is also known as ModelRDB [6]. I argue that the ModelRDB database layout, while being a good persistent backend for the Jena API, is not a good match for a SPARQL datastore. My recommendations for a better schema include:

1. using a normalized representation of RDF triples,
2. using an index on the GraphID column,
3. using a separate index on the Predicate column,
4. using the same node encoding for S, P, O and GraphID,
5. using separate tables for the default graph and the named graphs of an RDF dataset,
6. storing model metadata in a form accessible to SQL.

The recommendations put forward here are largely consistent with the findings presented in [3], which, at the time of writing, is the most comprehensive work on database layouts for SPARQL stores.

1 Normalized and denormalized layout

Contrary to common database design wisdom, ModelRDB uses a denormalized schema. URIs and literals are stored directly in the statement table. This decision is justified in [6]: ModelRDB has been optimized for `find` queries, that is, for finding all triples matching a simple triple pattern where subject, predicate and object may be wildcards. The denormalized schema avoids the need for database joins when performing find queries. This improves performance.

There are, however, several reasons why a normalized schema works better for more complex SPARQL queries.

SPARQL requires joins anyway. SPARQL queries typically involve several or many triple patterns. This means joins cannot be avoided. Furthermore, the columns used in join conditions represent RDF nodes. In a denormalized table, the joins are made over big string columns. In a normalized table, the joins are made over key columns, which are typically integers. Joins over integers are faster because less data has to be retrieved from disk.

SPARQL queries have higher selectivity. Find queries are not very discriminate. A find query will often return a sizable chunk of the datastore. Much filtering will be done in application code. This means lots of node data has to be accessed per find query. SPARQL queries are much more accurate in pinpointing the bits of data required by the client. Most of the processing is done by matching triples against each other; the actual node representations don't have to be accessed unless the node is in the results. The number of accesses into the nodes table of a normalized schema might actually be quite low.

Memory savings: A normalized schema occupies less space on disk because each node representation is stored only once, even if the node occurs in many triples.

For a store that mostly has to support find queries, a denormalized schema works better. For a store that mostly has to support SPARQL, a normalized schema should work better. The question is where the balance tips. Some informal tests suggest that a denormalized schema might be faster for queries involving one and two triple patterns, and might be slower for three or more triple patterns.

2 Indexed columns

The ModelRDB schema doesn't have an index on the GraphID column in the statements table. This is understandable with the ModelRDB case where you typically have a very low number of graphs (one!) in the same table.

With SPARQL, it is not uncommon to have thousands of graphs in the same dataset. This makes operations like *fetchGraph* and *deleteGraph* very slow. An index on GraphID is necessary.

The Jena schema has one combined index on Subject and Predicate, and one on Object. One group has found that individual indices on S, P and O are faster [4], but it is not known if this finding generalizes to other database layouts.

3 Node encoding

ModelRDB uses an intricate scheme to store a node in a single column. The first letter indicates the type of node. Offset numbers point to different parts of the node, like the language tag or datatype URI of a literal node. URI prefixing and long objects further complicate the issue.

This makes it nearly impossible to push value testing reliably down into the database. Complicated expressions involving substring functions and conditionals are necessary to extract the lexical value of a literal node or the URI of an URI node. Although value testing in the database is not strictly necessary, the schema should strive to make it possible.

(The same applies to `ORDER BY`, although I'm not aware of any attempt to do it inside the database.)

If all the value testing for a given query is done in the DB, then it's possible to execute `OFFSET` and `LIMIT` in the DB, which might be a large benefit for some queries. That's why I think pushing at least some common value testing idioms into the DB is important and worthwhile.

A database layout that facilitates value testing in the database has been proposed in [3].

4 Dataset Structure

SPARQL's data model is the RDF dataset, a set of graphs named by URIs plus a single unnamed default graph. This concept was not yet developed when ModelRDB was designed. Nevertheless, ModelRDB is very flexible and can be made to store RDF datasets: Its ability to store multiple models in the same statement table can be used to simulate a set of named graphs.

The exercise of shoehorning RDF datasets into a ModelRDB store has highlighted some properties that a SPARQL database layout should have:

One table for all named graphs: If they were stored in different tables, then SPARQL queries like this become much harder:

```
GRAPH ?any { ... }
```

This is because this type of query has to go through the triples of all named graphs. If they are stored in different tables, then the corresponding SQL must either name all these tables (might be thousands), or must make use of SQL's information schema, a feature that is not universally implemented and not well known.

Separate tables for default graph and named graphs: The default graph might be put in the same table with the named graphs, or in a different table. In SPARQL, query patterns over the named graphs are clearly distinguished from query patterns over the default graph, so the "`GRAPH ?any { ... }`" construct doesn't match the default graph. Having the default graph in a different statement table thus slightly simplifies queries (no need to add "`WHERE in_default_graph`" or "`WHERE in_named_graph`" to every triple pattern) and should have slight performance advantages because it keeps tables smaller. On the other side, it makes it harder to build a union over named graphs *and* default graph, but it's not clear that this would be useful.

Model metadata accessible to SQL: ModelRDB stores additional metadata about models as RDF triples in a special system model. This has a downside: To create and delete graphs, Java code must be run to insert or remove appropriate statements from the system model. Although not part of SPARQL, there's a strong case for features like this:

```
DELETE GRAPH ?g WHERE { ?g .... }
```

Efficient implementation is easier when the creation or deletion of graphs can be done completely in-database.[1]

Explicit per-dataset graph list: Aside from the statement table, some kind of graph list is needed because an RDF dataset might contain empty graphs, and these cannot be represented in a pure statements-plus-GraphID table. The graph list might be a simple table with a single column that indicates the node IDs of all graphs in the dataset.

In Jena, there's a graph list, but it is shared between all models in the database and not just the models in a particular dataset. This adds overhead when several datasets are stored in the same database. Queries over multiple datasets, and sharing of graphs between datasets are no goals at the moment, and so a separate graph list table for each dataset seems like the best approach.

Same node representation for S, P, O and GraphID: SPARQL requires joins from graph names to other nodes, e.g. in patterns like this:

```
?graph dc:date "..." .
GRAPH ?graph { ... }
```

With the ModelRDB layout, the node encoding used for S, P and O cannot be used for the GraphID if multiple RDF datasets are to be stored in the same database. This is because they all have to share the same graph name table. To avoid collisions, some kind of dataset ID has to be encoded together with the graph URI into the model name column. This makes joins for cases like the example above expensive.

5 Reified Statements and the RDF Dataset

ModelRDB has a dedicated table for reified statements to reduce the storage bloat that results from extensive use of reification. There are some reasons why this special support, while not harmful, might be less beneficial for a SPARQL datastore:

- SPARQL and the RDF dataset address some common use cases for reification in a more practical way. This might reduce usage of reification in the future.

[1] In gnowsis [5], which uses sparql2sql in recent development versions, there's a requirement to delete all graphs older than a certain date, and it's often thousands of graphs.

– If the SPARQL schema would be normalized, then the memory savings from a reified statements table would be small, because the nodes table dwarfs the statements table anyway. (But there should still be performance benefits.)
– The task of translating SPARQL into SQL is much more complex than RDQL or find queries. Specialized tables add further to this complexity.

6 Aknowledgements

Throughout the work leading to this report, Kevin Wilkinson and Andy Seaborne provided many invaluable comments and insights into the design of ModelRDB and SPARQL.

References

1. J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *13th World Wide Web Conference*, 2004.
2. R. Cyganiak. *sparql2sql: A query engine for SPARQL over Jena triple stores.* HP Labs, http://jena.sourceforge.net/sparql2sql/.
3. S. Harris. Sparql query processing with conventional relational database systems, 2005. Submitted to *International Workshop on Scalable Semantic Web Knowledge Base System (SSWS 2005)*. http://eprints.ecs.soton.ac.uk/11126/.
4. L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. Rstar: an rdf storage and query system for enterprise resource management. In *CIKM '04: Proceedings of the thirteenth ACM conference on Information and knowledge management*, pages 484–491, New York, NY, USA, 2004. ACM Press.
5. L. Sauermann and S. Schwarz. *The Gnowsis Semantic Desktop.* DFKI, http://www.gnowsis.org/.
6. K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in jena2. In *First International Workshop on Semantic Web and Databases (SWDB 2003)*, http://www.hpl.hp.com/techreports/2003/HPL-2003-266.html, 2003.