RDF Support in the Virtuoso DBMS

Orri Erling
oerling@openlinksw.com

Ivan Mikhailov
imikhailov@openlinksw.com

Abstract: This paper discusses RDF related work in the context of OpenLink Virtuoso, a general purpose relational / federated database and applications platform. We discuss adapting a relational engine for native RDF support with dedicated data types, bitmap indexing and SQL optimizer techniques. We further discuss mapping existing relational data into RDF for SPARQL access without converting the data into physical triples. We present conclusions and metrics as well as a number of use cases, from DBpedia to bio informatics and collaborative web applications.

1 Introduction And Motivation

Virtuoso is a multi-protocol server providing ODBC/JDBC access to relational data stored either within Virtuoso itself or any combination of external relational databases. Besides catering for SQL clients, Virtuoso has a built-in HTTP server providing a DAV repository, SOAP and WS* protocol end points and dynamic web pages in a variety of scripting languages. Given this background and the present emergence of the semantic web, incorporating RDF functionality into the product is a logical next step. RDF data has been stored in relational databases since the inception of the model [1][8]. Performance considerations have however led to the development of custom RDF engines, e.g. RDF Gateway [7], Kowari [9] and others. Other vendors such as Oracle and OpenLink have opted for building a degree of native RDF support into an existing relational platform.

The RDF work on Virtuoso started by identifying problems of using an RDBMS for triple storage:

- Data Types: RDF is typed at run time and IRI's must be distinct from other data.
- Unknown data lengths large objects mix with small scalar values in a manner not known at query compile time.
- More permissive cast rules than in SQL.
- Difficulty of computing query cost. Normal SQL compiler statistics are not precise enough for optimizing a SPARQL query if if all data is in a single table.
- Efficient space utilization.
- Need to map existing relational data into RDF and join between RDF data and relational data.

We shall discuss our response to all these challenges in the course of this paper.

2 Triple Storage

Virtuoso's initial storage solution is fairly conventional: a single table of four columns holds one quad, i.e. triple plus graph per row. The columns are G for graph, P for predicate, S for subject and O for object. P, G and S are IRI ID's, for which we have a custom data type, distinguishable at run time from integer even though internally this is a 32 or 64 bit integer. The O column is of SQL type ANY, meaning any serializable SQL object, from scalar to array or user defined type instance. Indexing supports a lexicographic ordering of type ANY, meaning that with any two elements of compatible type, the order is that of the data type(s) in question with default collation.

Since O is a primary key part, we do not wish to have long O values repeated in the index. Hence O's of string type that are longer than 12 characters are assigned a unique ID and this ID is stored as the O of the quad table. For example Oracle [10] has chosen to give a unique ID to all distinct O's, regardless of type. We however store short O values inline and assign ID's only to long ones.

Generally, triples should be locatable given the S or a value of O. To this effect, the table is represented as two covering indices, G, S, P, O and O, G, P, S. Since both indices contain all columns, the table is wholly represented by these two indices and no other persistent data structure needs to be associated with it. Also there is never a need for a lookup of the main row from an index leaf.

Using the Wikipedia data set [12] as sample data, we find that the O is on the average 9 bytes long, making for an average index entry length of 6 (overhead) + 3 * 4 (G, S, P) + 9 (O) = 27 bytes per index entry, multiplied by 2 because of having two indices.

We note however that since S is the last key part of P, G, O, S and it is an integer-like scalar, we can represent it as a bitmap, one bitmap per distinct P, G, O. With the Wikipedia data set, this causes the space consumption of the second index to drop to about a third of the first index. We find that this index structure works well as long as the G is known. If the G is left unspecified, other representations have to be considered, as discussed below.

For example, answering queries like

```
graph <my-friends> {
    ?s sioc:knows
          <http://people.com/people#John> ,
          <http://people.com/people#Mary> }
```

the index structure allows the AND of the conditions to be calculated as a merge intersection of two sparse bitmaps.

The mapping between an IRI ID and the IRI is represented in two tables, one for the namespace prefixes and one for the local part of the name. The mapping between ID's of long O values and their full text is kept in a separate table, with the full text or its MD5 checksum as one key and the ID as primary key. This is similar to other implementations.

The type cast rules for comparison of data are different in SQL and SPARQL. SPARQL will silently fail where SQL signals an error. Virtuoso addresses this by providing a special

QUIETCAST query hint. This simplifies queries and frees the developer from writing complex cast expressions in SQL, also enhancing freedom for query optimization.

Other special SPARQL oriented accommodations include allowing blobs as sorting or distinct keys and supporting the IN predicate as a union of exact matches. The latter is useful for example with FROM NAMED, where a `G` is specified as one of many.

Compression. We have implemented compression at two levels. First, within each database page, we store distinct values only once and eliminate common prefixes of strings. Without key compression, we get 75 bytes per triple with a billion-triple LUBM data set (LUBM scale 8000). With compression, we get 35 bytes per triple. Thus, key compression doubles the working set while sacrificing no random access performance. A single triple out of a billion can be located in less than 5 microseconds with or without key compression. We observe a doubling of the working set when using 32 bit IRI ID's. The benefits of compression are still greater when using 64 bit IRI ID's.

When applying gzip to database pages, we see a typical compression to a third, even after key compression. This is understandable since indices are by nature repetitive, even if the repeating parts are shortened by key compression. Over 99% of 8K pages filled to 90% compress to less than 3K with gzip at default compression settings. This does not improve working set but saves disk. Detailed performance impact measurement is yet to be made.

Alternative Index Layouts. Most practical queries can be efficiently evaluated with the GSPO and OGPS indices. Some queries, such as ones that specify no graph are however next to impossible to evaluate with any large data set. Thus we have experimented with a table holding `G`, `S`, `P`, `O` as a dependent part of a row id and made 4 single column bitmap indices for `G`, `S`, `P` and `O`. In this way, no combination of criteria is penalized. However, performing the bitmap AND of 4 given parts to check for existence of a quad takes 2.5 times longer than the same check from a single 4 part index. The SQL optimizer can deal equally well with this index selection as any other, thus this layout may prove preferable in some use cases due to having no disastrous worst case.

3 SPARQL and SQL

Virtuoso offers SPARQL inside SQL, somewhat similarly to Oracleś RDF_MATCH table function. A SPARQL subquery or derived table is accepted either as a top level SQL statement of wherever a subquery or derived table is accepted. Thus SPARQL inherits all the aggregation and grouping functions of SQL, as well as any built-in or user defined functions. Another benefit of this is that all supported CLI's work directly with SPARQL, with no modifications. For example, one may write a PHP web page querying the triple store using the PHP to ODBC bridge. The SPARQL text simply has to be prefixed with the SPARQL keyword to distinguish it from SQL. A SPARQL end point for HTTP is equally available.

Internally, SPARQL is translated into SQL at the time of parsing the query. If all triples are in one table, the translation is straightforward, with union becoming a SQL union and optional becoming a left outer join. Since outer joins can be nested to arbitrary depths

inside derived tables in Virtuoso SQL, no special problems are encountered. The translator optimizes the data transferred between parts of the queries, so that variables needed only inside a derived table are not copied outside of it. If cardinalities are correctly predicted, the resulting execution plans are sensible. SPARQL features like construct and describe are implemented as user defined aggregates.

SQL Cost Model and RDF Queries. When all triples are stored in a single table, correct join order and join type decisions are difficult to make given only the table and column cardinalities for the RDF triple or quad table. Histograms for ranges of `P`, `G`, `O`, and `S` are also not useful. Our solution for this problem is to go look at the data itself when compiling the query. Since the SQL compiler is in the same process as the index hosting the data, this can be done whenever one or more leading key parts of an index are constants known at compile time. For example, in the previous example, of people knowing both John and Mary, the `G`, `P` and `O` are known for two triples. A single lookup in log(n) time retrieves the first part of the bitmap for

```
((G = <my-friends>) and (P = sioc:knows) and
  (O = <http://people.com/people#John>) )
```

The entire bitmap may span multiple pages in the index tree but reading the first bitts and knowing how many sibling leaves are referenced from upper levels of the tree with the same `P`, `G`, `O` allows calculating a ballpark cardinality for the `P`, `G`, `O` combination. The same estimate can be made either for the whole index, with no key part known, using a few random samples or any number of leading key parts given. While primarily motivated by RDF, the same technique works equally well with any relational index.

Basic RDF Inferencing. Much of basic T box inferencing such as subclasses and subproperties can be accomplished by query rewrite. We have integrated this capability directly in the Virtuoso SQL execution engine. With a query like

```
select ?person where { ?person a lubm:Professor }
```

we add an extra query graph node that will iterate over the subclasses of lubm:Professor and retrieve all persons that have any of these as rdf:type. When asking for the class of an IRI, we also return any superclasses. Thus the behavior is indistinguishable from having all the implied classes explicitly stored in the database.

For A box reasoning, Virtuoso has special support for owl:same-as. When either an `O` or `S` is compared with equality with an IRI, the IRI is expanded into the transitive closure of its same-as synonyms and each of these is tried in turn. Thus, when same-as expansion is enabled, the SQL query graph is transparently expanded to have an extra node joining each `S` or `O` to all synonyms of the given value. Thus,

```
select ?lat where { <Berlin> has_latitude ?lat }
```

will give the latitude of Berlin even if `<Berlin>` has no direct latitude but `geo:Berlin` does have a latitude and is declared to be `owl:same-as <Berlin>`.

The `owl:same-as` predicate of classes and properties can be handled in the T box through the same mechanism as subclasses and subproperties.

Data Manipulation. Virtuoso supports the SPARUL SPARQL extension, compatible with JENA [8]. Updates can be run either transactionally or with automatic commit after each modified triple. The latter mode is good for large batch updates since rollback information does not have to be kept and locking is minimal.

Full Text. All or selected string valued objects can be full text indexed. Queries like

```
select ?person from <people> where {
    ?person a person ; has_resume ?r .
    ?r bif:contains 'SQL and "semantic web"' }
```

will use the text index for resolving the pseudo-predicate `bif:contains`.

Aggregates. Basic SQL style aggregation is supported through queries like

```
select ?product sum (?value) from <sales> where {
    <ACME> has_order ?o .   ?o has_line ?ol .
    ?ol has_product ?product ; has_value ?value }
```

This returns the total value of orders by ACME grouped by product.

For SPARQL to compete with SQL for analytics, extensions such as returning expressions, quantified subqueries and the like are needed. The requirement for these is inevitable because financial data become available as RDF through the conversion of XBRL [13].

RDF Sponge. The Virtuoso SPARQL protocol end point can retrieve external resources for querying. Having retrieved an initial resource, it can automatically follow selected IRI's for retrieving additional resources. Several modes are possible: follow only selected links, such as `sioc:see_also` or try dereferencing any intermediate query results, for example. Resources thus retrieved are kept in their private graphs or they can be merged into a common graph. When they are kept in private graphs, HTTP caching headers are observed for caching, the local copy of a retrieved remote graph is usually kept for some limited time. The sponge procedure is extensible so it can extract RDF data from non-RDF resource via microformat or other sort of filter. This provides common tool to traverse sets of interlinked documents such as personal FOAFs that refer to each other.

4 Mapping Legacy Relational Data into RDF for SPARQL Access

RDF and ontologies form the remaining piece of the enterprise data integration puzzle. Many disparate legacy systems may be projected onto a common ontology using different rules, providing instant content for the semantic web. One example of this is OpenLink's ongoing project of mapping popular Web 2.0 applications such as Wordpress, Mediawiki, PHP BB and others onto SIOC through Virtuoso's RDF Views system.

The problem domain is well recognized, with work by D2RQ [2], SPASQL [5], DBLP [3]

among others. Virtuoso differs from these primarily in that it combines the mapping with native triple storage and may offer better distributed SQL query optimization through its long history as a SQL federated database.

In Virtuoso, an RDF mapping schema consists of declarations of one or more quad storages. The default quad storage declares that the system table `RDF_QUAD` consists of four columns (`G`, `S`, `P` and `O`) that contain fields of stored triples, using special formats that are suitable for arbitrary RDF nodes and literals. The storage can be extended as follows:

An IRI class defines that an SQL value or a tuple of SQL values can be converted into an IRI in a certain way, e.g., an IRI of a user account can be built from the user ID, a permalink of a blog post consists of host name, user name and post ID etc. A conversion of this sort may be declared as bijection so an IRI can be parsed into original SQL values. The compiler knows that an join on two IRIs calculated by same IRI class can be replaced with join on raw SQL values that can efficiently use native indexes of relational tables. It is also possible to declare one IRI class `A` as `subClassOf` other class `B` so the optimizer may simplify joins between values made by `A` and `B` if `A` is bijection.

Most of IRI classes are defined by format strings that is similar to one used in standard C sprintf function. Complex transformations may be specified by user-defined functions. In any case the definition may optionally provide a list of sprintf-style formats such that that any IRI made by the IRI class always match one of these formats. SPARQL optimizer pays attention to formats of created IRIs to eliminate joins between IRIs created by totally disjoint IRI classes. For two given sprintf format strings SPARQL optimizer can find a common subformat of these two or try to prove that no one IRI may match both formats.

```
prefix : <http://www.openlinksw.com/schemas/oplsioc#>
create iri class :user-iri "http://myhost/sys/users/%s" (
    in login_name varchar not null ) .
create iri class :blog-home "http://myhost/%s/home" (
    in blog_home varchar not null ) .
create iri class :permalink "http://myhost/%s/%d" (
    in blog_home varchar not null,
    in post_id integer not null ) .
make :user_iri subclass of :grantee_iri .
make :group_iri subclass of :grantee_iri .
```

IRI classes describe how to format SQL values but do not specify the origin of those values. This part of mapping declaration starts from a set of table aliases, somehow similar to FROM and WHERE clauses of an SQL SELECT statement. It lists some relational tables, assigns distinct aliases to them and provides logical conditions to join tables and to apply restrictions on table rows. When a SPARQL query should select relational data using some table aliases, the final SQL statement contains related table names and all conditions that refer to used aliases and does not refer to unused ones.

```
from SYS_USERS as user from SYS_BLOGS as blog
where (^{blog.}^.OWNER_ID = ^{user.}^.U_ID)
```

A quad map value describes how to compose one of four fields of an RDF quad. It may be an RDF literal constant, an IRI constant or an IRI class with a list of columns of table aliases where SQL values come from. A special case of a value class is the identity class, which is simply marked by table alias and a column name.

Four quad map values (for `G`, `S`, `P` and `O`) form quad map pattern that specify how the column values of table aliases are combined into an RDF quad. The quad map pattern can also specify restrictions on column values that can be mapped. E.g., the following pattern will map a join of `SYS_USERS` and `SYS_BLOGS` into quads with `:homepage` predicate.

```
graph <http://myhost/users>
subject :user-iri (user.U_ID)
predicate :homepage
object :blog-home (blog.HOMEPAGE)
where (not ^{user.}^.U_ACCOUNT_DISABLED) .
```

Quad map patterns may be organized into trees. A quad map pattern may act as a root of a subtree if it specifies only some quad map values but not all four; other patterns of subtree specify the rest. A typical use case is a root pattern that specifies only the graph value whereas every subordinate pattern specifies `S`, `P` and `O` and inherits `G` from root, as below:

```
graph <http://myhost/users> option (exclusive) {
    :user-iri (user.U_ID)
      rdf:type foaf:Person ;
      foaf:name user.U_FULL_NAME ;
      foaf:mbox user.U_E_MAIL ;
      foaf:homepage :blog-home (blog.HOMEPAGE) . }
```

This grouping is not only a syntax sugar. In this example, `exclusive` option of the root pattern permits the SPARQL optimizer to assume that the RDF graph contains only triples mapped by four subordinates.

A tree of a quad map pattern and all its subordinates is called “RDF view” if the “root” pattern of the tree is not a subordinate of any other quad map pattern.

Quad map patterns can be named; these names are used to alter mapping rules without destroying and re-creating the whole mapping schema.

The top-level items of the data mapping metadata are quad storages. A quad storage is a named list of RDF views. A SPARQL query will be executed using only quad patterns of views of the specified quad storage.

Declarations of IRI classes, value classes and quad patterns are shared between all quad storages of an RDF mapping schema but any quad storage contains only a subset of all available quad patterns. Two quad storages are always defined: a default that is used if no storage is specified in the SPARQL query and a storage that refers to single table of physical quads.

The RDF mapping schema is stored as triples in a dedicated graph in the `RDF_QUAD` table so it can be queried via SPARQL or exported for debug/backup purposes.

5 Applications and Benchmarks

As of this writing, July 2007, the native Virtuoso triple store is available as a part of the Virtuoso open source and commercial offerings. The RDF Views system is part of the offering but access to remote relational data is limited to the commercial version.

Virtuoso has been used for hosting many of the data sets in the Linking Open Data Project [14], including Dbpedia [15], Musicbrainz [16], Geonames [17], PingTheSemanticWeb [18] and others. The largest databases are in the single billions of triples.

The life sciences demonstration at WWW 2007 [19] by Science Commons was made on Virtuoso, running a 350 million triple database combining diverse biomedical data sets.

Web 2.0 Applications. We can presently host many popular web 2.0 applications in Virtuoso, with Virtuoso serving as the DBMS and also optionally as the PHP web server.

We have presently mapped PHP BB, Mediawiki and Drupal into SIOC with RDF Views.

OpenLink Data Spaces (ODS). ODS is a web applications suite consisting of a blog, wiki, social network, news reader and other components. All the data managed by these applications is available for SPARQL querying as SIOC instance data. This is done through maintaining a copy of the relevant data as physical triples as well as through accessing the relational tables themselves via RDF Views.

LUBM Benchmark. Virtuoso has been benchmarked with loading the LUBM data set. At a scale of 8000 universities, amounting to 1068 million triples, the data without key compression size is 75G all inclusive and the load takes 23h 45m on a machine with 8G memory and two 2GHz dual core Intel Xeon processors. The loading takes advantage of SMP and parallel IO to disks. With key compression the data size drops to half.

Loading speed for the LUBM data as RDFXML is 23000 triples per second if all data fits in memory and 10000 triples per second with one disk out of 6 busy at all times. Loading speed for data in the Turtle syntax is up to 38000 triples per second.

6 Future Directions

Clustering. Going from the billions into the tens and hundreds of billions of triples, the insert and query load needs to be shared among a number of machines. We are presently implementing clustering support to this effect. The present clustering scheme can work in a shared nothing setting, partitioning individual indices by hash of selected key parts. The clustering support is a generic RDBMS feature and will work equally well with all RDF index layouts. We have considered Oracle RAC[20]-style cache fusion but have opted for hash partitioning in order to have a more predictable number of intra cluster messages and for greater ease in combining messages.

The key observation is that an interprocess round-trip in a single SMP box takes 50 microseconds and finding a triple takes under five. Supposing a very fast interconnect and a cluster of two machines, the break-even point after which cluster parallelism wins over

message delays is when a single message carries 10 lookups. Thus, batching operations is key to getting any benefit from a cluster and having a fixed partition scheme makes this much more practical than a shared disk/cache fusion architecture such as Oracle RAC.

Updating Relational Data by SPARUL Statements. In many cases, an RDF view contains quad map patterns that maps all columns of some table into triples in such a way that sets of triples made from different columns are "obviously" pairwise disjoint and invoked IRI classes are bijections. E.g., quad map patterns for RDF property tables usually satisfy these restrictions because different columns are for different predicates and column values are used as object literals unchanged. We are presently extending SPARUL compiler and run-time in order to make such RDF views updatable.

The translation of a given RDF graph into SQL data manipulation statement begins with extracting all SQL values from all calculatable fields of triples and partitioning the graph into groups of triples, one group per one distinct extracted primary key of some source table. Some triples may become members of more than one group, e.g., a triple may specify relation between two table rows. After integrity check, every group is converted into one insert or delete statement.

The partitioning of N triples requires $O(N \ln N)$ operations and keeps data in memory so it's bad for big dump/restore operations but pretty effecient for transactions of limited size, like individual bookeeping records, personal FOAF files etc.

7 Conclusion

Experience with Virtuoso has encountered most of the known issues of RDF storage and has shown that without overwhelmingly large modifications, a relational engine can be molded to efficiently support RDF. This has also resulted in generic features which benefit the relational side of the product as well. The reason is that RDF makes relatively greater demands on a DBMS than relational applications dealing with the same data.

Applications such as `www.pingthesemanticweb.com` and ODS for social networks and on-line collaboration have proven to be good test-beds for the technology. Optimizing queries produced by expanding SPARQL into unions of multiple storage scenarios has proven to be complex and needing more work. Still, it remains possible to write short queries which are almost impossible to efficiently evaluate, especially if they join between data that may come from many alternate relational sources. In complex EDI scenarios, some restrictions on possible query types may have to be introduced.

The community working on RDF storage will need to work on interoperability, standard benchmarks and SPARQL end point self-description. Through the community addressing these issues, the users may better assess which tool is best suited for which scale of problem and vendors may provide support for query and storage federation in an Internet-scale multi-vendor semantic web infrastructure.

Further details on the SQL to RDF mapping and triple storage performance issues are found in separate papers on the http://virtuoso.openlinksw.com site.

Literatur

[1] Beckett, D.: Redland RDF Application Framework. http://librdf.org/

[2] Bizer, C., Cyganiak, R., Garbers, J., Maresch, O.: D2RQ: Treating Non-RDF Databases as Virtual RDF Graphs. http://sites.wiwiss.fu-berlin.de/suhl/bizer/D2RQ/

[3] Chen, H., Wang, Y., Wang, H. et al.: Towards a Semantic Web of Relational Databases: a Practical Semantic Toolkit and an In-Use Case from Traditional Chinese Medicine. http://iswc2006.semanticweb.org/items/Chen2006kx.pdf

[4] Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Journal of Web Semantics 3(2), 2005, pp158–182.
Available via http://www.websemanticsjournal.org/ps/pub/2005-16

[5] Prudhommeaux E.: SPASQL: SPARQL Support In MySQL.
http://xtech06.usefulinc.com/schedule/paper/156

[6] 3store, an RDF triple store. http://sourceforge.net/projects/threestore

[7] Intellidimension RDF Gateway. http://www.intellidimension.com

[8] Jena Semantic Web Framework. http://jena.sourceforge.net/

[9] Northrop Grumman Corporation: Kowari Metastore. http://www.kowari.org/

[10] Oracle Semantic Technologies Center.
http://www.oracle.com/technology/tech/semantic_technologies/index.html

[11] Semantically-Interlinked Online Communities. http://sioc-project.org/

[12] Wikipedia3: A Conversion of the English Wikipedia into RDF.
http://labs.systemone.at/wikipedia3

[13] Extensible Business Reporting Language (XBRL) 2.1.
http://www.xbrl.org/Specification/XBRL-RECOMMENDATION-2003-12-31+Corrected-Errata-2006-12-18.rtf

[14] Bizer C., Heath T., Ayers D., Raimond Y.: Interlinking Open Data on the Web. 4th European Semantic Web Conference.
http://www.eswc2007.org/pdf/demo-pdf/LinkingOpenData.pdf

[15] Sören Auer, Jens Lehmann: What have Innsbruck and Leipzig in common? Extracting Semantics from Wiki Content 4th European Semantic Web Conference.
http://www.informatik.uni-leipzig.de/ auer/publication/ExtractingSemantics.pdf

[16] About MusicBrainz. http://musicbrainz.org/doc/AboutMusicBrainz

[17] About Geonames. http://www.geonames.org/about.html

[18] Ping The Semantic Web. http://pingthesemanticweb.com/about.php

[19] Alan Ruttenberg: Harnessing the Semantic Web to Answer Scientific Questions. 16th International World Wide Web Conference.
http://www.w3.org/2007/Talks/www2007-AnsweringScientificQuestions-Ruttenberg.pdf

[20] Oracle Real Application Clusters. http://www.oracle.com/database/rac_home.html