Main.VOSRDFWP

Implementing a SPARQL compliant RDF Triple Store using a SQL-ORDBMS

By: Orri Erling (Program Manager, OpenLink Virtuoso)

Abstract

Discussion of RDF-oriented customizations implemented in OpenLink Virtuoso to facilitate RDF-triple storage and the SPARQL query language, covering Data Types and Data Representation, SQL extensions for in-line SPARQL, SQL-Optimization techniques & Cost Model matters, Join Operation Algorithms, and other database-engine tuning matters.

Scope of discussion is limited to the ways the Virtuoso SQL ORDBMS Core was used to implement RDF data management. Issues such as translating pre-existing relational data to RDF and querying it via SPARQL (SQL to RDF Ontology Mapping) are not addressed. A different paper will address this aspect of Virtuoso's RDF functionality realm.

Contents

Table of Contents

Introduction

This paper discusses extending an existing Object-Relational Database Management System (ORDBMS) into an RDF Triple Store. Much work has been published concerning storing RDF triples in relational databases, most notably in relation to Oracle's recent exploits in version 10.2.x and to several Triple Stores that have been implemented atop MySQL and more generically atop ODBC and JDBC. This work concentrates primarily on how we have used pre-existing functionality to deliver a high-performance and SPARQL compliant RDF Triple Store.

However, in our case, the same individuals who created the underlying RDBMS are doing the RDF work, so we have more latitude than usual for low-level tailoring and extending the database for RDF use. Useful applications of these features are not limited to RDF, although RDF support may have been the initial catalyst for their introduction. We discuss both the database engine and query language aspects of the RDF-enhancing of Virtuoso, as these are often tightly interwoven.

What is RDF Data Management in Virtuoso?

RDF Data Sets are managed by an additional, dedicated module, within the Virtuoso ORDBMS core. This functionality is exposed to client applications through an implementation of the SPARQL Query Language and Protocol, plus a collection of Web Services and Virtuoso/PL based APIs for Creating, Updating, and Deleting RDF Data Sets.

Why is RDF Data Management Functionality Important?

Virtuoso offers an object-relational database, virtual/federated database, a powerful procedure language, Java and .NET run time hosting, and web and web services access to all of these. These features plus Virtuoso's strong XML support (built-in XQuery, XPath, XSLT, XML Schema) make it an attractive platform for the new crop of Web-2.0 and Semantic Web applications. For example, a single Virtuoso server can provide storage, dynamic web pages, and various XML feed formats for any of these application profiles.

We see Web 2.0 as a critical Semantic Web precursor, and complementary to RDF. It remains our public position that the relationship between Web 2.0 and the Semantic Web is symbiotic, since Web 2.0 needs the richness of the RDF Data Model, while the Semantic Web requires the loose coupling of Web 2.0. Without Web 2.0, the Semantic Web is difficult to comprehend in a broad sense as anything more than an egalitarian vision. Web 2.0 on the other hand, cannot scale in terms of value that it provides without the Open Data Access and Data Model flexibility of the Semantic Web. The issue of Data Access goes way beyond the generation of

URL accessible plain old RDF files such as FOAF and SIOC or RSS and Atom from relational databases.

We anticipate that fusion of the Web 2.0 and Semantic Web paradigms will be triggered by a realization on the part of Web 2.0 applications and services users that their Web applications (Blogs, Wikis, Feed Aggregators, Bookmark Mangers, Social Networks, Discussion Forums, Photo Sharing Services, etc.) require the same open data access and atomic data containment (Data Spaces) widely available to traditional desktop and enterprise applications. Building Silos and Walled-Gardens in some new context (i.e., Web 2.0) isn't the nirvana anticipated by Web users.

We see the transparent integration of Web 2.0 (and other) data sources with the RDF data model as the critical catalyst for the transition from a purely "programmable web" to a "programmable and queryable web of databases", which we call Data Spaces. In Semantic Web parlance, this implies representing non-RDF data as RDF data by way of Ontology mapping. It is important to note that this mapping should be achievable without imposing any burden at the user level.

How is it Implemented?

Transparently representing non-RDF data as RDF data via Ontology mapping entails either storing application triples in an RDF Triple Store or converting SPARQL queries to SQL (called SQL rewriting) and reformatting the results back into an RDF form. Naturally, Virtuoso provides solutions for both scenarios, but in this paper the focus is on the implementation of its native RDF Triple Store.

Virtuoso's RDF storage has been developed using existing infrastructure, providing significant advantages over starting from scratch. Especially, since Virtuoso possesses its own SQL-200n implementation; meaning we can process SPARQL queries using the existing query processor, thereby inheriting the query optimizer, and all of the time-tested prowess of the SQL query language. We do not have to extend SPARQL in anyway to facilitate this fusion, instead, we simply embed SPARQL in SQL by treating SPARQL query patterns as subqueries (derived tables) inside SQL constructs such as GROUP BY and others aggregate oriented clauses, which do not exist in SPARQL at present.

The RDF Triple Store implementation work in Virtuoso affect the following areas of the Virtuoso ORDBMS core:

- Data Types
- Data Dictionary (System Tables and Indexes)
- Virtuoso SQL and SPARQL Fusion
- Join Operation Algorithms
- Data Import and Index Compaction

Data Types

The basic requirement of RDF is supporting a type system compatible with XML Schema. In

specific, this means having a distinct, recognizable type for URIs and having strings with language tags.

To this end we have added a SQL type called IRI_ID, an unsigned 32-bit integer representing the ID of an IRI. These are mapped to the string representation by a normal relational table with an index over both the ID and the string. This allows for 4G distinct IRI's, which is sufficient for data sets of up to billions of triples. We also have a 64-bit version of the type, should this prove necessary.

Since Virtuoso's SQL engine is typed at run-time, we can have a column of type ANY, which will hold serializations of arbitrary SQL scalars, including IRI_IDs. The ANY type offers a lexicographic ordering of all values of all data types and thus is valid for use in keys. We do not need any flag for distinguishing between IRI and other types of values, as is the case in some other relational RDF storage schemes. Also, with IRI_IDs, different types of numbers and strings are guaranteed to be disjointed at run time.

The object of a triple can be an arbitrarily long value. This makes it impossible to have the full value as part of an index key. Thus, if the value is a short string or non-string scalar, it is stored in-line. If it is a longer string, a few first bytes plus an ID indexing a table of long values is stored as the key part in the object column of the triples table.

This is different from, for example, Oracle's scheme, where all values are given an ID regardless of type.

The SPARQL language itself does not presuppose a full text index, but we see the obvious usefulness of one for many applications. In our scheme, it is convenient to store all Os of given predicate/graph combinations as "long" strings, with a few leading characters and an ID stored in the string in the triple table and the rest in the object values table. This object values table can then maintain a text index on selected rows. Since we control the text index, this being part of the core Virtuoso engine, we can easily decide at run time which objects will be indexed and when.

Since SQL strings do not support language tags, we have added a few leading bytes to strings for encoding these. These bytes are added and removed by conversion functions between the SQL and SPARQL representations of data. No special datatype is added for this since this would entail adding support for it in all existing string functions, which is inconvenient.

RDF System Data Dictionary (System Tables & Indexes)

The default RDF representation consists of the below tables:

RDF_QUAD table

Column Name	Data Type	Description
G	IRI_ID	Graph - Primary Key
S	IRI_ID	Subject - Primary Key
P	IRI_ID	Predicate - Primary Key

O	ANY	Object - Primary Key

```
create table "DB"."DBA"."RDF_QUAD"
(
  "G" IRI_ID,
  "S" IRI_ID,
  "P" IRI_ID,
  "O" ANY,
  PRIMARY KEY ("G", "S", "P", "O")
);
     Create index RDF_QUAD_PGOS on RDF_QUAD ( P, G, O, S);
```

RDF_URL Table

The RDF_URL table contains a mapping between internal IRI ids and their external form.

Column Name	Data Type	Description
RU_IID	IRI_ID	References the IRI ID from the G, S or P column in the RDF_QUAD table
RU_QNAME	VARCHAR	External Value of the IRI example: http:myopenlinksw.com/ods/kidhen/blog/foaf.rdf

```
create table "DB"."DBA"."RDF_URL"
(
  "RU_IID" IRI_ID,
  "RU_QNAME" VARCHAR,
  PRIMARY KEY ("RU_IID")
);
```

RDF_OBJ Table

Column Name	Data Type	Description
RO_ID	integer	Primary Key
RO_VAL	varchar	
RO_LONG	long varchar	

```
create table "DB"."DBA"."RDF_OBJ"
(
  "RO_ID" INTEGER,
  "RO_VAL" VARCHAR,
  "RO_LONG" LONG VARCHAR,
  PRIMARY KEY ("RO_ID")
);
```

This is the default structure for storing triples. With the Wikipedia and Wordnet data sets, we have an average length for Objects (O) of 9 bytes with all other columns of RDF_QUAD being fixed 4 bytes. This gives a theoretical space requirement of 2*27 bytes. The 27 bytes are

obtained by adding a index entry overhead of 4 bytes, 2 bytes for the length field of the variable length O, 3 * 4 bytes for graph (G), subject(S), predicate (P) plus the average 8 bytes of O content. The data rows are stored directly in the leaves of the index tree. Since all columns are in both indices, there is never a need to join from an index to the primary key index that holds the dependent part.

A leaf page of 8K, if 80% full, will hold 242 entries. Thus the overhead of non-leaf nodes is small. There is no key compression, so as to have random access to data on index pages, all pre-aligned and in machine byte order, without any search time decompression. Also we do not do read-time decompression so as to have constant-size buffers.

Some additional space-savings are possible if we have a table per graph, obviating the need to store the G. Having less less key part to compare gives a small gain in index seek time.

Unlike Oracle, we do not have flags for reification or fields for "parent" of nodes linked via blank nodes. For these types of situations, we believe that a SPMJV (subject predicate matrix join view) can be used for customizing the representation of selected groups of triples. This subject will be fully addressed in the paper on RDF to relational mapping.

Virtuoso SQL and SPARQL Fusion

SPARQL produces result sets just like SQL does. The construct and describe queries may be viewed as an aggregate over a result set, still fitting well into the SQL model. This means that the basic structure of a SQL run time, as well as SQL client libraries, are suited for SPARQL with only minimal changes, if that.

Embedding SPARQL into Virtuoso SQL offers the following benefits:

- Reuse of all clients, including ODBC, JDBC, ADO .NET, OLE/DB;
- Reuse of SQL run-time and query optimizer, as described below;
- Implicit access to all SQL aggregate functions such as GROUP BYs etc. by simply embedding SPARQL code inside a SELECT statement with group by: without any non-standard SPARQL extensions. The same is true for SQL existence tests, EXCEPT and other functions. The semantic web purists will say that this is anathema to the open-world assumption of the semantic web. We note, however, that things like SELECT TOP ... ORDER BY and similar constructs applied to triples can serve useful purposes in generating reports and data snapshots.
- A natural possibility to join between RDF and relational data without application programming, such as the use of two cursors in a loop. This applies equally well to relational data in Virtuoso as well as relational data outside of it, in case the commercial version with virtual database capabilities is used. Naturally, a single query optimizer decides makes the execution plan across both SPARQL and SQL. This would have to be hard-coded if an external application were used to join triples and relational data.

Virtuoso's SQL parser recognizes the SPARQL keyword, which introduces a SPARQL string. If this is in a subquery or derived table position, the SPARQL text is enclosed in parentheses, providing a natural lexical boundary.

When a SPARQL fragment is found, it is parsed with a SPARQL parser and converted to SQL. The SQL processor picks up from this point on. The result is a SQL-only parse tree that is further processed by the SQL compiler.

All G, S, P values are IRI_IDs; they need to be converted into strings to allow applications to be able to make use of them. Likewise, an O value can be a string with language tag or an IRI_ID, in which case it too needs to be converted for an application to make use of it. The SPARQL processor adds these conversion functions around all result columns that become visible to SQL. In special cases, the internal IRI_IDs etc can be returned but then the application needs to have special knowledge of the RDF storage format.

A SPARQL filter expression can use SQL functions. When calling these functions, internal representations like IRI_IDs and strings with language are also converted back into the SQL form.

The SQL NULL value is the natural representation of SPARQL unbound. Thus, a SQL outer-join directly corresponds to SPARQL optional. If the operand of optional is complex, then the right operand of LEFT OUTER JOIN just becomes a derived table (subquery) representing all the logic of the complex optional.

A union in SPARQL corresponds directly to a SQL UNION ALL. Complex queries may easily result in joins between unions. This is especially true if multiple triple-storage tables are involved. Generally, the SPARQL to SQL transformer prefers making unions of joins rather than joins of unions. Thus, when we have

(a union b) join (c union d)

we get

(a join c) union (a join d) union (b join c) union (b join d)

This makes it simpler to remove terms that are known to be intrinsically empty based on SPARQL compile-time information. Also, the SQL-execution graphs are generally smaller and faster in this form.

Cost Model and Real Time Statistics and Sampling

Virtuoso, as with most databases, has a batch-mode statistics-gathering operation that counts rows in tables and counts of distinct value of columns. This is often adequate for choosing SQL join orders and join types. Sometimes a column histogram can be useful for knowing the selectivity of a comparison of a column with a constant. For example, when comparing date columns in business intelligence reports, it is good to know at compile time whether the condition will select the whole history table or only a few most recent rows: if it is a few rows, use the date index, if it is the whole table, do a full table scan.

With RDF data tending to all be stored in one large table, the SQL statistics become much less useful than they are with regular SQL.

For example, the knowledge that there are 100 distinct values of P says nothing about there being more rdf:type triples than foaf:mbti triples. Also a histogram is not very useful because the IRI ID values have no obvious order or collation. Thus it is simply best to go look.

```
    Select * from <people> where { ?p rdf:type foaf:Person . ?p
foaf:mbti="ESFJ"}
```

Could be done in two join orders: loop over people using PGOS, where G and P and O are known, getting all the Ss for people. Then check for existence using SGPO or PGOS where the S is given by the previous lookup. The other execution plan is take all the ESFJ Os using PGIOS and do an existence check to see if this S is a person.

Since the MBTI type is likely a relatively rare property, the second plan is faster. This can be known at compile-time by just inspecting the actual index.

For any indexed search criteria consisting of zero or more leading equalities plus optionally one <=, <, >, or >= condition, where leading conditions are comparisons of key parts with constants, make a cursor with these criteria and take a sample. Taking a sample is like an index-lookup but this additionally counts for each level of the index tree how many ways it branches and how many of the leaf pointers match the condition. At the leaf, this will count the leaves matching the condition. If at any point there was more than one leaf pointer matching the criteria, we assume that the sibling subtrees are equally populous as the leftmost branch that we followed.

A special case is when we have no search criteria and are just interested in the count of the table. For this we do the same type of lookup, now following a random leaf pointer at each level. We repeat this until we have seen up to 1% of our estimated row-count, but no more than 100 times. When doing this, we also count distinct values of columns. This gives a fast but fairly accurate estimate of the table overall. The results are remembered and another sample is done when inserts+deletes exceed 20% of the previous count estimate.

Any lookup with conditions is guaranteed not to seek more than one leaf. If this becomes a disk access, nothing is lost because the actual execution of the query would anyway have made the access.

We note that if the sample query had instead filtered for foaf:GeekScore > 10, the same estimates could have been done and this condition would still have been found more selective than being a person. The PGOS index would have an equality for P and G and a > for O.

Join Operations Algorithms

Merge Join and 'AND' of Triple Patterns

There is still one more way to resolve the sample join of P being a person with MBTI = ESFJ.

This is doing a merge join of PGOS for P = rdf_type and O = foaf:Person and PGOS for P = foaf:MBTI and O = "ESFJ", the Ss being declared equal.

This is a merge AND of the set of Ss where the S is a foaf:Person and where the S if has foaf:MBTI = "ESFJ". We note that this works for the ESFJ person query but not for the person with GeekScore > 10 query.

Whereas the loop join has the complexity of _O(log (n) + m * log (n))_, where n is the count of the index and m the cardinality of the first lookup, the merge join is closer to _O(log (n) + log (n) + x + y)_, where n is the count of the index and x and y are the cardinalities of the result sets of the two lookups.

This trick is sometimes done in relational databases when there are equality conditions on two different indexed columns. The indices are merge AND-ed before looking for the rows. This also allows for read-ahead in both indices, whereas a loop over one index and a nested lookup would allow read-ahead only on the driving index.

We have extended this trick for self-joins, as these are ubiquitous with SPARQL.

Data Import and Index Compaction

Experience shows that import of any large data set becomes IO-bound, with the GSPO index being inserted in roughly ascending order, as new S IRI_IDs get made. The other index, PGOS, is basically in random order, with some ascending sequences for individual Ss having the same value in the same predicate.

Having a multithreaded loader that allows for inserts that fit in cache to complete while IO is being served for one or more cache misses is essential for adequate performance.

This causes insert order on GSPO to become random within a few pages, thus causing pages to split in the middle instead of splitting 9:1, as they would for a purely ascending insert.

This leads to a fairly poor fill-rate on index pages. This can be ameliorated by taking groups of 3-4 "old dirty" index leaves, ones that are about to go to disk and compacting them to one less page. If this is limited to pages that are not in the hottest working set, the interference with other processing is minimal and the compaction cost is negligible in comparison with the cost of writing, let alone reading an extra page.

This optimization will be further measured in the future. This is not limited to RDF but specially RDF storage will benefit, as will text indexing, both consisting of randomly inserted usually short rows.

Conclusions and Future Directions

We have found that access to and mastery of the RDBMS core greatly facilitates efficient implementation of a triple store. Even in cases such as Oracle's, where the same company produces both elements, the size of the organization and the involvement of two different departments may make the integration less smooth.

Planned enhancements to Virtuoso's RDF support include:

- Increased integration with RDF front-end and visualization tools.
- Testing with the OpenLink Data Spaces applications suite, providing SIOC, FOAF, ATOM OWL and other ontologies for querying blogs, news feeds, wikis and other ODS application data.
- Mapping of SPARQL queries to pre-existing relational databases. A separate paper will discuss this effort. This will also cover native RDF storage optimizations a la SPMJV's of Oracle.
- Addition of text-indexing for selected triples with a SPARQL extension text_contains predicate.
- Comparisons with non-relational triple stores such as Kowari. To date we have to date seen no need for bitmap indices but later research may uncover such needs.
- Federating SPARQL queries across multiple external SPARQL end points. This is a complex subject and will have its own paper.

Related Items

Other OpenLink Resources

- ODS
- Virtuso RDF
- Metadata Extraction
- RDF FAQ
- Virtuoso RDF Triple Store
- Virtuoso SPARQL
- SPARQL Demonstration

Other SPARQL-Compliant RDF Triple Stores

- Kowari
- JENA
- Sesame

References

Articles & White Papers

- Eric P. Xing on relational
- Oracle on SPMJV's

Ontologies and Specs

- OWL
- ATOM
- FOAF
- SIOC

CategoryVOS CategorySPARQL CategoryODS CategoryRDF CategoryWhitepaper