Towards Web-Scale RDF

Orri Erling (Program Manager, OpenLink Virtuoso) and Ivan Mikhailov (Lead Developer, OpenLink Virtuoso) oerling{at}openlinksw.com and imikhailov{at}openlinksw.com

OpenLink Software, 10 Burlington Mall Road Suite 265 Burlington, MA 01803 U.S.A. http://www.openlinksw.com/

Table of Contents

Abstract

We are witnessing the first stages of the document Web becoming a data Web, with the implied new opportunities for discovering, re-purposing, "meshing up," and analyzing linked data. There is an increasing volume of linked open data, and the first data Web search engines are taking shape. Dealing with queries against the nascent data Web may easily add two orders of magnitude in computing power requirements on top of what a text search engine faces. Queries may involve arbitrary joining, aggregation, filtering, and so forth, compounded by the need for inference and on-the- fly schema mapping.

This is the environment for which Virtuoso Cluster Edition is intended. This paper presents the main challenges encountered and solutions arrived at during the development of this software.

We present adaptations of RDF load- and query-execution and query-planning suited for distributed memory platforms, with special emphasis on dealing with message latency and the special operations required by RDF.

Introduction

Virtuoso is a general-purpose RDBMS with extensive RDF adaptations.

In Virtuoso, RDF data may be stored as RDF quads (i.e., graph, subject, predicate, object tuples). All such quads are in one table, which may have different indexing depending on the expected query load.

RDF data may also be generated-on-demand by SPARQL queries against a virtual graph mapped from relational data, which may reside in Virtuoso tables or tables managed by any third party RDBMS. The "relational-to-RDF mapping" capability is further described in Declaring RDF Views of SQL Data [1]; this paper limits itself to discussing physically-stored RDF quads.

We recognize two main use cases of RDF data, which we may call the "Data Warehouse" and the "Open Web" scenarios. The Data Warehouse is built to serve a specific application, and can be laid out as a collection of relatively few graphs with well defined schemes. Since the application is known, domain experts can specify what inference is relevant, and the results of such inference can often be forward chained. Since data are loaded through custom ETL procedures, the identities of entities can often be homogenized at load time, so that the same URI ends up standing for the same thing even when the identifiers in the original data may differ.

The Open Web scenario is found when crawling data from the Web for search or Web analytics, or linked data mesh-ups. Data are often automatically discovered, provenance becomes important, and it is no longer possible to exhaustively list all graphs that may participate in a query's evaluation. Forward chaining inferred data becomes problematic due to large volumes, heterogeneous schemes, relative abundance of `owl:sameAs` links, and so forth. Also, Web-scale data volumes will typically require redundant infrastructure for uptime due to expected equipment and network failures.

Virtuoso Cluster Edition is intended to be configurable for both use cases.

Database Engine

The Virtuoso DBMS and its main RDF-oriented features are described in RDF Support in the Virtuoso DBMS [2].

Virtuoso has a general-purpose relational database engine enhanced with RDF-oriented data types such as IRIs and language- and type-tagged strings. Virtuoso makes extensive use of bitmap indices to improve space efficiency [3]. The default index layout uses `GSPO` (`graph, subject, predicate, object`) as the primary key, and `OPGS` as a bitmap index. These two indices are usually adequate for dealing with queries where the graph is known.

For cases where the graph is left open, the recommended index layout is `SPOG` as primary key, with `OPGS`, `GPOS`, and `POGS` as bitmap indices. The bitmap index means that in the case of `OPGS`, for example, for each distinct `OPG` (`object, predicate, graph`), there is a bitmap with 1 bit corresponding to each subject which has object `O` as a value of property `P` in graph `G`.

With typical RDF data, such as DBpedia [4] version 3, the bitmap index takes about 60% of the corresponding non-bitmap index space.

Both regular and bitmap indices use key compression, which collapses 64-bit IDs into 16 bits when the ID is within an integer increment of 16 from a previous ID on the same page. Common prefixes for strings are also eliminated. An index compressed in this manner, using 64-bit IDs takes 56% of the space of a non-compressed index with the same content but with 32-bit IDs.

After key compression is applied, using gzip gives a further almost 50% gain, i.e., 95% of all 8K pages drop to under 4K. Many pages compress to less than this but the percentage of pages that do not fit in the target compressed size must be kept small to maintain locality. The cost of compression is low — about 600 microseconds for compressing a page and a quarter of this for uncompressing. Pages in cache must be kept uncompressed since a random access of one triple out of hundreds of millions is only around 4-5 microseconds for data in memory; thus, applying gunzip at each of the usually 4 index tree levels would increase the time to about 600 microseconds. Stream compression is not good for database disk cache but does make for smaller files, easier backup, and better utilization of the hardware/OS disk cache.

Query Planning

Optimizing SPARQL queries against a quad store is not fundamentally different from optimizing SQL against a general purpose RDBMS. Still, regular SQL optimization statistics do not provide the requisite level of detail for RDF use cases. For a cost model to work well with RDF, it must be able to guess a match count for quads where any combination of `GSPO` is either equal to a constant, equal to a value known only at query time, or left open.

Pre-calculated histogram-style statistics do not answer these questions very well. This is why Virtuoso takes the approach of sampling the database at query optimization time. For example, when `G` and `S` are given, it is efficient to get a ballpark count of the matching `P, O` tuples by simply looking up the first match and counting matches on the same page. If the page ends with a match, also count the pages referenced from the parent of this page if they begin with a match, but do not read these; just assume their average row length to be the same as that of the first leaf page. For low cardinality cases, this often gives exact counts since all the matches are on a page; for high cardinality cases, taking a single page sample can often hit within 70% of the count when the count is in the millions.

Once the cardinalities are known, costing the queries is no different from SQL and is handled by the same code.

One difference from SQL is that hash joins are almost never preferred for RDF data. The reasons are that there pretty much always is an index that can be used, and that a full table scan is almost unknown.

On Shared Memory, MP, and Latency

A multi-core processor running a database will not deliver linear scale even when there is one query per core and the queries do not have lock contention. As a rule of thumb, a 4-core Xeon runs 4 query streams in 1.3 times the time it takes to run one of the streams, supposing the queries do not hit exactly the same data in the same order, the data is already in memory, and there is no wait for locks. This is roughly true of Virtuoso and other databases.

This is a best case. The worst case can easily destroy any benefit of SMP. If a thread has to wait for a mutex, the cost of the wait can be several microseconds even if the mutex is released 100 ns after the wait starts. If there is a pool of worker threads to serve a job queue, any time the queue goes empty will also cost about this much. We remember that a single triple lookup is about 4 μs; thus, spinning a thread to do a background single triple operation makes no sense. At least a dozen operations have to be dispatched together to absorb the cost of waiting for a thread to start and eventually blocking to wait for its completion. One must never think that multithreading is an end in itself.

On Networks and Latency

A round trip of sending one byte back and forth between processes on the same CPU takes as much as 20 μs real time over Unix domain sockets. Adding thread scheduling to this, as would be found in any real server process, makes the round trip 50 μs. The same takes about 140 μs on a 1Gbit Ethernet with no other traffic. We have a test program which runs n threads on each node of a cluster. Each of these threads sends a ping carrying x bytes of payload to every other node of the cluster and waits for the reply from all before sending the next ping. This creates a full duplex traffic pattern between all pairs of cluster nodes with intermittent sync.

4 processes on 4 core SMP box

Message length	Aggregate round trips/s	Aggregate MB/s
1,000	37,000	74
10,000	17,200	329
100,000	2,380	455

4 processes on 4 extra large AMIs on Amazon EC2

Message length	Aggregate round trips/s	Aggregate MB/s

1,000	10,000	20
10,000	3500	67
100,000	950	181

The round trips count is the number of messages sent by any node, divided by 2, multiplied by the duration in seconds. The MB/s is the sum total of data sent by all nodes during the interval, divided by the length of the interval.

Comparing these latencies with a single-triple in-memory random-access time of 4 μs shows that clustering is not an end in itself. The principal value of clustering is the fact that there is no limit to the amount of RAM nor RAM bandwidth.

Thus, it is evident that no benefit can be had from clustering unless messages are made to carry the maximum number of operations possible.

Partitioning vs Cache Fusion

Clustered databases have traditionally partitioned data between machines according to the values of one or more columns of a table. Another approach is cache fusion as in Oracle RAC [5]. With a cache fusion database, all machines of the cluster see the same disks but keep their local cache of this and have a cache-coherence protocol for managing concurrent update. We have not measured Oracle RAC but it is our impression that either an index lookup must be sent to the machine that holds the next page needed by the lookup or that the page must be transferred to the node making the lookup. In the latter case, we quickly get the same working set cached on all nodes. In the former case, we have a message round trip per page traversed, typically 4 round trips for a 4 level index tree. Either seems prohibitive in light of the fact that a single lookup is a few microseconds when all the data is local and in memory. This is true of Oracle as well as Virtuoso.

For this reason, we decided to go for partitioning. Most databases specify partitioning at the table level. We specify it at the index level, thus different keys of the same table may reside on different machines.

Proponents of cache fusion correctly point out that users do not know how to partition databases and that repartitioning a big database is next to impossible due to resulting downtime. The difficulty is reduced in the case of RDF since only a few tables are used for the data and they come pre-partitioned. The repartitioning argument is still valid in part.

We recognize that a Web-scale system simply cannot depend on a partitioning map that is set once-and-for-all, nor require reinserting the data when reallocating hardware resources. Google's Bigtable [6] and Amazon's Dynamo [7] each address this in different ways.

With Virtuoso, we have hash partitioning where the hash picks a logical partition out of a space of *n* logical partitions, where *n* is a number several times larger than the expected maximum machine count. Each logical partition is then assigned to a physical machine. When the machine allocation changes, logical partitions may be moved between nodes. When a partition is being moved, it continues to be served from the machine initially hosting it but a special log is kept for updates that hit already copied rows of the partition. Once the copy is complete, the partition is made read-only, the log is applied to the new host and subsequent queries are routed to the new host of the logical partition. Repartitioning is still a fairly heavy operation but does not involve downtime. Since one database file-set hosts many logical partitions, unequal slices can be allocated according to hardware capacity. Still, more flexibility could be had if each logical partition had its own database file-set. Then moving the partition would be a file copy instead of a database insert plus a delete of the logical content. The latter arrangement may be implemented later; it was not done now due to it involving more code.

Latency-Tolerant Load and Query Execution

Load

When loading RDF data, the database must translate between IRIs and literals and their internal IDs. It must then insert the resulting quad in each index. With a single process, as long as no data needs to be read from disk, the load rate is about 15Kt (kilotriples) per core.

Making a round trip per triple is out of the question. The load takes a series of 10,000 triples, and then for each unique IRI/literal, sends a request to allocate/return an ID for the node to the cluster node responsible for the partition given by the name. Whenever all the fields of the triple are known, each index entry of the triple gets put in the inserts queued for the box holding the partition. In this way, a batch of arbitrarily many triples can be inserted in a maximum of 4 round trips, each round trip consisting of messages that evenly fan out between machines.

In this way, even when all processes are on a single SMP box, clustered load is actually faster than single process load. The reason is that single process load suffers from waits for serializing access to shared data structures in the index. We remember that a single mutex wait takes as long as a full single-key insert, i.e., 5-6 μs.

Query

An RDF query primarily consists of single-key lookups grouped in nested-loop joins. Sometimes there are also bitmap intersections. Most result set columns are calculated by function calls since the internal IDs of IRIs and objects must be translated to text for return to the application.

The basic query is therefore a pipeline of steps, where most steps are individually partitioned operations. Sometimes consecutive steps can be partitioned together and dispatched as a unit.

The pattern

```
{?x a ub:Professor . ?x teacher_of <student> }
```

is a bitmap intersection where the `Professor` bits are merge-intersected with the `teacher_of` bits of <student>.

```
{ ?x a ub:Professor . ?x teaches_course ?c }
```

is a loop join starting with the `Professor` bitmap and then retrieving the courses taught from an index.

The whole query

```
select * from <lubm> where { ?x a ub:Professor ; ub:AdvisorOf ?y
}
```

is a pipeline of 4 steps:

1. translating the IRIs of the constants to IDs,
2. getting the professors,
3. getting the students they advise, and
4. translating the IDs to text.

```
select * from <lubm> where
{ ?x a ub:Professor ; ub:advisorOf ?y ; ub:telephone ?tel }
```

is still a pipeline of 4 steps because the two `ub:advisorOf` and `ub:telephone` property retrievals are co-located, since they have the same subject and the `GSPO` index is partitioned on subject.

The results have to be retrieved in deterministic order for result set slicing. If there is an explicit `ORDER BY` or an aggregate, this is no longer the case, and results can be processed in the order they become available.

Each step of the pipeline takes *n* inputs of the previous stage, partitions them, and sends a single message to each cluster node involved. If intermediate sets are large, they are processed in consecutive chunks. Execution of pipeline steps may overlap in time, and generally a step is divided over multiple partitions.

Normally, one thread per query per node is used. Making too many threads will simply congest the index due to possible mutex waits. On an idle machine, it may make sense to serve a batch of lookups on two threads instead of one, though. Further, since requests come in batches, if a lookup requires a disk read, the disk read can be started in the background and the next index lookup started until this too would need disk, and so on. This has the benefit of sorting a random set of disk cache misses into an ascending read sequence.

Distributed Pipe and Map-Reduce

As said before, RDF queries operate with IDs but must return the corresponding text. This implies a partitioned function call for each result column. Virtuoso SQL has a generic partitioned pipe feature. This takes a row of function/argument pairs, partitions these by some feature of the arguments, and returns the results for each input row once all the functions on the row have returned. This may be done preserving order or as results are available. It is possible also to block waiting for the whole pipe to be empty. The operations may have side effects and may either commit singly or be bound together in a single distributed transaction.

Aside from returning a result, the partitioned pipe function may return a set of follow-up functions and their arguments. These get partitioned and dispatched in turn. Thus, this single operation can juggle multiple consecutive map or reduce steps. There is a SQL procedure language API for this, but most importantly, the SQL compiler generates these constructs automatically when function calls occur in queries.

Inference for the Web — The Blessing and Bane of "sameAs"

When there is a well understood application and data is curated before import, entailed facts may often be forward-chained and identifiers made consistent. In a multiuser Web scenario, it is not possible for everybody to materialize the consequences of their particular rules over all the data. Thus, inference must take place when needed, not in anticipation of maybe being needed sometime.

Sub-properties and subclasses are easy to deal with at query run time. Given the proper pragmas, Virtuoso SPARQL will take

```
{ ?x ub:Professor . ?x ub:worksFor ?dept }
```

and generate the code for first looping over all subclasses of ub:Professor, and then all sub-properties of ub:worksFor. This stays a two step pipeline since the cluster node running the query knows the subclasses and sub-properties. With some luck, assistant professors and full professors will be in a different partition, adding some lateral parallelism to the operation.

The case of "sameAs" or transitive properties in general, such as part-of, are more complex. The principal problem is that for a pattern like {<subpart> part-of <whole>}, it is not self-evident whether one should go from <subpart> up or from <whole> down. Also, the cardinalities at each level, as well as the depth of the tree, are hard to guess.

"sameAs" is specially supported as an intermediate query graph node. It has no special cost model but it will take all fixed fields of the next join step and expand these into their "sameAs" statements, to full transitive closure. The feature is enabled for the whole query or for a single triple pattern with a SPARQL pragma. In a cluster situation, it is possible to just initialize the "sameAs" expansion when first reaching the place and to continue with the value one has as normally. In this event, if there are no "sameAs" statements, no extra pipeline step is added; the existing step just gets two more operations one looking for >?sas owl:sameAs ?thing and the other for ?thing owl:sameAs ?sas. If synonyms are found, they can be fed back into the step.

On Redundancy

A Web-scale RDF store will inevitably be quite large. One may count 16G of RAM per machine and about 1 billion triples per 16G RAM to keep a reasonable working set. 100 billion triples would be 100 machines. Of course, fitting infinitely many triples on disk is possible but when the memory to disk ratio deteriorates running queries of any complexity is not possible on-line.

As a basis for the above, one may consider that DBpedia with 198M triples is about 2M database pages, 16Gb without gzip. If data have strong locality, then about 5 times this could fit on a box without destroying working set. As machines are multiplied, failures become more common and failover becomes important.

We address this by allowing each logical partition to be allocated on multiple nodes. At query time, a randomly selected partition is used to answer the query if the data is not local. At update time, all copies are updated in the same transaction. This is transparent and is used for example for all schema information that is replicated on all nodes.

Storing each partition in duplicate or triplicate has little effect on load rate and can balance query load. Fault tolerance is obtained as a bonus. At present, the replicated storage is in regular use but a special RDF adaptation of this with administration, automatic reconstruction of failed partitions, etc., has to be done.

Some Metrics

Load

When loading data at a rate of 40 Ktriples/s, the network traffic is 170 messages/s and the aggregate throughput is 10MB/s. Since the load is divided evenly over all node-node connections, there is no real network congestion, and scale can be increased without hitting a network bottleneck.

Query

We have run the LUBM query mix against a 4-process Virtuoso cluster on one 4-core SMP box. With one test driver attached to each of the server processes, we get 330% of 400% server CPU load on servers and 30% on test drivers. During the test, the cluster interconnect cross sectional traffic is 1620 messages/s at 18MB/s while the aggregate query rate is 34 queries/s.

We see that we are not even near the maximum interconnect throughputs described earlier and that we can run complex queries with reasonable numbers of messages, about 1620/34 = 47 messages per query. The count includes both request and response messages.

The specifics of the test driver and query mix are given in the Virtuoso LUBM Load [8] blog post. The only difference was that a Virtuoso Cluster v6 instance was used instead.

Linked Data Applications

As of this writing, OpenLink hosts several billion triples worth of linked open data [9]. These are being transferred to Virtuoso Cluster v6 servers as of the time of this writing. In addition, the data aggregated from the Web by Zitgist are being moved to Virtuoso Cluster v6. Experiments are also being undertaken with the Sindice semantic Web search engine.

Future Directions

The linked data business model will have to do with timeliness and quality of data and references. Data are becoming a utility. Thus far there has been text search at arbitrary scale. Next there will be analytics and meshups at Web-scale. This requires a cloud data and cloud computing model since no single data center of Google, Yahoo, or any other, can accommodate such a diverse and unpredictable load. Thus the ones needing the analysis will have to pay for the processing power, but this must be adaptive and demand-based. Our work is to provide rapid deployment of arbitrary scale RDF and other database systems for the clouds. This involves also automatic partitioning and repartitioning as mentioned earlier. Google and Amazon have work in this direction but we may be the first to provide Bigtable- or Dynamo-like automatic adaptation for a system with general-purpose relational transaction semantics and full-strength query languages.

Conclusions

Aside from this, adapting query-planning cost models to data that contains increasing inference will be relevant for backward-chaining support of more and more complex inference steps. Also, we believe that common graph algorithms such as shortest-path, spanning-tree, and traveling-salesman, may have to become query language primitives, because their implementation in a cluster environment is non-trivial to do efficiently.

Appendix A — Metrics and Environment

We use version 2 of DBpedia [4] as a sample data set for RDF storage space unless otherwise indicated. When CPU speeds are discussed, they have been measured with a 2GHz Intel Xeon 5130 unless otherwise indicated. Networks are Gbit ethernet with Linksys switches.

References

1. Orri Erling: Declaring RDF Views of SQL Data, <http://www.w3.org/2007/03/RdfRDB/papers/erling.html>
2. Orri Erling, Ivan Mikhailov: RDF Support in the Virtuoso DBMS. In Franconi et al. (eds), Proc. of the 1st Conference on Social Semantic Web, Leipzig, Germany, Sep 26-28, 2007, CEUR Proceedings, ISSN 1613-0073, <http://CEUR-WS.org/Vol301/Paper%205%20Erling.pdf>.
3. Orri Erling: Advances in Virtuoso RDF Triple Storage (Bitmap Indexing) <http://virtuoso.openlinksw.com/wiki/main/Main/VOSBitmapIndexing>
4. Soeren Auer, Jens Lehmann: What have Innsbruck and Leipzig in common? Extracting Semantics from Wiki Content. In Franconi et al. (eds), Proceedings of European

Semantic Web Conference (ESWC07), LNCS 4519, pp. 503517, Springer, 2007, <http://www.informatik.uni-leipzig.de/~auer/publication/ExtractingSemantics.pdf>.
5. Oracle Real Application Clusters. <http://www.oracle.com/technology/products/database/clustering/index.html>
6. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al.: Bigtable: A Distributed Storage System for Structured Data. <http://labs.google.com/papers/bigtable-osdi06.pdf>
7. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani et al: Dynamo: Amazon's Highly Available Key-value Store. <http://www.scs.stanford.edu/08sp-cs144/sched/readings/amazon-dynamo-sosp2007.pdf>.
8. Orri Erling: Virtuoso LUBM Load <http://www.openlinksw.com/weblog/oerling/?id=1284>
9. Data set for the Billion Triples Challenge <http://www.cs.vu.nl/~pmika/swc/btc.html>