Towards Web Scale RDF

Orri Erling

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

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 product.
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.

1 Motivation

Scalability has become the key issue of deploying semantic/data web technologies. Virtuoso Cluster Edition is intended as a general purpose RDF platform scalable to the teratriple range. A web scale RDF infrastructure system must combine methods from across the database spectrum. No single algorithm or data structure or processing model will in itself be sufficient. Our emphasis is on the right combination of techniques rather than on any single one. We must cover from space efficient storage to latency tolerant parallel execution models.

Specifically, the issue of latency and latency tolerance becomes vital as soon as database operations are in the microsecond range. This issue is not widely covered in the RDF clustered database papers, e.g. Harth et al[13],

In designing Virtuoso cluster edition, we have had to consider issues of Oracle RAC -style cache fusion versus partitioning and of whether map-reduce is the scale out panacea it is sometimes touted as being. (DeWitt et al)[10]. Thus we discuss the rationale for choosing partitioning and the limits of map-reduce applicability to database workloads. We also review the basic facts of SMP (symmetric multiprocessing), multithreading, latency and network throughput as a basis for our engineering decisions.

2 Database Engine

Virtuoso is a general purpose RDBMS with extensive RDF adaptations, such as RDF-oriented data types (IRI's and language and type tagged strings) and SPARQL-to-SQL front-end compiler (SPARQL queries may appear in stored procedures and may be sent via ODBC, JDBC and other similar protocols). Its main RDF oriented features are described in[2].

In Virtuoso, RDF data can 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 can also be generated on demand by SPARQL queries against a virtual graph mapped from relational data in Virtuoso tables or tables managed by any third party RDBMS. The "relational-to-RDF mapping" capability is further described in[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 scenario.

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 schemas. 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 use case is found when crawling data from the web for search or web analytics or linked data mesh-ups. Data are often automatically discovered and 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 problematical due to large volumes, heterogeneous schemas, relative abundance of `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.

3 Quad Storage Layout

Virtuoso makes extensive use of bitmap indices for improving space efficiency[8]. The *bitmap* index means that in the case of `OPGS` for example, for each distinct `OPG` (object, predicate, graph) there is a bitmap with a 1 bit corresponding to each subject which has object `O` as a value of property `P` in graph `G`.

The default index layout is `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` for primary key, `OPGS`, `GPOS` and `POGS` as bitmap indices.

We leave the index choice up to the deployment[20].

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

Both regular and bitmap indices use key compression which collapses 64 bit id's 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 id's takes 56% of the space a non-compressed index with the same content but with 32 bit id's takes.

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, i.e. 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. Thus 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.

3.1 A Word on Vertical Storage

Vertical storage as in Monet[11], Vertica[12], Paraccel[18] and other recent analytics databases, has gained popularity in data warehouse circles and even been applied to RDF (Vertica[12], Abadi, et al[14]). Unlike closed data warehouse use cases, we find ourselves faced with unanticipated integration and ragged data, as on the open web, as experienced by Garlik, among others. Hence we do not wish to be bound to a table per predicate because this is hard to query and manage, specially if the predicate is left open or new predicates appear on the fly, which may happen. So we use a set of covering indices as Harth et al[13] (but we differ from their layout by the use of bitmap indices). A covering index means an index that contains all columns of the table or all columns that are needed by a specific query.

We have also experimented with quads stored vertically, i.e. a vertical table for each of `G`, `P`, `O`, `S` but find that a covering index with a multipart key performs better. With the RDF workload, we end up typically referencing all parts of each quad. This is not like in the business intelligence case where a typical query accesses 4 columns of a 15 column history table. Of the 4 RDF quad keys, at least 2 are generally given; Thus a common lookup becomes a merge intersection of two or three indices and random lookups by row id for the unspecified columns. We have measured the performance of such an index layout with Virtuoso and have found a multipart index covering GSPO to be about 3x more efficient.

Since we are not exclusively an RDF platform, vertical storage is of interest to us and we will do work in this space in addition to the row store functionality that has always been in Virtuoso.

4 Query Planning

Virtuoso supports SPARQL by means of a front-end compiler that translates SPARQL text into SQL text. The keyword `SPARQL` in SQL statement triggers the front-end compiler that cuts the text of SPARQL fragment after the keyword, composes equivalent SQL text and pastes it instead of the original fragment. The rest of the SQL processor sees no difference between SPARQL and SQL queries.

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. Therefore, 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.

Precalculated 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. When for example G and S are given, it is efficient to get a ballpark count of the P, O tuples matching 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 +/-30% 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 with SQL is that hash joins are almost never preferred for RDF data. The reason is that there pretty much always is an index that can be used and that a full table scan is almost unknown.

4.1 Related Work: Standalone RDF Frameworks

We have found tight integration between the quad store and the RDBMS engine to be necessary, which cannot be obtained by using MySQL, Postgres, Virtuoso or any other generic SQL database as a back-end connected via a client server connection to the SPARQL processor. This is done by default in Wilkinson et al[15], Broekstra et al[16], Harris et al[17] for example. In such a situation, Virtuoso would deliver little benefit as opposed to MySQL, primarily since interprocess latency would eat up the time, irrespective of DBMS engine speed. We measure 4-5us per single index lookup as opposed to a minimum of 50 us for client server round trip (multicore Xeon, Linux, Xeon 2GHz).

Furthermore, SQL optimization statistics do not give IRI-level cardinality estimates, which we can get by direct index sampling. This is crucial for query planning.

Virtuoso can be used as a back end of Jena[15], Sesame[16] and Redland. OpenLink has developed plug compatible drivers for using Virtuoso as a SPARQL

processor back end, thus bypassing the framework's native translation from SPARQL to SQL for MySQL etc. This allows benefiting from no latency between SPARQL execution and database and from RDF adapted statistics. Applications do not have to be altered for taking advantage of this. On the other hand, applications written using a triple pattern level API continue to be supported but do have to deal with the message latency between the RDF framework and the database.

5 On Shared Memory MP and Latency

A multicore processor running a database will not deliver linear scale even when there is a query per core and the queries do not have lock contention. As a rule of thumb that is roughly true for Virtuoso and other databases, 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, have the data in memory and do not wait for locks. This is a best case. The worst case can easily destroy any benefit from SMP. If a thread has to wait for a mutex, the cost of the wait can be several microseconds even if the mutex were released 100 ns after the wait started. If there is a pool of worker threads for serving a job queue, whenever the queue goes empty will cost about this much also. We remember that a single triple lookup is about 4 us. 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.

6 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 us 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 us. The same takes about 140 us 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 on between all pairs of cluster nodes with intermittent sync.

4 processes on 4 core SMP box:

Message length	1000	10000	100000
Aggregate round trips/s	37000	17200	2380
Aggregate MB/s	74	329	455

4 processes on 4 "extra large" AMI's on Amazon EC2:

Message length	1000	10000	100000
Aggregate round trips/s	10000	3500	950
Aggregate MB/s	20	67	181

The round trips count is the count of messages sent by any node divided by 2 x 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 us 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 or 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.

7 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[4]. 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.

Due to this 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 in part valid.

We recognize that a web scale system simply cannot depend on a once and for all set partitioning map or require reinserting the data when reallocating hardware resources. Google's Bigtable[5] and Amazon's Dynamo[6] both 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, databases can be allocated unequal slices 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 + delete of the logical content. The latter arrangement may be implemented later but was not done now due to it involving more code.

8 Latency Tolerant Load and Query Execution

8.1 Load

When loading RDF data, the database must translate from IRI's and literals to their internal id's. 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 CPU core.

Making a round trip per triple is out of the question. The load takes series of 10000 triples, and then for each unique IRI/literal, sends the 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 cluster node 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 us.

8.2 Query

An RDF query primarily consists of single key lookups grouped in nested loop joins. Sometimes there are also bitmap intersections. Most of result set columns are calculated by function calls since the internal ids of IRI's 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, one for translating the IRI's of the constants to id's, getting the professors, getting the students they advise, then translating the id's 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.

9 Distributed Pipe and Map-Reduce

As said before, RDF queries operate with id's 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 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.

10 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, everybody materializing the consequences of their particular rules over all the data is problematic. The obstacle is that different users/applications will have different needs and materializing everybody's entailed facts takes space and time, besides many of these facts may never be accessed if we have a lookup oriented application running off harvested data. Thus, when possible, inference should take place when needed, not in anticipation of maybe being needed.

Subproperties 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 subproperties of `ub:worksFor`. This stays a two step pipeline since the cluster node running the query knows the subclasses and subproperties. With some luck, assistant professors and full professors will be in a different partition, thus adding some laterall parallelism to the operation.

The `owl:equivalentClass` and `owl:equivalentPredicate` are taken into account when determining the set of sub/super classes or predicates.

Also, with

```
{ <AssistantProfessor1> a ?cl } ,
```

the bindings of `?cl` for all superclasses of `ub:AssistantProfessor` are automatically added to the result set without having to be stated in the database.

There is also support for limited inferencing in the A box for `owl:sameAs` and for more general transitivity.

`owl:sameAs` is specially supported as an intermediate query graph node. It has no special cost model but it will take all bound variables used as search criteria in the next join step and expand these into their `owl:sameAs`'s, 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 asynchronously initiate the `owl:sameAs` expansion when first reaching the place and to continue with the value one has as normally. In this event, if there are no `owl:sameAs`'s, 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. In this way, `owl:sameAs` adds some work but does not increase the number of synchronous message round trips needed for the query as long as there are no `owl:sameAs` assertions.

Virtuoso SQL and SPARQL have a special extension for transitive subqueries. In the general case, we have a derived table, i.e. in SQL a nested select in the from clause of a containing select. Such a derived table may be declared to be transitive. If so, some columns are declared as input and others as output. When the derived table is evaluated with a set of bindings for its input columns, it produces, for each such binding, zero or more bindings of its output columns. These bindings can then be fed back into the input columns for a transitive step. There are switches for specifying distinctness, cycle elimination, maximum depth and so forth.

The enclosing query must provide bindings for either the input columns, output columns or both. Since a derived table has no explicit direction, the transitive step can be evaluated from left to right or right to left or both in the event both ends are given. For example, if we have a social network defined with foaf:knows, we can write a query like

```
select * where { <John> foaf:knows ?person option (transitive) }
```

to get the transitive closure of all whom `<John>` directly or indirectly knows. There are further options for depth limits etc. The transitivity is determined at run time and does not need to be materialized.

```
select * where {
    { select ?person1 ?person2
      t_step (?person1) t_step ('step_no') t_step ('path_id')
      where { ?person1 foaf:knows ?person2 }
    } option (transitive t_in (1) t_out (2)) .
   filter (?person1 = <John> && person2 = <Mary> };
```

This returns all the paths from `<John>` to `<Mary>`. Within each path, the step number is given by the 4th column and the id of the path is given by the 5th column. Since there are multiple answers, each consisting of many rows, these are identified by the `path_id` column.

At evaluation time, the optimizer sees that both ends of the path are given and can thus attack the problem from both ends, following `foaf:knows` from `S` to `O` starting at `<John>` and from `O` to `S` starting at `<Mary>`.

At run time, transitivity is processed in a generally breadth first manner, resulting in good latency tolerance since many operations are shipped in a single message.

Transitivity of a single triple pattern does not have to be declared in the query text if the predicate is declared transitive in an ontology. Queries declare which ontologies they import with a special pragma. Complex cases with special options have to be written in full, as shown above.

A full explanation of the transitivity feature is to be published in the Virtuoso 6 documentation, to be online in late 2008. The other inference features are explained in the Virtuoso 5 documentation, online as of this writing.

11 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 be fitted 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.

12 Some Metrics

The BSBM benchmark[21] compares Virtuoso with other triple stores. The comparison are with a non-clustered version 5 and show that Virtuoso is generally faster than Sesame native or Jena TDB specially at larger scales. Virtuoso 6 has 5% less speed then Virtuoso 5 when all data is in memory but is two times more space efficient, thus can accommodate roughly double the data without going to disk. We note that a random lookup served from memory is about 5000-10000 times faster than one that does a single disk access.

12.1 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 over

all node-node connections evenly there is no real network congestion and scale can be increased without hitting a network bottleneck.

12.2 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 at[7]. The only difference was that a Virtuoso 6 Cluster was used instead.

13 Linked Data Applications

As of this writing, OpenLink hosts several billion triples worth of linked open data. These are being transferred to Virtuoso 6 cluster servers as of the time of this writing. In addition, the data aggregated from the web by Zeitgeist are being moved to Virtuoso 6 cluster. Experiments are also being undertaken with the Sindice semantic web search engine.

14 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.

Aside 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 travelling salesman may have to become

query language primitives because their efficient implementation in a cluster environment is non-trivial. To this effect, we have the transitive subquery feature discussed above which can be used as a building block for parallel graph algorithms.

15 Conclusions

The principal thing to consider in the design of software, specially databases, for clusters is latency. To combat latency, operations need to be sent in large batches and must be as asynchronous as possible. A database, most specially an RDF database, will have large numbers of random lookups whose individual cost is much less than the cost of a message round trip, even within a single computer. To deal with this, we have implemented a SQL and SPARQL execution engine that handles thousands of consecutive iterations of a loop join as a single message. Also, subquery evaluation may proceed concurrently for thousands of different bindings of input variables.

Having done this, we can absorb the cost of interprocess communication by added parallelism even for in-memory workloads where each consecutive join step is served by a different machine. Naturally, workloads which have aggregation or joining that stays in one partition scale linearly. Likewise, as soon as the workload is not all in memory, there is significant gain from independent disk access. Also, there is no limit to the amount of RAM one can have.

We have taken the basic subclass/subproperty/sameAs support of Virtuoso 5 and made it breadth first parallel for eliminating extra round trips. We have also added a generic transitivity feature which can be adapted for various search and graph traversal scenarios. The main contribution of this is that this encapsulates the complexity of concurrently advancing multiple parallel states of the algorithm, thus reducing latency. All this is entirely transparent to the developer.

As of September 2008, we have a full function clustered Virtuoso with both RDBMS and quad store capabilities. This will become commercially available later in the fall of 2008. We plan to publish performance figures using the TPC H benchmark for relational workloads and LUBM, BSBM and SP2B for RDF.

16 Appendix A — Metrics and Environment

We use version 2 of DBpedia[3] as a sample data set for RDF storage space unless otherwise indicated. When CPU processing speeds are discussed, they have been measured with a 2GHz Intel Xeon 5130 8Gb RAM, unless otherwise indicated. OS-specific times such as mutex wait "penalty" are measured using Linux 2.6 SMP (Fedora 7).

Networks are Gbit Ethernet with Linksys switches. 4 similar machines were used for cluster, 2 x 2.3 - 2.6 GHz AMD 285, 16G RAM each.

The LUBM based quotes, complete with test drivers, query mixes, etc. are covered in LUBM and Virtuoso[19].

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, CEUR-WS.org/Vol-301/Paper_5_Erling.pdf
3. Soren 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.
4. Oracle Real Application Clusters. http://www.oracle.com/technology/products/database/clustering/index.html
5. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al. Bigtable: A Distributed Storage System for Structured Data. labs.google.com/papers/bigtable-osdi06.pdf
6. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani et al. Dynamo: Amazon's Highly Available Key-value Store. www.scs.stanford.edu/08sp-cs144/sched/readings/amazon-dynamo-sosp2007.pdf
7. Orri Erling. Virtuoso LUBM Load. http://www.openlinksw.com/weblog/oerling/?id=1284
8. Orri Erling. Advances in Virtuoso RDF Triple Storage (Bitmap Indexing). virtuoso.openlinksw.com/wiki/main/Main/VOSBitmapIndexing
9. Data set for the Billion Triples Challenge www.cs.vu.nl/p̃mika/swc/btc.html
10. DeWitt, D.J., Stonebraker, M. MapReduce: A major step backwards. http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html
11. Monet — A Novel Main Memory Database Kernel (Monet). http://monetdb.cwi.nl/
12. Vertica - Column-store based DBMS. http://www.vertica.com/
13. Harth A., Decker S.: Optimized Index Structures for Querying RDF from the Web. LA-WEB, 2005.
14. Abadi D. J., Marcus A., Madden S., Hollenbach K. J. Scalable Semantic Web Data Management Using Vertical Partitioning. VLDB, 2007.
15. Wilkinson K., Sayers C., Kuno H., Reynolds D. Efficient RDF Storage and Retrieval in Jena2. SWDB, 2003.
16. Broekstra J., Kampman A., van Harmelen F. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. ISWC, 2002.
17. Harris S., Gibbins N. 3store: Efficient Bulk RDF Storage. PSSS, 2003.
18. Paraccel DBMS. http://www.paraccel.com/
19. Erling O. LUBM and Virtuoso. http://virtuoso.openlinksw.com/wiki/main/Main/VOSArticleLUBMBenchmark
20. Erling O. DBpedia Benchmark Revisited. http://www.openlinksw.com/weblog/oerling/?id=1358
21. Bizer C., Schultz A. Berlin SPARQL Benchmark (BSBM) Specification - V2.0 http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/20080912/