Andreas Harth, Katja Hose, Ralf Schenkel

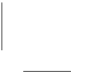
Linked Data Management: Principles and Techniques

List of Figures

1.1 An example of a distributed query plan with DFG 9

List of Tables

1.1 BSBM: BI Q1 micro-benchmark: vector size and DFG 11
1.2 BSBM: BI Q8 micro-benchmark 12
1.3 BSBM: BI Q3 micro-benchmark 13
1.4 Single vs Cluster with example query on 100G TPC-H dataset 14
1.5 BSBM data size and loading statistics 16
1.6 Business Intelligence Use Case: Detailed Results 18
1.7 Business Intelligence Use Case: Updated Results in March 2013 20
1.8 Explore Use Case: Detailed Results 21
1.9 Explore Results: Query Mixes Per Hour 22

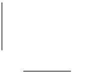


Contents

I This is a Part **1**

1 Experiences with Virtuoso Cluster RDF Column Store **3**
Peter Boncz, Orri Erling, and Minh-Duc Pham
1.1 Virtuoso Cluster 3
1.1.1 Architecture of a server process 5
1.1.2 Differences with Virtuoso 6 6
1.2 Query Evaluation 8
1.2.1 Vectoring 8
1.2.2 Cluster Tuning Parameters 10
1.2.3 Micro Benchmarks 10
1.3 Experimental Evaluation 14
1.3.1 BSBM results to date 14
1.3.2 Cluster Configuration 15
1.3.3 Bulk Loading RDF 16
1.3.4 Notes on the BI Workload 17
1.3.5 BSBM Benchmark Results 18
1.4 Conclusion 22

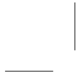
Bibliography **25**

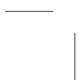


Symbol Description

Part I

This is a Part





Chapter 1

Experiences with Virtuoso Cluster RDF Column Store

Peter Boncz

CWI, The Netherlands

Orri Erling

OpenLink Software, U.K.

Minh-Duc Pham

CWI, The Netherlands

1.1 Virtuoso Cluster 3
  1.1.1 Architecture of a server process 4
  1.1.2 Differences with Virtuoso 6 6
1.2 Query Evaluation 8
  1.2.1 Vectoring 8
  1.2.2 Cluster Tuning Parameters 10
  1.2.3 Micro Benchmarks 10
1.3 Experimental Evaluation 14
  1.3.1 BSBM results to date 14
  1.3.2 Cluster Configuration 15
  1.3.3 Bulk Loading RDF 15
  1.3.4 Notes on the BI Workload 17
  1.3.5 BSBM Benchmark Results 18
1.4 Conclusion 21

1.1 Virtuoso Cluster

Virtuoso Column Store [4] introduces vectorized execution into the Virtuoso DBMS. Additionally, its scale-out version, that allows running the system on a cluster, has been significantly redesigned. This article discusses advances in scale-out support in Virtuoso and analyzes this on the Berlin SPARQL Benchmark (BSBM) [1]. To demonstrate the features of Virtuoso Cluster RDF Column Store, we first present micro-benchmarks on a small 2-node cluster with 10 billion triples. In the full evaluation we show one can now scale-out to a BSBM database of 150 billion triples. The latter experiment is a *750 times* increase over the previous largest BSBM report, and for the first time includes both its Explore and Business Intelligence workloads.

The storage scheme used by Virtuoso for storing RDF Subject-Property-Object triples pertaining to a Graph (hence we have quads, not triples) consists of five indexes: PSOG, POSG, SP, OP, GS. To be precise, PSOG is a B-tree with key (P,S,O,G), where P is an number identifying a property, S a subject, O an object and G the graph. Additionally, there is a B-tree holding URIs and a B-tree holding string literals, both of them used to encode string(-URI)s into numerical identifiers. Users may alter the indexing scheme of Virtuoso but this almost never happens. The three last indexes (SP, OP, GS) are projections of the first two *covering* indexes, containing only the unique combinations – hence these are much smaller. We note that Virtuoso Column Store Edition (V7) departs from the previous Virtuoso editions (V6) in that it uses a columnar data representation instead of a row-wise representation. In Virtuoso Column Store, the leaves of the B-trees refer to data pages which store the individual columns (e.g. P,S,O and G) separately, in highly compressed format (sequences of *compression entries*, with various compression methods supported which get automatically selected). One advantage of this is the possibility to compress data column-wise which leads to a much more dense data layout and more CPU cache locality and more RAM locality and therefore increased performance [4].

Virtuoso Cluster is based around a hash partitioning scheme where each index in the database is partitioned into *slices* by one or more partitioning columns. The partitioning column values determine which slice of the database the row belongs to. A server process manages several slices, usually one or two per core [2]. For each index, the partitioning column is either S or O, depending on which is first in key order. This produces a reasonably even distribution of data, except in cases of POSG where O is a very common RDF class (rdfs:type). In these cases, the distribution may be highly skewed but this is compensated for by the fact that such parts of the index compress very well, e.g. approximately 3 bits per index entry when the P and O are repeating and the S is densely ascending, e.g. average distance between consecutive values under three. The partitioning hash is calculated from bits 16-31 of the partitioning column. This way, each partition gets 64K consecutive key values and the next 64K values go to the next one and so forth. This works better than bits 8-23, since larger compression entries are made when there are longer runs of closely spaced values. Processing longer compression entries is faster.

Virtuoso stores data only in its B-tree indexes. All secondary indexes reference the table by incorporating the primary key columns. In a cluster setting this has the advantage of not forcing index entries of one row to reside in the same partition, as it would be difficult and time consuming to maintain integrity of physical references to data across process boundaries. Another advantage of having no physical references is that each index may be independently compacted to compensate for fragmentation, again without dealing with locking of anything more than a range of logically consecutive pages, i.e. no deadlocks can occur as a result.

1.1.1 Architecture of a server process

A Virtuoso cluster consists of multiple server processes all of which provide the same service and the same view on the same data. There is no distinction between coordinator and worker processes, each process will play either role statement by statement. Furthermore, queries with nesting or with arbitrary procedure calls themselves involving queries benefit from distributed coordination. Scheduling can further deal with arbitrary cases of recursive functions containing database operations [3].

Each process has separate listening ports for SQL CLI (ODBC, JDBC), HTTP and cluster internal traffic. A separate thread listens for all client traffic (SQL CLI and HTTP) and a different one listens for cluster traffic. Incoming cluster traffic is always read as soon as it becomes available. In order to guarantee that this always happens one should set thread CPU affinity so that the cluster listener thread had its own core and will not be displaced by other threads of the process. Typically there will be one server process per NUMA node, with one core for the listener and the other cores shared among the worker threads. Experience demonstrates that setting CPU affinity such improves performance by up to 20%.

A process has a pool of worker threads for cluster operations and a separate pool of worker threads for query parallelization. In most cases, a cluster operation occupies one thread from the cluster thread pool per involved process and additionally dispatches multiple tasks to be run by a separate thread pool (AQ or async queue threads). In this manner, the number of simultaneously runnable threads on a server process is maximally the number of AQ threads plus the number of concurrent client operations across the server processes.

Threads carry a transaction context so as to implement consistent visibility in read-committed isolation. Each transaction has a single such context object, in a server process. It allows the multiple threads that may be active for that transaction to acquire locks and wait on locks held by other transactions. One of the threads is the main branch for each transaction, and acts as the party in a possible two-phase commit between server processes.

When the cluster listener thread of a process sees a connection ready for reading it reads the header of the message. The header indicates whether this is a request or a response and whether this should be read on a separate thread. In the case of very long messages is may be appropriate to schedule the complete read on a different thread from the one listening for incoming requests. In this way a potentially slow transmission will not block processing of smaller, faster transmissions and overall responsiveness is improved. Further, the server process never blocks mid-message. If the read system call would block mid-message, the listener goes to a select system call on all connections and resumes reading on the first one to be ready, regardless of which it was reading before.

Outgoing traffic, whether request or response, is sent by the thread producing the data. Most outgoing traffic sends a partitioned operation to multiple

partitions. The messages may be long and are liable to block on write. In the case of a connection blocking on write, the sender will write to another connection and only block if all outgoing connections would block on write.

For write intensive workloads there will be times when dirty pages need to be written to disk. There is also a periodic recompression of disk pages when consecutive inserts and/or deletes have caused half full pages or loss of compression. In the event of a large write to disk, the OS may suspend the whole process until enough buffers are written out. This is not always predictable. A single process being suspended will potentially stop the whole cluster. Therefore it is important to have comparable IO performance in all nodes of the cluster and to synchronize flushing to disk across the cluster, otherwise it may be that one process is running at fractional CPU due to the OS blocking it while waiting for disk writes to finish, thereby slowing down all cluster nodes.

1.1.2 Differences with Virtuoso 6

Vectored execution is a natural fit for a cluster DBMS. The previous scale-out implementation did not have vectored execution but did buffer large numbers of tuples in a tuple by tuple execution model before sending them to a remote party. Thus from the message passing viewpoint the execution was vectored or batched but not from the query operator viewpoint. Eliminating this impedance mismatch leads in fact to some simplification of code. However the introduction of elasticity, i.e. decoupling the data partitions from the server process managing them and other improvements resulted in net increase in complexity. Also, threading was revisited, as there are now multiple threads per query per process whereas there used to be at most one.

A significant added complexity comes from arbitrary nesting and recursion between shipped operators. The Virtuoso 6 cluster capability relied on all operators shipped to remote locations to be able to complete within that location, without recourse to any resources in a third location. The amount of code is not very large but the functionality pervades the engine and gives rise to very complex special cases in thread allocation. For example, a server must keep track of which threads have been taken over by which descendant functions and cancellations and requests for a next batch of results must be directed without fail to the right thread. The data structures keeping track of this consist of several queues and hash tables, all serialized on a mutex. The mutex in question then becomes an obstacle to scaling, thus much attention has gone into optimizing the data structures used for scheduling. Even now, the mutex may be taken up to 10% of real time in a bad case, with about 20 executable threads all together.

In specific, if a computation on node 1 sends a request to node 2 which again needs a request on node 1 to complete, the thread originating the request on node 1 can be reused so as to accommodate arbitrary recursion without requiring infinite threads. This involves a global critical section that serializes

dispatching messages. The reader thread, even if it has a core of its own may not be able to read if it clocks on this mutex. Therefore an extra optimization consists of keeping on reading if the critical section is not immediately available. This trick drops the time for a 4-client BSBM BI at 10Gt from 1084 to 960 s. As a result of this, the aggregate time spent blocked on write drops from 1084 to 224 s. This clearly demonstrates the need to avoid threads going off CPU for any reason.

The “anytime feature” of Virtuoso [5] is a setting that places a soft timing cap on query execution, so that queries stop what they are doing after a given time and return whatever they have found so far. If an aggregation is interrupted, what has been aggregated so far on different threads or processes is added up and passed to the next aggregation or returned. This is a demo-only feature insofar the data that a query reaches in its allotted time depends on the plan and the physical order of the data, which in the case of RDF depends on its load order. Hence there are no semantic guarantees. The feature's use is limited to online sites that can offer arbitrary queries to the public with the provision that results might be incomplete. The feature has no other application and its cost of implementation is high. It is indirectly useful for testing error return paths, which are exceedingly many and complex in a cluster setting. The main difficulty lies in the fact that scheduling information on aborted operations must be correctly retracted, including any as yet unprocessed input for such.

Due to having multiple threads per transaction Virtuoso Column Store changes some aspects of cluster transaction management. The basic model remains 2PC with distributed deadlock detection but now logical transactions may have multiple physically distinct branches (threads) per process. There is always a main branch which owns the locks and rollback information but multiple threads are allowed to contribute to that state.

The scale-out logic is largely independent of storage organization, whether it be row or column-wise. It is tightly bound to the vectored execution model but this in turn applies equally to row- or column-wise storage. However column-wise compression does significantly interact with data partitioning. In order to preserve compression opportunities, the low bits of column values do not influence the partitioning. With the row-store, leaving out the low 8 bits worked well in V6, giving generally even balance and compression. With the column-store, the compression is still good if data comes in stretches of 256 values followed by a gap and then another window of 256 values but this tends to break compression entries into smaller chunks which is less efficient for reading. Thus leaving out the 16 low bits from the partitioning hash works better, giving longer stretches of near-consecutive values in key columns.

A scale-out system can approach full platform utilization only when coordination of operations is loose and maximally asynchronous. This is so regardless of the type interconnect being used (e.g. ethernet or infiniband). Of course, in trivial cases where operations are lookups or scans only, central coordination works well enough. For lookups, if the coordination is distributed between

enough front-end nodes dispatching small requests to back-end nodes, one can have linear scalability as long as data distribution is not too skewed. The same applies for scans. For complex joins, it becomes unworkable to direct all the intermediate states via any coordinating node, hence the worker nodes need to transfer data between themselves. We have not found any necessity for distinguishing between front-end and back-end nodes, all processes have the same functions. The only special cases that need to be allocated to privileged nodes are matters of distributed deadlock detection and arbitration in case of node failures in systems with redundancy.

1.2 Query Evaluation

The basic unit of query evaluation is the Distributed Fragment (DFG). This is a sequence of joins where consecutive steps may or may not be in the same partition. A stage of the DFG is a sequence of co-located joins and other operations. Between the stages there is a message exchange between all the slices that have data pertaining to the operation. There is a query state per slice, thus the natural parallelism is as many threads as there are distinct slices. This produces good platform utilization since there are usually as many slices as core threads on a server. A DFG can return values to its containing query or can end with an aggregation with or without group by or an order by. DFG's can be freely nested for complex subqueries and can be run with large numbers of input variable bindings in a single operation. The exact message passing having to do with DFG's and nesting is described in [3].

For the purpose of demonstration, we will here consider different queries against the 10G triple BSBM dataset and how they execute in a cluster setting as opposed to a single server. The test system consists of 2 machines with dual Xeon E5-2630 and 192G RAM and a QDR InfiniBand network. The cluster configuration has 2 processes per machine, one per NUMA node and each process has 12 slices of data for the 12 core threads on a NUMA node. Later in this chapter we will move our attention to larger cluster configurations.

1.2.1 Vectoring

All operators in a query plan attempt to produce a full vector worth of output before passing it to the next operator. Passing many rows of output variable bindings has the advantage of amortizing interpretation overhead and allowing cache-optimized algorithms that perform the same operation on large sets of values. If the operation is an index lookup, the inputs can be sorted and one can effectively transform a $n \log n$ set of random index lookups into a near-linear merge join. The result is in fact linear if there is sufficient locality between the input key values. This is described with some metrics in [4].

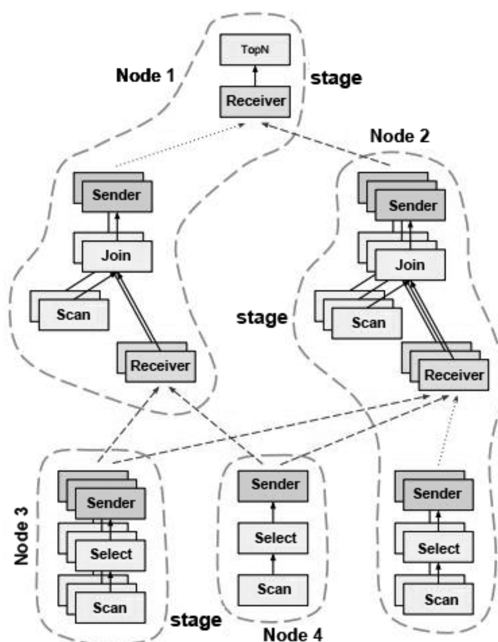


FIGURE 1.1: An example of a distributed query plan with DFG

Always running with large vector size has a high memory overhead, thus the default vector size is small, e.g. 10K values. The system automatically switches to a larger vector size, e.g. 500K if it finds that there is no locality in index lookups. Having larger batches of input increases the likelihood of having closely spaced hits even if the data comes in random order.

One vectored operation that occurs frequently in a cluster is partitioning. Given a set of vectors representing columns, the values in one provide a partitioning key and the result is a set of messages serializing the vectors split so that the rows going to one destination are all in one message. The message layout is column-wise. It is often the case that the vectors going into a message are not aligned, i.e. one vector may be produced before a join and another after a join. If the join does not produce exactly one row of output for one of input the vectors are not aligned. There is an additional mechanism of indirection allowing reassembling tuples after arbitrarily many cardinality and order-changing steps. The operation for partitioning a vector must thus do a gather-scatter pattern over a vector where values are fetched for logically consecutive places (many not physically consecutive) and are then appended into several target buffers. This is the single cluster-specific operation that stands out in a CPU performance profile, with up to 5% of CPU for cases where each join step needs a different partitioning. The operation is implemented with cache-friendly techniques with unrolled loops and many independent memory

accesses outstanding at the same time so as to optimize memory bandwidth. The data for large messages, e.g. 500K rows of 4 columns of 8-byte values will typically not come all from CPU cache, thus optimizing memory bandwidth does matter.

1.2.2 Cluster Tuning Parameters

Vector size. Due to high latency of any inter-process communication, it is desirable to pass as many values as possible in a single message. However if the network gets saturated and the writer blocks waiting for the reader, needless task switches are introduced and throughput actually suffers. The input vector size of a partitioned operation divided by the number of output partitions determines the average message size. Making the input vector arbitrarily large results in high memory overhead and in missing the CPU cache when assembling vectors, thus the size cannot be arbitrarily increased. The server process has a global cap on vector space, hence a query will not switch to larger vectors if space is short, even if this would be more efficient.

Separate thread. A parameter determines the minimum estimated cost of an operation for putting it on a separate thread. Thread switching is expensive, e.g. up to several microseconds for synchronizing with a thread if even minimal waiting is involved. Thus operations like single row lookups do not usually justify their own thread. A too high threshold misses out on parallelism opportunities, a too low one hits thread overhead. This is most significant for short queries, thus a low value is safe with workloads consisting of long queries.

Flow control. A DFG consists of multiple stages, the output of the previous is fed into the next. The processing speeds of these stages may vary. Generally, later stages are run in preference to earlier ones. However with data skew some producers may get ahead of consumers. Since the consumer always reads what is sent even if it cannot process this, there can be large transient memory consumption. If a producer stops after a small quota, it will have to be restarted by the query coordinator, after the latter determines that all in-flight results have been processed. This results in idle time. Thus the quota of output to produce should be large, e.g. 100MB per slice, e.g. 4.8G max for 48 slices in the test setup.

1.2.3 Micro Benchmarks

BSBM Q1 from the BI workload offers a relatively easy case of parallel execution. The query counts reviews per product type such that the producer is from a given country and the reviewer is from a different one.

```
SELECT ?productType (COUNT(?review) AS ?reviewCount) {
   ?product bsbm:producer ?producer .
   ?producer bsbm:country <http://downlode.org/rdf/iso-3166/countries#AT> .
   ?review bsbm:reviewFor ?product .
   ?review rev:reviewer ?reviewer .
```

	seconds	cpu%	msg/sec	bytes/msg
Single:	7.0	1578	n. a.	n. a.
Single 10K vec	32.5	1745	n. a.	n. a.
2-Cluster	4.96	3458	942K	1.6K
2-Cluster no DFG	10.94	585	16K	132K
2-Cluster 500K vec	4.3	2851	599	2.6K
2-Cluster 10K vec	20.3	2411	7.7M	0.3K

TABLE 1.1: BSBM: BI Q1 micro-benchmark: vector size and DFG

```
   ?reviewer bsbm:country <http://downlode.org/rdf/iso-3166/countries#US> .
   ?product a ?productType .
   ?productType a bsbm:ProductType . }
GROUP BY ?productType
ORDER BY DESC 2;
```

The patterns are arranged to reflect the actual join order. The single server parallelization works by splitting the initial scan of producer into n equal slices, one per core. The query then runs independently on each thread. If a thread finishes before others, then the remaining threads can parallelize more, i.e. having a vector of sorted lookups, these can then be split over many threads. All operations downstream proceed independently, including joins. Aggregations are added up by each thread that decides to spin more threads.

The cluster parallelization works by always sending the key to the slice where the next join is to be found and by having at most one thread per slice. Thus both schemes will set as many threads as there are cores into action. There are few distinct product types, thus the aggregation is done in both cases by all threads, with reaggregation into a single result at the end. In each case, the system does 104M key based lookups from the RDF quad table with locality varying depending on vector size. In the best case 96% of the rows retrieved fall in the same column compression segment as the previous one. A segment is about 16K rows in the present case.

This micro-benchmark shows in Table 1.1 reasonable behavior in the cluster when the query is run with default parameters: 7 seconds single versus 4.96 seconds on the 2 node cluster. The platform utilization (cpu%) with 2x12 hyper-threaded cores is maximally 4800%, but given that hyper-threaded cores are much less effective than real cores, any platform utilization exceeding 2400% is quite good. To show the effect of message flow, we also executed the same query on cluster without DFG ("2-Cluster no DFG"), i.e. so that the coordinator (the process to which the client is connected) performs a scatter-gather for each join step, so that all intermediate results flow via a single point. The time is more than doubled but there are fewer and longer messages. We see that the individual message size with DFG is short, as every one of the 48 slices usually makes a message to every other. However, all messages to the same process from one slice are combined, so the network in fact sees 12x longer transmissions (12 slices per process). The cluster result marked "500K vec" shows the data for a DFG execution with a fixed vector size of 500K

	seconds	cpu%	msg/sec	bytes/msg
Single	141.8	1736	n.a.	n.a.
2-cluster	60.1	2305	450K	51K
2-Cluster NP	75.2	2261	225K	99K

TABLE 1.2: BSBM: BI Q8 micro-benchmark

elements; here the performance improves slightly towards 4.3 seconds. The default behavior is to start with shorter vectors and automatically switch to longer vectors after noticing lack of locality. We also tested with small 10K vectors; which deteriorates performance in both single and cluster. The cluster is hit roughly as bad as single but finishes first due to having double the CPU, as short vectors increase CPU cost of index lookup.

We next look at the more complex query Q8, which determines for a product type the vendors with the less expensive products in each product type. The query does 1198M key-based lookups in the RDF quad table. The plan starts by taking all the products of the selected type and executes for each a subquery getting the average price across all offers of said product type. This produces 1.1 million prices. Table 1.2 shows the result of this experiment. As a future improvement, one should consider evaluation with GROUP BY partitioned on the grouping keys, such that re-aggregation is not required.

The cluster result with no group by partitioning is marked with NP. We notice that the number of messages is less and the size of the messages is larger but the overall run time is longer.

A query that achieves low platform utilization on both single and cluster is Q3 of BSBM BI, which finds the products with the greatest percentual growth in reviews between two given months.

```
SELECT ?product (xsd:float(?monthCount)/?monthBeforeCount AS ?ratio) {
{ SELECT ?product (COUNT(?review) AS ?monthCount) {
    ?review bsbm:reviewFor ?product .
    ?review dc:date ?date .
    FILTER(?date >= "2008-03-10"^^<http://www.w3.org/2001/XMLSchema#date>
        && ?date <  "2008-04-07"^^<http://www.w3.org/2001/XMLSchema#date>)
  } GROUP BY ?product }
{ SELECT ?product (COUNT(?review) As ?monthBeforeCount) {
     ?review bsbm:reviewFor ?product .
     ?review dc:date ?date .
     FILTER(?date >= "2008-02-11"^^<http://www.w3.org/2001/XMLSchema#date>
         && ?date <  "2008-03-10"^^<http://www.w3.org/2001/XMLSchema#date>)
 } GROUP BY ?product HAVING (COUNT(?review)>0) }}
ORDER BY DESC(xsd:float(?monthCount) / ?monthBeforeCount) ?product
LIMIT 10
```

The first subquery selects 17M products. The second takes these in batches of 1M and counts the reviews for each that fall in the previous month. The CPU% for both single and cluster is only about 500%.

Both aggregations are partitioned on the grouping column, thus the cluster does not need reaggregation. The platform utilization is low though since all the intermediate groups have to be brought to the coordinator process and again dispatched from there into the second aggregation. 1M products

	seconds	cpu%	msg/sec	bytes/msg
Single	134	586	n.a.	n.a.
2-Cluster	103.4	414	43.8K	156K

TABLE 1.3: BSBM: BI Q3 micro-benchmark

enter the second aggregation at a time, retrieving the dates of all reviews and counting those that fall in the chosen month. After all partitions are ready the counts are retrieved. The fact of having a synchronous barrier every 1M products lowers the platform utilization. A better plan would be to colocate the second aggregation with the first: Since each partition produces results of the first aggregation and these do not need to be merged with any other since already complete, each partition could coordinate the second aggregation, thus dividing the coordination. Each partition would still have synchronous barriers but there would be many independent threads each with its own instead of a single one.

In the single server case we could obtain better throughput by parallelizing the reaggregation. The single server parallelism model does not have partitioning by key value but such a construct could be added. Not partitioning by key value makes it possible to run each thread of the query without any synchronization with others but does entail the cost of reaggregation. Having an exchange operator would involve messaging between threads with queues and critical sections.

To explore the difference of centralized and distributed query coordination we take an example with the 100G TPC-H data set. We count all parts that have not been shipped to a customer in nation 5. Since the group by's have large numbers of groups, reaggregation is clearly a major cost, as the absolute CPU utilization of the single server exceeds the cluster's.

```
 select count (*) from part where not
exists
    (select 1 from lineitem, orders, customer
        where l_partkey = p_partkey
        and l_orderkey =  o_orderkey
        and c_custkey = o_custkey
        and c_nationkey = 5
        option (loop)
    );
```

The option at the end of the subquery specifies that nested loop join is to be used. The correct plan would make a hash table of orders by customers from the nation or at least a hash table of customers from the nation, depending on how much memory was available. However we disregard this in order to focus on cross partition joins.

For each slice of part, partitioned on p_partkey, there is a lookup using an index on l_partkey in lineitem. This index is local, being partitioned on l_partkey. The secondary index contains l_orderkey, as this is a primary key part. The next stage is a non-colocated lookup on orders to fetch o_custkey. The next step is also cross partition, since customer is partitioned on c_custkey.

	seconds	cpu%	msg/sec	bytes/msg
Single	46	1547	n.a.	n.a.
Single H	29	1231	n.a.	n.a.
2-cluster	32.7	3013	1559K	16.8K
2-cluster C	57.7	1370	548K	48K

TABLE 1.4: Single vs Cluster with example query on 100G TPC-H dataset

The cluster result with no distributed coordination is marked with C. The single result with a hash join with orders joined to customers on the build side is marked with H. As expected we see that a hash join plan is best. For parallel index based plans we see that cluster cannot beat single without distributed coordination.

With distributed coordination, each slice coordinates its share of the subquery, so that there is a nested DFG for each slice. In the central variant the parts are brought to the coordinator which then runs the subquery with a single DFG.

Cluster plans with hash join will produce the best results and will be explored in the future. We note that there are many possible variants as concerns partitioning or replicating the hash table.

1.3 Experimental Evaluation

In this section, we present full BSBM results [1] on the V3.1 specification, including both the Explore (transactional) and Business Intelligence (analytical) workloads.

1.3.1 BSBM results to date

The BSBM (Berlin SPARQL Benchmark) was developed in 2008 as one of the first open source and publicly available benchmarks for comparing the performance of storage systems that expose SPARQL endpoints. Such systems include native RDF stores, Named Graph stores, systems that map relational databases into RDF, and SPARQL wrappers around other kinds of data sources. The benchmark is built around an e-commerce use case, where a set of products is offered by different vendors and consumers have posted reviews about products. BSBM has been improved over this time and is current on release 3.1 which includes both Explore and Business Intelligence use case query mixes, the latter stress-testing the SPARQL1.1 group-by and aggregation functionality, demonstrating the use of SPARQL in complex analytical queries.

The following BSBM results have been published the last being in 2011,

all of which include results for the Virtuoso version available at that time (all but the last one being for Virtuoso row store) and can be used for comparison with the results produced in this evaluation:

- BSBM version 1 (July 2008) with 100 million triple datasets
- BSBM version 2 (Nov 2009) with 200 million triple datasets
- BSBM version 3 (Feb 2011) with 200 million triple datasets

The above results are all for the Explore use case query mix only. Apart from these official BSBM results, in published literature some BSBM results have appeared, though none of these complete BI runs or Explore runs on any larger size. The results of this evaluation benchmarking the BSBM Explore and BI use case query mixes against 50 and 150 billion triple datasets on a clustered server architecture represent a major step (750 times more data) in the evolution of this benchmark.

1.3.2 Cluster Configuration

RDF systems strongly benefit from having the working set of the data in RAM. As such, the ideal cluster architecture for RDF systems uses cluster nodes with relatively large memories. For this reason, we selected the CWI scilens[1] cluster for these experiments. This cluster is designed for high I/O bandwidth, and consists of multiple layers of machines. In order to get large amounts of RAM, we used only its "bricks" layer, which contains its most powerful machines. Virtuoso V7 Column Store Cluster Edition was set up on 8 Linux machines. Each machine has two CPUs (8 cores and hyperthreading, running at 2GHz) of the Sandy Bridge architecture, coupled with 256GB RAM and three magnetic hard drives (SATA) in RAID 0 (180 MB/s sequential throughput). The machines were connected by Mellanox MCX353A-QCBT ConnectX3 VPI HCA card (QDR IB 40Gb/s and 10GigE) through an InfiniScale IV QDR InfiniBand Switch (Mellanox MIS5025Q). The cluster setups have 2 processes per machine, 1 for each CPU. A CPU here has its own memory controller which makes it a NUMA node. CPU affinity is set so that each server process has one core dedicated to the cluster traffic reading thread (i.e. dedicated to network communication) and the other cores of the NUMA node are shared by the remaining threads. The reason for this set-up is that communication tasks should be handled with high-priority, because failure to handle messages delays all threads. These experiments have been conducted over many months, in parallel to the Virtuoso V7 Column Store Cluster Edition software getting ready for release. Large part of the effort spent was in resolving problems and tuning the software.

[1]This cluster is equipped with more-than-average I/O resources, achieving an Amdahl number > 1. See `www.scilens.org`.

nr triples	Size (.ttl)	Size (.gz)	Database Size	Load Time
50 Billion	2.8 TB	240 GB	1.8 TB	6h 28m
150 Billion	8.5 TB	728 GB	5.6 TB	n/a

TABLE 1.5: BSBM data size and loading statistics

1.3.3 Bulk Loading RDF

The original BSBM data generator was a single-threaded program. Generating 150B triples with it would have taken weeks. As part of this project, we modified the data generator to be able to generate only a subset of the dataset. By executing the BSBM data generator in parallel on different machines, each generating a different part of the dataset, BSBM data generation now has become scalable. In these experiments we generated 1000 data files with the BSBM data generator. Separate file generation is done using the `nof` option in the BSBM driver. These files are then distributed to each machine according to the modulo of 8 (i.e., the number of machine) so that files number 1, 9, 17, ... go to machine 1, file number 2, 10, 18,... go to machine 2, and so on. This striping of the data across the nodes ensures a uniform load, such that all nodes get an equal amount of similar data.

Each machine loaded its local set of files (125 files), using the standard parallel bulk-load mechanism of Virtuoso. This means that multiple files are read at the same time by the multiple cores of each CPU. The best performance was obtained with 7 loading threads per server process. Hence, with two server processes per machine and 8 machines, 112 files were being read at the same time. Also notice that in a cluster architecture there is constant need for communication during loading, since every new URIs and literals must be encoded identically across the cluster; hence shared dictionaries must be accessed. Thus, a single loader thread counts for about 250% CPU across the cluster. The load was non-transactional and with no logging, to maximize performance. Aggregate load rates of up to 2.5M quads per second were observed for periods of up to 30 minutes. The total loading time for the dataset of 50 billion triples is about 6h 28 minutes, which makes the average loading speed 2.14M triples per second.

The largest load (150B quads) was slowed down by one machine showing markedly lower disk write throughput than the others. On the slowest machine iostat showed a continuous disk activity of about 700 device transactions per second, writing anything from 1 to 3 MB of data per second. On the other machines, disks were mostly idle with occasional flushing of database buffers to disk producing up to 2000 device transactions per second and 100MB/s write throughput. Since data is evenly divided and 2 of 16 processes were not runnable because the OS had too much buffered disk writes, this could stop the whole cluster for up to several minutes at a stretch. Our theory is that these problems were being caused by hardware malfunction.

To complete the 150B load, we interrupted the stalling server processes, moved the data directories to different drives, and resumed the loading again.

The need for manual intervention, and the prior period of very slow progress makes it hard to calculate the total time it took for the 150B load.

For the 10B triples dataset used in the query microbenchmarks the load rate was 687Kt/s on two machines and 215Kt/s on one.

1.3.4 Notes on the BI Workload

For running the benchmark, we used the Business Intelligence Benchmark (BIBM) [2], an updated version of the original BSBM benchmark (BSBM) which provides several modifications in the test driver and the data generator. These changes have been adopted in the official V3.1 BSBM benchmark definition. The changes are as follows:

- The test driver reports more and more detailed metrics including "power" and "throughput" scores.
- The test driver has a drill down mode that starts at a broad product category, and then zooms in subsequent queries into smaller categories. Previously, the product category query parameter was picked randomly for each query; if this was a broad category, the query would be very slow; if it is a very specific category it would run very fast. This made it hard to compare individual query runs; and also introduced large variation in the overall result metric. The drill down mode makes it more stable and also tests a query pattern (drill down) that is common in practice.
- One query (i.e., BI Q6) was removed that returned a result that would increase quadratically with database scale. This query would become very expensive in the 1G and larger tests, so its performance would dominate the result.
- The text data in the generated strings is more realistic. This means you can do (more) sensible keyword queries on it.
- The new generator was adapted to enable parallel data generation. Specifically, one can let it generate a subset of the data files. By starting multiple data generators on multiple machines one can thus hand-parallelize data generation. This is convenient for generating large datasets such as 150 billion triples, which literally otherwise takes weeks.

As the original BSBM benchmark, the test driver can run with single-user run or multi-user run.

- *Single user* run: This simulates the case that one user executes the query mix against the system under test.

[2] See `www.sourceforge.net/projects/bibm`

	50 Billion triples		150Billion triples	
	Single-Client	4-Clients	Single-Client	4-Clients
runtime	3733s	9066s	12649s	29991s
Tput	12.052K	19.851K	10.671K	18.003K
	AQET	AQET	AQET	AQET
Q1	622.80s	1085.82	914.39s	1591.37s
Q2	189.85s	30.18	196.01s	507.02s
Q3	337.64s	2574.65	942.97s	8447.73s
Q4	18.13s	6.3s	183.00s	125.71s
Q5	187.60s	319.75s	830.26s	1342.08s
Q6	47.64s	34.67s	24.45s	191.42s
Q7	36.96s	39.37s	58.63s	94.82s
Q8	256.93s	583.20s	1030.73s	1920.03s

TABLE 1.6: Business Intelligence Use Case: Detailed Results

- *Multi-user* run: This simulates the case that multiple users concurrently execute query mixes against the system under test.

All BSBM BI runs were with minimal disk IO. No specific warm-up was used and the single user run was run immediately following a cold start of the multi-user run. The working set of BSBM BI is approximately 3 bytes per quad in the database. The space consumption without literals and URI strings is 8 bytes with Virtuoso column store default settings. For a single user run, typical CPU utilization was around 190 of 256 core threads busy. For a multi-user run, all core threads were typically busy. Hence we see that the 4 user run takes roughly 3 times the real time of the single user run.

1.3.5 BSBM Benchmark Results

The following terms will be used in the tables representing the results.

- *Elapsed runtime* (seconds): the total runtime of all the queries excluding the time for warm-up runs.

- *Throughput*: the number of executed queries per hour. This value is computed with considering the scale factor as in TPC-H. Specifically, the throughput is calculated using the following function: *Throughput = (Total # of executed queries) * (3600 / ElapsedTime) * scaleFactor*. Here, the scale factor for the 50 billion triples dataset and 150 billion triples dataset is 500 and 1500, respectively.

- *AQET*: Average Query Execution Time (seconds): The average execution time of each query computed by the total runtime of that query and the number of executions: *AQET(q) = (Total runtime of q) / (number of executions of q)*.

Some results seem noisy, for instance Q2@50B, Q4@50B, Q4@150B are significantly cheaper in the multi-client-setup. Given the fact that the benchmark was run in drill-down mode, this is unexpected. It could be countered by performing more runs, but, this would lead to very large run-times as the BI workload has many long-running queries.

In the following, we discuss the above performance results over several specific queries Q2 and Q3.

Query 2 in the BI use case:

```
SELECT ?otherProduct ?sameFeatures {
 ?otherProduct a bsbm:Product .
 FILTER(?otherProduct != %Product%)
 { SELECT ?otherProduct (COUNT(?otherFeature) AS ?sameFeatures) {
    %Product% bsbm:productFeature ?feature .
    ?otherProduct bsbm:productFeature ?otherFeature .
    FILTER(?feature=?otherFeature)
 } GROUP BY ?otherProduct }}
ORDER BY DESC(?sameFeatures) ?otherProduct
LIMIT 10
```

BSBM BI Q2 is a lookup for the products with the most features in common with a given product. The parameter choices (i.e., %Product%) produce a large variation in run times. Hence the percentage of the query's timeshare varies according to the repetitions of this query's execution. For the case of 4-clients, this query is executed for 4 times which can be the reason for the difference timeshare between single-client and 4-client of this query.

Query 3 in the BI use case:

```
SELECT ?product (xsd:float(?monthCount)/?monthBeforeCount AS ?ratio) {
{ SELECT ?product (COUNT(?review) AS ?monthCount) {
  ?review bsbm:reviewFor ?product .
  ?review dc:date ?date .
  FILTER(?date >= "%ConsecutiveMonth_1%"^^<http://www.w3.org/2001/XMLSchema#date>
      && ?date < "%ConsecutiveMonth_2%"^^<http://www.w3.org/2001/XMLSchema#date>) }
  GROUP BY ?product }
{ SELECT ?product (COUNT(?review) AS ?monthBeforeCount) {
  ?review bsbm:reviewFor ?product .
  ?review dc:date ?date .
  FILTER(?date >= "%ConsecutiveMonth_0%"^^<http://www.w3.org/2001/XMLSchema#date>
      && ?date < "%ConsecutiveMonth_1%"^^<http://www.w3.org/2001/XMLSchema#date>) }
  GROUP BY ?product
  HAVING (COUNT(?review)>0) }}
ORDER BY DESC(xsd:float(?monthCount) / ?monthBeforeCount) ?product
LIMIT 10
```

The query generates a large intermediate result: all the products and their review count on the latter of the two months. This takes about 16GB (in case of 150 billion triples), which causes this to be handled in the buffer pool, i.e. the data does not all have to be in memory. With multiple users connected to the same server process, there is a likelihood of multiple large intermediate results having to be stored at the same time. This causes the results to revert earlier to a representation that can overflow to disk. Supposing 3 concurrent instances of Q3 on the same server process, the buffer pool of approximately 80G has approximately 48G taken by these intermediate results. This causes pages needed by the query to be paged out, leading to disk access later in

the query. Thus the effect of many instances of Q3 on the same server at the same time decreases the throughput more than linearly. This is the reason for the difference in timeshare percentage between the single-user and multi-user runs. The further problem in this query is that the large aggregation on count is on the end result, which re-aggregates the aggregates produced by different worker threads. This re-aggregation is due to the large amount of groups quite costly; therefore it dominates the execution time: the query does not parallelize well. A better plan would hash-split the aggregates early, such that re-aggregation is not required.

	50 Billion triples	
	Single-Client	4-Clients
runtime	1988s	4690s
Tput	22.629K	38.375K
	AQET	AQET
Q1	58.93	72.26
Q2	2.15	20.14
Q3	449.42	656.52
Q4	36.35	75.09
Q5	95.37	312.33
Q6	0.31	25.85
Q7	7.72	27.96
Q8	154.47	292.77

TABLE 1.7: Business Intelligence Use Case: Updated Results in March 2013

The benchmark results in the Table 1.6 are taken from our experiments running in January 2013. With more tuning in the Virtuoso software, we recently have re-run the benchmark with the dataset of 50B triples. The updated benchmark results in Table 1.7 show that the current version of Virtuoso software, namely Virtuoso7-March2013, can run the BSBM BI with a factor of 2 faster than the old version (i.e., the Virtuoso software in January). We note that the Micro benchmark was also run with the Virtuoso7-March2013. Similar improvement on the benchmark results is also expected when we re-run the benchmark with the dataset of 150B triples.

We now discuss the performance results in the Explore use case. We notice that these 4-client results seem more noisy than the single-client results and therefore it may be advisable in future benchmarking to also use multiple runs for multi-client tests. What is striking in the Explore results is that Q5 dominates execution time.

Query 5 in the Explore use case:

```
SELECT DISTINCT ?product ?productLabel
WHERE {
 ?product rdfs:label ?productLabel .
 FILTER (%ProductXYZ% != ?product)
 %ProductXYZ% bsbm:productFeature ?prodFeature .
 ?product bsbm:productFeature ?prodFeature .
```

	50 Billion triples		150Billion triples	
	Single-Client	4-Clients	Single-Client	4-Clients
runtime	931s (100 runs)	15s (1run)	1894s (100 runs)	29s (1 run)
Tput	4.832M	11.820M	7.126M	18.386M
	AQET	AQET	AQET	AQET
Q1	0.066s	0.415s	0.113s	0.093s
Q2	0.045s	0.041s	0.066s	0.086s
Q3	0.112s	0.091s	0.111s	0.116s
Q4	0.156s	0.102s	0.308s	0.230s
Q5	3.748s	6.190s	8.052s	9.655s
Q7	0.155s	0.043s	0.258s	0.360s
Q8	0.100s	0.021s	0.188s	0.186s
Q9	0.011s	0.010s	0.011s	0.011s
Q10	0.147s	0.020s	0.201s	0.242s
Q11	0.005s	0.004s	0.006s	0.006s
Q12	0.014s	0.019s	0.013s	0.010s

TABLE 1.8: Explore Use Case: Detailed Results

```
 %ProductXYZ% bsbm:productPropertyNumeric1 ?origProperty1 .
 ?product bsbm:productPropertyNumeric1 ?simProperty1 .
 FILTER (?simProperty1 < (?origProperty1 + 120) &&
         ?simProperty1 > (?origProperty1 - 120))

 %ProductXYZ% bsbm:productPropertyNumeric2 ?origProperty2 .
 ?product bsbm:productPropertyNumeric2 ?simProperty2 .
 FILTER (?simProperty2 < (?origProperty2 + 170) &&
         ?simProperty2 > (?origProperty2 - 170))
}
ORDER BY ?productLabel
LIMIT 5
```

Q5 asks for the 5 most similar products to one given product, based on two numeric product properties (using range selections). It is notable that such range selections might not be computable with the help of indexes; and/or the boundaries of both 120 and 170 below and above may lead to many products being considered ‘similar’. Given the type of query, it is not surprising to see that Q5 is significantly more expensive than all other queries in the Explore use case (the other queries are lookups that are index computable. – this also means that execution time on them is low regardless of the scale factor). In the explore use case, most of the queries have the constant running time regardless of the scalefactor, thus computing the throughput by multiplying the qph (queries per hour) with the scalefactor may show a significant increase between the cases of 50-billion and 150-billion triples. In this case, instead of the throughput metric, it is better to use another metric, namely qmph (number of query mixes per hour).

	Single Client	4-Clients
50B	4253.157	2837.285
150B	2090.574	1471.032

TABLE 1.9: Explore Results: Query Mixes Per Hour

1.4 Conclusion

In this paper we have described the new Virtuoso Column Store Cluster Edition RDF store, and examined its architecture and performance, using the the BSBM benchmark for the experimental evaluation. This new cluster architecture allows to perform RDF data management on a scale that is unprecedented: the experimental results presented here[3] mark a 750 times increase on previously reported BSBM results[4]. Further, this demonstration not only contains short-running index lookup queries (The Explore use case), but also complex analytical queries that each touches a large fraction of the triples in the database (the BI use case).

Execution times in the BI use case are multiplied by a factor of three when going from a single client to 4 concurrent clients, which is consistent with a good degree of parallelism within a single query, a necessary prerequisite for complex analytic workloads. The queries themselves consist almost entirely of cross partition joins, thus we see that scalability does not result from an "embarrassingly parallel" workload, i.e. one where each partition can complete with no or minimal interaction with other partitions. We also see a linear increase in execution times when tripling the data size from 50 billion to 150 billion triples. The queries are generally in the order of $n\log n$, where n is the count of triples. The log component comes from the fact of using tree indexes where access times are in principle logarithmic to index size. However the log element is almost not felt in the results due to exploitation of vectoring for amortizing index access cost.

The execution is not significantly bound by interconnect, as we observe aggregate throughput of about 2GB/s on the 8 node QDR InfiniBand network, whose throughput in a n:n messaging pattern is several times higher. Latency is also not seen to cut on CPU utilization, as the CPU percent is high and execution has minimal synchronous barriers.

Having established this, several areas of potential improvement remain. Some queries produce intermediate results that are all passed via a central location when this is not in fact necessary (Q3 BI). Such aggregation can be partitioned better by using the GROUP BY key as partitioning key – a query optimization problem. All the joining in the benchmark, which consists almost only of JOIN's and GROUP BY's was done with index lookups. Use of

[3] Full latest BSBM report (April 2013), http://bit.ly/ZHtG5D
[4] Previous BSBM report (February 2011), http://bit.ly/12DpjMU

hash joins in many places could improve both throughput and locality, cutting down on network traffic.

Message compression may also reduce blocking on message passing, yielding smoother execution with less forced task switches.

In any case, the present results demonstrate that complex query loads on a schema-less data model are feasible at scale.

Bibliography

[1] Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.

[2] Orri Erling. LOD2 deliverable 2.2: Dynamic repartitioning. In *http://static.lod2.eu/Deliverables/LOD2_D2.2_Dynamic_Repartitioning.pdf*, 2012.

[3] Orri Erling. LOD2 deliverable 2.6: Knowledge store release with integrated bulk processing features. In *http://static.lod2.eu/Deliverables/D2.6_Knowledge_Store_Release_With_Integrated_Bulk_Processing_Features.pdf*, 2012.

[4] Orri Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull*, 35(1):3–8, 2012.

[5] Orri Erling and Ivan Mikhailov. Faceted views over large-scale linked data. In *LDOW*, 2009.