Collaborative Project

LOD2 – Creating Knowledge out of Interlinked Data

Project Number: 257943 **Start Date of Project:** 01/09/2010 **Duration:** 48 months

Deliverable 2.1.5

500 Billion Triple Dataset Hosted on the LOD2 Knowledge Store Cluster

Dissemination Level	Public
Due Date of Deliverable	Month 39, 30/11/2013
Actual Submission Date	Month 48, 18/08/2014
Work Package	WP 2, Storing and Querying Very Large Knowledge bases
Task	T2.1.5
Type	Software
Approval Status	Approved
Version	1.0
Number of Pages	51
Filename	LOD2_D2.1.5_LOD_Cloud_Hosted_On_The_LOD2_Knowledge_Store_Cluster_500B_Triples

Abstract: This report gives an overview of the Virtuoso Column Store Edition Cluster BSBM benchmarking testing against 500 billion triple generated datasets, to prove the ability to scale these size datasets with multiple concurrent user workloads.

Project co-funded by the European Commission within the Seventh Framework Programme (2007 – 2013)

History

Version	Date	Reason	Revised by
0.1	10/08/2014	Draft of 500B triples LOD Cloud hosted on the LOD2 Knowledge Store Cluster	Hugh Williams
0.2	28/08/2014	Draft of 500B triples LOD Cloud hosted on the LOD2 Knowledge Store Cluster	Orri Erling
0.9	29/08/2014	Final Draft of 500B triples LOD Cloud hosted on the LOD2 Knowledge Store Cluster	Orri Erling, Hugh Williams
1.0	29/08/2014	Final Review and Editing	Peter Boncz

Author List

Organisation	Name	Contact Information
OGL	Hugh Williams	hwilliams@openlinksw.com
OGL	Orri Erling	oerling@openlinksw.com
OGL	Ivan Mikhailov	imikhailov@openlinksw.com
CWI	Peter Boncz	P.Boncz@cwi.nl
CWI	Minh Pham Duc	P.Minh.Duc@cwi.nl

Table of Contents

Executive Summary 4

Introduction 5

- BSBM benchmark results to date 5
- Outline 6

Experiment Definition 7

- Dataset 7
- Cluster Partitioning 7
- Terms 8

BSBM 500B Bulk Load 9

BSBM 500B Explore 10

- Network Bottleneck 10
- Platform Utilization 11
- Query Plan Analysis 12
- Execution Profiles 13

BSBM 500B Update + Update & Explore 15

BSBM 500B Business Intelligence 18

- Query Plan Analysis 19
- Platform Utilization 20
- Execution Profiles 21

Conclusions 24

Appendix A – Query Plans 25

Appendix B – Virtuoso Configuration Files 46

References 51

Executive Summary

This document continues the series of experiments in LOD2 around management of large RDF data. The scale is now over 3 times larger than in the previous set of experiments, now totalling 500 billion triples. The experiments are carried out at the Scilens cluster of CWI. We show the BSBM explore, update and business intelligence workloads with extensive analysis of cluster performance dynamics and query execution profiles.

We demonstrate radical advances in query optimization, using diverse hash based techniques that were not in use in the previous experiments. These yield over an order of magnitude performance increase over the previous 150 billion triple runs. For analytics, the data is over 3x larger, yet execution times are a fraction of their former value.

Results of RDF query processing on both transactional (Explore & Update) and analytical (BI) workloads have never before been demonstrated on this scale, **marking absolute records** and demonstrating through the LOD2 EU project how RDF technology can be matured to match the level of sophistication and performance of relational database technology.

In particular, during the project RDF technology has been enriched with the following state-of-the-art database techniques:

- compressed columnar storage (better data locality, reduced RAM requirements)
- vectorized query execution (better I/O locality and reduced CPU query execution cost)
- hash-based joins and aggregations (instead of purely index-based processing)
- cost-based query optimization (proper operator ordering and use of hash vs index)
- cluster-based (MPP) data warehousing (full distribution of all SPARQL1.1 operators)
- distributed query optimization (partitioning locality, replication)

In spin-off work that will follow the LOD2 project, relational and RDF database technology will fully come together by adopting automatic tabular data structures in RDF data management, i.e. the work that automatically detects RDF structure regularity ("characteristic sets"). These expected advances are based on joint research conducted in LOD2 by CWI and Openlink.

Introduction

The original Virtuoso Server was developed as a row-wise transactional orientated RDBMS including built-in RDF Data storage , with clustered capabilities for scale out across commodity level hardware enabling the hosting of large RDF datasets in particular. As the RDF Datasets in the LOD cloud generally have grown in size, the need to host these datasets with increasing scale and performance has tested the row-wise RDF implementation to it limits.

Thus, with a view to trying to get RDF data storage and querying on a par with the relational model, as part of the LOD2 project in collaboration with our partner CWI, we have introduced the following innovations:

- compressed columnar storage (better data locality, reduced RAM requirements)
- vectorized query execution (better I/O locality and reduced CPU query execution cost)
- **hash-based joins and aggregations (instead of purely index-based processing)**
- cost-based query optimization (proper operator ordering and use of hash vs index)
- cluster-based (MPP) data warehousing (full distribution of all SPARQL1.1 operators)
- **distributed query optimization (partitioning locality, replication)**

With respect to the previous iteration of this deliverable (i.e., D2.1.4) the Virtuoso v7 Cluster Edition was enhanced with the techniques in boldface. In the course of 2014, Openlink changed its distribution model, introducing a "fasttrack" Virtuoso release, which is essentially a beta release program.

We note that the rested V7 results are based exclusively on this fasttrack release. Over time it will be incorporated in the official V7 product.

BSBM benchmark results to date

The BSBM (Berlin SPARQL BenchMark) was developed in 2008 as one of the first open source and publicly available benchmarks for RDF data stores. 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.

Results:

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 the deliverable:

- BSBM version 1 (July 2008) –100 million triples, BSBM Explore
- BSBM version 2 (Nov 2009) – **200 million triples**, BSBM Explore
- BSBM version 3 (Feb 2011) - 200 million triples, BSBM Explore + **Explore&Update**
- LOD2 D2.1.4 (January 2013) - **150 billion triples** (!), BSBM Explore + **BI**.

The results presented in this deliverable (LOD2 D2.1.5) mark two advances:

- an increase by x3.3 in scale, i.e. datasets of **500 billion triples**
- incorporation of the **Update** and **Explore &Update** workloads besides Explore + BI.

Outline

The remainder of this document is structured as follows:

- Experiment Definition: hardware platform, dataset and used terms
- BSBM Bulkload
- BSBM Explore workload
- BSBM Update + Update & Explore workloads
- BSBM BI workload
- Conclusion & Outlook

Experiment Definition

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 (www.scilens.org) 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, these experiments were carried with the new "stones" ring.

Virtuoso V7 Column Store Cluster Edition (incorporating fasttrack innovations) was set up on 12 Linux machines. The equipment consisted of 12 units with each dual Xeon E5 2650v2, 8 core, 16 threads per CPU, 2.6GHz, 256GB RAM, QDR InfiniBand. Each machine had 3 2TB magnetic disks in RAID 0 (striping - 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. It was verified experimentally that this configuration works best.

The experiments with 150 billion triples in January 2013 (LOD2 D2.1.4) were made on a slightly different set of machines (the "bricks" layer), 8 units of 2x Xeon E5 2650 (98 core, 16 thread, 2.0GHz) with QDR InfiniBand and 3 2TB magnetic disks each. Thus the clock is 30% faster and there are 50% more machines for a data size of 3.3 x the previous. We could not use the bricks layer again for lack of its availability for experimentation plus the fact that the extra 4x256GB memory is needed during query execution.

Dataset

The experiments were carried out with a 500 billion triple Berlin SPARQL Benchmark (BSBM) dataset. The scale factor was 500 * 2840000 products. The data generator was modified to drop the tens of millions of distinct namespace prefixes it generated, which were found to substantially hamper a previous set of experiments in January 2014, especially in bulkload.

The data generator was further modified to segment output into consecutive files and not to generate different parts of the dataset in parallel. Previously, the data generator wrote multiple files in round-robin fashion, thus effectively destroying locality. Therefore the generator now works as any other similar program.

This modified BSBM data generator with modifications is available as a part of v7fasttrack feature/analytics at **github**.com/v7fasttrack

Cluster Partitioning

The default Virtuoso index scheme was used, i.e. a single quads table with two covering indices on PSOG, POSG and distinct projections of SP, OP, GS. The partitioning key is S or O, whichever is first in key order.

The partitioning is by hash of the key value, omitting the low 8 bits.

Thus every consecutive 256 values will be in the same partition. Integers between 0 and 100000 are an exception. If such an integer occurs as O value and partitioning is on O, then the low 8 bits are not ignored. Since there is an index on all O values and small integers are common it is important not to have 1 and 2 in the same partition. This is enabled by the enable_small_int_part setting in the INI files.

The logical cluster consists of 24 processes spread over 12 machines. The processes are each assigned to a NUMA node on the machine so that mostly local memory is used.

Each process has 16 partition slices, one per hardware thread. The entire cluster thus consists of 384 partition slices. The server configuration files are in the appendix.

Each process has an independent HTTP listener and SPARQL end point.

Terms

The following terms will be used in the tables representing the benchmark 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 500 billion triples dataset is 5000.

- **Timeshare**(%): The percentage of the query runtime in total runtime of all queries.

  *Timeshare(query q) = (Total runtime of q) / (Total runtime of all queries) * 100*

(It also can be calculated as Timeshare(query q) = 100*aqet*runsPerQuery/totalRuntime, where

aqet is the average query execution time for a query, runsPerQuery is the number of runs of that query)

  - **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)

  - **QMpH: Queries Mixes per Hour**. Number of query mixes with different parameters that are executed per hour against the SUT

BSBM 500B Bulk Load

Each machine loaded its local set of 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 12 machines, 168 files were being read at the same time.

The total uncompressed size of teh dataset is 28TB yet the database size (which contains the data x in different clustered indexes) ends up 18TB, thanks to compression.

Data loading was done in many steps, with different sets of machines as 2 of the initial 12 machines were defective (extremely slow disks). Parts of the load were done with a single 1Gbit Ethernet per unit, others with InfiniBand. The best sustained load rate with Ethernet was 5.2Mt/s and 6.8Mt/s with InfiniBand.

The initial load experiment suggests that if the data can be loaded as a single bulk load this will finish under 24h with the cluster used.

Following is a sample starting 6 hours before the end of the load and covering 4 hours. The numbers are triples per second measured in one minute windows, first average (, i.e. 5.8Mt/s), then the rate for the slowest and the fastest minute:

```
select avg (lm_rows_per_s), min (lm_rows_per_s), max (lm_rows_per_s)  from ld_metric where lm_dt between cast ('2014-8-16 00:00:00' as datetime) and cast ('2014-8-16 04:00:00' as datetime);

5804637  39072  7149827
```

BSBM 500B Explore

This section discusses the single and multiuser runs of the Explore workload.

Runs are made under the following conditions. All runs are warm i.e., the second run with the same seed is reported:

- 1 client, 100 mixes
- 16 clients 64 mixes, all clients on same server
- 32 clients 64 mixes, all clients on same server
- 48 clients, 2 clients per server

The BSBM driver data on this run produced the following table of results:

500B triples								
	Single-client		**Multi-client (16)**		**Multi-client (32)**		**Multi-client (48)**	
runtime	1840		233.38		229.36		224.85sec	
Tput	24447087.13		185100899.40		188347545.99		192130607.00	
Qmph	195.57		1480.80		1506.78		1537.04	
	AQET	**Timeshare**	**AQET**	**Timeshare**	**AQET**	**Timeshare**	**AQET**	**Timeshare**
Q1	0.118	0.64%	0.201	0.58%	0.366	0.56%	0.461	0.52%
Q2	0.016	0.53%	0.035	0.62%	0.045	0.42%	0.095	0.64%
Q3	0.211	1.15%	0.122	0.35%	0.150	0.23%	0.296	0.33%
Q4	0.241	1.31%	0.272	0.76%	0.356	0.55%	0.634	0.71%
Q5	8.601	93.55%	16.626	95.77%	31.429	97.15%	43.158	96.77%
Q7	0.083	1.81%	0.102	1.85%	0.105	0.65%	0.144	0.64%
Q8	0.027	0.29%	0.042	0.25%	0.039	0.12%	0.0488	0.11%
Q9	0.006	0.13%	0.013	0.15%	0.012	0.07%	0.013	0.06%
Q10	0.027	0.30%	0.042	0.24%	0.041	0.13%	0.083	0.19%
Q11	0.036	0.20%	0.011	0.03%	0.010	0.01%	0.009	0.01%
Q12	0.009	0.05%	0.011	0.03%	0.047	0.07%	0.013	0.02%

We note that the Tput numbers achieved 24M (single-client) and ~190M (multi-client) are significantly increased from the previous cluster-baed explore experiments on teh 150B triple dataset (which were 7M resp 18M).

Network Bottleneck

The motivation for the tested configurations, specifically with respect to the capabilities of the cluster network hardware, is as follows. In the 48 client case we observe CPU at 7000% in terms of the Linux top utility (out of a maximum of 19200% for 12*16=192 real cores) and 500K m/s (messages per second) and around 1GB/s of interconnect throughput. The average message size is thus around 2KB.

In a separate microbenchmark, we measured the peak interconnect throughput for 2KB messages so that 3 threads on each server send a message to every other server and wait for all to have responded before sending the next. This is a close approximation of the synchronous pattern of resolving RDF literals in Q5, which is most of the work. The throughput is 800K m/s and 1.7GB/s, without any other work. From this we estimate that a 72 client simulation would be surely network bound, therefore we stopped at 48 clients.

We also microbenchmarked the interconnect throughput with 20K messages. The resulting throughput is lower, 709K m/s. The interconnect is clearly congested without other use of the switch. The point to point ping with long messages does however get close to 2GB/s throughput, which is consistent with the nominal 40Gbit at 110/88 bits, i.e. 32Gbit/s, further subtracting TCP protocol overheads. The switch carries this throughput port to port on multiple independent pairs of ports, as intended. More efficient use of interconnect could be obtained via busy waiting and use of lower level API's, right now the cluster uses the standard TCP/IP calls. The protocol is TTCP, SDP is not used. Prior experiments did not show gain from SDP or zero copy.

Platform Utilization

The Explore workload brings the working set in memory. Below is a sample of a 60s window showing the cluster activity during a run on 96 query mixes on 48 threads, two connected to each of the 24 SPARQL end points:

```
Cluster 24 nodes, 66 s. 171773 m/s 1048171 KB/s  5452% cpu 112%  read 60579% clw threads 711r 0w 668i buffers 104235251 1019 d 43 w 0 pfs

cl 1: 2584 m/s 2620 KB/s  174% cpu 5%  read 3017% clw threads 33r 0w 31i buffers 4320122 15 d 9 w 0 pfs

cl 2: 4007 m/s 2979 KB/s  179% cpu 6%  read 2610% clw threads 30r 0w 30i buffers 4306404 11 d 0 w 0 pfs

cl 3: 4701 m/s 28199 KB/s  200% cpu 7%  read 3029% clw threads 21r 0w 21i buffers 4254030 10 d 0 w 0 pfs

cl 4: 3208 m/s 1649 KB/s  190% cpu 7%  read 2605% clw threads 34r 0w 34i buffers 4240268 12 d 0 w 0 pfs

cl 5: 3242 m/s 3242 KB/s  183% cpu 5%  read 2802% clw threads 36r 0w 36i buffers 4322908 10 d 0 w 0 pfs

cl 6: 33512 m/s 597837 KB/s  715% cpu 6%  read 270% clw threads 50r 0w 13i buffers 4776538 11 d 32 w 0 pfs

cl 7: 2245 m/s 2942 KB/s  185% cpu 5%  read 2036% clw threads 28r 0w 28i buffers 4206852 10 d 0 w 0 pfs

cl 8: 25852 m/s 52278 KB/s  256% cpu 5%  read 2578% clw threads 35r 0w 32i buffers 4542751 65 d 0 w 0 pfs

cl 9: 14942 m/s 52646 KB/s  251% cpu 5%  read 3398% clw threads 26r 0w 26i buffers 4668849 227 d 0 w 0 pfs

cl 10: 10858 m/s 26787 KB/s  225% cpu 4%  read 1922% clw threads 25r 0w 25i buffers 4366502 11 d 0 w 0 pfs

cl 11: 3975 m/s 11121 KB/s  187% cpu 2%  read 3201% clw threads 33r 0w 33i buffers 4427374 14 d 0 w 0 pfs

cl 12: 1573 m/s 1046 KB/s  180% cpu 1%  read 2654% clw threads 23r 0w 23i buffers 4221077 9 d 0 w 0 pfs

cl 13: 14395 m/s 66537 KB/s  225% cpu 3%  read 2905% clw threads 23r 0w 23i buffers 4236593 12 d 0 w 0 pfs

cl 14: 2158 m/s 1677 KB/s  172% cpu 3%  read 3082% clw threads 30r 0w 30i buffers 4209747 10 d 0 w 0 pfs

cl 15: 1835 m/s 2135 KB/s  181% cpu 4%  read 2247% clw threads 29r 0w 29i buffers 4157306 10 d 0 w 0 pfs

cl 16: 3518 m/s 1423 KB/s  176% cpu 3%  read 3555% clw threads 39r 0w 39i buffers 4243402 13 d 0 w 0 pfs

cl 17: 4224 m/s 27842 KB/s  209% cpu 4%  read 2133% clw threads 27r 0w 27i buffers 4280584 14 d 0 w 0 pfs
cl 18: 1608 m/s 799 KB/s  174% cpu 4%  read 2023% clw threads 28r 0w 28i buffers 4221572 18 d 0 w 0
pfs

cl 19: 1337 m/s 1493 KB/s  179% cpu 3%  read 3299% clw threads 32r 0w 32i buffers 4240334 10 d 0 w 0
pfs

cl 20: 12365 m/s 92368 KB/s  277% cpu 4%  read 2583% clw threads 29r 0w 29i buffers 4637691 485 d 0
w 0 pfs

cl 21: 5673 m/s 7000 KB/s  204% cpu 4%  read 2441% clw threads 34r 0w 34i buffers 4337884 8 d 1 w 0
pfs

cl 22: 2903 m/s 9996 KB/s  178% cpu 4%  read 3014% clw threads 34r 0w 34i buffers 4263713 13 d 1 w 0
pfs

cl 23: 8133 m/s 49978 KB/s  266% cpu 4%  read 2164% clw threads 15r 0w 14i buffers 4371113 8 d 0 w 0
pfs

cl 24: 2910 m/s 3562 KB/s  273% cpu 4%  read 996% clw threads 17r 0w 17i buffers 4381637 13 d 0 w 0
pfs
```

We see that the load is even with many processes working at or around 200% CPU and that the platform utilization is around 1/4 (since each Virtuoso process "has" 8 real cores, hence 800% CPU max). The cross sectional interconnect traffic of over 1GB/s suggests, as mentioned, an interconnect bound situation, rather than an I/O bound situation.

Further, the 66000% clw number on the top line indicates that on the average 660 threads are waiting for a reply from another partition. This is caused almost exclusively by the sorting on product label in Q5. Notice that string literals, due to their encoding into O numbers (SPOG storage), must be looked up when their real value is used, which requires partitioned network traffic.

As an experiment, we replaced the sort on label in Q5 by a sort on a numeric property. The platform utilization is as follows:

```
The clw% drops to 4200 but the run time stays within 10%.
```

The situation remains is interconnect bound.

We remind that in previous deliverables we have already written about the undesired property of Q5 to dominate the BSBM Explore workload. It remains our recommendation to change the workload with respect to this query.

Query Plan Analysis

The Explore workload offers fairly little opportunity for query optimization. We note that all queries have an order by and a limit. Thus, after the order by has some initial content, an extra condition can be pushed down into the selection. So, if the results are sorted by descending date and there is a limit of 5 and 5 results exist, no results with a date greater than the 5th in the order by need be explored. This restriction can be pushed down as early as possible. This is done but the gains are not large since the queries touch relatively few rows.

Since the explore mix consists almost exclusively of Q5, we will briefly look at how this works.

We look at plans for Q1 and Q5.

Q1 finds products which have two given features. The most selective feature is used as a hash build side. Then the second most selective probes this, after which there is a range check and fetch of the label. This goes into another hash build side. Then the product type is scanned and the second hash table is probed.

An index based plan that starts with the most selective feature is 5% faster than this hash plan. The break-even between index and hash is a constant problem of RDF.

The plan is in Appendix A (r500qp.txt)

Q5 gets all products that have a feature in common with a given product and have numeric quantities close to those of the first product. The top 5 are returned in order of label. First the properties of the fixed product are fetched. Then it is joined to products sharing features with it, then the properties are compared.

Finally there is a top k order by on the label. The join covers 3 cross partition stages, first the initial product, then the product features of this, then the similar product. There is a distinct but since the partitioning key is in the distinct (the similar product), the distinct is collocated. There is a top k restriction on the name, so if are already 5 names a product with name that would fall after these is rejected early. The check is late in the plan and does not help much.

We note that the operation checking the label against the last label so far is 85% of the time. This is not wasted per se since the label string is used anyway in the sorting. The mapping from the O, a literal identifier, to the string is a cross partition round trip. It is done on a large vector of O's at a time but still has latency. In the case at hand this is done on 133K rows. These are however spread over 384 partitions, thus the single vector is not very long. Each element of this vector may go to any of the 24 servers, hence a very large number of short messages is generated. The effect is seen in the mentioned 66000% cluster wait (clw) in the run statistics of Explore.

For this reason some RDF stores inline short string literals, e.g. Big Data, earlier Virtuoso's. Virtuoso does not presently inline short strings because this is bad for compression.

The solution to the problem is eventually found in adaptive schema where some properties of some characteristic sets may be declared inlined if under a certain lenBh.

Execution Profiles

Below is a oprofile profile on the BSBM Explore workload:

This profile was obtained the 37.5G triple scale, but we think the results prepresentative of higher scales as well.

Commentary on the function represented is inline.

```
CPU: Intel Sandy Bridge microarchitecture, speed 2299.98 MHz (estimated)

Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (No unit
mask) count 100000

samples  %        symbol name

1847991   9.7017  itc_col_seg

1477473   7.7565  cs_decode

1084011   5.6909  ce_filter

-- Index lookup, mostly range lookup in Q5 on p = constant s = ?  and o > ? and o < ?


951535    4.9954  clrg_partition_dc

-- partitioning a vector into multiple messages, one per target partition


785421    4.1234  dv_compare_so

783786    4.1148  cr_n_rows

-- index lookup


633497    3.3258  memcpy_16

628326    3.2986  session_buffered_write

-- Partitioning


507507    2.6643  ce_col_cmp

448827    2.3563  dc_append_bytes

-- index lookup and fetching columns from index
```

```
438940    2.3044  ce_search_bm
423946    2.2257  ce_search_cmp
340257    1.7863  itc_ce_value_offset
339494    1.7823  ce_bm_nth
331564    1.7407  itc_next_ce_skip
312521    1.6407  itc_set_row_spec_param_row
-- index lookup, bm is for bitmap compression, e.g. any psog pon ps  where ps is unique, i.e. functional properties


296657    1.5574  ks_vec_partition_fast
-- partitioning


284161    1.4918  dc_any_value
279483    1.4672  ds_add
-- index lookup


272228    1.4292  clib_vec_read_into_slots
-- partitioning


269357    1.4141  itc_any_param
239101    1.2552  box_equal
236278    1.2404  itc_next_ce
225411    1.1834  itc_next_set_cmp
224176    1.1769  ce_result
222729    1.1693  ce_intd_range_ltB
-- More index lookup
```

We see that the directly scale-out related functions are under 20% of CPU.   The rest is index access. Hash operations play no role here even though they occur in the plans, however Q5 index access obscures them.

The workload is very network intensive, as we have seen.

BSBM 500B Update + Update & Explore

The update is run for 100 update mixes after multiuser Explore, however the working set is not warm. The CPU is just 72% across the whole cluster (from a max of 19200%, hence each core is just 0.3% busy) at throughout the run. The first run takes 150s. Repeating the same updates, which has no effect, takes 4.1s, hence 40x more CPU usage.

The Update average execution times are negligible compared to the Explore times.

Next, 32 update threads are run for 3200 update mixes. All are connected to the same process. The CPU is at 360% with an average 120 reads in parallel. The run is disk bound, interconnect traffic is 11K m/s and 1MB/s. The 3200 update mixes take 72s.

Small updates are slower than large ones due to less benefit from vectoring, i.e. few nearby rows are likely to be hit. To measure the overhead in message passing and parsing, we run the same updates again. The run takes 12.5s. The message traffic and scheduling is much the same but deletes are not found and inserts are already in place, so database operations are reduced to between 1/3rd - 1/5th, i.e. just the action of finding the place for the insert/delete.

It appears that the BSBM driver is not adequate for getting to a steady state in updates, since not enough updates can be generated. Hence to see steady state update throughput, another driver is used, a Virtuoso stored procedure running on a separate instance outside of the cluster. The driver has any number of threads which connect to a given SPARQL end point and execute updates similar to the BSBM ones. The update mix is 2 deletes for every 3 inserts. The insert makes 5 new products, 6 new offers and 1 review for each of the 5 products, for a total of 505 triples. The deletes delete a single product from one of the inserted ones, with 36 triples in each delete. The identifiers are random and the random update can be run indefinitely to produce a steady state without any dependence on the data generator.

A first run of the 96 explore query mixes with 48 clients is executed as first warm-up. After this finishes updates are run with 3 clients per end point.

This is continued until steady state is reached 120 minutes into the run the status line is:

```
Cluster 24 nodes, 45 s. 314761 m/s 52487 KB/s  3733% cpu 5593%  read 5673% clw threads 110r 0w 70i
buffers 140699103 49402531 d 493 w 0 pfs
```

We note that disk predominates, with network at just 314K m/s. While eventually the update workload will be in memory a warm up time of hours is unworkable for benchmarking. In this case, we had to accept that despite the warm up the measured experiment would not be warm. This suggests that future BSBM Update advances should pursue the use of SSD hardware, which cut down I/O time considerably.

It is worth noting that both Explore and BI do not suffer from prohibitive warm-up times even though they read from disk. This is due to the very high inherent parallelism in either large scans or large batches of index lookups. Up to 2 million reads may be concurrently scheduled across the cluster. Each of these in turn brings in quite often a whole extent worth of pages. An extent in this case is a set of 256 consecutive 8K pages used for the same column.

The random update however lacks all such opportunities since it by definition consists of point lookups that become known a few at a time. Such a worst case may only be handled by increasing random IO throughput in the hardware.

After 160 minutes of warmup, the working set is still not entirely hot but the read experiment had to be start to limitation we had in terms of SCILENS computation time. The relevant status line is:

```
Cluster 24 nodes, 19 s. 564898 m/s 94072 KB/s  6141% cpu 3007%  read 4746% clw threads 97r 0w 71i
buffers 154107505 60776225 d 260 w 0 pfs
```

We note that messages are numerous and short. Interconnect (564K m/s) at this stage has at this stage possibly become more limiting than disk.

And excerpt of the triples/s averaged over each run minute from the update driver log table follows. This is without explore workload:

```
RUN_MINUTE        aggregate
INTEGER NOT NULL  INTEGER NOT NULL
________________________________________________________________________________


0                 220351
1                 230703
2                 236693
3                 238131
4                 242943
5                 235805
6                 196003
7                 206448
8                 216047
9                 220980
10                225089
11                228730
```

While the Update workload from the previous section keeps running, a concurrent Explore workload is now started with 48 clients, 2 per end point.

Explore & Update do not hit lock contention since the reads are non-locking read committed.

A status line during the Explore:

```
Cluster 24 nodes, 49 s. 155650 m/s 1050665 KB/s  6086% cpu 1841%  read 68557% clw threads 896r 0w
687i buffers 158093819 62758195 d 10285 w 1 pfs
```

The Explore completes in 249 seconds, producing the following table of results:

	500B triples	
	Multi-client (48)	
runtime	249.00sec	
Tput	73493326.42	
Qmph	1387.94	
	AQET	Timeshare
Q1	3.863	(3.90%)
Q2	0.247	(1.49%)
Q3	3.958	(4.00%)
Q4	7.953	(8.04%)
Q5	31.077	(62.81%)
Q7	2.882	(11.65%)
Q8	2.122	(4.29%)
Q9	0.013	(0.29%)

Q10	1.696	(3.43%)
Q11	0.042	(0.04%)
Q12	0.043	(0.04%)

The update rates per minute fall during the explore as follows:

```
minute,         triples/s
15                 238278
16                 245461
17                 245638
18                 164704
19                 39853
20                 23365
21                 86720
22                 191348
23                 230824
```

The explore is 100% warm, having been run before but the update continues at about 20 disk reads pending at any one time. The Explore is hardly slowed down (**Qmph** 1387 vs 1537) but the update rate takes a hit. Neither pre-empts the other and the behaviour is smooth.

The update rate continues to grow after the read experiment, with the status like:

```
Cluster 24 nodes, 54 s. 626889 m/s 104420 KB/s  6871% cpu 1764%  read 4321% clw threads 77r 0w 70i
buffers 167890634 69598578 d 62 w 0 pfs
```

corresponding to a rate of 263K triples/s.

We note that the random insert rate is about 16-20x lower than the bulk load rate. This does not come from transactionality but from lack of locality and from predominance of large numbers of short messages. There is hardly any gain from vectoring in column store inserts and deletes if all hits are spread far apart, as is the case here. Lock or latch contention is minimal.

The update driver is found at binsrc/tests/bibm/bibm_update.sql in the Virtuoso v7fasttrack feature/analytics on github.

BSBM 500B Business Intelligence

The BSBM BI workload is run with a single user and with 4 concurrent users. The workload consists in both cases of 4 query mixes each with 7 different queries, with drill down behaviour enabled.

A single query mix is in both cases executed as warm-up before the reported results.

	500B triples			
	Single-client		**Multi-client (4)**	
runtime	5936sec		7854sec	
Tput	278960		210827	
Qmph	2.426		1.833	
	AQET	Timeshare	AQET	Timeshare
Q1	60.01	(10.78%)	1264.88	(17.34%)
Q2	20.04	(1.35%)	36.03	(0.49%)
Q3	57.29	(3.86%)	846.69	(11.61%)
Q4	127.82	(51.68%)	500.25	(41.16%)
Q6	5.10	(0.34%)	54.24	(0.74%)
Q7	5.88	(2.77%)	60.84	(5.84%)
Q8	72.24	(29.21%)	277.22	(22.81%)

This section discusses advances in query optimization and execution since the last large BSBM BI run from January 2013.

The cost model is greatly changed and supports diverse hash based operators which were not previously supported. The TPC-H blog series on the Virtuoso blog discusses the query optimization advances in depth. It cannot be overstated that RDF databasing can only make significant progress by incorporating the best of relational technology. TPC-H is thus an appropriate illustration of what a database needs to do, whether for relational or RDF workloads.

The supported hash based operators include:

- group by with and without partitioning
- distinct
- hash join with different cases for:
    - single integer key existence check
    - single integer key with dependent data, either unique or non unique
    - multipart key in unique and non-unique variants.

Each of these can occur in a partitioned or replicated variant. In the partitioned case, a high cardinality key is used to decide which partition the hash entry goes to, in replication, all server processes have a local copy of the hash table.

Each variant of hash table may occur with and without a Bloom filter. A Bloom filter is made if the build side involves a join or a selection, which is most often the case.

Any single key hash table probe where the probe value is a column can be merged inline in a table scan or index lookup. Thus the hash operation is done before fetching (materializing) dependent data, thus effectively supplying late materialization.

If a hash table is not guaranteed unique (cardinality restricting), it cannot be merged into the probing table access but its Bloom filter can, since this is always cardinality reducing. Thus Bloom filters are separated from the actual hash table probe.

If a hash table in a cluster setting is partitioned, its Bloom filter, which is much smaller can be replicated. This offers very effective

Pre-filtering also for large hash joins and cuts down on interconnect traffic.

A special right outer join variant of a hash join operator is provided, this is described in detail in the TPC-H blog series under Q13. This is an always partitioned hash join where the left side (mandatory) builds a hash and is probed by the right (optional) side.

The items not hit in the probe are then the ones for which the optional side has no match.

A hash join build side is first materialized as a table or row-wise tuples. A second pass constructs a linear hash table from this, with full parallelism. In this way, a hash join build always knows the optimal size and never needs to rehash.

A group by or distinct hash table shares the same structure but does need to rehash when growing. These hash tables are also linear and their layout is similar to the hash join build sides, so a hash join operator can probe these.

The hash join implementation is according to best relational practices. This does have some RDF adaptations, though, since the set of data types is larger and includes run time typed columns.

The initial 150 B BSBM runs in Jan 2013 were done without any hash join and also without partitioned group by. By the correct use of hash join in and partitioned group by the performance of the BI workload has increased by at least an order of magnitude, often cutting hours into minutes.

Query Plan Analysis

We take a sample of Q1, Q3 and Q8 of the BI mix. The query texts and execution profiles are in Appendix A (r500qp.txt).

Q1: One of the principal problems of RDF query optimization is the proper choice of index vs hash based plans. Depending on parameter choices plans will greatly vary.

We consider BI Q1: This counts reviews of products whose manufacturer is from one country and where the reviewer is from another country. The counts are grouped by the product type.

In the attached plan, we see three hash tables are built and replicated into all partitions:

1. all product types
2. all things from Austria
3. all things from Japan.

The plan then scans all producers and leaves the ones in the Austria hash table. The next join to reviews is collocated. The next step from review to reviewer is partitioned on the reviewer, hence cross partition. The partition break is denoted by stage <nn>. The reviewers are hash joined to the entities from the country, this is not a partition break since the hash table is replicated. The next step is in a different partition, by product, getting the type. The check for this being a product type is by hash. The final stage is partitioning on the product type, which is the grouping key. Since this is a relatively high cardinality item, a few hundred thousand distinct values, the group by is partitioned.

With other parameter choices, index based plans are generated. We note that checking whether a reviewer is from a country based on index has both S and O specified, with O as constant. It is imperative to use an index partitioned on S and not one on O, as the latter would create a bottleneck where all intermediate results pass via a single partition.

If the query is run entirely without hash join, the performance is extremely bad since checking whether a product type is in fact a product type hits extreme partition skew, as all products have one type in common. Doing this check by replicated hash table resolves this.

Thus, the index only plan takes 509s with 1150% CPU and the hash plan shown here takes 55s with 3500% CPU.

Q3: Lists the products with the greatest increase in review count between two given months. The same group by is made twice with different parameters and the results are joined.

The grouping key is of high cardinality, with around a billion values, hence the operation is partitioned. Since a group by is a hash table basically identical to one that is used for hash join, the plan uses one of the group by's to probe the other and then sorts the products in both hash tables by the ratio of the counts.

The index based plan starts with a group by, then reads it, then for each batch of grouping keys it makes a nested group by. The index based plan has poor platform utilization and is not well distributed, hence runs for 8 hours. In specific, all the tuples from te first group by pass via a single point before being again spread out to do the second group by. This is the reason why Q3 was the longest running of the Jan 2013 runs.

The hash plan has perfect parallelism and executes in minutes.

Q8: Selects vendors with prices below an average for a product within a certain category.

The plan makes two hash tables, one for the count of offers for products in the category and another or the average price of the product in the category. The average is done by sum and count, the division is done after the probe. Both hash build sides are group by's.

The plan then takes the average price per vendor and product. This is read and probes the hash table with the average price for the product. This generates the final group by.

We note that using a group by space as hash build side is much better than making a hash table of the derived table feeding the group by.

For one, the group by's has less rows than the data producing it. TPC-H has little dependence on this trick, with only slight gains possible in Q17 and Q20. BSBM BI and TPC DS have more use for this. The typical case is BSBM BI Q3, with a comparison of two time windows, a common business question.

This is one of the heavier queries of the workload and has a good platform utiilization with the hash based plan shown. We note that this single query reaches much higher platform utilization than the explore mix regardless of how many concurrent clients are added.

Platform Utilization

The per server status follows for one execution of Q8:

```
Cluster 24 nodes, 304 s. 2588136 m/s 989517 KB/s  17276% cpu 0%  read 3% clw threads 1r 0w 0i
buffers 239996317 1022 d 0 w 2044 pfs

cl 1: 10699 m/s 27811 KB/s  529% cpu 0%  read 3% clw threads 1r 0w 0i buffers 9996487 15 d 0 w 2042
pfs

cl 2: 84471 m/s 37699 KB/s  578% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 9 d 0 w 2 pfs

cl 3: 99450 m/s 39817 KB/s  589% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 10 d 0 w 0
pfs

cl 4: 111218 m/s 41414 KB/s  612% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 11 d 0 w 0
pfs

cl 5: 118444 m/s 42347 KB/s  604% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 10 d 0 w 0
pfs

cl 6: 113105 m/s 41628 KB/s  612% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 11 d 0 w 0
pfs
```

```
cl 7: 100352 m/s 39953 KB/s  590% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 11 d 0 w 0
pfs

cl 8: 86098 m/s 38049 KB/s  588% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 65 d 0 w 0
pfs

cl 9: 81477 m/s 37144 KB/s  575% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 227 d 0 w 0
pfs

cl 10: 88042 m/s 38015 KB/s  586% cpu 0%  read 0% clw threads 0r 0w 0i buffers 9999948 12 d 0 w 0
pfs

cl 11: 100786 m/s 39707 KB/s  588% cpu 0%  read 0% clw threads 0r 0w 0i buffers 9999989 13 d 0 w 0
pfs

cl 12: 113968 m/s 41504 KB/s  611% cpu 0%  read 0% clw threads 0r 0w 0i buffers 9999979 9 d 0 w 0
pfs

cl 13: 101323 m/s 39803 KB/s  587% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 12 d 0 w 0
pfs

cl 14: 114378 m/s 41517 KB/s  610% cpu 0%  read 0% clw threads 0r 0w 0i buffers 9999989 11 d 0 w 0
pfs

cl 15: 103065 m/s 40022 KB/s  588% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 11 d 0 w 0
pfs

cl 16: 98809 m/s 39452 KB/s  596% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 12 d 0 w 0
pfs

cl 17: 101625 m/s 39839 KB/s  588% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 15 d 0 w 0
pfs

cl 18: 115044 m/s 41624 KB/s  611% cpu 0%  read 0% clw threads 0r 0w 0i buffers 9999989 19 d 0 w 0
pfs

cl 19: 143729 m/s 45367 KB/s  1103% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 9 d 0 w 0
pfs

cl 20: 147593 m/s 45837 KB/s  1130% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 485 d 0 w
0 pfs

cl 21: 144081 m/s 45398 KB/s  1111% cpu 0%  read 0% clw threads 0r 0w 0i buffers 9999946 9 d 0 w 0
pfs

cl 22: 151321 m/s 46338 KB/s  1120% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 13 d 0 w 0
pfs

cl 23: 107580 m/s 52905 KB/s  1025% cpu 0%  read 0% clw threads 0r 0w 0i buffers 10000000 9 d 0 w 0
pfs

cl 24: 151466 m/s 46314 KB/s  1131% cpu 0%  read 0% clw threads 0r 0w 0i buffers 9999990 14 d 0 w 0
pfs
```

The load is over half the platform, as each process has 8 real cores plus another 8 as core threads, most of the real cores are busy. All 16 threads busy is around 30% higher in throughput than 8 threads busy when running memory intensive workloads like hash joins.

The messages are again many and short but there is much less synchronization than with the literal to string mapping problem of Explore Q5. Hence there is more choice of work to do and longer batches of running without depending on network.

Execution Profiles

Next we look at execution profiles for the BI mix:

```
CPU: Intel Sandy Bridge microarchitecture, speed 2299.98 MHz (estimated)

Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (No unit
mask) count 100000

samples  %        symbol name

2645238   6.0826  setp_chash_run

-- group by

1886395   4.3377  clrg_partition_dc

-- splitting a vector into per-partition messages
```

```
1873830   4.3088  itc_col_seg
-- index lookup, table scan
1597807   3.6741  dk_alloc
-- memory allocation, small blocks
1372047   3.1549  cs_decode
-- Table scan, column decompression
1281678   2.9471  ce_vec_iri32_sets_hash
-- invisible hash join, i.e. hash probe/Bloom filter merged into table lookup
1163678   2.6758  memcpy_16
-- memcpy, mostly for messages
984760    2.2644  cha_cmp
-- compare of hash join keys with keys in hash table or group by
885944    2.0372  ce_result
-- fetching column values, decompression
882550    2.0294  ds_add
846434    1.9463  ce_search_bm
694173    1.5962  itc_ce_value_offset
-- index lookup
672532    1.5464  cha_insert_1i
-- hash join build, single integer/iri id key with dependent
668279    1.5367  page_wait_access
-- buffer cache lookup
654763    1.5056  ce_search_cmp
647860    1.4897  itc_next_ce_skip
-- index lookup
641572    1.4753  ks_vec_partition_fast
-- partitioning
625931    1.4393  itc_next_set_cmp
587607    1.3512  itc_single_row_opt
573211    1.3181  ce_bm_nth
-- index lookup
555493    1.2773  qn_result
545691    1.2548  dc_append_bytes
-- query execution
496181    1.1409  itc_any_param
-- index lookup
488830    1.1240  cha_inline_1i_iri32
-- hash join merged into table lookup, special case for array of iri id's in robing column
483049    1.1107  box_hash
434832    0.9999  itc_fetch_col_vec
-- buffer cache lookup
427897    0.9839  dk_free_tree
-- small memory free
```

As expected we see more hash based operators with still more index than hash. This is appropriate to the workload, the plans for the BI mix are appropriate.

The dk_alloc/dk_free_tree pair is for allocating single values and is an extra overhead that can in many cases be replaced by allocating a vector of values as a single block. Small individual allocations are mostly bad for performance. Normal query execution does not have such.

The TPC-H blog series contains more examples of CPU profiles for analytical workloads.

We see here that while TPC-H is predominantly hash joins, the BSBM BI still has a large fraction of index access.

Conclusions

The present work defines the limits of accomplishment when applying best of breed analytics DB techniques to RDF workloads. The scale out implementation is rich in features and different execution patterns and does not suffer from any easily avoidable naivety or stupidity or lack of support for something that is evidently needed.

However, the present work remains naive insofar it uses the RDF quad as the basis of its physical data representation. Better results, this time on a true par with the best in relational analytics are possible when abandoning the quad as the unit of storage and moving to a physical representation based on characteristic sets.

Thus, a quads based representation has been taken to its natural limits and its shortcomings have been compensated for as much as feasible, through the use of vectoring, hash joins, partitioned operations, query optimization that optimizes based on physical data order and colocation in partitions and so forth. This is highly competent core database work by any standard.

The next steps, which are in part already taken consist of breaking free from the idea that data whose meaning is defined by triples and graphs would have to be stored as such. In this way, RDF will find its natural use as an interchange format and a schema-less way of expressing queries and applications. The schema-less-ness and self-description of RDF is in our experience the principal basis of its adoption. This must be preserved, hence a rigid schema-first approach with SPARQL a language is ruled out. Diverse such implementations have existed against relational back ends and predictably have failed to deliver RDF's promise of freedom and flexibility. However, keeping this promise does not imply forever carrying the basically needless burden of endless self-joins and the resulting overheads.

Thus the next steps will consist of using the structure inherent in RDF to guide physical storage. Initial results show a threefold increase in bulk load rates and a halving of space consumption with the BSBM data discussed here.

Appendix A – Query Plans

Explore

```
Q1   sparql {#Q1
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT ?product ?label
WHERE {
    ?product rdfs:label ?label .
    ?product a <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType220032> .
    ?product                       bsbm:productFeature                        <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature3485024> .
    ?product                       bsbm:productFeature                        <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature12834> .
    ?product bsbm:productPropertyNumeric1 ?value1 .
        FILTER (?value1 > 347)
        }
ORDER BY ?label
LIMIT 10

}  {
time    0.0017% fanout         1 input          1 rows
time      0.21% fanout         1 input          1 rows
{ hash filler
wait time       6.7% of exec real time, fanout         0
QF {
time          0% fanout         0 input          0 rows
Stage 1
time     0.045% fanout       777 input          1 rows
RDF_QUAD_POGS   7.8e+02 rows(t4.S)
 P =  #/productFeature   ,  O =  #/ProductFeature3485024
time       1.6% fanout         2 input        777 rows
Stage 2
time       1.3% fanout         0 input      18648 rows
Sort hf 39 replicated     8e+02 rows(t4.S)
}
}
```

```
time      0.25% fanout         1 input         1 rows
{ hash filler
Subquery 45
{
wait time        26% of exec real time, fanout         0
QF {
time         0% fanout         0 input         0 rows
Stage 1
time       4.3% fanout       204 input         1 rows
RDF_QUAD_POGS   9.5e+04 rows(t5.S)
 P =  #/productFeature  ,  O =  #/ProductFeature12834
hash partition+bloom by 43 (tmp)hash join merged if unique card   6.3e-07 -> ()
time    0.0061% fanout         1 input       204 rows
Hash source 39  not partitionable   6.3e-07 rows(t5.S) -> ()

After code:
      0: t4.S :=  := artm t5.S
      4: BReturn 0
time       1.6% fanout 0.0686275 input       204 rows
Stage 2
time       7.3% fanout  0.754902 input       204 rows
RDF_QUAD      0.96 rows(t6.S)
 P =  #/productPropertyNumeric1  ,  S = q_t5.S ,  O >  347  O >  347
time         5% fanout         1 input       154 rows
RDF_QUAD       1.4 rows(t2.S, t2.O)
 inlined  P =  ##label  ,  S = k_q_t4.S

After code:
      0: t4.S :=  := artm t4.S
      4: t6.S :=  := artm t6.S
      8: t5.S :=  := artm t5.S
      12: t2.S :=  := artm t2.S
      16: t2.O :=  := artm t2.O
      20: BReturn 0
time        73% fanout   17.7403 input       154 rows
Stage 3
time       4.2% fanout         0 input      3696 rows
Sort hf 88 replicated     8e+02 rows(t4.S) -> (t6.S, t5.S, t2.S, t2.O)

}
}
}
Subquery 94
{
time    0.0092% fanout         1 input         1 rows
```

```
{ fork
wait time      2.3% of exec real time, fanout      154
QF {
time      0.14% fanout      229 input         1 rows
RDF_QUAD_POGS     3e+03 rows(s_20_10_t1.S)
 P =  ##type  ,  O =  #/ProductType220032
hash partition+bloom by 92 ()
time     0.053% fanout  0.672489 input      229 rows
Hash source 88                 0  rows(s_20_10_t1.S)  ->  (s_20_10_t4.S,  s_20_10_t3.S,  s_20_10_t0.S,
s_20_10_t0.O)
time     0.039% fanout         1 input      154 rows
END Node
After test:
      0: if (s_20_10_t1.S = s_20_10_t4.S) then 4 else 13 unkn 13
      4: if (s_20_10_t1.S = s_20_10_t3.S) then 8 else 13 unkn 13
      8: if (s_20_10_t0.S = s_20_10_t1.S) then 12 else 13 unkn 13
      12: BReturn 1
      13: BReturn 0
time     0.014% fanout         0 input      154 rows
 qf select node output: (qf_set_no, s_20_10_t0.S, s_20_10_t0.O)
}
time     0.063% fanout         1 input      154 rows
Distinct (s_20_10_t0.S, s_20_10_t0.O)
time       1.1% fanout         0 input      154 rows

Precode:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
s_20_10_t0.O -> __RO2SQ -> __ro2sq
}

      2: BReturn 0
Sort (__ro2sq) -> (s_20_10_t0.S)

}
time     0.043% fanout        10 input         1 rows
top order by read (s_20_10_t0.S, __ro2sq)

After code:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
s_20_10_t0.S -> __ID2In -> __id2in
}
```

```
      2: label :=  := artm __ro2sq
      6: product :=  := artm __id2in
      10: BReturn 0
time   0.00079% fanout         0 input        10 rows
Subquery Select(product, label)
}

After code:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
label -> __RO2SQ -> label
product -> __RO2SQ -> product
}

      2: BReturn 0
time   0.00073% fanout         0 input        10 rows
Select (product, label)
}



Q5  sparql {#Q5

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>

SELECT DISTINCT ?product ?productLabel
WHERE {
       ?product rdfs:label ?productLabel .
    FILTER                                                                  (<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer/Product1294222259> != ?product)
       <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer/Product1294222259>           bsbm:productFeature
?prodFeature .
       ?product bsbm:productFeature ?prodFeature .
       <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer/Product1294222259>  bsbm:productPropertyNumeric1
?origProperty1 .
       ?product bsbm:productPropertyNumeric1 ?simProperty1 .
       FILTER (?simProperty1 < (?origProperty1 + 120) && ?simProperty1 > (?origProperty1 - 120))
       <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer/Product1294222259>  bsbm:productPropertyNumeric2
?origProperty2 .
       ?product bsbm:productPropertyNumeric2 ?simProperty2 .
       FILTER (?simProperty2 < (?origProperty2 + 170) && ?simProperty2 > (?origProperty2 - 170))
```

```
}
ORDER BY ?productLabel
LIMIT 5

}


{
 time   2.8e-07% fanout         1 input         1 rows
Subquery 28
{
time   7.3e-05% fanout         1 input         1 rows
{ fork
wait time   6.1e+03% of exec real time, fanout         0
QF {
time         0% fanout         0 input         0 rows
Stage 1
time     7e-06% fanout         1 input         1 rows
RDF_QUAD         1 rows(s_23_14_t3.O)
 inlined  P =  #/productPropertyNumeric1  ,  S =  #/Product1294222259
time   9.5e-06% fanout         1 input         1 rows

Precode:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
s_23_14_t3.O -> __RO2SQ -> __ro2sq
}

      2: temp := artm __ro2sq +  120
      6: temp := artm __ro2sq -  120
      10: BReturn 0
RDF_QUAD         1 rows(s_23_14_t5.O)
 inlined  P =  #/productPropertyNumeric2  ,  S =  #/Product1294222259
time   9.5e-06% fanout        19 input         1 rows

Precode:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
s_23_14_t5.O -> __RO2SQ -> __ro2sq
}

      2: temp := artm __ro2sq +  170
      6: temp := artm __ro2sq -  170
      10: BReturn 0
```

```
RDF_QUAD        19 rows(s_23_14_t1.O)
 inlined  P =  #/productFeature   ,  S =  #/Product1294222259
time   3.9e-05% fanout 0.0526316 input        19 rows
Stage 2
time     0.061% fanout    980979 input        19 rows
RDF_QUAD_POGS   3.4e+03 rows(s_23_14_t2.S)
 P =  #/productFeature   ,  O = q_s_23_14_t1.O
time       2.5% fanout  0.136491 input 3.83044e+07 rows
Stage 3
time       8.4% fanout  0.275013 input 3.83044e+07 rows
RDF_QUAD      0.11 rows(s_23_14_t6.S)
 inlined   P =   #/productPropertyNumeric2   ,   S = q_s_23_14_t2.S ,   O > q_q_temp < q_q_temp O >
q_q_temp < q_q_temp
time         3% fanout 0.0795689 input 1.05342e+07 rows
RDF_QUAD       0.1 rows(s_23_14_t4.S)
 inlined  P =  #/productPropertyNumeric1   ,  S = k_q_s_23_14_t2.S ,   O > k_q_q_temp < k_q_q_temp O >
k_q_q_temp < k_q_q_temp
time      0.82% fanout  0.159096 input    838196 rows
RDF_QUAD       1.3 rows(s_23_14_t0.O, s_23_14_t0.S)
 <inlined  P =  ##label   ,  S = k_q_s_23_14_t2.S
top k on O
time        85% fanout   0.99955 input    133354 rows
END Node
After test:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
s_23_14_t0.O -> __RO2SQ -> __ro2sq
}

      2: __topk := Call __topk (__ro2sq,  0 ,  3 ,  5 ,  0 ,  140391824282016 ,  0 )
      7: if ( 1  = __topk) then 11 else 12 unkn 12
      11: BReturn 1
      12: BReturn 0
time    0.0055% fanout  0.999887 input    133294 rows
END Node
After test:
      0: if (s_23_14_t0.S =  #/Product1294222259 ) then 5 else 4 unkn 5
      4: BReturn 1
      5: BReturn 0
time     0.016% fanout  0.901297 input    133279 rows
Distinct (s_23_14_t0.S, s_23_14_t0.O)
time     0.029% fanout         0 input    120124 rows
Sort (__ro2sq) -> (s_23_14_t0.S)

time   2.5e-07% fanout         0 input         0 rows
```

```
 ssa iterator
time   7.4e-05% fanout         5 input        24 rows
top order by read (s_23_14_t0.S, __ro2sq)
time   2.8e-05% fanout         0 input       120 rows
 qf select node output: (__ro2sq, s_23_14_t0.S)
}
}
time     0.012% fanout         5 input         1 rows
  cl fref read
 output: (__ro2sq, s_23_14_t0.S)
order:  0

After code:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
s_23_14_t0.S -> __ID2In -> __id2in
}


      2: productLabel :=  := artm __ro2sq
      6: product :=  := artm __id2in
      10: BReturn 0
time     3e-07% fanout         0 input         5 rows
Subquery Select(product, productLabel)
}

After code:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
productLabel -> __RO2SQ -> productLabel
product -> __RO2SQ -> product
}


      2: BReturn 0
time   3.5e-07% fanout         0 input         5 rows
Select (product, productLabel)
}
```

Business Intelligence

```
sparql {#Q7
  prefix bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
  prefix bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
  prefix xsd: <http://www.w3.org/2001/XMLSchema#>

  Select ?product
  {
    { Select ?product
      {
        { Select ?product (count(?offer) As ?offerCount)
          {
            ?product a <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType1> .
            ?offer bsbm:product ?product .
          }
          Group By ?product
        }
      }
      Order By desc(?offerCount)
      Limit 1000
    }
    FILTER NOT EXISTS
    {
      ?offer bsbm:product ?product .
      ?offer bsbm:vendor ?vendor .
      ?vendor bsbm:country ?country .
      FILTER(?country=<http://downlode.org/rdf/iso-3166/countries#KR>)
    }
  }

}                                                                                  {
Subquery 28
{
{ fork
{ fork
QF {
RDF_QUAD_POGS   2.9e+10 rows(s_17_4_t1.O)
 inlined  P =  #/product
RDF_QUAD       0.8 rows(s_17_4_t0.S)
 inlined  P =  ##type  ,  S = k_s_17_4_t1.O ,  O =  #/ProductType1
Sort (s_17_4_t0.S) -> (inc)

}
}
QF {
group by read node
(s_17_4_t0.S, aggregate)

After code:
      0: product :=  := artm s_17_4_t0.S
      4: offerCount :=  := artm aggregate
      8: BReturn 0
Subquery Select(product, offerCount)
Sort (offerCount) -> (product)

 ssa iterator
top order by read (offerCount, product)
 qf select node output: (offerCount, product)
}
}
  cl fref read
 output: (offerCount, product)
order:  0  desc

After code:
      0: product :=  := artm product
      4: BReturn 0
Subquery Select(product)
}
END Node
After test:
      0: if ({
QF {
Stage 1
RDF_QUAD_POGS        13 rows(s_28_14_t2.S)
 P =  #/product  ,  O = lcast
Stage 2
RDF_QUAD         1 rows(s_28_14_t3.O)
 inlined  P =  #/vendor  ,  S = q_s_28_14_t2.S
Stage 3
RDF_QUAD_POGS      0.15 rows()
 inlined  P =  #/country  ,  O =  ##KR  ,  S = q_s_28_14_t3.O
 qf select node output: (qf_set_no)
 qf select node output: (qf_set_no)
}
Subquery Select( <none> )
}
) then 5 else 4 unkn 5
      4: BReturn 1
      5: BReturn 0

After code:
      0: QNode {
dpipe
```

```
product -> __ID2In -> product
}

      2: BReturn 0
Select (product)
}

sparql {#Q1
prefix bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
prefix rev: <http://purl.org/stuff/rev#>

Select ?productType ?reviewCount
{
 { Select ?productType (count(?review) As ?reviewCount)
  {
   ?productType a bsbm:ProductType .
   ?product a ?productType .
   ?product bsbm:producer ?producer .
   ?producer bsbm:country <http://downlode.org/rdf/iso-3166/countries#AT> .
   ?review bsbm:reviewFor ?product .
   ?review rev:reviewer ?reviewer .
   ?reviewer bsbm:country <http://downlode.org/rdf/iso-3166/countries#JP> .
  }
  Group By ?productType
 }
}
Order By desc(?reviewCount) ?productType
Limit 10

}

{
time     9e-08% fanout         1 input         1 rows
time   0.00027% fanout         1 input         1 rows
{ hash filler
wait time   1.5e-05% of exec real time, fanout         0
QF {
time         0% fanout         0 input         0 rows
Stage 1
time   4.8e-05% fanout    231715 input         1 rows
RDF_QUAD_POGS    2.3e+05 rows(s_1_14_t0.S)
 P =  ##type   ,  O =  #/ProductType
time    0.0036% fanout         2 input    231715 rows
Stage 2
time     0.003% fanout         0 input 5.56116e+06 rows
```

```
Sort hf 39 replicated   2.3e+05 rows(s_1_14_t0.S)
}
}
time       0.24% fanout         1 input         1 rows
{ hash filler
wait time    0.0016% of exec real time, fanout         0
QF {
time          0% fanout         0 input         0 rows
Stage 1
time    0.00026% fanout    630000 input         1 rows
RDF_QUAD_POGS    3.5e+07 rows(s_1_14_t6.S)
 P =  #/country  ,  O =  ##JP
time       0.42% fanout         2 input 7.72273e+07 rows
Stage 2
time       0.73% fanout         0 input 1.85346e+09 rows
Sort hf 56 replicated   3.5e+07 rows(s_1_14_t6.S)
}
}
time       0.12% fanout         1 input         1 rows
{ hash filler
wait time   0.00067% of exec real time, fanout         0
QF {
time          0% fanout         0 input         0 rows
Stage 1
time    0.00013% fanout    630000 input         1 rows
RDF_QUAD_POGS    1.5e+07 rows(s_1_14_t3.S)
 P =  #/country  ,  O =  ##AT
time       0.19% fanout         2 input 3.86109e+07 rows
Stage 2
time       0.33% fanout         0 input 9.2666e+08 rows
Sort hf 73 replicated   1.5e+07 rows(s_1_14_t3.S)
}
}
Subquery 79
{
time      5e-08% fanout         1 input         1 rows
time    7.5e-05% fanout         1 input         1 rows
{ fork
time    6.3e-07% fanout         1 input         1 rows
{ fork
wait time   4.1e-05% of exec real time, fanout         0
QF {
time     0.0015% fanout         0 input         0 rows
Stage 1
time          1% fanout    185530 input       384 rows
RDF_QUAD   8.6e+08 rows(s_1_14_t2.O, s_1_14_t2.S)
 inlined  P =  #/producer
hash partition+bloom by 77 (tmp)hash join merged if unique card     0.025 -> ()
time    0.0028% fanout         1 input 7.12435e+07 rows
Hash source 73       0.025 rows(cast) -> ()
time       1.3% fanout   10.0027 input 7.12435e+07 rows
RDF_QUAD_POGS        29 rows(s_1_14_t4.S, s_1_14_t4.O)
 P =  #/reviewFor   ,  O = k_s_1_14_t2.S
time        38% fanout  0.952667 input 7.12624e+08 rows
Stage 2
time        32% fanout 0.0999923 input 7.12624e+08 rows
RDF_QUAD         1 rows(s_1_14_t5.O)
 inlined  P =  ##reviewer   ,  S = q_s_1_14_t4.S
hash partition+bloom by 60 (tmp)hash join merged if unique card     0.059 -> ()
time    0.0059% fanout         1 input 7.1257e+07 rows
Hash source 56       0.059 rows(cast) -> ()
time       2.8% fanout         1 input 7.1257e+07 rows
Stage 3
time       3.4% fanout         7 input 7.1257e+07 rows
RDF_QUAD       7.9 rows(s_1_14_t1.O)
 inlined  P =  ##type   ,  S = q_q_s_1_14_t2.S
hash partition+bloom by 43 (tmp)hash join merged if unique card       0.8 -> ()
time     0.065% fanout         1 input 4.98799e+08 rows
Hash source 39         0.8 rows(cast) -> ()

After code:
      0: s_1_14_t0.S :=  := artm s_1_14_t1.O
      4: BReturn 0
time        19% fanout  0.992588 input 4.98799e+08 rows
Stage 4
time      0.49% fanout         0 input 4.98799e+08 rows
Sort (q_s_1_14_t0.S) -> (inc)

}
}
wait time         0% of exec real time, fanout         0
QF {
time   0.00038% fanout   603.424 input       384 rows
group by read node
(s_1_14_t0.S, aggregate)

After code:
      0: productType :=  := artm s_1_14_t0.S
      4: reviewCount :=  := artm aggregate
      8: BReturn 0
```

```
time    8.1e-06% fanout          1 input    231715 rows
Subquery Select(productType, reviewCount)
time     0.0054% fanout          0 input    231715 rows

Precode:
      0: QNode {
time          0% fanout          0 input         0 rows
dpipe
productType -> __ID2In -> __id2in
}

      2: BReturn 0
Sort (reviewCount, __id2in)
time    1.1e-08% fanout          0 input         0 rows
 ssa iterator
time    2.8e-05% fanout    9.32249 input       369 rows
top order by read (__id2in, reviewCount)
time    9.9e-06% fanout          0 input      3440 rows
 qf select node output: (__id2in, reviewCount)
}
}
time    7.6e-05% fanout         10 input         1 rows
  cl fref read
 output: (__id2in, reviewCount)
order:  1  desc  0

After code:
      0: productType :=  := artm __id2in
      4: reviewCount :=  := artm reviewCount
      8: BReturn 0
time    1.8e-08% fanout          0 input        10 rows
Subquery Select(productType, reviewCount)
}

After code:
      0: QNode {
time          0% fanout          0 input         0 rows
dpipe
reviewCount -> __RO2SQ -> reviewCount
productType -> __RO2SQ -> productType
}

      2: BReturn 0
time    3.3e-08% fanout          0 input        10 rows
Select (productType, reviewCount)
```

```
}




sparql {#Q3
  prefix bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
  prefix bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
  prefix rev: <http://purl.org/stuff/rev#>
  prefix dc: <http://purl.org/dc/elements/1.1/>
  prefix xsd: <http://www.w3.org/2001/XMLSchema#>

  Select ?product (xsd:float(?monthCount)/?monthBeforeCount As ?ratio)
  {
    { Select ?product (count(?review) As ?monthCount)
      {
        ?review bsbm:reviewFor ?product .
        ?review dc:date ?date .
        Filter(?date >= "2007-12-22"^^<http://www.w3.org/2001/XMLSchema#date> && ?date < "2008-01-
19"^^<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 >= "2007-11-24"^^<http://www.w3.org/2001/XMLSchema#date> && ?date < "2007-12-
22"^^<http://www.w3.org/2001/XMLSchema#date>) #
      }
      Group By ?product
      Having (count(?review)>0)
    }
  }
  Order By desc(xsd:float(?monthCount) / ?monthBeforeCount) ?product
  Limit 10

}                                                                                  {
time   4.5e-08% fanout         1 input         1 rows
time   0.00014% fanout         1 input         1 rows
{ hash filler
wait time      0.55% of exec real time, fanout         0
QF {
time   5.1e-05% fanout         0 input         0 rows
Stage 1
time       2.3% fanout 3.70867e+07 input       384 rows
```

```
RDF_QUAD   1.4e+10 rows(t6.S, t6.O)
 inlined  P =  #/reviewFor
time        23% fanout 0.0481733 input 1.42413e+10 rows
RDF_QUAD     0.016 rows()
 P =  #/date   ,  S = t6.S ,  O >= <tag 211 c 2007-11-24> < <tag 211 c 2007-12-22> O >= <tag 211 c
2007-11-24> < <tag 211 c 2007-12-22>
time       4.7% fanout  0.949941 input 6.86051e+08 rows
Stage 2
time       2.4% fanout         0 input 6.86051e+08 rows
Sort (q_t6.O) -> (inc)

}
}
Subquery 77
{
time     2e-08% fanout         1 input         1 rows
time   6.4e-05% fanout         1 input         1 rows
{ fork
time   3.4e-07% fanout         1 input         1 rows
{ fork
wait time      0.89% of exec real time, fanout         0
QF {
time   0.00055% fanout         0 input         0 rows
Stage 1
time       2.4% fanout 3.70867e+07 input       384 rows
RDF_QUAD   1.4e+10 rows(s_17_4_t0.S, s_17_4_t0.O)
 inlined  P =  #/reviewFor
time        23% fanout 0.0600904 input 1.42413e+10 rows
RDF_QUAD     0.018 rows()
 P =  #/date   ,  S = s_17_4_t0.S ,  O >= <tag 211 c 2007-12-22> < <tag 211 c 2008-01-19> O >= <tag
211 c 2007-12-22> < <tag 211 c 2008-01-19>
time         6% fanout  0.948689 input 8.55765e+08 rows
Stage 2
time       3.3% fanout         0 input 8.55765e+08 rows
Sort (set_no, q_s_17_4_t0.O) -> (inc)

}
}
wait time        11% of exec real time, fanout         0
QF {
time   1.7e-06% fanout         0 input         0 rows
Stage 1
time      0.13% fanout    986742 input       384 rows
group by read node
(gb_set_no, s_17_4_t0.O, aggregate)
After code:
      0: product :=  := artm s_17_4_t0.O
      4: monthCount :=  := artm aggregate
      8: BReturn 0
time    0.0033% fanout         1 input 6.09854e+08 rows
Subquery Select(product, monthCount)
time      0.42% fanout         1 input 6.09854e+08 rows
Stage 2
time      0.85% fanout  0.422257 input 6.09854e+08 rows
Hash source 54            1 rows(q_product) -> (monthBeforeCount)
time     0.046% fanout         1 input 2.57515e+08 rows

Precode:
      0: product :=  := artm product
      4: BReturn 0
END Node
After test:
      0: if (product = product) then 4 else 5 unkn 5
      4: BReturn 1
      5: BReturn 0
time        31% fanout         0 input 2.57515e+08 rows

Precode:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
product -> __RO2SQ -> __ro2sq
}

      2: _cvt := Call _cvt (<constant>, monthCount)
      7: temp := artm _cvt / monthBeforeCount
      11: BReturn 0
Sort (temp, __ro2sq)
time   9.7e-09% fanout         0 input         0 rows
 ssa iterator
time   2.1e-06% fanout        10 input        24 rows
top order by read (__ro2sq, temp)
time   1.1e-06% fanout         0 input       240 rows
 qf select node output: (temp, __ro2sq, set_no)
}
}
time   8.9e-05% fanout        10 input         1 rows
  cl fref read
 output: (temp, __ro2sq, set_no)
order:  2   0  desc  1
```

```
After code:
      0: product :=  := artm __ro2sq
      4: ratio :=  := artm temp
      8: BReturn 0
time   1.2e-08% fanout         0 input        10 rows
Subquery Select(product, ratio)
}

After code:
      0: QNode {
time         0% fanout         0 input         0 rows
dpipe
ratio -> __RO2SQ -> ratio
product -> __RO2SQ -> product
}

      2: BReturn 0
time   8.3e-09% fanout         0 input        10 rows
Select (product, ratio)
}

sparql {#Q8
  prefix bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
  prefix bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
  prefix xsd: <http://www.w3.org/2001/XMLSchema#>

  Select ?vendor (xsd:float(?belowAvg)/?offerCount As ?cheapExpensiveRatio)
  {
    { Select ?vendor (count(?offer) As ?belowAvg)
      {
        { ?product a <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType4> .
          ?offer bsbm:product ?product .
          ?offer bsbm:vendor ?vendor .
          ?offer bsbm:price ?price .
          { Select ?product (avg(?price) As ?avgPrice)
            {
              ?product a <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType4> .
              ?offer bsbm:product ?product .
              ?offer bsbm:price ?price .
            }
            Group By ?product
          }
        } .
        FILTER (?price < ?avgPrice)
      }
      Group By ?vendor
    }
    { Select ?vendor (count(?offer) As ?offerCount)
      {
        ?product a <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType4> .
        ?offer bsbm:product ?product .
        ?offer bsbm:vendor ?vendor .
      }
      Group By ?vendor
    }
  }
  Order by desc(xsd:float(?belowAvg)/?offerCount) ?vendor
  limit 10

}                                                                                   {
time   1.3e-09% fanout         1 input         1 rows
time   9.6e-05% fanout         1 input         1 rows
{ hash filler
wait time      0.13% of exec real time, fanout         0
QF {
time         0% fanout         0 input         0 rows
Stage 1
time   0.00027% fanout 1.351e+07 input         1 rows
RDF_QUAD_POGS     1e+08 rows(t12.S)
 P =  ##type   ,  O =  #/ProductType4
time      0.15% fanout 0.0834747 input 1.27187e+08 rows
Stage 2
time      0.32% fanout   20.0004 input 1.27187e+08 rows
RDF_QUAD_POGS        13 rows(t13.S)
 P =  #/product   ,  O = lcast
time       1.9% fanout  0.963308 input 2.5438e+09 rows
Stage 3
time        21% fanout         1 input 2.5438e+09 rows
RDF_QUAD         1 rows(t14.O)
 inlined  P =  #/vendor  ,  S = q_t13.S
time        29% fanout  0.999996 input 2.5438e+09 rows
Stage 4
time      0.74% fanout         0 input 2.5438e+09 rows
Sort (q_t14.O) -> (inc)

}
}
time   2.6e-05% fanout         1 input         1 rows
{ hash filler
```

```
wait time       6.7% of exec real time, fanout         0
QF {
time          0% fanout         0 input         0 rows
Stage 1
time   0.00023% fanout 1.401e+07 input         1 rows
RDF_QUAD_POGS     1e+08 rows(t8.S)
 P =  ##type  ,  O =  #/ProductType4
time     0.077% fanout 0.0834747 input 1.27187e+08 rows
Stage 2
time      0.22% fanout   20.0004 input 1.27187e+08 rows
RDF_QUAD_POGS        13 rows(t9.S)
 P =  #/product  ,  O = lcast
time       1.2% fanout  0.984298 input 2.5438e+09 rows
Stage 3
time        34% fanout         1 input 2.5438e+09 rows
RDF_QUAD         1 rows(t10.O)
 inlined  P =  #/price  ,  S = q_t9.S
time       9.3% fanout         1 input 2.5438e+09 rows

Precode:
      0: temp := artm t10.O +  0
      4: BReturn 0
Stage 4
time       2.3% fanout         0 input 2.5438e+09 rows
Sort (q_q_q_t8.S) -> (temp, inc)

}
}
Subquery 145
{
time     3e-09% fanout         1 input         1 rows
time   2.6e-07% fanout         1 input         1 rows
{ fork
time   3.5e-08% fanout         1 input         1 rows
{ fork
wait time   0.00018% of exec real time, fanout         0
QF {
time          0% fanout         0 input         0 rows
Stage 1
time   7.5e-09% fanout         0 input         1 rows
RDF_QUAD_POGS     1e+08 rows(s_22_16_t3.S)
 P =  ##type  ,  O =  #/ProductType4
hash partition+bloom by 143 ()
time          0% fanout         0 input         0 rows
Stage 2
time          0% fanout          0 input          0 rows
Hash source 122           1 rows(q_s_22_16_t3.S) -> (a15, a16)
time          0% fanout          0 input          0 rows

Precode:
      0: product :=  := artm s_22_16_t3.S
      4: temp := artm a15 / a16
      8: BReturn 0
END Node
After test:
      0: if (s_22_16_t3.S = product) then 4 else 5 unkn 5
      4: BReturn 1
      5: BReturn 0
time          0% fanout          0 input          0 rows
RDF_QUAD_POGS        13 rows(s_22_16_t4.S)
 P =  #/product  ,  O = k_q_s_22_16_t3.S
time          0% fanout          0 input          0 rows
Stage 3
time          0% fanout          0 input          0 rows
RDF_QUAD         1 rows(s_22_16_t5.S, s_22_16_t5.O)
 inlined  P =  #/vendor  ,  S = q_s_22_16_t4.S
time          0% fanout          0 input          0 rows
RDF_QUAD      0.31 rows()
 P =  #/price  ,  S = k_s_22_16_t5.S ,  O < k_q_temp O < k_q_temp
time          0% fanout          0 input          0 rows
Stage 4
time          0% fanout          0 input          0 rows
Sort (set_no, q_s_22_16_t5.O) -> (inc)

}
}
wait time   0.00059% of exec real time, fanout         0
QF {
time   9.1e-08% fanout          0 input          0 rows
Stage 1
time   2.5e-07% fanout          0 input        384 rows
group by read node
(gb_set_no, s_22_16_t5.O, aggregate)

After code:
      0: vendor :=  := artm s_22_16_t5.O
      4: belowAvg :=  := artm aggregate
      8: BReturn 0
time          0% fanout          0 input          0 rows
Subquery Select(vendor, belowAvg)
```

```
time          0% fanout          0 input          0 rows
Stage 2
time          0% fanout          0 input          0 rows
Hash source 61            1 rows(q_vendor) -> (offerCount)
time          0% fanout          0 input          0 rows

Precode:
      0: vendor :=  := artm vendor
      4: BReturn 0
END Node
After test:
      0: if (vendor = vendor) then 4 else 5 unkn 5
      4: BReturn 1
      5: BReturn 0
time          0% fanout          0 input          0 rows

Precode:
      0: QNode {
time          0% fanout          0 input          0 rows
dpipe
vendor -> __RO2SQ -> __ro2sq
}

      2: _cvt := Call _cvt (<constant>, belowAvg)
      7: temp := artm _cvt / offerCount
      11: BReturn 0
Sort (temp, __ro2sq)
time   9.2e-10% fanout          0 input          0 rows
 ssa iterator
time          0% fanout          0 input          0 rows
top order by read (__ro2sq, temp)
time          0% fanout          0 input          0 rows
 qf select node output: (temp, __ro2sq, set_no)
}
}
time   6.4e-07% fanout          0 input          1 rows
  cl fref read
 output: (temp, __ro2sq, set_no)
order:  2   0  desc  1

After code:
      0: vendor :=  := artm __ro2sq
      4: cheapExpensiveRatio :=  := artm temp
      8: BReturn 0
time          0% fanout          0 input          0 rows
```

```
Subquery Select(vendor, cheapExpensiveRatio)
}

After code:
      0: QNode {
time          0% fanout          0 input          0 rows
dpipe
cheapExpensiveRatio -> __RO2SQ -> cheapExpensiveRatio
vendor -> __RO2SQ -> vendor
}

      2: BReturn 0
time          0% fanout          0 input          0 rows
Select (vendor, cheapExpensiveRatio)
}
```

Appendix B – Virtuoso Configuration Files

Following are the configuration files for the system. The files are identical for each process, except that even numbered processes have different affinity settings in virtuoso.global.ini.

```
$ cat cluster.ini
[Cluster]
Master                  = Host1
ThisHost                = Host1

[ELASTIC]
Slices = 16
Segment1 = 1024, cl1/cl1_1.db = q1, cl1/cl1_2.db = q2, cl1/cl1_3.db = q3, cl1/cl1_4.db = q4

[gast749@stones04 01]$ cat cluster.global.ini
[Cluster]
Threads = 200
Master = Host1
ReqBatchSize = 10000
BatchesPerRPC = 4
BatchBufferBytes = 20000
LocalOnly = 2
MaxKeepAlivesMissed = 2000
Host1                   = 192.168.64.220:22101
Host2                   = 192.168.64.220:22102
Host3                   = 192.168.64.221:22103
Host4                   = 192.168.64.221:22104
Host5                   = 192.168.64.222:22105
Host6                   = 192.168.64.222:22106
```

```
Host7                   = 192.168.64.223:22107
Host8                   = 192.168.64.223:22108
Host9                   = 192.168.64.224:22109
Host10                   = 192.168.64.224:22110
Host11                   = 192.168.64.225:22111
Host12                   = 192.168.64.225:22112
Host13                   = 192.168.64.219:22113
Host14                   = 192.168.64.219:22114
Host15                   = 192.168.64.228:22115
Host16                   = 192.168.64.228:22116
Host17                   = 192.168.64.217:22117
Host18                   = 192.168.64.217:22118
Host19                   = 192.168.64.230:22119
Host20                   = 192.168.64.230:22120
Host21                   = 192.168.64.231:22121
Host22                   = 192.168.64.231:22122
Host23                   = 192.168.64.232:22123
Host24                   = 192.168.64.232:22124
[gast749@stones04 01]$ cat virtuoso.ini
; virtuoso.ini
;
; Configuration file for the OpenLink Virtuoso VDBMS Server
;
;
; Database setup
;
[Database]
DatabaseFile     = virtuoso.db
TransactionFile = virtuoso.trx
ErrorLogFile     = virtuoso.log
ErrorLogLevel    = 7
Syslog           = 0
TempStorage      = TempDatabase
FileExtend       = 200
Striping         = 0

[TempDatabase]
DatabaseFile     = virtuoso.tdb
TransactionFile = virtuoso.ttr
FileExtend       = 200

;
; Server parameters
;
[Parameters]
```

```
ServerPort                 = 12201
ServerThreads              = 100
CheckpointSyncMode         = 2
CheckpointInterval         = 0
NumberOfBuffers            = 10000000
MaxDirtyBuffers            = 8000000
MaxCheckpointRemap         = 5000000
DefaultIsolation           = 2
UnremapQuota               = 0
AtomicDive                 = 1
PrefixResultNames          = 0
CaseMode                   = 2
DisableMtWrite             = 0
;MinAutoCheckpointSize = 4000000
;CheckpointAuditTrail  = 1
DirsAllowed                = /
PLDebug                    = 0
TestCoverage               = cov.xml
;Charset=ISO-8859-1
ResourcesCleanupInterval   = 1
ThreadCleanupInterval      = 1
TransactionAfterImageLimit = 1500000000
FDsPerFile                 = 4
;StopCompilerWhenXOverRunTime = 1
MaxMemPoolSize             = 40000000
AdjustVectorSize           = 1
ThreadsPerQuery            = 16
AsyncQueueMaxThreads       = 24
IndexTreeMaps              = 64

[VDB]
VDBDisconnectTimeout = 1000
ArrayOptimization    = 2
NumArrayParameters   = 10

[Client]
SQL_QUERY_TIMEOUT   = 0
SQL_TXN_TIMEOUT     = 0
SQL_ROWSET_SIZE     = 10
SQL_PREFETCH_BYTES = 12000

[AutoRepair]
BadParentLinks = 0
BadDTP         = 0
```

```
[Replication]
ServerName   = VIRTUOSO_CLUSTER
ServerEnable = 1

[HTTPServer]
ServerPort                   = 8892
ServerRoot                   = ../vsp
ServerThreads                = 40
MaxKeepAlives                = 10
KeepAliveTimeout             = 10
MaxCachedProxyConnections    = 10
ProxyConnectionCacheTimeout = 10
DavRoot                      = DAV
HTTPLogFile                  = logs/http28082014.log

[!URIQA]
DefaultHost = lod.openlinksw.com

[SPARQL]
;ExternalQuerySource = 1
;ExternalXsltSource = 1
ResultSetMaxRows      = 100000
;DefaultGraph = http://localhost:8892/dataspace
;MaxQueryCostEstimationTime = 120 ; in seconds
MaxQueryExecutionTime = 0     ; in seconds
ExecutionTimeout      = 0     ; in seconds
LabelInferenceName    = facets
ImmutableGraphs       = inference-graphs, *
ShortenLongURIs       = 1
;EnablePstats = 0
[gast749@stones04 01]$ cat virtuoso.global.ini
[Parameters]
MaxQueryMem      = 30G
MaxVectorSize    = 500000
Affinity         = 1-7 16-23
ListenerAffinity = 0
HashJoinSpace    = 10G

[Flags]
hash_join_enable          = 2
dfg_max_empty_mores       = 10000
dfg_empty_more_pause_msec = 5
qp_thread_min_usec        = 100
cl_dfg_batch_bytes        = 20000000
enable_high_card_part     = 1
```

```
enable_vec_reuse          = 1
mp_local_rc_sz            = 0
dbf_explain_level         = 0
enable_feed_other_dfg     = 0
;enable_cll_nb_read = 1
dbf_no_sample_timeout     = 1
enable_subscore           = 0
;enable_cl_compress = 1
enable_conn_fifo          = 0
enable_mt_transact        = 1
enable_mt_txn             = 0
iri_seqs_used             = 3
enable_small_int_part     = 1
dbf_max_itc_samples       = 1
enable_dt_hash            = 1
[gast749@stones04 01]$
```

The even numbered server affinity lines are:

```
[gast749@stones04 01]$ fgrep Affinity ../02/virtuoso.global.ini
Affinity         = 9-15 24-31
ListenerAffinity = 8
```

References

[BSBM] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/

[BIBM] https://sourceforge.net/projects/bibm/

[BSBM 500B Results] https://dl.dropboxusercontent.com/u/106414290/bsbm-500B-triple-results.zip

[Virtuoso Colum Store] http://sites.computer.org/debull/A12mar/vicol.pdf

[Virtuoso TPC-H Blog Series] http://www.openlinksw.com/weblog/oerling/?id=1739

[LOD2 Final: The 500 Giga-triples Blog] http://www.openlinksw.com/weblog/oerling/?id=1805