Virtuoso, a Hybrid RDBMS/Graph Column Store

Orri Erling[*]

Abstract

We discuss applying column store techniques to both graph (RDF) and relational data for mixed workloads ranging from lookup to analytics in the context of the OpenLink Virtuoso DBMS. In so doing, we need to obtain the excellent memory efficiency, locality and bulk read throughput that are the hallmark of column stores while retaining low-latency random reads and updates, under serializable isolation.

1 Introduction

OpenLink Virtuoso was first developed as a row-wise transaction oriented RDBMS with SQL federation. It was subsequently re-targeted as an RDF graph store with built-in SPARQL and inference [1]. Lastly, the product has been revised to take advantage of column-wise compressed storage and vectored execution. This article discusses the design choices met in applying column store techniques under the twin requirements of performing well on the unpredictable, semi-structured RDF data and more typical relational BI workloads.

The largest Virtuoso applications are in the RDF space, with terabytes of RDF triples that usually do not fit in RAM. The excellent space efficiency of column-wise compression was the greatest incentive for the column store transition. Additionally, this makes Virtuoso an option for relational analytics also. Finally, combining a schema-less data model with analytics performance is attractive for data integration in places with high schema volatility. Virtuoso has a shared nothing cluster capability for scale-out. This is mostly used for large RDF deployments. The cluster capability is largely independent of the column-store aspect but is mentioned here because this has influenced some of the column store design choices.

Column Store. Virtuoso implements a clustered index scheme for both row and column-wise tables. The table is simply the index on its primary key with the dependent part following the key on the index leaf. Secondary indices refer to the primary key by including the necessary key parts. In this the design is similar to MS SQL Server or Sybase. The column store is thus based on sorted multi-column column-wise compressed projections. In this, Virtuoso resembles Vertica [2]. Any index of a table may either be represented row-wise or column-wise. In the column-wise case, we have a row-wise sparse index top, identical to the index tree for a row-wise index, except that at the leaf, instead of the column values themselves is an array of page numbers containing the column-wise compressed values for a few thousand rows. The rows stored under a leaf row of the sparse index are called a segment. Data compression may radically differ from column to column, so that in some cases multiple segments may fit in a single page and in some cases a single segment may take several pages.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]OpenLink Software, 10 Burlington Mall Road, Suite 265, Burlington, MA 01803, U.S.A. *oerling@openlinksw.com*

The index tree is managed as a B tree, thus when inserts come in, a segment may split and if all the segments post split no longer fit on the row-wise leaf page this will split, possibly splitting the tree up to the root.

This splitting may result in half full segments and index leaf pages. These are periodically re-compressed as a background task, resulting in 90+% full pages with still room for small inserts.

This is different from most column stores, where a delta structure is kept and then periodically merged into the base data [3]. Virtuoso also uses an uncommonly small page size for a column store, only 8K, as for the row store. This results in convenient coexistence of row-wise and column wise structures in the same buffer pool and in always having a predictable, short latency for a random insert.

While the workloads are typically bulk load followed by mostly read, using the column store for a general purpose RDF store also requires fast value based lookups and random inserts. Large deployments are cluster based, which additionally requires having a convenient value based partitioning key. Thus, Virtuoso has no concept of a table-wide row number, not even a logical one. The identifier of a row is the value based key, which in turn may be partitioned on any column. Different indices of the same table may be partitioned on different columns and may conveniently reside on different nodes of a cluster since there is no physical reference between them. A sequential row number is not desirable as a partition key since we wish to ensure that rows of different tables that share an application level partition key predictably fall in the same partition.

The column compression applied to the data is entirely tuned by the data itself, without any DBA intervention. The need to serve as an RDF store for unpredictable, run time typed data makes this an actual necessity, while also being a desirable feature for a RDBMS use case.

The compression formats include: *(i)* Run length for long stretches of repeating values. *(ii)* Array of run length plus delta for tightly ascending sequences with duplicates. *(iii)* Bitmap for tightly ascending sequences without duplicates. *(iv)* Value sequence 11, 12, 14, 15 would get 11 as base value and 0b1011 indicating increments of one, two and one. *(v)* Array of 2-byte deltas from an integer base, e.g., mid cardinality columns like dates. *(vi)* Dictionary for anything that is not in order but has a long stretch with under 256 distinct values. *(vii)* Array of fixed or variable length values. If of variable length, values may be of heterogeneous types and there is a delta notation to compress away a value that differs from a previous value only in the last byte.

Type-specific index lookup, insert and delete operations are implemented for each compression format.

Updates and Transactions. Virtuoso supports row-level locking with isolation up to serializable with both row and column-wise structures. A read committed query does not block for rows with uncommitted data but rather shows the pre-image. Underneath the row level lock on the row-wise leaf is an array of row locks for the column-wise represented rows in the segment. These hold the pre-image for uncommitted updated columns, while the updated value is written into the primary column.

RDF updates are always a combination of delete plus insert since there are no dependent columns, all parts of a triple make up the key. Update in place with a pre-image is needed for the RDB case.

Checking for locks does not involve any value-based comparisons. Locks are entirely positional and are moved along in the case of inserts or deletes or splits of the segment they fall in. By far the most common use case is a query on a segment with no locks, in which case all the transaction logic may be bypassed.

In the case of large reads that need repeatable semantics, row-level locks are escalated to a page lock on the row-wise leaf page, under which there are typically some hundreds of thousands of rows.

Vectored Execution. Column stores generally have a vectored execution engine that performs query operators on a large number of tuples at a time, since the tuple at a time latency is longer than with a row store.

Vectored execution can also improve row store performance, as we noticed when remodeling the entire Virtuoso engine to always running vectored. The benefits of eliminating interpretation overhead and improved cache locality, improved utilization of CPU memory throughput, all apply to row stores equally. When processing one tuple at a time, the overhead of vectoring is small, under 10% for a single row lookup in a large table while up to 200% improvement is seen on row-wise index lookups or inserts as soon as there is any locality.

Consider a pipeline of joins, where each step can change the cardinality of the result as well as add columns to the result. At the end we have a set of tuples but their values are stored in multiple arrays that are not aligned. For this one must keep a mapping indicating the row of input that produced each row of output for every stage in the pipeline. Using these, one may reconstruct whole rows without needing to copy data at each step. This tuple reconstruction is fast as it is nearly always done on a large number of rows, optimizing memory bandwidth.

Virtuoso vectors are typically long, from 10000 to 1000000 values in a batch of the execution pipeline. Shorter vectors, as in Vectorwise [4], are just as useful for CPU optimization, besides fitting a vector in the first level of cache is a plus. Since Virtuoso uses vectoring also for speeding up index lookup, having a longer vector of values to fetch increases the density of hits in the index, thus directly improving efficiency: Every time the next value to fetch is on the same segment or same row-wise leaf page, we can skip all but the last stage of the search. This naturally requires the key values to be sorted but the gain far outweighs the cost as shown later. An index lookup keeps track of the hit density it meets at run time. If the density is low, the lookup can request a longer vector to be sent in the next batch. This adaptive vector sizing speeds up large queries by up to a factor of 2 while imposing no overhead on small ones. Another reason for favoring large vector sizes is the use of vectored execution for overcoming latency in a cluster.

RDF Specifics. RDF requires supporting columns typed at run time and the addition of a distinct type for the URI and the typed literal. A typed literal is a string, XML fragment or scalar with optional type and language tags. We do not wish to encode all these in a single dictionary table since at least for numbers and dates we wish to have the natural collation of the type in the index and having to look up numbers from a dictionary would make arithmetic near unfeasible.

Virtuoso provides an 'any' type and allows its use as a key. In practice, values of the same type will end up next to each other, leading to typed compression formats without per-value typing overhead. Numbers can be an exception since integers, floats, doubles and decimals may be mixed in consecutive places in an index.

2 Experimental Analysis

We compare bulk load speed, space consumption and performance with different mixes of random and sequential access with the TPC H data set and DBpedia [5] for RDF. The test system is a dual Xeon 5520 with 144GB of RAM. All times are in seconds and all queries run from memory.

Relational Microbenchmarks. We use the 100G TPC-H data set as the base for comparing rows and columns for relational data. In both row and column-wise cases the table is sorted on the primary key and there are value based indices on `l_partkey`, `o_custkey`, `c_nationkey` and the `ps_suppkey`, `ps_partkey` pair. Data sizes are given as counts of allocated 8K pages.

The bulk load used 16 concurrent streams, and is RAM-resident, except for the IO-bound checkpoint. Using column storage, bulk load takes 3904s + 513 checkpoint, the footprint is 8904046 pages; wheres in row storage it takes 4878 + 450 checkpoint, and the footprint is 13687952 pages. We would have expected the row store to outperform columns for sequential insert. This is not so however because the inserts are almost always tightly ascending and the column-wise compression is more efficient than the row-wise.

Since the TPC-H query load does not reference all columns, only 4933052 pages = 38.5MB are read to memory. The row store does not have this advantage. The times for Q1, a linear scan of lineitem are 6.1s for columns and 66.7 for rows. TPC-H generally favors table scans and hash joins. As we had good value-based random access as a special design goal, let us look at the index lookup/hash join tradeoff. The query is:

```
select sum (l_extendedprice * (1 - l_discount)) from lineitem, part
where l_partkey = p_partkey and p_size < ?
```

If the condition on part is selective, this is better done as a scan of part followed by an index lookup on `l_partkey` followed by a lookup on the lineitem primary key. Otherwise this is better done as a hash join

Table 1: Sum of revenue for all parts smaller than a given size, execution times in seconds

	Column store			Row store		
% of part selecteds	index	vectored hash join	invisible hash join	index	vectored hash join	invisible hash join
1.99997%	4.751	7.318	7.046	5.065	39.972	21.605
3.99464%	6.300	9.263	8.635	9.614	40.985	24.165
5.99636%	7.595	9.754	10.310	14.029	42.175	27.402
7.99547%	10.620	11.293	11.947	18.803	42.28	30.325
9.99741%	12.080	10.944	11.646	22.597	42.399	31.570
11.99590%	14.494	11.054	12.741	26.763	42.473	32.998
13.99850%	16.181	11.417	12.630	31.119	41.486	34.738

where part is the build side and lineitem the probe. In the hash join case there are two further variants, using a non-vectored invisible join [6] and a vectored hash join. The difference between the two is that in the event of the invisible join, the cardinality restricting hash join is evaluated before materializing the `l_extendedprice` and `l_discount` columns. For a hash table not fitting in CPU cache, we expect the vectored hash join to be better since it will miss the cache on many consecutive buckets concurrently even though it does extra work materializing prices and discounts.

We see that the index plan keeps up with the hash surprisingly long, only after selecting over 10% is the lineitem scan with hash filtering clearly better. In this case, the index plan runs with automatic vector size, i.e. it begins with a default vector size of 10000. It then finds that there is no locality in accessing lineitem, since `l_partkey` is not correlated to the primary key. It then switches the vector size to the maximum value of 2000000. In this case the second batch of keys is still spread over the whole table but now selects one of 300 (600M lineitems / 2M vector) rows, thus becoming again relatively local.

We note that the invisible hash at the high selectivity point is slightly better than the vectored hash join with early materialization. The better memory throughput of the vectored hash join starts winning as the hash table gets larger, compensating for the cost of early materialization.

It may be argued that the Virtuoso index implementation is better optimized than the hash join. The hash join used here is a cuckoo hash with a special case for integer keys with no dependent part. Profiling shows over 85% of the time spent in the hash join. For a hash lookups that mostly find no match, Bloom filters could be added and a bucket chained hash would probably perform better as every bukcet would have an overflow list.

The experiment was also repeated with a row-wise database. Here, the indexed plan is best but is in all cases slower than the column store indexed plan. The invisible hash is better than vectored hash with early materialization due to the high cost of materializing the columns. To show a situation where rows perform better than columns, we make a stored procedure that picks random orderkeys and retrieves all columns of lineitems of the order. 3/4ths of the random orderkeys have no lineitem and the remainder have 1-7 lineitems. We retrieve 1 million orderkeys, single threaded, without any vectoring; this takes 16.447s for columns, 8.799s for rows. The column store's overhead is in making 16+x more page number to buffer cache translations, up to a few per column, as a single segment of a long column like `l_comment` spans many pages. Column stores traditionally shine with queries accessing large fractions of the data. We clearly see that the penalty for random access need not be high and can be compensated by having more of the data fit in memory.

RDF Microbenchmarks. We use DBpedia 3.7, a semi-structured collection of 256.5 million RDF triples extracted from Wikipedia. The RDF is stored as quads of subject, predicate, object, graph (SPOG). In both cases there are two covering indices, one on PSOG and the other on POGS, plus the following distinct projections: OP, SP and GS. Dictionary tables mapping ids of URI's and literals to the external form are not counted in the size figures. The row-wise representation compresses repeating key values and uses a bitmap for the last key part in POGS, GS and SP, thus it is well compressed as row stores go, over 3x compared to uncompressed.

Bulk load on 8 concurrent streams with column storage takes: 945s, resulting in in 479607 pages, down to 302423 pages after automatic re-compression. With row storage, it takes 946s resulting in 1021470 pages. The Column store wins hands down, with equal bulk load rate and under 1/3 of the memory footprint.

Table 2: The breakdown of space utilization in the column store by compression type

Compression type	run length delta	array	run length	bitmap	dictionary	2-byte delta
% of values	15.3	10.8	47.4	8.9	8.7	8.6
% of bytes	4.6	51.6	0.4	2.4	15.4	25.5

Next we measure index lookup performance by checking that the two covering indices contain the same data. All the times are in seconds of real time, with up to 16 threads in use (one per core thread).

```
select count (*) from rdf_quad a table option (index rdf_quad)
where not exists (
  select 1 from rdf_quad b table option (loop, index rdf_quad_pogs)
  where a.g = b.g and a.p = b.p and a.o = b.o and a.s = b.s );
```

Vector size	10K	1M
columns	103	48
rows	110	100
hash join columns	60	63
hash join rows	94	94

Store and vector size	rnd	seq	same seg	same pg	same par
column store, 10K vector	256.5M	256.5M	199.1M	55.24M	2.11M
column store, 1M vector	256.5M	256.5M	255M	1.472M	32.29K
row store, 10K vector	256.6M	256.6M	0	165.2M	0
row store, 1M vector	256.6M	256.6M	0	237.7M	0

Vectoring introduces locality to the otherwise random index access pattern. The locality is expressed in the *rnd* and *seq* columns of the rightmost table, with the count of rows accessed at random and sequentially. The next 3 numbers respectively show how many times the next match was in the same segment, in a different segment on the same row-wise leaf page and lastly on a sibling page of the row-wise leaf page.

We next join an index to itself by random index lookup. In the case of vectoring, this effectively becomes a merge join, comparing the sorted key values in the index to the sorted lookup values. POGS merge join to itself took 21s for columns and 103s for rows.

We notice that the column store is generally more sensitive to vector size. The average segment size in the POGS index, the one accessed by index lookup, is 7593 rows, i.e. there is one row-wise index leaf entry for so many rows in the index. From the locality stats we see that 74% of the time the next row was found in the same segment. Thus the join accessed one row every 1546 rows on the average. With this density of access it outperformed the row store with 103s against 110s. As the hit density increased, the column store more than doubled its throughput with a 1 million vector size. When every row of the index was selected by the join, the throughput doubled again, with 21 against 48s.

As expected, the hash join, which anyhow does not exhibit locality of access is not sensitive to vector size. The fact that column store outperforms rows with 60s vs 94s is surprising. Profiling demonstrates this to be due to the greater memory bandwidth needed in both build and probe for reading the table. We note that extracting all values of a column is specially efficient in a column store. The temporary space utilization of the build side of the hash join was 10GB. For a large join, we can roughly say that vectored index lookup outperforms hash when more than 2% of the rows get picked in each vectored lookup. The hash join implementation is a vectored cuckoo hash where each thread does explicit prefetches for consecutive lookups so as to have multiple cache misses outstanding at all times.

Space Utilization. The database has 14 columns, 4+4 for the covering indices and 3x2 for the distinct projections. All together these contain 2524447546 values and occupy 2137 MB.

We notice that run length compression predominates since the leading key parts of the indices have either low cardinality (P and G) or highly skewed value distributions with long runs of the same value (O). Since there are only two graphs in this database, the G column almost always get run length compression. The worst compression is for the third key of PSOG and POSG, specially when storing wiki links (random connections).

Analytical Queries. Here we compare rows and columns with a pseudo-realistic workload, the Berlin SPARQL Benchmark business intelligence mix [7]. The scale is 1 billion triples. Vector size is set to 1000000.

We also repeat the experiment with the column store in a cluster configuration on the same test machine (dual Xeon 5520). The database is divided in 32 partitions, with indices partitioned on S or O, whichever is first in key order. The cluster consists of 4 server processes each managing 8 partitions.

	Bulk load + checkpoint	1 user run, QMpH	8 user run, QMpH
single server, columns	2508s + 318s	7653	17092
single server, rows	2372s + 495s	3225	7138
cluster, columns	3005s + 230s	7716	13411

Data fits in memory in both bulk load and queries. The query numbers are in queries per hour multiplied by the scale (10). Space does not allow us to further analyze these results but we note that for large joins the column store delivers a gain that is roughly on par with what we saw in the query comparing the two covering indices of the quad table. With cluster, the bulk load takes slightly longer than single server due to more copying of data and the extra partitioning step. The single server number is about even with the single server configuration. What is lost in partitioning and message passing is gained in having more runnable threads. The multi-user number shows a 21% drop in throughput as opposed to the single server configuration. Both configurations are at full CPU through the test but the cluster has extra partitioning and message passing to do. We note that almost all joins in the workload are cross partition, i.e. the consecutive steps do not have an equality on the partitioning key. In future work we will use the TPC-H in its original SQL formulation and a 1:1 SPARQL translation for the same analysis. This will allow us to also quantify the performance cost of using a schema-less data model as opposed to SQL.

3 Conclusions

Adopting column store techniques in Virtuoso is amply justified by the present use and future development of the product. The results confirm all our initial expectations when embarking on the column store/vectoring rewrite. This work owes much to the excellent work and publications on other column stores, specially Vectorwise and Vertica. Future work may add more compression formats, specifically for strings and automation in cluster and cloud deployments, for example automatically commissioning new cloud servers based on data size and demand. While this is not column store specific, the column store with its performance and efficiency gains is a necessary basis for a competitive multi-model data warehouse like Virtuoso.

References

[1] Erling O., Mikhailov I. Virtuoso: RDF Support in a Native RDBMS. Semantic Web Information Management 2009, pp. 501-519

[2] Vertica Systems. Vertica Database for Hadoop and MapReduce.
http://www.vertica.com/MapReduce

[3] Heman S., Nes N. J., Zukowski M., Boncz P. Positional Delta Trees To Reconcile Updates With Read-Optimized Data Storage. CWI Technical Report INS-E0801, CWI, August 2008.

[4] Actian Corporation. Vectorwise Technical White Paper.
http://www.actian.com/whitepapers/download-the-vectorwise-technical-white-paper-today

[5] Soren Auer, Jens Lehmann. What have Innsbruck and Leipzig in common? Extracting Semantics from Wiki Content. In Franconi et al. (eds), Proceedings of European Semantic Web Conference (ESWC07), LNCS 4519, pp. 503517, Springer, 2007.

[6] Abadi, Daniel J. Query Execution in Column-Oriented Database Systems. MIT PhD Dissertation, February, 2008.
http://cs-www.cs.yale.edu/homes/dna/papers/abadiphd.pdf

[7] Bizer C., Schultz A. Berlin SPARQL Benchmark (BSBM) Specification - V2.0
http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/20080912/