Bulletin of the Technical Committee on

Data Engineering

March 2012 Vol. 35 No. 1

IEEE Computer Society

Letters

Special Issue on Column Store Systems

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TC on Data Engineering web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

IEEE Computer Society News

We in the database community perhaps are not fully aware of how our “prosperity" depends upon the surrounding technical society infrastructure and regulations. But this is surely true for us as it impacts on major conferences, whether they be VLDB, SIGMOD, or ICDE, the conference sponsored by the IEEE Computer Society. Our relationship to the Computer Society also impacts how the Technical Committee on Data Engineering (TCDE)can operate and what it might accomplish. So I will use part of this letter to provide news, some relatively good, some not so good, about our relationship with the Computer Society.

As I informed you earlier, a subcommittee of TC chairs, who constitute the membership of the Technical Activities Committee (TAC) within the Computer Society had suggested that TC chairs elect the TAC chair, who will represent the TC's at “higher level" boards of the Computer Society. Sadly, that proposal has been rejected. The Computer Society is NOT an agile and flexible organization, and it struggles. This is a case of a self-inflicted and unnecessary wound, risking the disaffection of TC chairs and conference organizers.

Not all news is bad, however, and there is some evidence that the Computer Society door to change may be slightly ajar. A recent change in how the Computer Society sponsors and profits from conferences has shifted some money to conference organizers and hence to the benefit of conference attendees. Further, there are proposals, not yet approved, to permit TC's to carry over, in a limited way, a fund balance from year to year. This is something they are not currently permitted to do. So change comes slowly. I will keep you posted.

The Current Issue

One of the exciting and unique characteristics of the database technical area is the flow of ideas between research and industry. This is due, in large part, to the huge role that databases play in the market and in society in general. Because money is at stake, technical work can move very rapidly from the idea stage to the “shipping" stage.

It would be hard to find an area where this is more true than with column store technology. More than 20 years ago, a small company, Expressway Technologies, introduced a column-based database product. Around the same time, research work on column-based databases began with the MonetDB project. The pioneering work was followed, after some delay, by an explosion of work, both in research and in industry. Major vendors and many researchers have explored the area and the impact has been enormous.

Peter Boncz, who with Martin Kersten, was among the early research pioneers in column-based databases is the editor for the current issue. What I particularly like about the issue is how it demonstrates that what may have once been considered a “fringe" technology has blossomed into a thriving industry. This issue includes articles on most commercial column-based databases, and gives both a great snapshot of current industrial practice and clues to where the industry is heading. I had hoped that Peter would “do" a column-store" issue when I appointed him an editor. Hence, I find the current issue very gratifying, and I want to thank Peter for the fine job he has done in organizing the issue.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

In the past five years, columnar storage technologies have gained momentum in the analytical marketplace. This resulted from (i) a number of new successful entrants in the database market with products based on columnar technology, as well as (ii) established vendors introducing columnar products or even integrating columnar technology deep inside their existing products.

In this issue, you find a number of papers that describe systems that fit one or even both of these catagories: Infobright and MonetDB as pure column stores; and Virtuoso, Vectorwise, IBM's Blink based products, Microsoft SQLserver, and SAP HANA (through its P*TIME subsystem) are products that combine row- and column-technology; this also holds for the research systems HYRISE and Hyper. Given that column store systems have been on the market now for a few years, the commercial system papers also describe customer reactions, so we can assess how column stores are holding up in practice. These papers contain concise system descriptions that hopefully will inspire the reader, and also form an entry point for further reading.

Previous to these five years of major commercial adoption, there has been a long research track in column stores, most visibly in the MonetDB system, which is completing its second decade of research history. MonetDB has proven to be a powerful platform for new interesting systems research such as on-the-fly indexing ("cracking") and adaptive result caching ("recycling"). But not only at CWI does columnar technology inspire new academic research in new topics, evidenced by descriptions of HYRISE and Hyper. Both latter papers address, in different ways, the issue of combining row and columnar formats, among other topics.

While the success of specialized columnar systems seemed to underline the end of the "one system fits all" paradigm as proclaimed by Michael Stonebraker, this issue clearly shows that this is still a debatable proposition. Both the Microsoft SQLserver as well as the Openlink Virtuoso systems show that tight integration of columnar technology in row-based systems is both possible and desirable. Both systems are deeply integrated, as they do not stop at only superficially adopting columnar storage, but also vectorized large parts of their execution systems to reap its query processing benefits. Though Virtuoso probably is the lesser well-known system (database practitioners working with RDF will surely know it), it is an especially interesting case, as the upcoming version 7 described and microbenchmarked here integrates columnar and vectorized technology fully throughout data storage, query execution and event MPP cluster infrastructure. In this system, rows and columns fully co-exist, which enables interesting apples-to-apples comparisons.

The idea that one would have to choose between row- or columnar-systems is also contradicted by the work in HYRISE and Vectorwise, where even during execution tuples are represented partly columnar and partly row-wise – this also true in Blink due to packing of columns in machine words. The Hyper paper confronts "one system fits all" head on, arguing for the contrary. It shows in proof-of-concept that by machine-supported forms of transaction isolation, and efficient query compilation techniques, one system can both compete or exceed the best specialized alternatives in OLTP and OLAP workloads.

In all, columnar systems research leads to interesting questions and will continue to influence future developments in database architecture.

Let me hereby thank all authors for their efforts, and express my hope that you will enjoy reading the resulting material.

Peter Boncz
CWI
Amsterdam

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 (ESWCŠ07), LNCS 4519, pp. 503Ű517, 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/

Business Analytics in (a) Blink

Ronald Barber[†] Peter Bendel[‡] Marco Czech[•*] Oliver Draese[‡] Frederick Ho[#] Namik Hrle[‡]
Stratos Idreos[§*] Min-Soo Kim[◁*] Oliver Koeth[‡] Jae-Gil Lee[◇*] Tianchao Tim Li[‡] Guy Lohman[†]
Konstantinos Morfonios[△*] Rene Mueller[†] Keshava Murthy[#] Ippokratis Pandis[†] Lin Qiao[○*]
Vijayshankar Raman[†] Richard Sidle[†] Knut Stolze[‡] Sandor Szabo[‡]

[†]IBM Almaden Research Center [‡]IBM Germany [•]SIX Group, Ltd. [#]IBM
[§]CWI Amsterdam [◁]DGIST, Korea [◇]KAIST, Korea [△]Oracle [○]LinkedIn

Abstract

The Blink project's ambitious goal is to answer all Business Intelligence (BI) queries in mere seconds, regardless of the database size, with an extremely low total cost of ownership. Blink is a new DBMS aimed primarily at read-mostly BI query processing that exploits scale-out of commodity multi-core processors and cheap DRAM to retain a (copy of a) data mart completely in main memory. Additionally, it exploits proprietary compression technology and cache-conscious algorithms that reduce memory bandwidth consumption and allow most SQL query processing to be performed on the compressed data. Blink always scans (portions of) the data mart in parallel on all nodes, without using any indexes or materialized views, and without any query optimizer to choose among them. The Blink technology has thus far been incorporated into two IBM accelerator products generally available since March 2011. We are now working on the next generation of Blink, which will significantly expand the "sweet spot" of the Blink technology to much larger, disk-based warehouses and allow Blink to "own" the data, rather than copies of it.

1 Introduction

Business Intelligence (BI) queries typically reference *data marts* that have a "star"- or "snowflake"-shaped schema, i.e., with a huge *fact table* having billions of rows, and a number of smaller *dimension tables*, each representing some aspect of the fact rows (e.g., geography, product, or time). Traditional DBMSs, and even some column stores, rely upon a *performance layer* of indexes [4] and/or materialized views (or "projections" [1]) to speed up complex BI queries. However, determining this layer requires knowing the query workload in advance, anathema to the *ad hoc* nature of BI queries, and increases the variance in response time between those queries that the performance layer anticipates and those it does not.

The Blink project's ambitious goal is to answer *all* BI queries in mere seconds, regardless of the database size, without having to define any performance layer. Blink is a database system optimized for read-mostly BI queries. Blink was built from the ground up to exploit the scale-out made possible by modern commodity

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]Work done while the author was at IBM

multi-core processors and inexpensive DRAM main memories, together with meticulous, hardware-conscious engineering of query processing. What differentiates Blink is that its proprietary dictionary encoding of data and its cache-conscious algorithms combine to minimize consumption of memory bandwidth, to perform most SQL query processing on the encoded data, and to enable single-instruction multiple-data (SIMD) operations on vectors of those compressed values.

This paper provides an overview of the Blink technology, describes the IBM products that have incorporated the first generation of that technology, gives one example of the performance gains on customer data of those products, briefly introduces the research behind the second generation of Blink, and finally compares Blink to related systems before concluding.

2 Blink technology

This section summarizes the essential aspects of Blink technology – how it compresses and stores data at load time, and how it processes that data at query time.

Compression and Storage. In Blink, every column is compressed by encoding its values with a fixed-length, order-preserving dictionary code. Blink uses a proprietary compression method called *frequency partitioning* [12] to horizontally partition the domain underlying each column, based upon the frequency with which values occur in that column at load time. A separate dictionary is created for each partition. Since each dictionary need only represent the values of its partition, each dictionary can use shorter column codes. More precisely, partition P of column C can encode all its values in a dictionary $D_{C,P}$ using only $\lceil \log_2 |D_{C,P}| \rceil$ bits. In [12] we show that frequency partitioning approaches the efficiency of Huffman coding as the number of partitions grow, but has the advantage of generating fixed-length codes. Furthermore, encoded values are assigned in each dictionary in an order-preserving way, so that range and equality predicates can be applied directly to the encoded values.

Rows of tables are then loaded in encoded and packed form, as described below, horizontally partitioned according to the partitioning of their values for each column. Since each column has fixed width within a partition, each row in that partition therefore has the same fixed format.

While relational DBMSs since System R have laid out tables in row-major order for maximal update efficiency, many recent read-optimized systems use a column-major order, so that each query need only scan the columns it references (e.g., Sybase IQ [10], MonetDB [4], C-Store [1]). We argue that both row-major and column-major layouts are suboptimal: the former because queries have to scan unreferenced columns, and the latter because encoded columns must be padded to word boundaries for efficient access. Instead, Blink vertically partitions the bit-aligned, encoded columns of each horizontal partition into a family of fixed-size, byte-aligned *banks* of 8, 16, 32, or 64 bits, to allow efficient ALU operations. Since the width of each column varies from one horizontal partition to the next, so too may this assignment of columns to banks. The bin-packing algorithm that performs this assignment seeks to minimize padding, not bank accesses, and is therefore insensitive to how often columns in each bank are referenced together in queries. We then lay out the table in storage in *bank-major* order – each *tuplet* of a bank contains the encoded values of one row for the columns assigned to that bank. Large, fixed-size blocks are formed with all the banks for a range of RIDs, in a PAX-like format [2].

Banked layouts permit a size trade-off. Wide banks are more compact, because we can pack columns together with less wasted space. When a query references many of the columns in the bank, this compactness results in efficient utilization of memory bandwidth, much as in a row-major layout. On the other hand, narrow banks are beneficial for queries that reference only a few columns within each bank. In the extreme case, when each column is placed in a separate bank, we get a column-major layout, padded to the nearest machine word size. Blink's banked layouts exploit SIMD instruction sets on modern processors to allow a single ALU operation to operate on as many tuplets in a bank as can be packed into a 128-bit register. Narrow banks rely on SIMD processing for greater efficiency.

Overview of Query Processing. Query processing in Blink is both simpler and more complex than in traditional DBMSs. It's simpler because Blink has only one way to access tables – scans (recall Blink has no indexes or materialized views) – and always performs joins and grouping using hash tables in a pre-specified order. So it needs no optimizer to choose among alternative access paths or plans. On the other hand, it's more complex because the query must be compiled for the different column widths of each (horizontal) partition. To limit this overhead, we set an upper limit on the number of partitions at load time.

Query processing first breaks an SQL query into a succession of scans called *single-table queries (STQs)*. Although we originally anticipated denormalizing dimension tables (at load time) to completely avoid joins at run time in Blink [12], customer data proved that the redundancy introduced by denormalizing would more than offset our excellent compression [3]. We therefore abandoned this assumption and implemented (hash) joins. Joins are performed by first performing an STQ that scans each dimension table and builds a hash table of the rows that survive any *local predicates* to that table. Then the fact-table STQ scans and applies predicates local to the fact table, probes the hash table for each of its dimensions to apply its *join predicate(s)*, and finally hashes the grouping columns and performs the aggregates for that group. Queries against snowflake queries repeat this plan recursively, "outside-in", starting with the outer-most dimension tables. Between these outer-most tables and the central fact table, the intermediate "hybrid" dimension tables will act both as a "fact" and as a "dimension" table in the same STQ. That is, each hybrid STQ: (a) probes all tables that are "dimension" tables relative to it; and then (b) builds a hash table for the subsequent join to the table that is "fact" relative to it.

Blink compiles each STQ from value space to code space and then runs the STQ directly on the compressed data without having to access the dictionary. This compilation from value space to code space has to be done separately for each partition, because the dictionaries are different for each partition. However, a benefit of this dictionary-specific compilation is that all rows in an entire partition may be eliminated at compile time if the value(s) for a local predicate cannot be found in that partition's dictionary. For example, a predicate `StoreNo = 47` cannot be true for any row in any partition not having `47` in its dictionary for `StoreNo`, which will be the case for all but one partition.

Blink executes an STQ by assigning blocks of a partition to threads, each running on a core. Since partitions containing more frequent values will, by construction, have more blocks, threads that finish their assigned partition early will help out heavier-loaded threads by "stealing" some unprocessed blocks from the bigger partitions to level the load automatically. In each block, a Blink thread processes all the rows in three stages, with the set of RIDS of qualifying tuples passed from one to the next as a bit vector: (a) fastpath predicate evaluation: applies conjuncts and disjuncts of range and short IN-list selection predicates; (b) residual predicate evaluation: applies remaining predicates, including join predicates, with a general-purpose expression interpreter; (c) grouping and aggregation: each thread maintains its own hash table for grouping and aggregation, to avoid locking or latching, and the resulting aggregates are combined at the end to produce the final answer.

Most modern ALUs support operations on 128-bit registers. By packing the codes for multiple instances of multiple columns into a single 128-bit unit and applying predicates on these multi-tuplets simultaneously, Blink can evaluate the column comparisons on N column values that have been compressed to B bits each using only $N/\lfloor 128/B \rfloor$ operations, as compared to N operations using the standard method.

Wherever possible, operators process compressed codes. Because we have constructed a dictionary of values, we can convert complex predicates on column values, such as LIKE predicates, into IN-lists in code space by evaluating the predicate on each element of the dictionary. Due to order-preserving dictionary coding, all the standard predicates ($=, <, >, \leq, \geq$) map to integer comparisons between codes, irrespective of data type. As a result, even predicates containing arbitrary conjunctions and disjunctions of atomic predicates can be evaluated using register-wide mask and compare instructions provided by processors, as described in [8]. A Decode operator decompresses data only when necessary, e.g., when character or numeric expressions must be calculated.

Blink batch-processes a large gulp of rows, called a *stride*, at each stage of query processing, as in [11, 4], to exploit ILP and SIMD. Short-circuiting for non-qualifying tuples only occurs between stages, to minimize the high cost of mispredicting branches that short-circuiting causes. Run-time operators produce a bit vector

of predicate results, with one bit for each input RID. The final step of the Residual stage uses this bit vector to produce the final RID list of qualifying tuples, which is then passed to grouping and aggregation.

3 Accelerator Products Based on Blink

Although prototyped initially as a stand-alone main-memory DBMS [12], the Blink technology has been incorporated into two IBM products thus far as a parallelized, main-memory accelerator to a standard, disk-based host DBMS. Once the user has defined the mart of interest to be accelerated, a bulk loader extracts a copy of its tables automatically from the data warehouse, pipes the data to the accelerator, analyzes it, and compresses it for storage on the nodes of the accelerator, which can be either blades in a commodity blade center or segments of a single server. The assignment of data to individual nodes is not controllable by the user. Fact tables are arbitrarily partitioned among the nodes, and dimensions are replicated to each. Once the data has been loaded in this way, SQL queries coming into the host DBMS that reference that data will be routed automatically by the host optimizer to Blink, where it will be executed on the accelerator's compressed data, rather than on the host DBMS. The SQL query is first parsed and semantically checked for errors by the host DBMS before being sent to the accelerator in a pre-digested subset of SQL, and results are returned to the host DBMS, and thence to the user. Note that users need not make any changes to their SQL queries to get the router to route the SQL query to the accelerator; the query simply has to reference a subset of the data that has been loaded. Below, we briefly describe each of these IBM products in a bit more detail.

IBM Smart Analytics Optimizer (ISAO). The *IBM Smart Analytics Optimizer for DB2 for z/OS V1.1 (ISAO)* is an appliance running Blink, called the *zEnterprise Blade eXtension (zBX)*, which is network-attached to the *zEnterprise* mainframe containing a standard, disk-based data warehouse managed by DB2 for z/OS. The user doesn't really see this accelerator, as there are no externalized interfaces to it. The zBX is a modified Blade Center H containing up to 14 blades, and ISAO can accommodate multiple such zBXs for scale-out. Blades are (soft-) designated as either *coordinators* or *workers*. Coordinator blades receive queries from DB2 and broadcast them to the workers, then receive partial answers from the workers, merge the results, and return them to DB2. There are always at least 2 active coordinator blades to avoid a single point of failure, plus one held in reserve that can take over for any worker blade that might fail by simply loading its memory image from a disk storage system that only backs up each worker node.

Each blade contains two quad-core Nehalem chips and 48 GB of real DRAM. Only worker blades dedicate up to 32 GB of DRAM for storing base data; the rest is working memory used for storing the system code and intermediate results. A fully-populated zBX having 11 worker blades would therefore be capable of storing 11 * 32 GB = 352 GB of compressed data, or at least 1 TB of raw (pre-load) data, conservatively estimating Blink compression at 3x (though much higher compression has been measured, depending upon the data).

Informix Warehouse Accelerator (IWA). Blink technology is also available via a software offering, called the *Informix Ultimate Warehouse Edition (IUWE)*. The Blink engine, named *Informix Warehouse Accelerator (IWA)*, is packaged as a main-memory accelerator to the *Informix database server*. The Informix server, when bundled with IWA as IUWE, is available on Linux, IBM AIX© , HP-UX, and Oracle Solaris. When running on Linux, the database server and IWA can be installed on the same or different computers. When both Informix and IWA are running on the same machine, the coordinator and worker nodes simply become processes on the same machine that communicate via loopback. Informix and IWA can be on distinct SMP hardware, with IWA running both the coordinator and worker processes on the same hardware. IWA can also be deployed on a blade server supporting up to 80 cores and 6 TB of DRAM, each blade supporting up to 4 sockets and 640 GB of DRAM. The IUWE software was packaged flexibly enough to run directly on hardware or in a virtualized/cloud environment. Each database server can have zero, one, or more IWAs attached to it.

Query #	1	2	3	4	5	6	7
Informix 11.50	22 mins	3 mins	3 mins 40 secs	>30 mins	2 mins	30 mins	>45 mins
IWA + Informix 11.70	4 secs	2 secs	2 secs	4 secs	2 secs	2 secs	2 secs
Speedup	330x	90x	110x	>450x	60x	900x	>1350x

Table 3: Execution times for Skechers queries with and without on Informix Warehouse Accelerator

4 Performance

Blink is all about query performance. Space constraints limit us to a brief summary of one customer's initial experience; for more, see [3]. Skechers, a U.S. shoe retailer, uses Informix for their inventory and sales data warehouse, which has fact tables containing more than a billion rows. Queries took anywhere from a few minutes to 45 minutes to run on their production server running Informix 11.50. During the IWA Beta program, Skechers tested IWA with the same data and workload. The queries took just 2 to 4 seconds on the Informix Warehouse Accelerator, a 60x to 1400x speed-up, as shown in Table 3. Note IWA's low variance.

5 The Next Generation of Blink

Leveraging our experience with the initial products based upon the first generation of Blink, we are now prototyping the next generation of Blink, to widen the "sweet spot" provided by Blink. First of all, we are relaxing the requirement that all data fit in main memory and allowing Blink tables to be stored on disk, while retaining our main-memory legacy of high-performance cache-conscious algorithms and multi-core exploitation. This re-introduces disk I/O concerns, arguing for Blink to be a more pure column store by favoring "thin" banks, i.e., allocating a single column to each block by default. Since each partition of each column may be represented by a different number of bits, each block may therefore contain a different number of tuplets. Stitching together all the referenced columns for a particular row now becomes quite a challenge, as they're not all in the same block, or even in the same relatively-numbered block for each column. And over-sized intermediate results must be able to spill to disk. Secondly, since disk storage is persistent, Blink tables can now "own" the base data, rather than a copy of the data. This obviates the problem of keeping multiple copies in sync, but raises performance issues for "point" queries to Blink tables. Will we have to backtrack on indexes? And we still need a mechanism for updates, inserts, and deletes, both in batches and as a "trickle-feed", including ways to evolve the dictionaries by which Blink encodes data. Thirdly, we are rethinking our algorithms and data structures for both joins and grouping to minimize cache misses and still avoid any locking or latching between threads, to ensure good scaling as the number of cores increase exponentially. Fourthly, interrupts for disk I/Os once again create opportunities for context switching among multiple concurrent queries, necessitating careful allocation of resources among these queries for maximum efficiency. These and other issues are the focus of our current research.

6 Related Work

Numerous new systems in industry and academia target fast processing of BI queries, but unfortunately most of the commercial systems have very limited or no documentation in the refereed literature. Compared to existing systems such as Vertica [1], which is based upon the C-store academic prototype [13], and VectorWise [7], which is derived from MonetDB [5] and X100 [4], Blink is closer to the vectorized storage and processing model pioneered by VectorWise. C-store creates *projections*, a redundant copy of the base data sorted on a leading

[1] Vertica, an HP Company: http://www.vertica.com

column and resembling an index that can exploit run-length encoding to reduce storage and processing overhead. Blink introduces a more advanced order-preserving compression scheme, frequency partitioning, which allows it to achieve a good balance between reducing size while still maintaining fixed-width arrays and performing most database operations on the encoded data. HYRISE [6] and HyPer [9] are both recent academic main-memory DBMSs for mixed (OLTP and BI) workloads, but their real-world feasibility remains to be proven. The main feature of Hyper is that it exploits the ability of modern hardware and operating systems to create virtual memory snapshots by duplicating pages on demand when BI queries conflict with OLTP queries. This allows BI queries to see a very recent snapshot of the data, while OLTP queries can continue in parallel. The main contributions of HYRISE seem to be an offline analysis tool for deciding the proper grouping of columns and physical layout to optimize performance for a given mixed workload, which may be valuable for organizing banks in Blink, and a detailed cost model for each operator in terms of cache misses. SAP's HANA[2] and its predecessor Netweaver Business Warehouse Accelerator (BWA) are main-memory DBMSs that resemble Blink by using dictionary encoding to pack multiple values in a register and exploit SIMD operations to do decompression and (local) predicate evaluation. However, unlike Blink, the values of only one column are packed without padding to align to register boundaries, so that values may span register boundaries, creating challenges for processing values on those boundaries and extracting results using a clever series of complicated bit shifts [14].

7 Conclusions

Radical changes in hardware necessitate radical changes in software architecture. Blink is such a radically novel architecture–a main-memory, special-purpose accelerator for SQL querying of BI data marts that exploits these hardware trends. It also exploits proprietary order-preserving compression techniques that permit SQL query processing on the compressed values and simultaneous evaluation of multiple predicates on multiple columns using cache-conscious algorithms. As a result, Blink can process queries in simple scans that achieve near-uniform execution times, thus speeding up the most problematic queries the most, without requiring expensive indexes, materialized views, or tuning. Completely obviating the need for a tunable "performance layer" is the best way to lower administration costs, and hence the total cost of ownership and time to value.

References

[1] D. Abadi et al. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[3] R. Barber et al. Blink: Not your father's database! In *BIRTE*, 2011.

[4] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[5] P. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51, 2008.

[6] M. Grund et al. HYRISE—A main memory hybrid storage engine. *PVLDB*, 4, 2010.

[7] D. Inkster, M. Zukowski, and P. Boncz. Integration of VectorWise with Ingres. *SIGMOD Rec.*, 40, 2011.

[8] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1, 2008.

[9] A. Kemper and T. Neumann. HyPer – a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[10] R. MacNicol and B. French. Sybase IQ Multiplex - Designed for analytics. In *VLDB*, 2004.

[11] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, 2001.

[12] V. Raman et al. Constant-time query processing. In *ICDE*, 2008.

[13] M. Stonebraker et al. C-store: a column-oriented DBMS. In *VLDB*, 2005.

[14] T. Willhalm et al. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2, 2009.

[2] http://www.intel.com/en_US/Assets/PDF/whitepaper/mc_sap_wp.pdf

Columnar Storage in SQL Server 2012

Per-Ake Larson
palarson@microsoft.com

Eric N. Hanson
ehans@microsoft.com

Susan L. Price
susanp@microsoft.com

Abstract

SQL Server 2012 introduces a new index type called a column store index and new query operators that efficiently process batches of rows at a time. These two features together greatly improve the performance of typical data warehouse queries, in some cases by two orders of magnitude. This paper outlines the design of column store indexes and batch-mode processing and summarizes the key benefits this technology provides to customers. It also highlights some early customer experiences and feedback and briefly discusses future enhancements for column store indexes.

1 Introduction

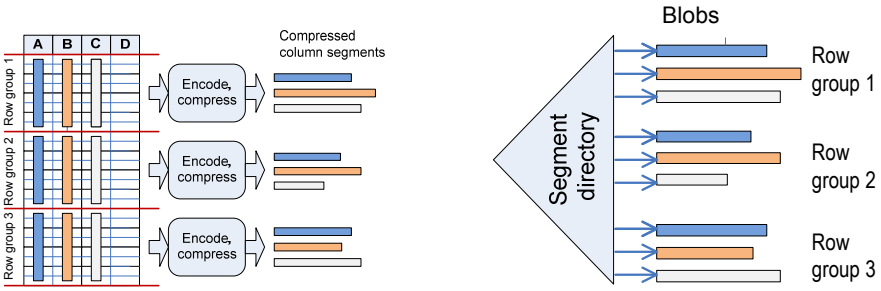
SQL Server is a general-purpose database system that traditionally stores data in row format. To improve performance on data warehousing queries, SQL Server 2012 adds columnar storage and efficient batch-at-a-time processing to the system. Columnar storage is exposed as a new index type: a column store index. In other words, in SQL Server 2012 an index can be stored either row-wise in a B-tree or column-wise in a column store index. SQL Server column store indexes are "pure" column stores, not a hybrid, because different columns are stored on entirely separate pages. This improves I/O performance and makes more efficient use of memory.

Column store indexes are fully integrated into the system. To improve performance of typical data warehousing queries, all a user needs to do is build a column store index on the fact tables in the data warehouse. It may also be beneficial to build column store indexes on extremely large dimension tables (say more than 10 million rows). After that, queries can be submitted unchanged and the optimizer automatically decides whether or not to use a column store index exactly as it does for other indexes. Some queries will see significant performance gains - even as much as 100X - while others will show smaller or no gains.

The idea of storing data column-wise goes back to the seventies. In 1975 Hoffer and Severance [3] investigated how to decompose records into smaller subrecords and storing them in separate files. A 1985 paper by Copeland and Khoshafian [2] proposed fully decomposed storage where each column is stored in a separate file. The development of MonetDB, a column store pioneer, began in the early nineties at CWI [4]. Sybase launched Sybase IQ, the first commercial columnar database system, in 1996. More recent entrants include Vertica, Exasol, Paraccel, InfoBright and SAND.

SQL Server is the first general-purpose database system to fully integrate column-wise storage and processing into the system. Actian Vectorwise Analytical Database (from Actian Corporation) is a pure column store and engine embedded within the Ingres DBMS but it does not appear to interoperate with the row-oriented Ingres engine, that is, a query cannot access data both in the Vectorwise column store and the standard Ingres row store

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering



(a) Convert rows to column segment (b) Store segments in blobs

Figure 1: Creation of a column store index

[5]. Greenplum Database (from EMC Greenplum) and Aster Database (from Teradata's Aster Data) began as pure row stores but have now added column store capabilities. However, it is unclear how deeply column-wise processing has been integrated into the database engines. Teradata has announced support for columnar storage in Teradata 14 but, at the time of writing, this has not yet been released.

2 Technical Overview

SQL Server has long supported two storage organization: heaps (unordered) and B-trees (ordered), both row-oriented. A table or a materialized view always has a primary storage structure and may have additional secondary indexes. The primary structure can be either a heap or a B-tree; secondary indexes are always B-trees. SQL Server also supports filtered indexes, that is, an index that stores only rows that satisfy a selection predicate.

Column store capability is exposed as a new index type: a column store index. A column store index stores its data column-wise in compressed form and is designed for fast scans. The initial implementation has restrictions but, in principle, any index can be stored as a column store index, be it primary or secondary, filtered or non-filtered, on a base table or on a view. A column store index can, in principle, support all the same index operations (scans, lookups, updates, and so on) that heaps and B-tree indexes support. All index types can be made functionally equivalent but they do differ in how efficiently various operations can be performed.

2.1 Column-wise Index Storage

Figure 1 illustrates how a column store index is created and stored. The first step is to convert a set of rows to column segments. The rows are first divided into row groups of about one million rows each. Each row group is then encoded and compressed independently, producing one compressed column segment for each column included in the index. Figure 1(a) shows a table divided into three row groups where three of the four columns are included in the index. This yields nine compressed column segments, three segments for each of columns A, B, and C. Further details about encoding and compression can be found in reference [6].

The column segments are then stored using existing SQL Server storage mechanisms as shown in Figure 1(b). Each column segment is stored as a separate blob (LOB). Segment blobs may span multiple disk pages but this is automatically handled by the blob storage mechanisms. A segment directory keeps track of the location of segments so all segments comprising a column can be easily located. The directory contains additional metadata about each segment such as number of rows, size, how data is encoded, and min and max values. Dictionary compression is used for (large) string columns and the resulting dictionaries are stored in separate blobs.

Storing the index in this way has several important benefits. It leverages the existing blob storage and catalog implementation - no new storage mechanisms - and many features are automatically available. Locking, logging, recovery, partitioning, mirroring, replication and other features immediately work for the new index type.

2.2 I/O and Caching

Column segments and dictionaries are brought into memory as needed. They are stored not in the page-oriented buffer pool but in a new cache designed for handling large objects (columns segments, dictionaries). Each object in the cache is stored contiguously and not scattered across discrete pages. This simplifies and speeds up scanning of a column because there are no "page breaks" to worry about.

A segment storing a blob may span multiple disk pages. To improve I/O performance, read-ahead is applied aggressively both within and among segments. In other words, when reading a blob storing a column segment, read-ahead is applied at the page level. A column may consist of multiple segments so read-ahead is also applied at the segment level. Finally, read-ahead is also applied when loading data dictionaries.

2.3 Batch Mode Processing

Standard query processing in SQL Server is based on a row-at-a-time iterator model, that is, a query operator processes one row at a time. To reduce CPU time a new set of query operators were introduced that instead process a batch of rows at a time. They greatly reduce CPU time and cache misses on modern processors when processing a large number of rows.

A batch typically consists of around a thousand rows. As illustrated in Figure 2, each column is stored as a contiguous vector of fixed-sized elements. The "qualifying rows" vector is used to indicate whether a row has been logically deleted from the batch. Row batches can be processed very efficiently. For example, to evaluate a simple filter like $Col1 < 5$, all that's needed is to scan the Col1 vector and, for each element, perform the comparison and set/reset a bit in the "qualifying rows" vector. As convincingly shown by the MonetDB/X100 project [1], this type of simple vector processing is very efficient on modern hardware; it enables loop unrolling and memory prefetching and minimizes cache misses, TLB misses, and branch mispredictions.

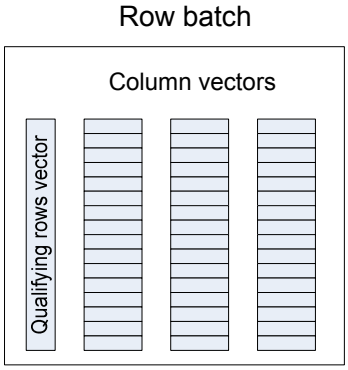


Figure 2: A row batch object

In SQL Server 2012 only a subset of the query operators are supported in batch mode: scan, filter, project, hash (inner) join and (local) hash aggregation. The hash join implementation consists of two operators: a (hash table) build operator and an actual join operator. In the build phase of the join, multiple threads build a shared in-memory hash table in parallel, each thread processing a subset of the build input. Once the table has been built, multiple threads probe the table in parallel, each one processing part of the probe input.

Note that the join inputs are not pre-partitioned among threads and, consequently, there is no risk that data skew may overburden some thread. Any thread can process the next available batch so all threads stay busy until the job has been completed. In fact, data skew actually speeds up the probing phase because it leads to higher cache hit rates.

The reduction in CPU time for hash join is very significant. One test showed that regular row-mode hash join consumed about 600 instructions per row while the batch-mode hash join needed about 85 instructions per row and in the best case (small, dense join domain) was a low as 16 instructions per row. However, the initial version of batch-mode hash join has limitations: the hash table must fit entirely in memory and it supports only inner join. These limitations will be addressed in future releases.

The scan operator scans the required set of columns from a segment and outputs batches of rows. Certain filter predicates and bitmap filters are pushed down into scan operators. (Bitmap filters are created during the build phase of a hash join and propagated down on the probe side.) The scan operator evaluates the predicates directly on the uncompressed data, which can be significantly cheaper and reduces the output from the scan.

The query optimizer decides whether to use batch-mode or row-mode operators. Batch-mode operators are typically used for the data intensive part of the computation, performing initial filtering, projection, joins and aggregation of the inputs. Row-mode operators are typically used on smaller inputs, higher up in the tree to finish the computation, or for operations not yet supported by batch-mode operators.

For queries that process large numbers of rows, the net result of the improvements in query processing is an order of magnitude better performance on large OLAP queries.

3 Customer Experiences

A customer using SQL Server 2012 on a star schema database with a two billion row fact table achieved a remarkable speedup of over 200 times. Their original strategy with a prior version of SQL Server was to run reports at night and cache the results for users to retrieve during the business day. The nightly report generation process took 18 hours, running on a four processor, 16 core machine with 512GB RAM and a good I/O system. After upgrading to SQL Server 2012 and creating a column store index on the fact table, they were able to generate the nightly reports in 5 minutes, on the same hardware. Individual queries that scan the fact table now run in about three seconds each, versus up to 17 minutes each before using column store indexes. They are thus going to give their users the ability to do interactive reporting rather than merely allowing them to retrieve pre-generated reports, clearly adding business value.

A Microsoft IT group that manages a data warehouse containing financial information ran their standard test set of 133 real end-user queries before and after building column store indexes on 23 fact tables. Two-thirds of the queries ran faster - three queries by as much as 50X. The number of queries running longer than ten minutes decreased by 90% (from 33 to 3). The queries that benefited from column store indexes were the queries that are most painful for end users - the longest-running queries. All but one of the queries that did not improve were queries that ran in 40 seconds or less. Only one of the queries with a fast baseline regressed to longer than 40 seconds (to about 42 seconds).

The average compression ratio (size of base table/size of column store index on all columns) was 3.7. Ten of the 23 tables were already using SQL Server ROW compression. Compared to the base table plus existing non-clustered B-tree indexes, the column store index was 11.2X smaller (i.e. total size of row-based structures/size of column store index = 11.2).

Another Microsoft IT group that runs a data warehouse containing current and historical employee data, created column store indexes on their fact tables and larger dimension tables. Average query response time dropped from 220 seconds to 66 seconds. In addition, the team was able to eliminate most of the row-based indexes on the tables with column store indexes. Another early user reported that creating a column store index sped up an ETL query to extract data from a billion-row source table from 15 minutes to 3 minutes.

We have begun formulating best practices based on our early customers' experiences.

1. Use a star schema when possible. We have optimized query execution for star-join style queries.
2. Include all the columns in the column store index. Although it is possible to look up data from missing columns in the base table, such usage is not very efficient and query performance will suffer. The cost of including all columns is very small since only columns touched by the query are retrieved from disk.
3. Ensure that enough memory is available to build the column store index. Creating the index is memory-intensive, especially for wide tables.
4. Create the column store index from a clustered index to get segment elimination for queries with predicates on the clustering key. Although the column store index is not ordered, the clustered index is scanned in index order, naturally resulting in data locality when rows are assigned to row groups.

5. Check query plans for use of batch mode processing and consider tuning queries to get increased benefit of batch mode processing. Because not all operators can be executed in batch mode yet, query rewriting can sometimes yield large performance gains.

Customer feedback has identified several improvements that would be highly beneficial such as extending the repertoire of batch-mode operators, especially outer join, union all, scalar aggregates, and global aggregation. SQL Server includes many performance optimizations for row-mode queries. To provide exceptional performance across a wide range of scenarios, batch-mode processing needs to match many of those optimizations. For example, we currently push some filters into the column store scan; customers would like us to match or exceed the range of filter types that are pushed in row-mode queries. Also, B-tree indexes can be rebuilt online so this option should be provided for column store indexes too.

4 Customer Benefits

The dramatically improved query performance enabled by column store indexes provides significant benefits to customers. Most importantly, it allows a much more interactive and deeper exploration of data which ultimately leads to better insight and more timely and informed decisions.

It has been common practice to use summary aggregates, whether in the form of materialized views or user-defined summary tables, to speed up query response time. The improved performance using column store indexes also means that the number of summary aggregates can be greatly reduced or eliminated completely. Furthermore, OLAP cubes, if they had been used strictly to improve query performance due to their internal summary aggregate maintenance and aggregate navigators, also may be eliminated in favor of SQL reporting directly on the DBMS.

Users typically don't have the patience to wait for more than half a minute for a query result. Hence, reporting applications need to pre-prepare reports if they run more slowly than this. In addition, users will modify their information requests to accommodate the system almost subconsciously to get faster response. For example, they will ask for a summary of activity on one day a month for the last six months instead of a summary of activity every day for the last six months, if that lets the query run in seconds instead of minutes. The column store index mechanism can allow them to get the interactivity they want for the question they really want to ask. In summary, the key customer benefits of column store indexes and more efficient query execution are as follows.

1. Faster data exploration leading to better business decisions.
2. Lower skill and time requirements; designing indexes, materialized views and summary tables requires time and a high level of database expertise.
3. Reduced need to maintain a separate copy of the data on an OLAP server.
4. Faster data ingestion due to reduced index and aggregate maintenance.
5. Reduced need to move to a scale-out solution.
6. Lower disk space, CPU, and power requirements.
7. Overall lower costs.

The use of an OLAP (BI) mechanism still will be warranted if it makes the end-user reporting environment richer, and the business value of that outweighs IT cost savings. We expect the ability of the relational DBMS to run data warehouse queries with interactive response time will drive additional effort into relational OLAP (ROLAP) systems. Such systems can provide rich interactive data exploration environments on top of a single copy of the data warehouse or mart maintained by the relational DBMS.

5 Future Enhancements

For reasons of scope and schedule, the implementation of column store indexes had to be scoped down in the initial release, leaving important features unsupported. These limitations will be addressed in future releases though we cannot at this stage disclose on what schedule.

Direct update and load of a table with a column store index is not supported in the initial release. Even so, data can be added to such a table in a number of ways. If the table is not too large, one can drop its column store index, perform updates, and then rebuild the index. Column store indexes fully support range partitioning. So for large tables, the recommended way is to use partitioning to load a staging table, index it with a column store index, and switch it in as the newest partition. SQL Server 2012 allows up to 15,000 partitions per table so this approach can handle many loads per day, allowing data to be kept current.

In SQL Server, the primary organization of a table can be either a heap or a B-tree. However, a column store index cannot be used as the primary organization in this release; they can only be used for secondary indexes. This restriction may in some cases result in wasted disk space so it will be lifted.

Batch-mode processing is crucial to realize the full performance gains but the initial repertoire of batch-mode operators is limited. Batch-mode hash join will be extend to support all join types (inner, outer, semijoin, anti-semijoin) and additional operators will be implemented in batch-mode.

6 Acknowledgements

Many people have contributed to the success of this project. We thank Ted Kummert and the senior leadership team in SQL Server for their ongoing support and sponsorship of the project. Amir Netz and Cristian Petulescu generously shared their ideas and insights from PowerPivot. Hanuma Kodavalla initiated the project and urged us to show that "elephants can dance". Development was lead by Srikumar Rangarajan with Aleksandras Surna, Artem Oks, and Cipri Clinciu contributing major pieces and Qingqing Zhou monitoring and tracking performance.

References

[1] P. Boncz, M. Zukowski, and N. Nes. MonetDB/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[2] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *SIGMOD Conference*, pages 268–279, 1985.

[3] J. A. Hoffer and D. G. Severance. The use of cluster analysis in physical data base design. In *VLDB*, pages 69–86, 1975.

[4] M. Holsheimer and M. L. Kersten. Architectural support for data mining. In *KDD Workshop*, pages 217–228, 1994.

[5] D. Inkster, M. Zukowski, and P. Boncz. Integration of vectorwise with Ingres. *SIGMOD Record*, 40(3):45–53, 2011.

[6] P.-Å. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL server column store indexes. In *SIGMOD Conference*, pages 1177–1184, 2011.

Vectorwise: Beyond Column Stores

Marcin Zukowski, Actian, Amsterdam, The Netherlands
Peter Boncz, CWI, Amsterdam, The Netherlands

Abstract

This paper tells the story of Vectorwise, a high-performance analytical database system, from multiple perspectives: its history from academic project to commercial product, the evolution of its technical architecture, customer reactions to the product and its future research and development roadmap.
One take-away from this story is that the novelty in Vectorwise is much more than just column-storage: it boasts many query processing innovations in its vectorized execution model, and an adaptive mixed row/column data storage model with indexing support tailored to analytical workloads.
Another one is that there is a long road from research prototype to commercial product, though database research continues to achieve a strong innovative influence on product development.

1 Introduction

The history of Vectorwise goes back to 2003 when a group of researchers from CWI in Amsterdam, known for the MonetDB project [5], invented a new query processing model. This *vectorized query processing* approach became the foundation of the X100 project [6]. In the following years, the project served as a platform for further improvements in query processing [23, 26] and storage [24, 25]. Initial results of the project showed impressive performance improvements both in decision support workloads [6] as well as in other application areas like information retrieval [7]. Since the commercial potential of the X100 technology was apparent, CWI spun-out this project and founded Vectorwise BV as a company in 2008. Vectorwise BV decided to combine the X100 processing and storage components with the mature higher-layer database components and APIs of the Ingres DBMS; a product of Actian Corp. After two years of cooperation between the developer teams, and delivery of the first versions of the integrated product aimed at the analytical database market, Vectorwise was acquired and became a part of Actian Corp.

2 Vectorwise Architecture

The upper layers of the Vectorwise architecture consist of Ingres, providing database administration tools, connectivity APIs, SQL parsing and a cost-based query optimizer based on histogram statistics [13]. The lower layers come from the X100 project, delivering cutting-edge query execution and data storage [21], outlined in Figure 1. The details of the work around combining these two platforms are described in [11]. Here we focus on how the most important feature of Vectorwise, dazzling query execution speed, was carefully preserved and improved from its inception in an academic prototype into a full-fledged database product.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

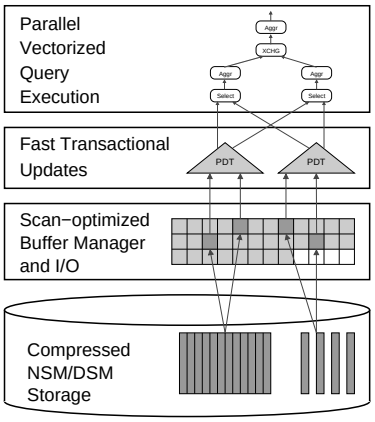


Figure 1: A simplified architecture of the Vectorwise kernel.

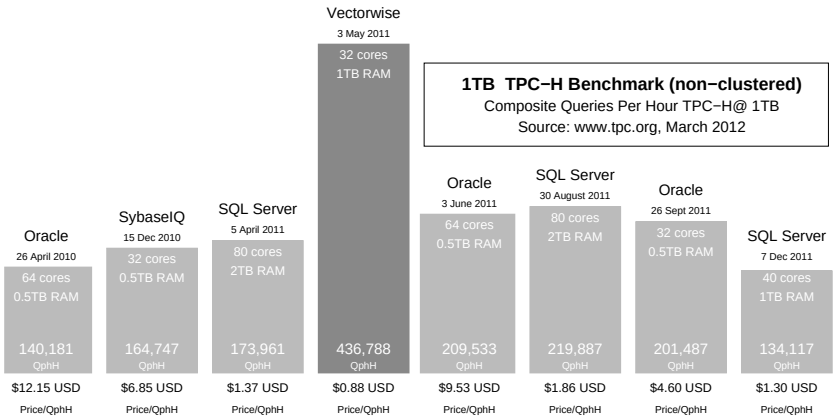


Figure 2: Latest official 1TB TPC-H performance results (non-clustered), in publication order, as of March 18, 2012.

Data Storage. While Vectorwise provides great performance for memory-resident data sets, when deployed on a high-bandwidth IO subsystem (typically locally attached), it also allows efficient analysis of much larger datasets, often allowing processing of disk-resident data with performance close to that of buffered data. To achieve that, a number of techniques are applied.

Vectorwise stores data using a generalized row/column storage based on PAX [2]. A table is stored in multiple PAX partitions, each of which contains a group of columns. This allows providing both "DSM/PAX" (with each column in a separate PAX group) and "NSM/PAX" (with all columns in one PAX group), as well as all options in between. We argue here from the IO perspective: disk blocks containing data from only one column we call DSM/PAX, and containing all columns we call NSM/PAX (this is called PAX in [2]).

The exact grouping of a given table in PAX partitions can be controlled by explicit DDL, but in absence of this, it is self-tuned. One odd-ball example of this are nullable columns. For query processing efficiency, Vectorwise represents nullable columns internally as a column containing values, and a boolean column that indicates whether the value is NULL or not. The motivation behind this is query processing efficiency: testing each tuple for being NULL slows down predicate evaluation, due to hard-to-predict branching CPU instructions and because the presence of NULLs prevents the use of SIMD instructions. Often, however, NULL testing can be skipped and queries can be processed by ignoring the NULL column altogether, so separating the data representations makes sense. The boolean NULL columns are one of the ways PAX partitions are used automatically in Vectorwise: each nullable attribute stores the value and NULL column in the same PAX partition.

Another default PAX policy is to store composite (multi-column) primary keys automatically in the same partition. NSM/PAX storage (i.e. a single PAX partition) is used in case of small tables, where a DSM representation would waste a lot of space using one (almost) empty disk block per column. The overhead of such empty blocks is higher in Vectorwise than in traditional systems, since Vectorwise uses relatively large block-sizes; typically 512KB on magnetic disks (or 32KB for SSDs [4]).

A more advanced PAX grouping algorithm is used in case of very wide tables, to automatically cluster certain columns together in PAX partitions. The reason to limit the amount of PAX groups in which a table is stored lies in the buffer memory needed for table scans. For each PAX group, a scan needs to allocate multiple blocks per physical disk device *for each* PAX partition; which in the widest tables encountered in practice (hundreds of columns) otherwise would lead to many GBs needed for a single scan.

Data on disk is stored in compressed form, using automatically selected compression schemes and automatically tuned parameters. Vectorwise only uses compression schemes that allow very high decompression ratios,

with a cost of only a few cycles per tuple [24]. Thanks to the very low overhead of decompression, it is possible to store data compressed in buffer pool and decompress immediately before query processing. This allows increasing the effective size of the buffer pool further reducing the need for IO. We initially stayed away from compressed execution [1], because it can complicate the query executor and our extremely high decompression speed and fast vectorized execution limits the benefit of compressed execution in many common cases. However, there are some high-benefit cases for compressed execution, such as aggregation on RLE (which can be an order of magnitude less effort) or operations on dictionary-compressed strings (which convert a string comparison into a much cheaper integer or even SIMD comparison), so recently we have worked on simple forms of compression execution [14].

While its strength is in fast scans, Vectorwise allows users in its DDL to declare one *index* per table; this simply means that the physical tuple order becomes determined by the index keys; rather than insertion order. As Vectorwise keeps a single copy of all data, only one such index declaration is allowed per table, such that for users it is similar to a *clustered index*. The main benefit of a clustered index is push-down of range-predicates on the index keys. When an index is declared on a foreign key, treatment is special as the tuple order then gets derived from that of the referenced table, which accelerates foreign key joins between these tables. In the future, we expect to improve this functionality towards multi-dimensional indexing where tables are co-clustered on multiple dimensions such that multiple kinds of selection predicates get accelerated, as well as foreign key joins between co-clustered tables (tables clustered on a common dimension).

Vectorwise automatically keeps so-called *MinMax* indices on all columns. MinMax indices, based on the idea of *small materialized aggregates* [15], store simple metadata about the values in a given range of records, such as Min and Max values. They allow quick elimination of ranges of records during scan operations. MinMax indices are heavily consulted during query rewriting, and are effective in eliminating IO if there are correlations between attribute values and tuple position. In data warehouses, fact table order is often time-related, so date-time columns typically have such correlations. The previously mentioned use of sorted tables (clustered indices) are a direct source of correlation between position and column key values. The rewriter will restrict table scans with range-selections on any position correlated columns, thanks to MinMax indexes.

In the IO and buffer pool layers, Vectorwise focuses on providing optimal performance for *concurrent* scan-intensive queries, typical for analytical workloads. For this purpose, the X100 project originally proposed *cooperative scans*[25], where table scans accept data out-of-order, and an Active Buffer Manager (ABM) determines the order to fetch tuples at runtime, depending on the interest of all concurrent queries, optimizing both the average query latency and throughput. The ABM is a quite complex component that influences the system architecture considerably, such that in the product version of Vectorwise we have switched to a less radical, but still highly effective variant of intelligent data buffering, that to a large extent achieves the same goals [19].

The final element of Vectorwise storage layer are high-performance updates, using a differential update mechanism based on *Positional Delta Trees* (PDT) [10]. A three-level design of PDTs, with one very small PDT, private to the transaction, one shared CPU-cache resident PDT and one potentially large RAM-resident PDT, offers snapshot isolation without slowing down read-only queries in any way. The crucial feature of PDTs as differential structure is the fact that they organize differences by position rather than by key value, and therefore the task of merging in differences during a table scan has virtually no cost, as it does not involve costly key comparisons (nor key scans).

Query Execution. The core technology behind the high processing speeds of Vectorwise is its *vectorized processing model* [6]. It dramatically reduces the interpretation overhead typically found in the tuple-at-a-time processing systems. Additionally, it exposes possibilities of exploiting performance-critical features of modern CPUs like super-scalar execution and SIMD instructions. Finally, thanks to its focus on storing data in the CPU cache, main-memory traffic is reduced which is especially important in modern multi-core systems.

The vectorized execution model was further improved including (i) lazy vectorized expression evaluation, (ii) choosing different function implementations depending on the environment, (iii) pushing *up* selections if this

enables more SIMD predicate evaluation [9], and (iv) NULL-processing optimizations. Further, strict adherence to a vertical (columnar) layout in all operations was dropped and now Vectorwise uses a NSM record layout during the execution for (parts of) tuples where the access pattern makes this more beneficial (mostly in hash tables) [26]. Additionally, Volcano-based parallelism based on exchange operators has been added, allowing Vectorwise to efficiently scale to multiple cores [3]. Many internal operations were further improved, e.g. highly efficient Bloom-filters were applied to speed-up join processing. Another example of such improvements was cooperation with Intel, to use new CPU features such a large TLB pages, and exploit the SSE4.2 instructions for optimizing processing of text data [20].

On the innovation roadmap for query execution are execution on compressed data, and introducing the intelligent use of just-in-time (JIT) compilation of complex predicates – only in those situations where this actually brings benefits over vectorized execution [17, 18]. All these innovations are aimed at bolstering the position of Vectorwise as the query execution engine that gets most work done per CPU cycle. An additional project to introduce MPP cluster capabilities is underway to make Vectorwise available in a scale-out architecture.

3 Vectorwise Experiences

While the history of X100 goes back all the way in 2003, for many years it was only a research prototype offering very low-level interfaces for data storage and processing [22]. After the birth of Vectorwise BV in the summer of 2008 this prototype quickly started evolving into an industry-strength component, combined with the mature Ingres upper layers. Over the course of less than two years the system grew into a full-fledged product, leading to the release of Vectorwise 1.0 in June of 2010. This release of the product was met with highly positive reception thanks to its unparalleled processing performance. Still, it faced a number of challenges typical for young software projects. A number of users missed features available in other products necessary to migrate to a new system. Also, some features initially did not meet expectations, especially around updates. Finally, exposing the product to a large number of new users revealed a sizable number of stability problems.

Over the following 18 months, a number of updates have been released providing a lot of requested features, optimizing many elements of the system and dramatically increasing the system stability. Vectorwise 2.0, released in November 2011, is a solid product providing features like optimized loading, full transactional support, better storage management, parallel execution, temporary tables, major parts of analytical SQL 1999 and disk-spilling operations. It also supports dozens of new functions, both from the SQL standard as well as some used by other systems, making the migrations easier.

To make Vectorwise adoption easier, a lot of effort has been invested to make sure it works well with popular tools. As a result it is now certified with products like Pentaho, Jaspersoft, SAP Business Objects, MicroStrategy, IBM Cognos, Tableau and Yellowfin. Another critical milestone was a release of the fully functional Windows version, making Vectorwise one of the very few analytical DBMS systems for that platform.

To demonstrate the performance and scalability of Vectorwise, a series of record-breaking TPC-H benchmarks were published – as of March 18, 2012, Vectorwise continues to hold the leadership in the single-node 100GB to 1TB results (see Figure 2 for 1TB results).

Customer Reactions. Since the early releases of Vectorwise, users and analysts have been highly impressed with its performance. In a number of cases, the improvement was so large customers believed the system must be doing some sort of query result caching (which it does not). High performance also resulted in customers adopting previously impossible approaches to using their databases. Typical examples include:

- ***Removing indices.*** Combination of efficient in-memory and on-disk scan performance with optimized filtering delivers performance better than previous systems when using indexing.

- ***Normalizing tables.*** Thanks to quick data transformations, large-volume data normalizations are now possible, improving performance and reducing storage volume.

- ***De-normalizing tables.*** In contrast to above, some users find Vectorwise performance with de-normalized tables more than sufficient and prefer that approach due to a simplified schema and loading.

- ***Running on the raw data.*** Many customers now avoid performing expensive data precomputation, as raw-data processing performance is efficient enough.

- ***Full data reloads.*** All above features, combined with high loading performance of Vectorwise, make data loading process faster and simpler. As a result, for many customers full data reload is now a feasible method.

Improved efficiency combined with the possibility to simplify data storage and management, translate directly into reduced need for hardware and human resources.

While high performance delivered by Vectorwise receives a lot of praise, system adoption would not be possible without the technical and organizational contributions of the much more mature Ingres product. On the technical side, users appreciate a wide range of connectivity options, solid SQL support and an ever-growing number of available tools. Even more importantly, users praise the worldwide, 24/7 support capability and active involvement of pre-sales, support and engineering teams with their POC and production systems.

Adoption Challenges. While Vectorwise capabilities provide a great value to many customers, its adoption also faces a number of challenges. Very many issues are related to migrations from older DBMS systems. Off-line data migration is relatively easy, either using manual methods or with support of ETL tools. On-line data transfer from transactional systems poses a bigger challenge, and is discussed below. Migration of different SQL flavors with a plethora of non-standard functions turns out to be a relatively simple, but laborious process. The hardest problem is the application logic stored in DBMSs, e.g. PL/SQL – migration from complex systems using this approach turns out extremely labor intensive.

Vectorwise was originally designed with an idea that the data will be loaded relatively rarely. However, once the users got accustomed to high processing speeds, they requested the ability to use it on much more up-to-date data, including sub-second data loading latency. To address that, Vectorwise quickly improved its incremental-load as well as data-update capabilities, also providing full ACID properties. Additionally, Actian offers Vectorstream: a separate product/service that enables very low-latency data loads into Vectorwise.

Another set of challenges was related to complex database schemas. Scenarios with hundreds of databases, many thousands of tables, and tables with many thousands of attributes stressed the system capabilities, calling for schema reorganizations as well as numerous system improvements.

4 Vectorwise Research Program

Vectorwise has strong roots in academia, and a continued research track is an important part of its technical innovation roadmap. Vectorwise offers academic institutions source code access under a research license. Apart from CWI, licensees include the universities of Ilmenau, Tuebingen and Edinburgh, and the Barcelona Supercomputing Center. Actian Corp. is also sponsoring a number of PhD students at these institutes.

In cooperation with Vrije Universiteit Amsterdam and University of Warsaw, multiple MSc projects have been pursued. Completed topics include: Volcano-style multi-core parallelism in Vectorwise [3], just-in-time compilation of predicates [17, 18], non-intrusive mechanisms for query execution on compressed data [14], materialization and caching of interesting intermediate query results in order to accelerate a query workload by re-using results [16] (i.e. adapting the Recycler [12] idea to pipelined query execution), and buffer management policies that make concurrent queries cooperate rather than fight for IO [19], using an approach that is less system-intrusive than so-called Cooperative Scans [25]. There has also been work on XML storage and processing in Vectorwise [8].

Research activities continue with a number of projects including further improving vectorized execution performance, accelerating processing with multi-dimensional data organization, and improving performance and scalability of Vectorwise in an MPP architecture.

5 Conclusion

This paper gave a short overview of the Vectorwise system, focusing on its origins, technology, user reception and adoption challenges. It shows that achieving really high performance requires much more than just "column storage". Additionally, we discuss other elements required to find adoption in the market: functionality, usability, support capabilities and strong future roadmap.

Less than two years since its first product release, Vectorwise continues to make rapid progress. This includes usability advances such as storage management, backup functionality and rich SQL support encompassing functions, data types and analytical features. Internal and external research activities have created a solid innovation pipeline that will bolster and improve the performance of the product in the future.

References

[1] D. J. Abadi. *Query Execution in Column-Oriented Database Systems*. PhD thesis, MIT, 2008.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, 2001.

[3] K. Anikiej. Multi-core parallelization of vectorized query execution. *MSc thesis, Vrije Universiteit Amsterdam*, 2010.

[4] S. Baumann, G. de Nijs, M. Strobel, and K.-U. Sattler. Flashing databases: expectations and limitations. In *DaMoN*, 2010.

[5] P. Boncz. *Monet: a next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, 2002.

[6] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[7] R. Cornacchia, S. Héman, M. Zukowski, A. P. de Vries, and P. Boncz. Flexible and Efficient IR using Array Databases. *VLDB Journal*, 17(1), 2008.

[8] T. Grust, J. Rittinger, and J. Teubner. Pathfinder: XQuery Off the Relational Shelf. *DEBULL*, 31(4), 2008.

[9] S. Héman, N. Nes, M. Zukowski, and P. Boncz. Vectorized data processing on the cell broadband engine. In *DaMoN*, 2007.

[10] S. Héman, M. Zukowski, N. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *SIGMOD*, 2010.

[11] D. Inkster, M. Zukowski, and P. Boncz. Integration of VectorWise with Ingres. *SIGMOD Record*, 40(3), 2011.

[12] Ivanova, M. and Kersten, M. and Nes, N. and Gonçalves, R. An architecture for recycling intermediates in a column-store. In *SIGMOD*, 2009.

[13] R. Kooi. *The Optimization of Queries in Relational Database Systems*. PhD thesis, Case Western Reserve University, 1980.

[14] A. Luszczak. Simple Solutions for Compressed Execution in Vectorized Database System. *MSc thesis, Vrije Universiteit Amsterdam*, 2011.

[15] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB*, 1998.

[16] F. Nagel. Recycling Intermediate Results in Pipelined Query Evaluation. *MSc thesis, Tuebingen University*, 2010.

[17] J. Sompolski. Just-in-time Compilation in Vectorized Query Execution. *MSc thesis, Vrije Universiteit Amsterdam*, 2011.

[18] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *DaMoN*, 2011.

[19] M. Switakowski. Integrating Cooperative Scans in a column-oriented DBMS. *MSc thesis, Vrije Universiteit Amsterdam*, 2011.

[20] Vectorwise. Ingres/VectorWise Sneak Preview on the Intel Xeon 5500 Platform. Technical report, 2009.

[21] M. Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. PhD thesis, Universiteit van Amsterdam, 2009.

[22] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *DEBULL*, 28(2), 2005.

[23] M. Zukowski, S. Héman, and P. Boncz. Architecture-Conscious Hashing. In *DaMoN*, 2006.

[24] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, 2006.

[25] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, 2007.

[26] M. Zukowski, N. Nes, and P. Boncz. DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing. In *DaMoN*, 2008.

The SAP HANA Database – An Architecture Overview

Franz Färber Norman May Wolfgang Lehner Philipp Große Ingo Müller
Hannes Rauhe Jonathan Dees
SAP AG

Abstract

Requirements of enterprise applications have become much more demanding. They require the computation of complex reports on transactional data while thousands of users may read or update records of the same data. The goal of the SAP HANA database is the integration of transactional and analytical workload within the same database management system. To achieve this, a columnar engine exploits modern hardware (multiple CPU cores, large main memory, and caches), compression of database content, maximum parallelization in the database kernel, and database extensions required by enterprise applications, e.g., specialized data structures for hierarchies or support for domain specific languages. In this paper we highlight the architectural concepts employed in the SAP HANA database. We also report on insights gathered with the SAP HANA database in real-world enterprise application scenarios.

1 Introduction

A holistic view on enterprise data has become a core asset for every organization. Data is entered in batches or by-record via multiple channels, such as enterprise resource planning systems (e.g., SAP ERP), sensors used in production environments, or web-based interfaces. For example in a sales process, orders are created, modified, and deleted. These orders are the basis for production planning and delivery. Hence, during the sales process records are looked up, inserted, and updated. This kind of data processing is typically referred to as Online Transactional Processing (OLTP). OLTP has been the strength of current disk-based and row-oriented database systems.

Upon closer inspection, a supposedly simple sales process exhibits a significant amount of complex analytical processing. For example, checking the availability of a product for delivery as part of a sales process requires aggregating expected sales, expected delivery, and completion of production lots, as well as comparing the resulting inventory with the customer demand. Similarly, a sales organization would be interested in profitability measures for planning based on most recent sales and cost information. This kind of workload is considered Online Analytical Processing (OLAP). Periodical tasks, such as quarter-end closing or customer segmentation, are executed by replicating data into a read-optimized data warehouse. For those types of reporting, column-stores have become more and more popular [3].

Additionally, analytical applications require procedural logic, which cannot be expressed with plain SQL, e.g., clustering sales number of different products or classifying customer behavior. The natural approach is to transfer all the data needed from the database to the application and process it there. Therefore, optimized data

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

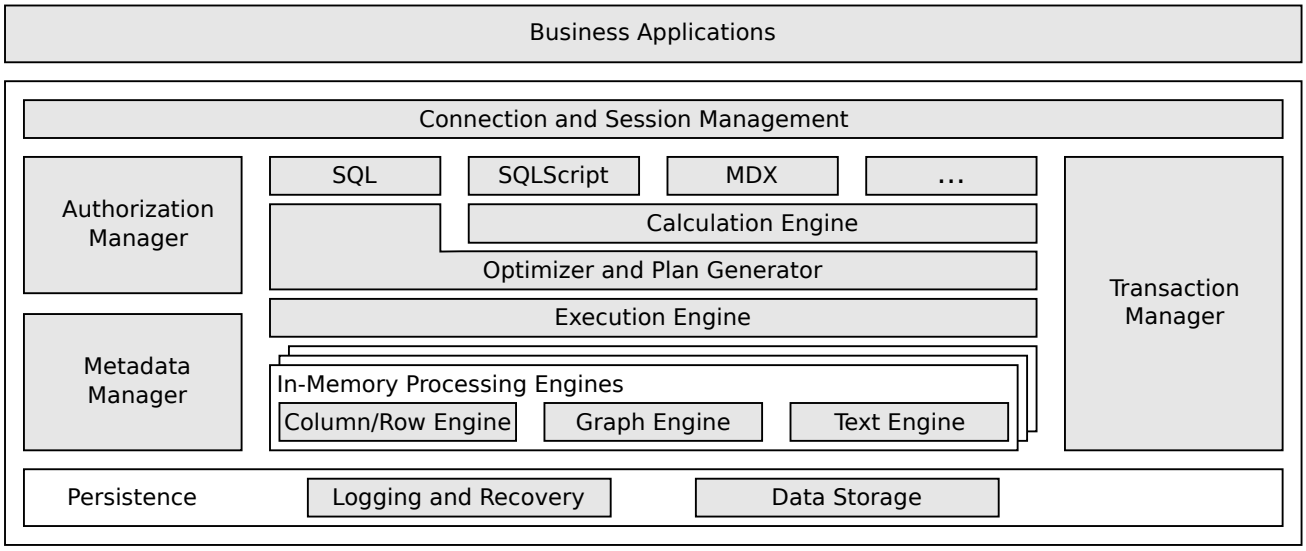


Figure 1: Overview of the SAP HANA DB architecture

structures and metadata cannot be used and intermediate results have to be transferred back to the database if they are needed in the following business process steps.

Ideally, a database shall be able to process all of the above-mentioned workloads and application-specific logic in a single system [13]. This observation sparked the development of the SAP HANA database (SAP HANA DB). Historically, the in-memory columnar storage of the SAP HANA DB is based on the SAP TREX text engine [15] and the SAP BI Accelerator (SAP BIA) [10], which allows for fast processing of OLAP queries. The high-performance in-memory row-store of the SAP HANA DB is derived from P*Time [2] and specially designed to address OLTP workload. The persistence of the SAP HANA DB originated from the proven technology of SAP's MaxDB database system providing logging, recovery, and durable storage. As of today, the SAP HANA DB is commercially available as part of the SAP HANA appliance.

In the next section, we give a brief overview about the architectural components of the SAP HANA DB. Section 3 discusses the ability to execute analytical application-specific logic. In section 4 we outline how the SAP HANA DB accelerates traditional data warehouse workloads. We discuss how we address challenges on transactional workloads in enterprise resource planning systems in section 5 and summarize our work on the SAP HANA DB in section 6.

2 SAP HANA DB Architecture

The general goal of the SAP HANA DB is to provide a main-memory centric data management platform to support pure SQL for traditional applications as well as a more expressive interaction model specialized to the needs of SAP applications [4, 14]. Moreover, the system is designed to provide full transactional behavior in order to support interactive business applications. Finally, the SAP HANA DB is designed with special emphasis on parallelization ranging from thread and core level up to highly distributed setups over multiple machines.

Figure 1 provides an overview of the general SAP HANA DB architecture. The heart of the SAP HANA DB consists of a set of in-memory processing engines. Relational data resides in tables in column or row layout in the combined column and row engine, and can be converted from one layout to the other to allow query expressions with tables in both layouts. Graph data and text data reside in the graph engine and the text engine respectively; more engines are possible due to the extensible architecture [4]. All engines keep all data in main memory as long as there is enough space available. As one of the main distinctive features, all data structures are optimized for cache-efficiency instead of being optimized for organization in traditional disk

blocks. Furthermore, the engines compress the data using a variety of compression schemes. When the limit of available main memory is reached, entire data objects, e.g., tables or partitions, are unloaded from main memory under control of application semantic and reloaded into main memory when it is required again.

From an application perspective, the SAP HANA DB provides multiple interfaces, such as standard SQL for generic data management functionality or more specialized languages as SQLScript (see section 3) and MDX. SQL queries are translated into an execution plan by the plan generator, which is then optimized and executed by the execution engine. Queries from other interfaces are eventually transformed into the same type of execution plan and executed in the same engine, but are first described by a more expressive abstract data flow model in the calculation engine. Irrespective of the external interface, the execution engine can use all processing engines and handles the distribution of the execution over several nodes.

As in traditional database systems, the SAP HANA DB has components to manage the execution of queries. The session manager controls the individual connections between the database layer and the application layer, while the authorization manager governs the user's permissions. The transaction manager implements snapshot isolation or weaker isolation levels – even in a distributed environment. The metadata manager is a repository of data describing the tables and other data structures, and, like the transaction manager, consists of a local and a global part in case of distribution.

While virtually all data is kept in main memory by the processing engines for performance reasons, data has also to be stored by the persistence layer for backup and recovery in case of a system restart after an explicit shutdown or a failure. Updates are logged as required for recovery to the last committed state of the database and entire data objects are persisted into the data storage regularly (during savepoints and merge operations, see section 4).

3 Support for Analytical Applications

A key asset of the SAP HANA DB is its capability to execute business and application logic inside the database kernel. For this purpose the calculation engine provides an abstraction of logical execution plans, called calculation models. For example SQLScript, a declarative and optimizable language for expressing application logic as data flows or using procedural logic, is compiled into calculation models. Following this route, multiple domain-specific languages can be supported as long as a compiler generates the intermediate calculation model representation.

The primitives of a calculation model constitute a logical execution plan consisting of an acyclic data flow graph with nodes representing operators (plan operations) and edges reflecting the data flow (plan data). One class of operators implements the standard relational operators like join and selection. In addition, the SAP HANA DB supports a huge variety of special operators for implementing application-specific components in the database kernel. Almost all these operators are only able to accelerate data processing because they exploit the columnar data layout. By implementing special operators in the calculation engine, several application domains can be supported:

Statistical Algorithms can be attached to calculation models to perform complex statistical computations inside an associated R runtime. This includes different statistical methods, such as linear and nonlinear models, statistical tests, time series analyses, classification, and clustering. At the same time, the calculation model allows to leverage the capabilities to pre- and post-process large data in the database kernel and thereby interweave the statistical algorithms with database operations [5].

Planning provides a set of commonly used and generic planning functions that allow to model and execute complex planning scenarios directly in the database. Planning logic is expressed using data flow operators of the calculation engine. In addition, special operators perform specific planning algorithms such as disaggregation and custom formulas [7, 8].

Other Special Operators provided within the calculation engine include business logic which requires complex operations that are hard to implement efficiently using SQL, e.g. currency conversion. Another example are large hierarchies describing, e.g., relations between employees and associated information in the human capital management of an enterprise. Here, the SAP HANA DB provides application-specific operators to return query results on these hierarchies almost instantaneously exploiting alternative internal data structures.

A specific calculation model or logical execution plan—once submitted to SAP HANA DB (e.g., by using SQLScript)—can be accessed in the same way as a database view, making the calculation model a kind of parameterized view. A query consuming such a view invokes the database plan execution to process an execution plan. This plan is derived from the logical data flow description provided by the calculation model. If the calculation model contains independent data flow paths, the derived execution plan implicitly contains inter-operator parallel execution. This is explored by SQLScript and the domain-specific languages compiled into calculation models.

4 Analytical Query Processing

As generally agreed, column-stores are well suited for analytical queries on massive amounts of data [1]. For high read performance the SAP HANA DB's column-store uses efficient compression schemes in combination with cache-aware and parallel algorithms. Every column is compressed with the help of a sorted dictionary, i.e., each value is mapped to an integer value (the valueID). These valueIDs are further bit-packed and compressed. By resorting the rows in a table, the most beneficial compression (e.g., run-length encoding (RLE), sparse coding, or cluster coding) for the columns of this table can be used [11, 12]. Compressing data does not only allow to keep more data on a single node, but it also allows for faster query processing, e.g., by exploiting the RLE to compute aggregates. Scans are accelerated by excessively using SIMD algorithms working directly on the compressed data [16].

Since single updates are expensive in the described layout, every table has a delta storage, which is designed to balance between high update rates and good read performance. Dictionary compression is used here as well, but the dictionary is stored in a Cache Sensitive B+-Tree (CSB+-Tree). The delta storage is merged periodically into the main data storage. To minimize the period of time where tables are locked, write operations are redirected to a new delta storage when the delta merge process starts. Until it is finished, read operations access new and old delta storage as well as the old main storage [9].

Query execution exploits the increasing number of available execution threads within a node by using intra-operator parallelism. For example, grouping operations scale almost linearly with the number of threads until the CPU is saturated. Additionally the SAP HANA DB also exploits parallelism inside a query execution plan and across many cores and nodes. Large tables can be partitioned using various partitioning criteria. These parts or complete tables can then be assigned to different nodes in the landscape [10]. The execution engine schedules operators in parallel if they can be processed independently and—if possible—executes them on the node that holds the data. In case of changing workload, the partitioning scheme and assignment of tables to nodes can be adapted while the database is available for queries and updates. Joins involving tables distributed across multiple nodes are processed using semi-join reduction [6].

5 Transactional Query Processing

While it is clear that column-stores work well for OLAP workloads, we also argue that there are several reasons to consider the column-store for OLTP workloads, especially in ERP systems [9]:

1) OLTP scenarios can greatly benefit from the compression schemes available in column-stores: In a highly customizable system like SAP ERP, many columns are not used, and thus only contain default values or no values at all. Similarly, some columns typically have a small domain, e.g., status flags. In both cases, compression is very efficient, which can be a decisive advantage for OLTP scenarios: By reducing the memory consumption, the necessary landscape size becomes smaller, so either fewer or smaller nodes are required. Moreover, compression also leads to lower memory bandwidth utilization.

2) Real-world transactional workloads have larger portions of read operations than standard benchmarks like TPC-C define. Hence, the read-optimized column-oriented storage layout may be more appropriate for OLTP workloads than suggested by the benchmarks.

3) Column-stores usually follow the simple "append-only" scheme: When an existing row is updated, the current version is invalidated and a new version is appended. This scheme is simpler than in-place updates as it neither requires reordering nor encoding the values.

4) Column-stores greatly reduce the need for indices: As a matter of fact, the high scan performance of column-stores on modern hardware permits us to have indices only for primary keys, columns with unique constraints and frequent join columns. In all other cases, scan performance is good enough without indices, especially in small tables or small partitions with up to a few hundred thousand rows. The advantages are a significantly simplified physical database design, reduced main memory consumption, and eliminated effort in the maintenance of indices, which in turn speed up the overall query throughput.

Beside these intrinsic advantages of column-stores for OLTP, there are several challenges we have discovered in this context. One challenge emerges directly from the column data layout. Although it allows for a more fine-grained data access pattern, it can result in a significant performance overhead to allocate the memory per columns to handle a large number of columns, for example when constructing a single result row consisting of 100 columns or more. As of now, the SAP HANA DB combines memory allocations for multiple columns into a single one whenever it helps to reduce the performance overhead.

As a major challenge, we see that in ERP applications, a substantial number of updates is performed concurrently. In contrast to data warehouses, where updates are executed in batches, frequent updates of single records constantly add new entities into the delta storage, implying frequent merges from the delta into the main data storage. As the merge operation is a CPU and memory intensive operation, the challenge is to minimize its impact on the requests that are processed concurrently. The SAP HANA faces this problem by careful scheduling and parallelization [9].

6 Summary

In this paper we summarize the principles guiding the design and implementation of the SAP HANA DB. Our analysis shows that in-memory processing using a columnar engine is the most promising approach to cope with analytical and transactional workloads at the same time. The strong support of business application requirements and both kinds of workloads differentiate the SAP HANA DB from other column-stores. After all, the majority of OLAP and OLTP operations are read operations, which benefit from column-wise compression. Moreover, fewer indices are required leading to a simplified physical database design and reduced memory consumption. To further hold the massive amount of data produced by today's enterprise applications in memory, SAP HANA DB allows to distribute it in a cluster of nodes. As a result, the analysis of large data sets is orders of magnitude faster than on conventional database systems.

However, an in-memory column-store supporting distribution raises a number of challenges including the need for partitioning, support for distributed transactions and a carefully designed process for merging updates into the read-optimized storage layout. But these challenges are just the tip of the iceberg because as we consider

data-intensive applications in the cloud, elasticity requirements, multi-tenancy, or scientific computing further challenges have to be addressed.

References

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *Proc. SIGMOD*, pages 967–980, 2008.

[2] S. K. Cha and C. Song. P*TIME: Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload. In *Proc. VLDB*, pages 1033–1044, 2004.

[3] S. Chaudhuri, U. Dayal, and V. Narasayya. An Overview of Business Intelligence Technology. *CACM*, 54(8):88–98, 2011.

[4] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database - Data Management for Modern Business Applications. *SIGMOD Record*, 40(4):45–51, 2011.

[5] P. Große, W. Lehner, T. Weichert, F. Färber, and W.-S. Li. Bridging Two Worlds with RICE – Integrating R into the SAP In-Memory Computing Engine. *Proc. VLDB*, 4(12):1307–1317, 2011.

[6] G. Hill and A. Ross. Reducing outer joins. *VLDB Journal*, 18(3):599–610, 2009.

[7] B. Jäcksch, F. Färber, and W. Lehner. Cherry Picking in Database Languages. In *Proc. IDEAS*, pages 117–122, 2010.

[8] B. Jäcksch, W. Lehner, and F. Färber. A Plan for OLAP. In *Proc. EDBT*, pages 681–686, 2010.

[9] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, P. Dubey, H. Plattner, and A. Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *Proc. VLDB*, 5(1):61–72, 2011.

[10] T. Legler, W. Lehner, and A. Ross. Data Mining with the SAP NetWeaver BI Accelerator. In *Proc. VLDB*, pages 1059–1068, 2006.

[11] C. Lemke, K.-U. Sattler, F. Färber, and A. Zeier. Speeding Up Queries in Column Stores – A Case for Compression. In *Proc. DaWak*, pages 117–129, 2010.

[12] M. Paradies, C. Lemke, H. Plattner, W. Lehner, K.-U. Sattler, A. Zeier, and J. Krüger. How to Juggle Columns: An Entropy-Based Approach for Table Compression. In *Proc. IDEAS*, pages 205–215, 2010.

[13] H. Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *Proc. SIGMOD*, pages 1–2, 2009.

[14] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. Springer, Berlin Heidelberg, 2011.

[15] F. Transier and P. Sanders. Engineering Basic Algorithms of an In-Memory Text Search Engine. *ACM TOIS*, 29(1):2, 2010.

[16] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on- Chip Vector Processing Units. *Proc. VLDB*, 2(1):385–394, 2009.

A Rough-Columnar RDBMS Engine – A Case Study of Correlated Subqueries

Dominik Ślęzak
University of Warsaw &
Infobright Inc., Poland
slezak@infobright.com

Piotr Synak
Infobright Inc., Poland
synak@infobright.com

Janusz Borkowski
Infobright Inc., Poland
januszb@infobright.com

Jakub Wróblewski
Infobright Inc., Poland
jakubw@infobright.com

Graham Toppin
Infobright Inc., Canada
toppin@infobright.com

Abstract

Columnar databases provide a number of benefits with regard to both data storage (e.g.: data compression) and data processing (e.g.: optimized data access, parallelized decompression, lazy materialization of intermediate results). Their characteristics are particularly advantageous for exploratory sessions and ad hoc analytics. The principles of columnar stores can be also combined with a pipelined and iterative processing, leading toward modern analytic engines able to handle large, rapidly growing data sets. In this paper, we show how to further enrich such a framework by employing metadata layers aimed at minimizing the need of data access. In particular, we discuss the current implementation and the future roadmap for correlated subqueries in Infobright's RDBMS, where all above-mentioned architectural features interact with each other in order to improve the query execution.

1 Introduction

One of the main challenges of today's enterprises is to integrate ad hoc reporting and analysis features within applications and services that they deliver. It becomes necessary to support the data mining processes and analytically intensive query workloads, including exploratory SQL statements, which were not anticipated during the database design stages. Moreover, users need to operate on rapidly growing data without delays caused by time consuming model re-optimizations related to evolving data or query patterns.

The discussed RDBMS technology relies on the principles of columnar databases [15], with an additional usage of an automatically generated metadata layer aimed at improving a flow of data accesses while resolving queries [12]. The columnar approach enables to apply (de)compression algorithms that are better adjusted to analytical query execution characteristics [16] and can significantly improve the process of assembling results of ad hoc queries [1]. On the other hand, an appropriate choice and usage of metadata may lead toward a better query plan optimization [7], which can also adaptively influence the process of a query execution [4].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

In our approach, metadata is applied throughout the whole process of the query execution in order to minimize the intensity of data accesses. The content of each column is split onto collections of values of some consecutive rows. Each of the data packs created this way is represented by its basic statistics at the metadata level. This approach led us toward developing algorithms identifying data packs that are sufficient to complete particular query execution stages. The size of metadata grows linearly with the size of data and remains several orders of magnitude smaller. Metadata units corresponding to newly formed data packs are computed independently from (meta)data already stored in a database. Moreover, the whole framework makes it possible to combine the classical benefits of columnar engines with a more iterative data processing [3].

The paper is organized as follows: In Section 2, we explain how metadata describing blocks of rows can assist in speeding up a query execution. We show how to operate with approximations of query results and we refer to correlated subqueries as an illustration. In Section 3, we recall advantages of the columnar databases with regard to analytic queries. We also discuss the compression of data decomposed with respect to both columns and blocks of rows. In Section 4, we introduce the rough-columnar architecture of Infobright's RDBMS[1]. We show how the principles of the columnar databases and rough operations can complement each other. In Section 5, we discuss the current implementation and the roadmap for further improvements of correlated subqueries. We also provide more details about their usefulness in practice. Section 6 concludes the paper.

2 Rough Operations on Granulated Data

In this section we discuss an idea of using granulated tables – specific metadata descriptions of data tables in a relational model. Rows of a granulated table correspond to some partition blocks of the original data rows. Columns of a granulated table (further called rough columns) store statistics (further called rough values) describing the original data columns within particular blocks of rows. The rough values may contain information such as min/max values (interpreted specifically for different data types), a sum of values (or, e.g., a total length of string values), a number of nulls et cetera. They can be employed, e.g., as a support of a cardinality estimation during the query plan optimization, or as a metadata layer assisting in the query execution.

Although such framework can be compared to a number of approaches to data block-level indexing, the first realistic implementation of a granulated metadata support of a query execution seems to refer to Netezza's nearly ordered maps, also known as the zone maps [9]. Netezza's invention was to partition data into blocks of consecutively loaded rows, annotate each of such blocks with its min/max values with respect to particular data columns, and use such rough values against `WHERE` clauses in order to eliminate the blocks that were for sure out of the scope of a given query. In this paper, we focus on the technology developed by Infobright, which goes further by means of applied types of statistics, identification of cases where blocks of data do not need to be accessed, and data processing operations for which such identification can be taken into account.

There is also another, not so broadly studied case when a block of rows does not need to be accessed. It occurs when it is enough to use its statistics. It may happen, e.g., when we are sure that all rows in a block satisfy query conditions and, therefore, some of its rough values can represent its contribution into the final result. In our research, we noted an analogy between the two above-discussed cases of blocks that do not require accessing and positive/negative regions used in the theory of rough sets to learn classification models [10]. It helped us to understand how to employ various AI-based heuristics for our own purposes. It also explains why we use terms such as a rough column and a rough value when introducing foundations of our approach. Throughout the rest of the paper, we will refer to the both of above cases as to the solved blocks of rows.

The efficiency of rough computations depends on an ability to classify blocks as solved by rough values. Quality of granulated tables understood in this way relies on regularities within the collections of column values in particular blocks. In the case of solutions such as Netezza or Infobright, blocks are assembled along the flow of rows loaded into a database. Thus, the highest quality is expected for rough values corresponding to columns

[1] www.infobright.com

that are loosely ordered. Even if one knows that some other columns are going to be more frequently queried, it is hard for users to accept any time consuming reorganizations of rows that would improve the quality of the corresponding parts of metadata. Fortunately, it is also possible to consider more on-the-fly strategies for re-ordering the loaded rows [11], basing on ideas borrowed from the data stream cluster analysis [2].

The rough values can be also computed dynamically, as descriptions of intermediate results of the query execution. As a case study, consider a correlated subquery: a query nested in some other query – called an outer query – which often operates on another table – called an outer table – and which is parameterized by some outer table's values. Correlated subqueries can occur in a number of SQL clauses. Here, for illustrative purposes, consider the following example of the `WHERE` clause of the outer query: `T.a<(SELECT MIN(U.x) FROM U WHERE U.y=T.b)`. If `U` is large, then the execution of the outer query on `T` may be time consuming. However, as pointed out in [14], one can quickly derive approximate answers to particular subqueries and, for each row in `T`, check whether the above condition could be successfully resolved by using such dynamically obtained rough values. Let us also add that such a mechanism of fast production of the rough values estimating the query results is available as so called rough SQL in the 4.x^{++} versions of Infobright's RDBMS[2].

3 Toward Columnar Storage / Processing

Columnar databases seem to be particularly useful in database analytics. Complex reports can be realized by highly optimized column scans. Ad hoc filters can be processed directly against particular columns or projections [13]. Only those values that are associated with columns involved in a query need to be processed. Some of components of results can be kept as non-materialized for a longer time during the query execution [1]. Since each column stores a single data type, it is possible to apply better adjusted compression algorithms. It is also possible to execute queries against not fully decompressed data [16]. Thus, for analytic queries, columnar databases are able to manage available resources efficiently. Also, e.g. with regard to minimizing the disk I/O, the columnar approaches turn out to have the same optimization goals as those described in Section 2.

The basic idea underlying integration of columnar and rough methodologies is to split data both vertically and horizontally. However, such a solution is worth considering also because of other reasons. First of all, it allows to combine the above advantages of columnar databases with pipelined mechanisms of the data processing [3]. Also, data compression techniques are then able to adjust better to smaller, potentially more homogeneous contents [15]. In general, a behavior of compression algorithms may vary not only with respect to different columns but also with respect to patterns automatically detected inside different collections of values.

Let us also note that one of important challenges is to efficiently compress and process alphanumeric collections. Our motivation to look at this issue arises from an analysis of machine-generated data sets, which grow extremely fast and often require ad hoc querying. Such data sets include web logs, computer events, connection details et cetera. They often correspond to long varchar columns. Interestingly, machine-generated data values often follow some semantic constructs that are difficult to detect even for sophisticated compression algorithms [6]. Accordingly, we designed interfaces allowing domain experts to express their knowledge about internal structures of values stored in varchar columns. We also developed methods that apply the acquired knowledge to improve both the alphanumeric data compression and the corresponding rough values' quality [8].

4 Infobright as a Rough-Columnar Engine

Infobright's RDBMS engine combines techniques presented in Sections 2 and 3 in order to provide efficient executions of analytic SQL statements over large amounts of data. New rough values are computed after receiving each block of 2^{16} freshly loaded rows. The gathered rows correspond to a new entry in the granulated table. The

[2] www.dbms2.com/2011/06/14/infobright-4-0/

engine first decomposes rows onto particular columns' values and creates data packs – collections of 2^{16} values of each column. In other words, each data pack corresponds to a single block of rows and a single data column. Each data pack is represented by a rough value summarizing its statistics at a metadata level. Last but not least, each of data packs can be compressed independently.

There are actually two metadata layers. The first one stores information about a location and status of each data pack. It also contains lower level statistics assisting in pack's decompression or, if applicable, translating incoming requests in order to resolve them against only partially decompressed pack's content. This layer is often referred to as a data grid. The second layer, called a knowledge grid, maintains statistics corresponding to granulated tables. Besides elements already mentioned in Section 2, it contains some more advanced structures resembling those used in other approaches [4, 7]. A common feature of all stored structures is that their sizes are orders of magnitude smaller than the sizes of the corresponding data packs, and that they are all designed to be used in rough operations. Also, the granulated tables residing in the knowledge grid are internally structured in a columnar fashion, so particular rough columns can be efficiently employed.

Infobright's knowledge grid can be applied to minimize portions of data required to resolve queries along the lines discussed in Section 2. However, the data access reduction process is now conducted at a more fine-grained level of particular data packs rather than blocks of rows. Rough values are applied against query components in order to identify the data packs that are solved at particular data processing stages. The meaning of solved data packs is analogous to solved blocks. However, within a single block of rows, classifications of the data packs corresponding to different columns may vary. For instance, in case of an ad hoc multi-column condition, some data packs can be identified as solved while others may require a more thorough analysis. However, while combining such classifications, more data packs can change their status to solved. The same may happen after examining some of data packs value by value. Generally, a number of solved data packs can grow during a given query execution, basing on additional information acquired iteratively [12].

In summary, the discussed rough-columnar approach has already enabled Infobright to gather positive feedback from users in areas such as online analytics, financial services and telecommunications, with the largest current customer approaching one petabyte of processed data[3]. Still, there are more opportunities to use the above methods for a faster analytic querying against large data. For example, we use rough values not only for purposes of identifying solved data packs but also for heuristic ordering of packs to be decompressed, as well as for grouping packs into data areas corresponding to better-defined processing sub-tasks. We assign rough values not only to physical data packs but also to intermediate structures produced during the query execution (e.g.: hash tables constructed during aggregations and joins). We produce rough values that can be applied at further execution stages (e.g.: already-mentioned rough execution of correlated subqueries or the recently implemented computation of rough values for expressions). Finally, we can adapt a number of useful techniques developed within the main stream research on columnar databases, as visible also in the next section.

5 A Roadmap for Correlated Subqueries

There are various approaches to the optimization of correlated subqueries [5]. Their importance is visible in several areas of business intelligence, such as the trend or predictive analysis. Complex nested queries may occur at the reporting stages focused on identifying anomalies and risk indicators. They may be useful also in applications requiring storage of large sets of hierarchical objects in a relational format. In many cases, e.g. for large metadata repositories, a dynamic reconstruction of such objects may be also supported by massive self-join operations or some recursive SQL extensions, if available. However, there are cases where the usage of correlated subqueries seems to be particularly convenient, if they perform fast enough. For instance, for a purpose of analysis of documents stored in an EAV-like type of format, the correlated subqueries may be applied to extract information about documents that posses some specific semantic and/or statistical features.

[3] www.infobright.com/customer/jdsu

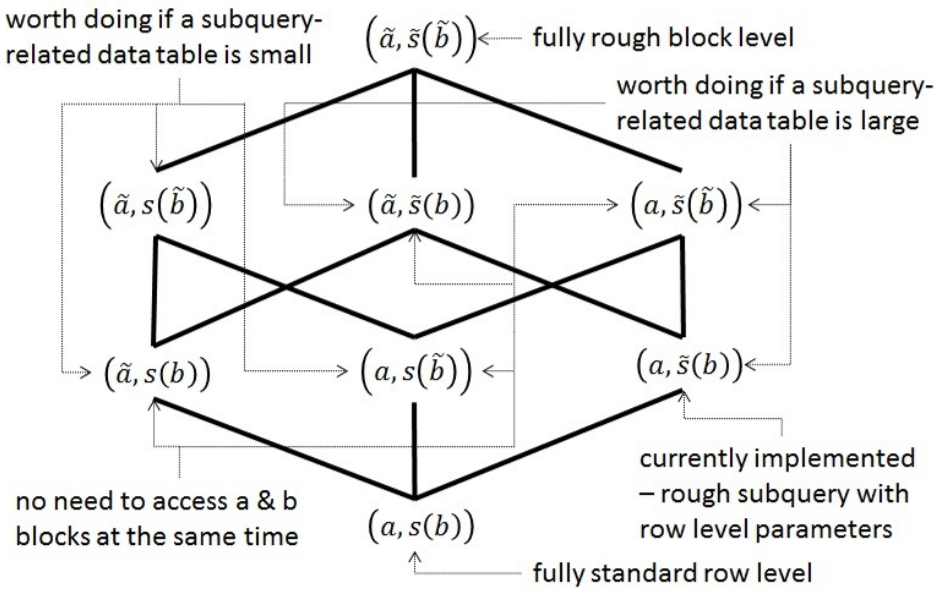


Figure 1: The roadmap for improving the execution of correlated subqueries. An example of a correlated subquery in the `WHERE` clause, where outer table's column a is compared with a result of subquery parameterized by outer table's column b. a and $\widetilde{a}$ denote, respectively, an exact value (for a row) or a rough value (for a block) of a. (The same applies to b and $\widetilde{b}$.) s and $\widetilde{s}$ denote, respectively, the standard and the rough execution of the subquery, which may be parameterized by b or $\widetilde{b}$. The pairs in brackets denote which modes of values and/or rough values are compared to each other while resolving the `WHERE` part of a query.

The current Infobright's implementation of the correlated subqueries was informally outlined in Section 2. Sticking to an example of a subquery in the `WHERE` clause, we launch its fast but inexact version with parameters induced by each consecutive row in the outer table and attempt to use its rough outcome to avoid the exact mode of execution. In Figure 1, it is illustrated by the cases $(a, \widetilde{s}(b))$ and $(a, s(b))$. The first of them symbolizes, for a given row, the comparison of its value on the column a with a rough result of the subquery parameterized by its value on the column b. If such a comparison is not sufficient to decide whether that row satisfies the `WHERE` clause, then, for that particular row, we need to proceed with the standard subquery execution $s(b)$.

Figure 1 represents also other cases waiting for implementation. Let us first consider $(\widetilde{a}, \widetilde{s}(\widetilde{b}))$, where we confront the rough value obtained as a result of $\widetilde{s}(\widetilde{b})$ with the rough value of a. If such a comparison fails to classify the block of rows as solved, the algorithm should proceed with either of: 1) the standard subquery execution $s(\widetilde{b})$ and a comparison of its result with the a's rough value; 2) decompression of the a's data pack and comparison of the a's values on particular rows with the previously computed $\widetilde{s}(\widetilde{b})$; 3) decompression of the b's data pack and comparison of the a's rough value with the rough result of $\widetilde{s}(b)$ computed for particular values of b. The possible computation strategies correspond to the top-down paths in Figure 1. For instance, the strategy $(\widetilde{a}, \widetilde{s}(\widetilde{b}))\rightarrow(\widetilde{a}, \widetilde{s}(b))\rightarrow(a, \widetilde{s}(b))\rightarrow(a, s(b))$ would proceed with the current implementation if the above-described step #3 fails to categorize some of the rows that are important for the final query result.

The choice of the computation strategy shall depend on the expected gain and cost related to the standard subquery execution and/or the decompression of data packs of particular columns. It can be decided for each of blocks of rows separately or for the entire table, depending on how the underlying column scans can be optimally arranged. Let us also note that in order to proceed with $s(\widetilde{b})$ and $\widetilde{s}(\widetilde{b})$, the subquery syntax needs to be modified. For instance, for a query considered in Section 2, its part `U.y = T.b` would need to be automatically replaced by `U.y BETWEEN T.b_min AND T.b_max`, where `T.b_min` and `T.b_max` denote the min/max statistics of b for a given block. This requires further research for a broader class of queries.

6 Conclusion

Our primary goal in this paper was to show that it is worth combining the ideas behind columnar databases and rough operations on granular metadata. We interpreted Infobright's solution as an example of a rough-columnar RDBMS. In particular, we investigated the execution of correlated subqueries in order to show a variety of possibilities provided by the coexistence of presented methods within the same framework.

References

[1] Abadi, D.J., Myers, D.S., DeWitt, D.J., Madden, S.: Materialization Strategies in a Column-oriented DBMS. In: Proc. of ICDE (2007) 466–475

[2] Aggarwal, C.C. (ed.): Data Streams: Models and Algorithms. Springer (2007)

[3] Boncz, P., Zukowski, M., Nes, N.: MonetDB/X100: Hyper-pipelining Query Execution. In: Proc. of CIDR (2005) 225–237

[4] Deshpande, A., Ives, Z.G., Raman, V.: Adaptive Query Processing. Foundations and Trends in Databases **1**(1) (2007) 1–140

[5] Elhemali, M., Galindo-Legaria, C.A., Grabs, T., Joshi, M.: Execution Strategies for SQL Subqueries. In: Proc. of SIGMOD (2007) 993–1004

[6] Inenaga, S., Hoshino, H., Shinohara, A., Takeda, M., Arikawa, S., Mauri, G., Pavesi, G.: On-line Construction of Compact Directed Acyclic Word Graphs. Discrete Applied Mathematics **146**(2) (2005) 156–179

[7] Ioannidis, Y.E.: The History of Histograms (Abridged). In: Proc. of VLDB (2003) 19–30

[8] Kowalski, M., Ślęzak, D., Toppin, G., Wojna, A.: Injecting Domain Knowledge into RDBMS – Compression of Alphanumeric Data Attributes. In: Proc. of ISMIS (2011) 386–395

[9] Metzger, J.K., Zane, B.M., Hinshaw, F.D.: Limiting Scans of Loosely Ordered and/or Grouped Relations Using Nearly Ordered Maps. US Patent 6,973,452 (2005)

[10] Pawlak, Z., Skowron, A.: Rudiments of Rough Sets. Information Sciences **177**(1) (2007) 3–27

[11] Ślęzak, D., Kowalski, M., Eastwood, V., Wróblewski, J.: Method and System for Database Organization. US Patent Application 2009/0106210 A1 (2009)

[12] Ślęzak, D., Wróblewski, J., Eastwood, V., Synak, P.: Brighthouse: An Analytic Data Warehouse for Ad-hoc Queries. PVLDB **1**(2) (2008) 1337–1345

[13] Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.J., O'Neil, P.E., Rasin, A., Tran, N., Zdonik, S.B.: C-Store: A Column-oriented DBMS. In: Proc. of VLDB (2005) 553–564

[14] Synak, P.: Rough Set Approach to Optimisation of Subquery Execution in Infobright Data Warehouse. In: Proc. of SCKT (PRICAI Workshop) (2008)

[15] White, P., French, C.: Database System with Methodology for Storing a Database Table by Vertically Partitioning all Columns of the Table. US Patent 5,794,229 (1998)

[16] Zukowski, M., Héman, S., Nes, N., Boncz, P.: Super-scalar RAM-CPU Cache Compression. In: Proc. of ICDE (2006) 59

MonetDB: Two Decades of Research in Column-oriented Database Architectures

Stratos Idreos Fabian Groffen Niels Nes Stefan Manegold Sjoerd Mullender Martin Kersten

Database Architectures group*, CWI, Amsterdam, The Netherlands

Abstract

MonetDB is a state-of-the-art open-source column-store database management system targeting applications in need for analytics over large collections of data. MonetDB is actively used nowadays in health care, in telecommunications as well as in scientific databases and in data management research, accumulating on average more than 10,000 downloads on a monthly basis. This paper gives a brief overview of the MonetDB technology as it developed over the past two decades and the main research highlights which drive the current MonetDB design and form the basis for its future evolution.

1 Introduction

MonetDB[1] is an open-source database management system (DBMS) for high-performance applications in data mining, business intelligence, OLAP, scientific databases, XML Query, text and multimedia retrieval, that is being developed at the CWI database architectures research group since 1993 [19].

MonetDB was designed primarily for data warehouse applications. These applications are characterized by large databases, which are mostly queried to provide business intelligence or decision support. Similar applications also appear frequently in the area of e-science, where observations are collected into a warehouse for subsequent scientific analysis. Nowadays, MonetDB is actively used in businesses such as health care and telecommunications as well as in sciences such as in astronomy. It is also actively used in data management research and education. The system is downloaded more than 10,000 times every month.

MonetDB achieves significant speed up compared to more traditional designs by innovations at all layers of a DBMS, e.g., a storage model based on vertical fragmentation (column-store), a modern CPU-tuned query execution architecture, adaptive indexing, run-time query optimization, and a modular software architecture. The rest of the paper gives a brief overview of the main design points and research directions.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*__Acknowledgments.__ The research and development reported here has been made possible by all former and current members of CWI's Database Architectures group, most notably Peter Boncz, Romulo Goncalves, Sandor Heman, Milena Ivanova, Erietta Liarou, Lefteris Sidirourgos, Ying Zhang, Marcin Zukowski.

[1] `http://www.monetdb.org/`

2 MonetDB Design

From a user's point of view, MonetDB is a full-fledged relational DBMS that supports the SQL:2003 standard and provides standard client interfaces such as ODBC and JDBC, as well as application programming interfaces for various programming languages including C, Python, Java, Ruby, Perl, and PHP.

MonetDB is designed to exploit the large main memories of modern computer systems effectively and efficiently during query processing, while the database is persistently stored on disk. With respect to performance, MonetDB mainly focuses on analytical and scientific workloads that are read-dominated and where updates mostly consist of appending new data to the database in large chucks at a time. However, MonetDB also provides complete support for transactions in compliance with the SQL:2003 standard.

Internally, the design, architecture and implementation of MonetDB reconsiders all aspects and components of classical database architecture and technology by effectively exploiting the potentials of modern hardware. MonetDB is one of the first publicly available DBMSs designed to exploit column-store technology. MonetDB does not only use the column-oriented logic for the way it stores data; it provides a whole new design for an execution engine that is fully tailored for columnar execution, deploying carefully designed cache-conscious data structures and algorithms that make optimal use of hierarchical memory systems [2].

In addition, MonetDB provides novel techniques for efficient support of a priori unknown or rapidly changing workloads over large data volumes. Both the fine-grained flexible intermediate result caching technique "recycling" [12] and the adaptive incremental indexing technique "database cracking" [8] require minimal overhead and investment to provide maximal benefit for the actual workload and the actual hot data.

The design also supports extensibility of the whole system at various levels. Via extension modules, implemented in C or MonetDB's MAL language, new data types and new algorithms can be added to the system to support special application requirements that go beyond the SQL standard, or enable efficient exploitation of domain-specific data characteristics. Additionally, MonetDB provides a modular multi-tier query optimization framework that can be extended with domain specific optimizer rules.

Finally, the core architecture of MonetDB has proved to provide efficient support not only for the relational data model and SQL, but also for, e.g., XML and XQuery [1]. In this line, support for RDF and SPARQL, as well as arrays [14] is currently under development.

Physical Data Model. The storage model is a significant deviation of traditional database systems. Instead of storing all attributes of each relational tuple together in one record (aka. *row-store*), MonetDB represents relational tables using vertical fragmentation (aka. *column-store*), by storing each column in a separate `(surrogate,value)` table, called a *BAT* (*Binary Association Table*). The left column, often the surrogate or `OID` (object-identifier), is called the *head*, and the right column, usually holding the actual attribute values, is called the *tail*. In this way, every relational table is internally represented as a collection of BATs. For a relation R of k attributes, there exist k BATs, each BAT storing the respective attribute as (`OID`,`value`) pairs. The system-generated `OID` identifies the relational tuple that the attribute `value` belongs to, i.e., all attribute values of a single tuple are assigned the same `OID`. `OID` values form a dense ascending sequence representing the *position* of a value in the column. Thus, for base BATs, the `OID` columns are not materialized, but rather implicitly given by the position. This makes base BATs essentially equal to typed arrays in C with optional metadata. For each relational tuple t of R, all attributes of t are stored in the *same* position in their respective column representations. The position is determined by the insertion order of the tuples. This tuple-order *alignment* across all base columns allows the column-oriented system to perform tuple reconstructions efficiently in the presence of tuple order-preserving operators.

For fixed-width data types (e.g., integer, decimal and floating point numbers), MonetDB uses a plain C array of the respective type to store the `value` column of a BAT. For variable-width data types (e.g., strings), MonetDB applies a kind of dictionary encoding. All distinct values of a column are stored in a BLOB and the `value` column of the BAT is an integer array with references to BLOB positions where the actual values exist.

MonetDB uses the operating system's memory mapped files support to load data in main memory and

exploit extended virtual memory. Thus, all data structures are represented in the same binary format on disk and in memory. Furthermore, MonetDB uses late tuple reconstruction, i.e., during the entire query evaluation all intermediate results are in a column format. Only just before sending the final result to the client, N-ary tuples are constructed. This approach allows the query engine to exploit CPU- and cache-optimized vector-like operator implementations throughout the whole query evaluation relying on a bulk processing model as opposed to the typical Volcano approach, allowing to minimize function calls, type casting, various metadata handling costs, etc. Intermediate results need to be materialized, but those can be reused [12].

Execution Model. The MonetDB kernel is an abstract machine, programmed in the *MonetDB Assembly Language* (*MAL*). The core of MAL is formed by a closed low-level two-column relational algebra on BATs. N-ary relational algebra plans are translated into two-column BAT algebra and compiled to MAL programs. These MAL programs are then evaluated in a *operator-at-a-time* manner, i.e., each operation is evaluated to completion over its entire input data, before subsequent data-dependent operations are invoked. Each BAT algebra operator maps to a simple MAL instruction, which has zero degrees of freedom in its behavior (obviously, it may be parameterized where necessary): it does not take complex expressions as parameter. Complex operations are broken into a sequence of BAT algebra operators that each perform a simple operation on an entire column of values ("bulk processing"). This allows the implementation of the BAT algebra to avoid an expression interpreting engine; rather, all BAT algebra operations in the implementation map onto simple array operations. The figure on the left shows such an implementation of a select operator. The BAT algebra operators have the advantage that tight for-loops without function calls create high instruction locality which eliminates the instruction cache miss problem. Such simple loops are amenable to compiler optimization (loop pipelining, blocking, strength reduction), and CPU out-of-order speculation.

```
select(B:bat[:oid,:int], V:int) ⟶ for (i = j = 0; i < n; i++)
                                    if (B.tail[i] == V)
                                      R.tail[j++] = i;
```

System Architecture. MonetDB's query processing scheme is centered around three software layers.

Front-end. The top layer or *front-end* provides the user-level data model and query language. While relational tables with SQL and XML with XQuery [1] are readily available, arrays with SciQL [14] and RDF with SPARQL are on our research and development agenda. The front-end's task is to map the user-space data model to MonetDB's BATs and to translate the user-space query language to MAL. The query language is first parsed into an internal representation (e.g., SQL into relational algebra), which is then optimized using domain-specific rules. In general, these domain-specific *strategic optimizations* aim primarily at reducing the amount of data to be processed, i.e., the size of intermediate results. In the case of SQL & relational algebra, such optimizations include heuristics like pushing down selections and exploiting join indexes. The optimized logical plan is then translated into MAL and handed over to the back-end for general MAL-optimization and evaluation.

Back-end. The middle layer or *back-end* consists of the MAL optimizers framework and the MAL interpreter as textual interface to the kernel. The MAL optimizers framework consists of a collection of optimizer modules that each transform a given MAL program into a more efficient one, possibly adding resource management directives. The modules provide facilities ranging from symbolic processing up to just-in-time data distribution and execution. This *tactical optimization* is more inspired by programming language optimization than by classical database query optimization. It breaks with the hitherto omnipresent cost-based optimizers, recognizing that not all decisions can be cast together in a single cost formula. Operating on the common binary-relational back-end algebra, these optimizer modules are shared by all front-end data models and query languages.

Kernel. The bottom layer or *kernel* (aka. *GDK*) provides BATs as MonetDB's bread-and-butter data structure, as well as the library of highly optimized implementations of the binary relational algebra operators. Due to the operator-at-a-time "bulk-processing" evaluation paradigm, each operator has access to its entire input including known properties. This allows the algebra operators to perform *operational optimization*, i.e., to choose at runtime the actual algorithm and implementation to be used, based on the input's properties. For instance, a `select` operator can exploit sorted-ness of a BAT by deploying binary search, or (for point-selections) use an existing hash-index, and fall back to a scan otherwise. Likewise, a join can at runtime decide to, e.g., perform a merge-join if the join attributes happen to be sorted, and fall-back to hash-join otherwise.

3 Research

In this section, we briefly summarize the highlights of column-oriented research in the context of MonetDB. Part of the topics discussed below reflect fundamental research in database architectures which has lead to the current design of MonetDB and its spin-offs. Another part of the research topics reflects high risk research aiming at future innovations in database architectures and big data analytics. Both fundamental and high risk projects are fully materialized within the MonetDB kernel and are disseminated as open source code together with the rest of the MonetDB code family.

Hardware-conscious Database Technology. A key innovation in MonetDB is its reliance on hardware conscious algorithms. In the past decades, advances in speed of commodity CPUs have far outpaced advances in RAM latency. Main-memory access has therefore become a performance bottleneck for many computer applications, including database management systems; a phenomenon widely known as the "memory wall".

The crucial aspect in order to overcome the memory wall is good use of CPU caches, i.e., careful tuning of memory access patterns is needed. This led to a new breed of query processing algorithms, in particular for join processing, such as *partitioned hash-join* [3] and *radix-cluster* [18]. The key idea is to restrict any random data access pattern to data regions that fit into the CPU caches to avoid cache misses, and thus, performance degradation. For query optimization to work in a cache-conscious environment, and to enable automatic tuning of our cache-conscious algorithms on different types of hardware, we developed a methodology for creating cost models that take the cost of memory access into account [17]. The key idea is to abstract data structures as *data regions* and model the complex data access patterns of database algorithms in terms of simple compounds of a *few basic data access patterns*.

Vectorized Execution and Light-weight Compression. The X100 project explored a compromise between classical tuple-at-a-time pipelining and operator-at-a-time bulk processing [4]. The idea of vectorized execution is to operate on chunks (vectors) of data that are large enough to amortize function call overheads, but small enough to fit in CPU caches and to avoid materialization of large intermediates into main memory. Combined with just-in-time light-weight compression, it lowers the memory wall significantly [21]. The X100 project has been commercialized into the Actian/VectorWise company.

Reusing Intermediate Results with Recycler. Bulk processing in a column-store architecture implies materialization of intermediate results. The *Recycler* project adaptively stores and reuses those intermediates when possible, i.e., when a select operator is covered by a stored intermediate of a past query, then MonetDB avoids touching the base column [**?**]. Intermediates are kept around as long as they fit in the allocated space for the Recycler and as long as they are hot.

Adaptive Indexing and Database Cracking. Modern business applications and scientific databases call for inherently dynamic data storage environments. Such environments are characterized by two challenging features: (a) they have little idle system time to devote on physical design; and (b) there is little, if any, a priori workload knowledge, while the query and data workload keeps changing dynamically. In such environments, traditional approaches to index building and maintenance cannot apply.

To solve this problem, MonetDB research pioneered *Database cracking* that allows on-the-fly physical data reorganization, as a collateral effect of query processing [8]. Cracking continuously and automatically adapts indexes to the workload at hand, without human intervention. Indexes are built incrementally, adaptively, and on demand as part of select operators, join operators and projection operators; the more queries are processed the more the relevant indexes are optimized.

Cracking was proposed in the context of modern column-stores and has been hitherto applied for boosting the select operator performance [8], maintenance under updates [9], and arbitrary multi-attribute queries with *sideways cracking* [10]. With *partial cracking*, a storage threshold can be applied and cracking adaptively maintains its auxiliary structures within this threshold [10]. A new variant, *stochastic cracking* maintains and expands the adaptive behavior across various workloads by opportunistically introducing random index refinement optimization actions during query processing [7]. Furthermore, contrary to first impressions, database cracking

allows for queries to run concurrently even though at the conceptual level it turns read queries into write queries. With careful and adaptive management of short term latches and with early release of those latches multiple queries can operate in parallel over the cracking indexes [6]. In addition, more recently, database cracking has been extended to exploit a partition/merge -like logic as well as to study the various tradeoffs of adaptive indexing in more detail [11]. When to optimize an adaptive index and by how much are fundamental questions that this research tries to answer.

DataCyclotron. One of the grand challenges of distributed query processing is to devise a self-organizing architecture which exploits all hardware resources optimally to manage the database hot-set, to minimize query response time, and to maximize throughput without single point global co-ordination. The Data Cyclotron architecture [5] addresses this challenge using turbulent data movement through a storage ring built from distributed main memory and capitalizing on the functionality offered by modern remote-DMA network facilities. Queries assigned to individual nodes interact with the storage ring by picking up data fragments that are continuously flowing around, i.e., the hot-set.

Adaptive Sampling and Exploration with SciBORQ. In modern applications, not all data is equally useful all the time. A strong aspect in query processing is exploration. In the SciBORQ project, we explore a route based on exactly this knowledge that only a small fraction of the data is of real value for any specific task [20]. This fraction becomes the focus of scientific reflection through an iterative process of ad-hoc query refinement. However, querying a multi-terabyte database requires a sizeable computing cluster, while ideally the initial investigation should run on the scientist's laptop. We work on strategies on how to make biased *snapshots* of a science warehouse such that data exploration can be instigated using precise control over all resources.

MonetDB/DataCell: Stream Processing in a Column-store. New applications are increasingly transformed into online and continuous streaming applications, e.g., new data continuously arrives, and we need to do analytics in a small time window until the next data batch arrives. This scenario occurs in network and big clusters monitoring, in web logs, in the financial market and in many more applications. In the DataCell project, we design a stream engine on top of MonetDB [16] with a target to allow for advanced analytics "as the data arrives". The major challenge is the efficient support for specialized stream processing features such as window based processing and fast responses as data arrive as well as support for advanced database features such as indexing over the continuously changing data stream.

Graph Databases and Run Time Optimization. Optimization of complex XQueries combining many XPath steps and joins is currently hindered by the absence of good cardinality estimation and cost models for XQuery. Additionally, the state of the art of even relational query optimization still struggles to cope with cost model estimation errors that increase with plan size, as well as with the effect of correlated joins and selections. With ROX, we propose to radically depart from the traditional path of separating the query compilation and query execution phases, by having the optimizer execute, materialize partial results, and use sampling based estimation techniques to observe the characteristics of intermediates [13]. While run-time optimization with sampling removes many of the vulnerabilities of classical optimizers, it brings its own challenges with respect to keeping resource usage under control, both with respect to the materialization of intermediates, as well as the cost of plan exploration using sampling.

4 Deployment

Being freely available in open source, only a fraction of the MonetDB users provide us with detailed information about their deployment of and experiences with MonetDB. In academic environments, MonetDB is used for education and numerous research projects. We use MonetDB in several projects ranging from emergency management (EMILI: `http://emili-project.eu/`) over earth observation (TELEIOS: `http://earthobservatory.eu/`) to astronomy (LOFAR: `http://lofar.org/`) with a general focus on scientific databases (SciLens: `http://scilens.org`), including an implementation of the Sloan Digital Sky

Survey's Data Release 7 (`http://www.scilens.org/skyserverdemo/introduction`). Known commercial deployments range from off-line data analysis for business consultants on 32-bit Windows laptops to large-scale call-detail-record management for telecommunication operators on a farm of large Linux servers.

5 Future Paths

The MonetDB system and continuous research mainly targets the arena of data intensive applications over massive amounts of data such as in scientific databases and the need for analytics in modern businesses. Our future efforts are mainly towards distributed and highly parallel processing as well as towards adaptive and exploratory processing where database systems may *interpret queries by their intent*, rather than as a contract carved in stone for complete and correct answers [15].

References

[1] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD*, 2006.

[2] P. Boncz, M.L. Kersten, and S. Manegold. Breaking the Memory Wall in MonetDB. *Communications of the ACM (CACM)*, 51(12), Dec. 2008.

[3] P. Boncz, S. Manegold, and M.L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*, 1999.

[4] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[5] R. Goncalves and M.L. Kersten. The Data Cyclotron Query Processing Scheme. In *EDBT*, 2010.

[6] G. Graefe, F. Halim, S. Idreos, H. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *PVLDB*, 2012.

[7] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 2012.

[8] S. Idreos, M.L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.

[9] S. Idreos, M.L. Kersten, and S. Manegold. Updating a Cracked Database. In *SIGMOD*, 2007.

[10] S. Idreos, M.L. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-stores. In *SIGMOD*, 2009.

[11] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9), 2011.

[12] M. Ivanova, M.L. Kersten, N. Nes, and R. Goncalves. An Architecture for Recycling Intermediates in a Column-store. In *SIGMOD*, 2009.

[13] R. A. Kader, P. Boncz, S. Manegold, and M. van Keulen. ROX: Run-time Optimization of XQueries. In *SIGMOD*, 2009.

[14] M.L. Kersten, Y. Zhang, M. Ivanova, and N. Nes. SciQL, a query language for science applications. In *EDBT Workshop on Array Databases*, 2011.

[15] M.L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *PVLDB*, 4(12), 2011.

[16] E. Liarou, R. Goncalves, and S. Idreos. Exploiting the Power of Relational Databases for Efficient Stream Processing. In *EDBT*, 2009.

[17] S. Manegold, P. Boncz, and M.L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *VLDB*, 2002.

[18] S. Manegold, P. Boncz, and M.L. Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(4):709–730, 2002.

[19] S. Manegold, M.L. Kersten, and P. Boncz. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB*, 2(2), 2009.

[20] L. Sidirourgos, M.L. Kersten, and P. Boncz. SciBORQ: Scientific data management with Bounds On Runtime and Quality. In *CIDR*, 2011.

[21] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.

HyPer: Adapting Columnar Main-Memory Data Management for Transactional AND Query Processing

Alfons Kemper Thomas Neumann Florian Funke Viktor Leis Henrik Mühe
TU München, Faculty of Informatics, firstname.lastname@in.tum.de

Abstract

Traditionally, business applications have separated their data into an OLTP data store for high throughput transaction processing and a data warehouse for complex query processing. This separation bears severe maintenance and data consistency disadvantages. Two emerging hardware trends allow the consolidation of the two disparate workloads onto the same database state on one system: the increasing main memory capacities of several terabytes per server and multi-threaded processing based on multi-core parallelism. The prevalent data representation of hybrid OLTP&OLAP main memory database systems is columnar in order to achieve best possible query execution performance for OLAP applications. In order to shield the OLTP transaction processing from long-running queries without costly locking/latching, all queries are executed on an arbitrarily recent snapshot of the data. The paper contrasts several snapshotting techniques for columnar data (twin block, versioning) with the hardware-supported shadow paging we employ in the HyPer system. While OLAP-query processing can rely mostly on columnar scans, high OLTP throughput requirements necessitate index techniques for exact match and small range queries. Despite the ever growing capacity, main memory is still a scare resource. Therefore, compressing main memory resident databases is beneficial. The paper will devise techniques that achieve good compression ratios without hurting the mission-critical OLTP throughput by adaptively separating cold (i.e. immutable) data for aggressive compression from the hot (i.e. mutable) working set data that remains uncompressed.

1 Introduction

In this paper we want to highlight the architecture of our hybrid OLTP&OLAP main memory database system HyPer which – against common belief – achieves world-record transaction processing throughput and best-of-breed OLAP query response times in **one** system **in parallel** on the **same** database state. The two workloads of online transaction processing (OLTP) and online analytical processing (OLAP) present different challenges for database architectures. Currently, users with high rates of mission-critical transactions have split their data into two separate systems, one database for OLTP and one so-called *data warehouse* for OLAP. While allowing for decent transaction rates, this separation has many disadvantages including data freshness issues due to the delay caused by only periodically initiating the *Extract Transform Load*-data staging and excessive resource consumption due to maintaining two separate information systems. We present an efficient hybrid system, called *HyPer*, that can handle both OLTP and OLAP simultaneously by using hardware-assisted replication mechanisms to

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

maintain consistent snapshots of the transactional data. HyPer is a main-memory database system that guarantees the full ACID properties for OLTP transactions and executes OLAP query sessions (multiple queries) on arbitrarily current and consistent snapshots. The utilization of the processor-inherent support for virtual memory management (address translation, caching, copy-on-write) yields both at the same time: unprecedentedly high transaction rates as high as 100,000 per second and very fast OLAP response times on a single system executing both workloads in parallel. The performance analysis is based on a combined TPC-C and TPC-H benchmark.

In the novel hybrid OLTP&OLAP database system HyPer we separate the OLTP from the OLAP processing by way of snapshotting transactional data via the virtual memory management of the operating system [14]. In this architecture the OLTP process "owns" the database and periodically (e.g., in the order of seconds or minutes) forks an OLAP process. This OLAP process constitutes a fresh transaction consistent snapshot of the database. Thereby, we exploit operating systems functionality to create virtual memory snapshots for new, cloned processes. In Unix, for example, this is done by creating a child process of the OLTP process via the `fork` system call.

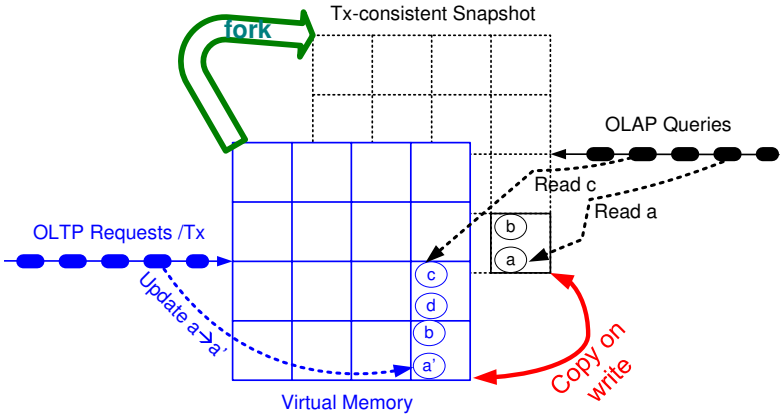


Figure 1: Virtual Memory Snapshots to Separate OLTP & OLAP.

The forked child process obtains an exact copy of the parent processes address space, as exemplified in the Figure by the overlayed page frame panel. This virtual memory snapshot that is created by the `fork`-operation will be used for executing a session of OLAP queries. These queries can be executed in parallel threads or serially, depending on the system resources or client requirements. In essence, the virtual memory snapshot mechanism constitutes a OS/hardware supported shadow paging mechanism as proposed decades ago for disk-based database systems by Lorie [17]. However, the original proposal incurred severe costs as it had to be software-controlled and it destroyed the clustering on disk. Neither of these drawbacks occurs in the virtual memory snapshotting as clustering across RAM pages is not an issue. Furthermore, the sharing of pages and the necessary copy-on-update/write is managed by the operating system with effective hardware support of the MMU (memory management unit) via the page table that translates VM addresses to physical pages and traps necessary replication (copy-on-write) actions. Therefore, the page replication is extremely efficiently done in 2 μs as we measured in a micro-benchmark.

HyPer's OLTP throughput is better than that of dedicated OLTP engines (like VoltDB) and HyPer's OLAP query response times match those of the best pure OLAP engines (e.g., MonetDB). It should be emphasized that HyPer can match (or beat) these two best-of-breed transaction (VoltDB) and query (MonetDB) processing engines **at the same time** by performing both workloads in parallel on the same database state. This performance evaluation was carried out on the basis of a new business intelligence benchmark, called the CH-benCHmark [6], that combines the transactional workload of TPC-C with the OLAP queries of TPC-H – executed against the **same** database state. Hyper's performance is due to the following design choices:

- HyPer relies on in-memory data management without the ballast of traditional database systems caused by DBMS-controlled page structures and buffer management. The SQL table definitions are transformed into simple vector-based virtual memory representations – which constitutes a column-oriented physical storage scheme.
- The OLAP processing is separated from the mission-critical OLTP transaction processing by **fork**-ing virtual memory snapshots. Thus, no concurrency control mechanisms other than the hardware-assisted

VM management are needed to separate the two workload classes.

- Transactions and queries are specified in SQL and are efficiently compiled into LLVM assembly code [20]. The transactions are specified in an SQL scripting language and registered as precanned, stored procedures. For the JIT-compiled interactive queries the very fast compilation into assembly language (LLVM) as opposed to a slow cross-compilation (into C/C++) is essential. The query evaluation follows a data-centric paradigm by applying as many operations on a data object as possible in between pipeline breakers. This evaluation scheme goes one step beyond cache-locality towards register-locality.
- As in VoltDB, the parallel transactions are separated via lock-free admission control that allows only non-conflicting transactions at the same time. Parallelism in this serial execution model is achieved by logically partitioning the database and admitting multiple partition-constrained transactions in parallel. However, for executing partition-crossing transactions the scheduler resorts to strict serial execution, rather than costly locking-based synchronization.
- HyPer relies on logical logging where, in essence, the invocation parameters of the stored procedures / transactions are logged via a high-speed network. The serial execution model in combination with partitioning and group committing achieves extreme scalability in terms of transaction throughput – without compromising the "holy grail" of ACID (that was sacrificed by the recent NoSQL/key value stores).
- While in-core OLAP query processing can be based on sequential scans, this is not possible for transaction processing as we require execution times of a few microseconds only. Therefore, we have developed sophisticated main-memory indexing structures based on hashing, balanced search trees (e.g., red black trees) and radix trees [15]. Hash indexes are indispensable for exact match (e.g., primary key) accesses that are most common in transactional processing while the tree structured indexes are essential for small-range queries, that are commonly encountered in transactional scripts as well.

2 Snapshotting

Since HyPer relies on the ability to efficiently create a transaction-consistent snapshot of the database, we compare different mechanisms for snapshot creation which do not diminish OLTP performance. These techniques are our hardware page shadowing approach as illustrated in Section 1 which we refer to as `vm-fork`, tuple shadowing which was used – for instance – by SolidDB [16] and a variant of the so called ZigZag approach as evaluated by Cao [5]. For brevity, we refer to implementation details in [19].

We extended all mechanisms to enable high performance query execution on snapshots as their most important use – instead of just recovery as previously suggested, e.g., by Molina et al. [9] or in [22]. Because of this, our approaches yield higher order snapshots for query processing as opposed to those snapshots primarily used for recovery.

Additionally, we adapted the mechanisms for use in main memory database systems: In case of Lorie's [17] page shadowing approach, we can show that the limitations when used in on-disk database systems can completely be alleviated in main memory. With Cao et al.'s twin objects approach (called ZigZag approach in [5]), we extended the implementation for use in a general purpose database system instead of a specialized application. We evaluate the different approaches taking both OLTP as well as OLAP throughput into account, since we want to evaluate their suitability for a hybrid OLTP & OLAP database system like HyPer.

The three techniques discussed here can be subdivided by the method they use to achieve a consistent snapshot while still allowing high throughput OLAP transactions on the data. The hardware page shadowing approach uses a hardware supported copy on write mechanism to create a snapshot. In contrast to that, tuple shadowing as well as the twin object approach use software mechanisms to keep a consistent snapshot of the data intact while modifications are stored separately.

Figure 2 shows the throughput for OLTP transactions without parallel OLAP execution ("raw" column) as well as OLTP and OLAP throughput while run at the same time. The OLTP transactions correspond to those of

the TPC-C. The OLAP queries consist of queries semantically equivalent to queries 1 and 5 of the TPC-H. The two representative OLAP queries are repeatedly executed in an alternating pattern.

Looking at OLTP throughput, it can be observed that techniques based on Hardware Page Shadowing yield higher throughput. This has two major reasons: First, Hardware Page Shadowing allows for faster reorganization than software controlled mechanisms. Second, there is no indirection as opposed to Tuple Shadowing where a shadow tuple has to be checked, or Twin Tuples, where the tuple to be read has to be found using a bit flag.

OLAP query performance is influenced less by the choice of snapshotting mechanism. Compared to a 50% slowdown as seen in OLTP throughput, OLAP queries run about 25% slower when a software controlled snapshotting mechanism is employed.

Here, the slowdown is caused by two main factors: Reorganization time and the delay caused by quiescing OLAP queries. All backends have been architected so that OLAP query performance is as high as possible. This is achieved by maintaining tuples included in the snapshot in their original form and position and adding redirection only for new, updated or deleted tuples which can only be seen by transactions, not OLAP queries.

Backend	Raw OLTP	Comb. OLTP	Comb. OLAP
VM-Fork	85k	60k	10.3
Tuple	25k	22k	6.8
Twin	33k	29k	7.0

Figure 2: OLTP and OLAP throughput per second combining TPC-C and TPC-H.

3 Database Compaction: Hot/Cold Clustering and Compression

Compression techniques for columnar database systems is a topic extensively studied, primarily in the context of analytical systems [1, 12, 26, 21]. While some of the existing work addresses the problem of updates in compressed databases [11, 3], none of the techniques developed for OLAP systems can be easily adapted for OLTP-style workloads. Modern in-memory database systems have fast and lean transaction models that penalize additional processing severely which often prevents them from compressing data in favor of transaction throughput. A good example is the lock-free transaction processing model pioneered by H-Store/VoltDB [13, 24] that executes transactions serially on private partitions without any overhead from buffer management or locking. This model allows for record-breaking transaction throughput, but necessitates that all transactions execute quickly to prevent the serial execution pipeline from becoming congested.

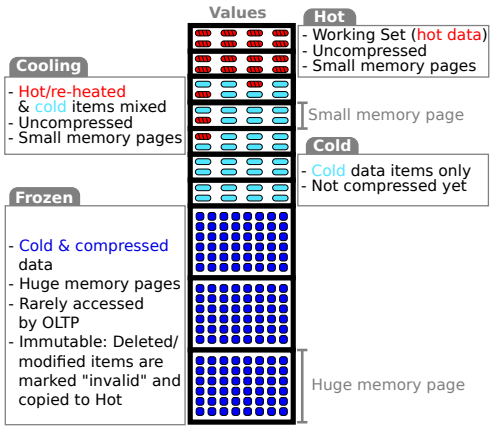


Figure 3: Hot/cold clustering for compression. = hot (volatile) data item, = cold data item, = cold & compressed data item.

To avoid hurting transactional throughput, OLTP engines often refrain from compressing their data and thus waste memory space. The lack of compression becomes even more impeding, when the database system is capable of running OLAP-style queries on the transactional data, like the HyPer system [14] or SAP HANA [8]. In this scenario, compression can not only reduce memory consumption significantly due to columnar storage, but also promises faster query execution [25, 1, 4, 10].

Approaches that maintain two separate data stores, an uncompressed store for freshly inserted data and a compressed store for older data, require costly merge phases that require exclusive locking of tables when moving data to the compressed store. They also tend to complicate and slow down query and transaction processing.

Our approach to compression in hybrid OLTP & OLAP column stores is based on the observation that while OLTP workloads frequently modify the dataset, they often follow the working set assumption [7]: Only a small subset of the data is accessed and an even smaller subset of this working set is being modified (cf. Figure 3). In business applications, this working set is mostly comprised of tuples that were added to the database in the recent past, as it can be observed in the TPC-C workload [23].

Our system uses a lightweight monitoring component to observe accesses to the dataset and identify opportunities to reorganize data such that it is clustered into hot and cold parts. After clustering, the database system compresses cold chunks to reduce memory consumption and streamline queries.

While compression reduces the memory consumption for the data set, we perform further optimizations to narrow the memory consumption in the presence of snapshots. Lorie's shadow paging [17] and its modern, hardware-assisted reincarnation, HyPer's `fork`-based snapshot mechanism, both maintain separate page tables for the current database state as seen by OLTP workers and snapshots. Creating a new snapshot requires the duplication of the current page table. Maintaining the snapshot requires the replication of those pages that are modified by transactions in order to preserve the consistent snapshot state from the time of its creation. The performance of both operations, page table duplication and page replication, highly depends on the size of the pages: While huge pages result in smaller page tables, and thus faster snapshot creation, smaller pages incur less replication overhead. Hot/cold clustering is an elegant solution to this problem, as the cold bulk of the data can be stored on huge memory pages (and thus shrink the page table size) while the hot, frequently modified working set remains on regular memory pages that can be replicated inexpensively. The frozen, huge data pages are never modified; if a frozen data object is changed, after all, it is invalidated (via an invalidation vector) in the frozen partition and re-inserted into the hot working set. The invalidation vector can be maintained as a position tree [11] to retain the fast scanning speed without having to check the validity of every data object upon access.

In addition to the smaller page table, the use of huge pages has further advantages:

1. Scanning huge pages is faster than scanning regular pages. Manegold et al. [18] analyze the impact of translation lookaside buffer (TLB) misses on query performance and conclude with regard to the page size that the use of huge pages reduces the probability of TLB misses.
2. The TLB of the processor's memory management unit has separate sections for huge and normal pages on most platforms. Since the bulk of the database is cold and resides on huge pages, table-scans in OLAP queries mostly access the huge pages TLB. Transactions operate almost entirely on hot data that is stored on normal pages. Thus, separate hardware resources are used and no "TLB-thrashing" occurs.

4 Conclusion and Ongoing Work

In-memory technology has facilitated a reunion of OLTP and OLAP systems by separating the two disparate workloads via snapshotting. So, after all, a "one size fits all" system appears possible. We have presented a comparison of snapshotting techniques demonstrating the benefits of hardware-assisted shadow paging over software approaches. We have made the case for hot/cold clustering to store frequently accessed tuples together on regular memory pages while cold, immutable tuples can reside on huge pages. This leads to the advantageous combination of page table size (and thus snapshot creation costs) and replication overhead. In addition, it allows to compress the majority of the database without causing OLTP throughput declines.

In the future, we will investigate the following issues:

Long-running transactions: Currently, HyPer focuses on the execution of short, pre-canned transactions which are known in advance. Future research will extend this set of workloads to include transactions of arbitrary length. For the execution of these long-running transactions, an optimistic execution strategy which does not interfere with the execution of good-natured short transactions has to be designed.

Scale-out: So far, HyPer was designed as a single server system as we believe it to be more economical to scale up before you scale out. Nevertheless, for very large data volumes scaling out the database across multiple servers is necessary. To enable this, the query engine is being redesigned to distribute query plans and a query coordinator is being designed to manage intermediate query results. A global transaction manager is needed to control the execution of inter-site transactions and for creating globally consistent snapshots. The synchronization of global transactions relies on the execution model for long-running transactions.

Memory efficient index structures: Unlike systems which concentrate purely on OLAP, HyPer does not store data in a sorted fashion to allow for high OLTP throughput. Index structures are used both for efficient access to

single tuples using their key as well as ordered access to a relation. Since index structures make up a significant share of the total memory used by the DBMS, we are developing an efficient, compact index structure which works well with our snapshotting approach. As we regularly keep snapshots of index structures as well as the data, modifications to an index need to be as local as possible to cause only few copy-on-write operations.

Multi-core parallel query processing: To make operational BI a reality, it is necessary to exploit the vast computing power of modern multi-core servers effectively. For this purpose, the OLAP query engine of HyPer is currently extended by intra-operator parallelism. We are developing a new massively parallel sort merge-join algorithm [2] that scales linearly in the number of cores involved in the join computation.

References

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[2] M. Albutiu, A. Kemper, and T. Neumann. Massively Parallel Sort Merge Join in Main Memory Multi-core Database Systems. Technical report, 2012, TU München, 2012.

[3] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, 2009.

[4] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[5] T. Cao, M. Salles, B. Sowell, Y. Yue, J. Gehrke, A. Demers, and W. White. Fast Checkpoint Recovery Algorithms for Frequently Consistent Applications. In *SIGMOD*, 2011.

[6] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. A. Kuno, R. O. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload ch-benchmark. In *DBTest 2011*, 2011.

[7] P. J. Denning. The Working Set Model for Program Behaviour. *CACM*, 11(5):323–333, 1968.

[8] F. Färber, S. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[9] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE TKDE*, 4(6):509–516, 1992.

[10] G. Graefe and L. Shapiro. Data Compression and Database Performance. In *Symp. On Applied Computing*, 1991.

[11] S. Héman, M. Zukowski, N. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *SIGMOD*, 2010.

[12] A. Holloway, V. Raman, G. Swart, and D. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *SIGMOD*, 2007.

[13] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[14] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, 2011.

[15] V. Leis. Main-memory index structures for modern hardware. Master's thesis, TU Muenchen, 2012.

[16] A.-P. Liedes and A. Wolski. SIREN: A Memory-Conserving, Snapshot-Consistent Checkpoint Algorithm for in-Memory Databases. In *ICDE*, 2006.

[17] R. A. Lorie. Physical Integrity in a Large Segmented Database. *TODS*, 2(1), 1977.

[18] S. Manegold, P. Boncz, and M. L. Kersten. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In *VLDB*, 2000.

[19] H. Mühe, A. Kemper, and T. Neumann. How to efficiently snapshot transactional data: hardware or software controlled? In *DaMoN*, 2011.

[20] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[21] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *ICDE*, 2008.

[22] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *ICDE*, 1989.

[23] Transaction Processing Performance Council. TPC-C specification, 2010.

[24] VoltDB. Technical Overview. `http://www.voltdb.com`, March 2010.

[25] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Record*, 29(3):55–67, 2000.

[26] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, 2006.

An overview of HYRISE - a Main Memory Hybrid Storage Engine

Martin Grund[1], Philippe Cudre-Mauroux[2], Jens Krueger[1], Samuel Madden[3], Hasso Plattner[1]
[1]Hasso Plattner Institute, 14482 Potsdam, Germany
[2]eXascale Infolab, University of Fribourg, Switzerland,
[3]Database Group, MIT CSAIL, USA

Abstract

HYRISE is a new relational storage engine for main memory database systems. It is built on the premise that enterprise application workloads can benefit from a dedicated main-memory storage engine. The key idea behind HYRISE is that it provides dynamic vertical partitioning of the tables it stores. Since enterprise applications typically use a large number of very wide tables, we designed a novel layout algorithm specifically tailored for enterprise data. Our algorithm uses a main memory cost model that is able to precisely estimate the physical access cost of database operators and to determine a vertical partitioning that yields the best execution performance.

1 Introduction

Traditionally, the database market is divided into transaction processing (OLTP) and analytical processing (OLAP) workloads. OLTP workloads are characterized by a mix of reads and writes to a few rows at a time, typically through a B+Tree or other index structures. Conversely, OLAP applications are characterized by bulk updates and large sequential scans spanning few columns but many rows of the database, for example to compute aggregate values. Typically, those two workloads are supported by two different types of database systems – transaction processing systems and warehousing systems.

This simple categorization of workloads, however, does not entirely reflect modern enterprise computing. First, there is an increasing need for "real-time analytics" – that is, up-to-the-minute reporting on business processes that have traditionally been handled by warehousing systems. Although warehouse vendors are doing as much as possible to improve response times (e.g., by reducing load times), the explicit separation between transaction processing and analytics systems introduces a fundamental bottleneck in analytics response times. For some applications, directly answering analytics queries from the transactional system is preferable. For example "available-to-promise" (ATP) applications process OLTP-style queries while aggregating stock levels in real-time using OLAP-style queries to determine if an order can be fulfilled. Recently two new benchmarks TPC-CH[6] and a mixed workload benchmark presented in[3] emerged underlining the current trend in modern database systems towards mixed workloads scenarios.

Unfortunately, existing databases are not optimized for such mixed query workloads because their storage structures are usually optimized for one workload or the other. To address such workloads, we have built a

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

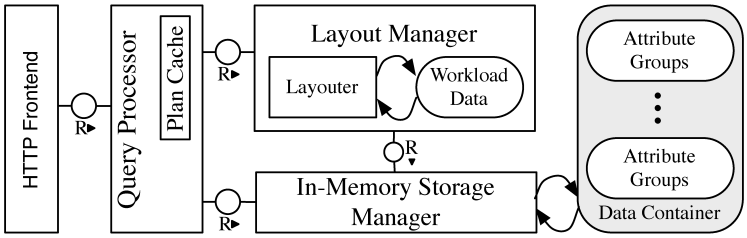


Figure 1: Architecture of HYRISE

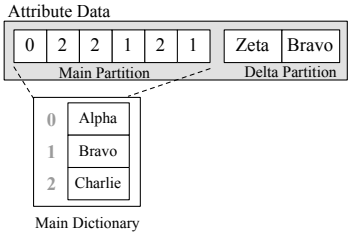


Figure 2: Logical structure of an attribute

main memory hybrid database system, called HYRISE, which partitions tables into vertical partitions of varying widths depending on how the columns of the tables are accessed (e.g., transactionally or analytically).

We focus on main memory systems because, like other researchers [11], we believe that many future databases – particularly those that involve enterprise entities like customers, outstanding orders, products, stock levels, and employees – will fit into the memory of a small number of machines.

Main memory systems present a unique set of challenges and opportunities. Due to the architecture of modern CPUs and their complex cache hierarchy, comparing the performance of different main memory layouts can be challenging.

Our system captures the idea that it is preferable to use narrow partitions for columns that are accessed as a part of analytical queries, as is done in pure columnar systems [4, 5]. In addition, HYRISE stores columns that are accessed in OLTP-style queries in wider partitions, to reduce cache misses when performing single row retrievals. Though others have noted the importance of cache locality in main memory systems [5, 2, 8, 12], we believe we are the first to build a dedicated hybrid database system based on a detailed model of cache performance in mixed OLAP/OLTP settings.

2 HYRISE Overview

In this section we provide an overview of HYRISE and its main components. The general architecture of HYRISE is depicted in Figure 1 and consists of the following main components: a unique hybrid storage engine, a hybrid layout manager and a query engine.

Hybrid Storage Engine The key concept of HYRISE is to natively support vertical partitioning of relational tables into disjoint sets of attributes. Each partition is internally represented as a *container*. Containers can in turn be freely composed to build the relational table. We built HYRISE to optimize data access to these containers. Traditionally, databases uses page structures to store data. Inside a page, tuples are stored one after the other. In contrast, tuples in a HYRISE containers are stored inside a single compressed block of main memory. Additionally, HYRISE uses dictionary compression to replace actual values with an encoded value id. The dictionaries are stored as sorted lists allowing offset-based mapping from encoded value to actual values. This solution has two advantages: First, the effect of each access to a container can be calculated independently of the actual value or value distribution. Second, expensive value-based comparisons, e.g. on string values, can be executed in the dictionary directly using efficient binary operations on integers. Due to the explicit ordering of value ids, HYRISE only materializes intermediate results as values when it is explicitly required to do so during query execution.

Modifications to a container are stored in an uncompressed delta partition to avoid having to re-compress or re-encode partitions when data is modified. To optimize memory consumption and query execution performance, data is cyclically re-compressed[9]. The logical structure of an attribute with a sorted main dictionary and the delta partition is shown in Figure 2. All tuples in HYRISE are stored following an insert-only approach. Deletes

and updates are transformed into operations setting an invalid attribute on the delta container. Depending on the chosen strategy, invalid tuples are either kept to allow time-travel operations[10] or they are simply removed during the re-compression phase.

Query Execution Query execution in HYRISE is flexible and allows choosing from different materialization and execution strategies. Initially, query plans in HYRISE are hand-coded operator graphs represented in JSON.

Parallelization of analytical queries is achieved by parallellizing the data flow of the query plan. Relational plan operators are split into smaller independent tasks that are then executed in parallel. Update operations are serialized per container, which allows us to reduce latch and lock costs for read-only queries. Intermediate results in HYRISE are modeled as temporary tables, which are either directly materialized or stored as position lists. The advantage of position lists is that expensive data copying can be avoided as long as possible[1].

Layout Manager The layout manager in HYRISE takes as input a sample workload, and returns as output a specification of the physical layout of the database. This specification is used to generate the containers that are stored in HYRISE, or to transform the layout of a previously loaded containers into a different representation.

Our HYRISE layout engine currently supports three different algorithms: a candidate based layout algorithm, a divide and conquer algorithm and an incremental layout algorithm. The first two algorithms are optimal, in the sense that they always find the best layout, while the divide and conquer algorithm is a scalable but approximate algorithm. We give more detail on our layout algorithms below in Section 3.2.

Workload Evolution Taking advantage of the layout manager described above, we were able to build a system that can easily adapt in case the workload changes. Our incremental layout algorithm allows us to compute a new optimal layout based on the delta between two separate workloads. With this incremental algorithm, our system can frequently recompute the layout and detect if a change of the layout is required, even for very large applications and frequently changing workloads.

3 Automated Logical Database Design

The design goal for our layout algorithm is to scale both with the number of attributes in tables of large enterprise applications and with the huge number of tables in these applications.

The number of possible hybrid physical designs (combinations of non- overlapping containers containing all of the columns) for a particular table is huge. For a table of n attributes, there exist $a(n)$ possible hybrid designs, where $a(n) = (2n-1)a(n-1) - (n-1)(n-2)a(n-2)$, where $a(0) = a(1) = 1$. There are for instance $3,535,017,524,403$ possible layouts for a table of 15 attributes. The only automated hybrid designer we are aware of, the HillClimb Data Morphing algorithm [8], does not work for wide tables in practice since it scales exponentially (2^n) with the number of attributes in both time and space. We propose a new set of algorithms that can efficiently determine the most appropriate physical design for tables of many tens or hundreds of attributes given a database and a query workload.

3.1 Layouts

A layout in HYRISE is defined by a set of containers that implicitly splits a relational table into a set of disjoint partitions $P_1, \ldots, P_n$. A single attribute can only occur in one and only one partition. While the attributes inside a single partition are ordered, a valid layout consists of an unordered set of partitions, such that such the layouts $\lambda_1 = \{(a_1, a_2), (a_3, a_4)\}$ and $\lambda_2 = \{(a_3, a_4), (a_1, a_2)\}$ are considered identical.

Formally, given a database DB and a workload W, our goal is to determine the list of layouts $\boldsymbol{\lambda_{opt}}$ minimizing the workload cost. The cost model we use in this case is based on cache misses and presented in more detail in [7].

3.2 Layout Selection

Based on our cost model, which allows us to estimate the number of cache misses occurred by an operation, we make two observations: First, a relational projection that retrieves $\pi.w$ bytes out of a $C.w$-byte container generally incurs an overhead that is caused by loading bytes into cache that are not read by the projection. This overhead is proportional to $C.w - \pi.w$ for full scans, and can vary for selections depending on the exact alignment of the projection and the cache lines. We call this overhead *container overhead* cost.

Second, when output tuples can be reconstructed without any cache eviction, the cost expression *distributes* over the set of queries, in the sense that the total access cost of a workload can be computed by summing the cost of each query individually on the partitions it accesses.

3.3 Candidate Based Layout Selection

Based on our model and the above observations, our layout algorithm works in three phases called *candidate generation*, *candidate merging*, and *layout generation* phases.

Candidate Generation The first phase of our layout algorithm determines all primary partitions for all participating tables. A primary partition is defined as the largest partition that does not incur any container overhead cost. For each relation $\mathcal{R}$, we start with the complete set of attributes $\{a_1, \ldots, a_m\}$ in $\mathcal{R}$. Each operation op_j implicitly splits this set of attributes into two subsets: the attributes that are accessed by the operation, and those that are ignored. The order in which we consider the operations does not matter in this context. By recursively splitting each set of attributes into subsets for each operation op_j, we end up with a set of $|P|$ *primary* partitions $\{P^1_1, \ldots, P^1_{|P|}\}$, each containing a set of attributes that are always accessed together. The cost of accessing a primary partition is independent of the order in which the attributes are laid out, since all attributes are always queried together in a primary partition.

Candidate Merging: The second phase of the algorithm inspects permutations of primary partitions to generate additional candidate partitions that may ultimately reduce the overall cost of the workload. Our cost model shows us that merging two primary partitions P^1_i and P^1_j is advantageous for wide, random access to attributes since corresponding tuple fragments are co-located inside the same partition; for projections, the merging process is usually detrimental due to the additional access overhead (which occurs unless both primary partitions are perfectly aligned to cache lines.)

This tension between reduced cost of random accesses and penalties for large scans of a few columns allows us to prune many of the potential candidate partitions by comparing the cost of the workload for the set of primary partitions with their merged counterpart. If the cost for the merged partitions is equal to or greater than the sum of the individual costs of the partitions (due to the container overhead), then this candidate partition can be discarded. If a candidate partition is not discarded by this pruning step, it is added to the current set of partitions and will be used to generate valid layouts in the following phase.

Layout Generation The third and last part of our algorithm generates the set of all valid layouts by exhaustively exploring all possible combinations of the partitions returned by the second phase. The algorithm evaluates the cost of each valid layout consisting of a covering but non-overlapping set of partitions, discarding all but the physical layout yielding the lowest cost. This last layout is the optimal layout according to our cost

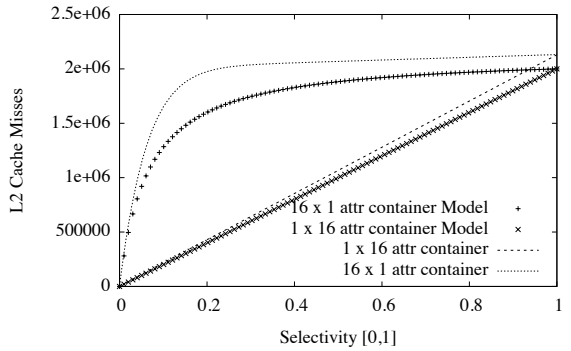


Figure 3: Comparing predicted and measured Level 2 cache misses for the selection operator with varying selectivity.

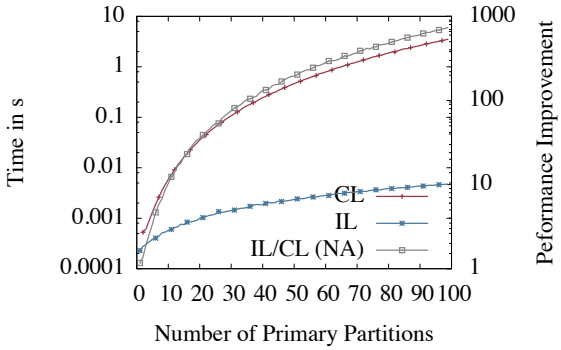


Figure 4: Comparing layout generation performance for Candidate Layouter (CL) and Incremental Layouter (IL)

model, since all potentially interesting permutations of attributes are examined by our algorithm (only irrelevant permutations, such as subsets of primary partitions or suboptimal merges from Section 3.3, are discarded).

The worst-case space complexity of our layout generation algorithm is exponential with the number of candidates partitions $|P|$. However, it performs very well in practice since very wide relations typically consist of a small number of sets of attributes that are frequently accessed together (thus, creating a small number of primary partitions) and since operations across those partitions are often relatively infrequent (thus drastically limiting the number of new partitions generated by the second phase above.)

3.4 Divide and Conquer Partitioning

For large relations and complex workloads involving hundreds of different frequently-posed queries, the running time of the above algorithm may still be high. To address this situation, HYRISE provides an approximate layout algorithm that clusters the primary partitions that are frequently co-accessed together. To retrieve the partitions that are accessed together, we build a symmetric affinity matrix that is transformed into a weighted graph. Based on the weights, we partition the graph in order to obtain a series of *min- cut* subgraphs to minimize the total cost (weight) of all edges that must be removed. We then determine the optimal layout for each of the subgraphs, which is computationally much lighter than considering the whole graph. The intermediate layouts of the subgraphs are used as candidates for the merging phase. Finally, we combine the partitions of the sub-layouts from the previous step to find a near-optimal overall layout.

4 Comparison and Performance Characteristics

To evaluate our system we used a mixed workload benchmark consisting out of 13 queries, including 9 transactional queries, and 4 analytical queries. Using our layout algorithm, we were quickly able to determine the optimal layout based on our accurate cache-miss cost model (Figure 3.) In total for the above scenario our hybrid partitioning is 4 times better than storing the data using rows and about 60% faster compared to storing the data fully decomposed as columns. The results of the individual queries as shown in [7] show a strong tension between the different layouts. It is important to mention that our algorithm optimizes for the best result of the overall observed workload and might sacrifice the performance of an individual query for this goal.

In addition, we evaluated the performance of our layout algorithm to validate its applicability in the area of enterprise applications with very wide schemas and many thousand tables. A table with 100 primary partitions, with hundreds of attributes, can be handled in under 10s, when we apply our incremental layout algorithm (Figure 4.)

5 Conclusions

In this paper, we gave an overview of HYRISE, a new hybrid in-memory storage engine we have built that can outperform both row and column-wise storage systems. To do this, HYRISE employs workload-aware layout algorithm that determines the optimal physical layout of a database and scales to large-scale of enterprise data. The overall goal of our work is to provide an application-specific storage layer for enterprise data. Based on further analyses of enterprise applications, we see are beginning to explore different storage containers that may not be strictly relational, but that also integrate semi-structured and unstructured data into one main-memory system.

References

[1] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, pages 466–475, 2007.

[2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180. Morgan Kaufmann, 2001.

[3] Anja Bog, Kai Sachs, Alexander Zeier, and Hasso Plattner. Normalization in a Mixed OLTP and OLAP Workload Scenario. In *Third TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC)*, 2011.

[4] Peter Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*, pages 54–65. Morgan Kaufmann, 1999.

[5] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[6] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. The mixed workload CH-benCHmark. In *DBTest*, page 8. ACM, 2011.

[7] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[8] Richard A. Hankins and Jignesh M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, pages 417–428, 2003.

[9] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB*, 5(1):61–72, 2011.

[10] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD*, pages 1–2. ACM, 2009.

[11] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, pages 1150–1160. ACM, 2007.

[12] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Workshop on Data Management on New Hardware*, pages 47–54. ACM, 2008.

29th IEEE International Conference on

DATA ENGINEERING

BRISBANE, AUSTRALIA | 8 – 11 APRIL 2013

A leading forum for researchers, practitioners, developers and users to explore cutting-edge ideas and to exchange techniques, tools and experiences.

GENERAL CHAIRS

Rao Kotagiri *The University of Melbourne, Australia*

Beng Chin Ooi *National University of Singapore, Singapore*

PROGRAM COMMITTEE CHAIRS

Christian S. Jensen *Aarhus University, Denmark*

Chris Jermaine *Rice University, USA*

Xiaofang Zhou *The University of Queensland, Australia*

IMPORTANT DATES

Research & Industry Papers
Abstracts due
JULY 16 2012

Full paper submissions due
JULY 23 2012

Notification to authors
OCTOBER 14 2012

Final versions due
NOVEMBER 30 2012

www.icde2013.org

IEEE computer society

icde 2013

29th IEEE International Conference on
DATA ENGINEERING
BRISBANE, AUSTRALIA | 8 – 11 APRIL 2013

Images courtesy of Brisbane Marketing and Tourism Australia

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903