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

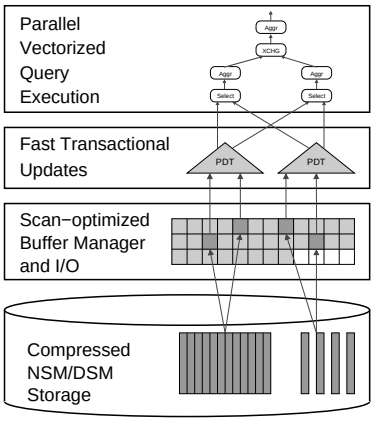

Figure 1: A simplified architecture of the Vectorwise kernel.

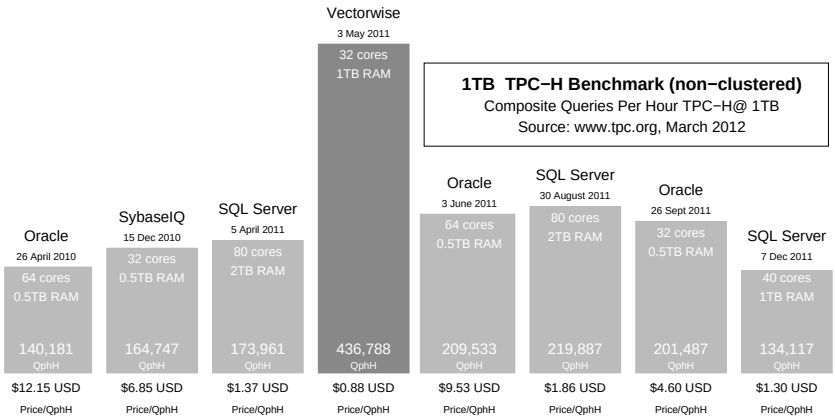

Figure 2: Latest official 1TB TPC-H performance results (non-clustered), in publication order, as of March 18, 2012.

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.