Introduction to Column-Oriented Database Systems (Nov 2012)

S.G.Yaman
Univ. Helsinki, Helsinki, Finland
sezin.yaman@helsinki.fi

Abstract— Column-oriented database systems, also known as column-stores, has an important demand in the past few years. Basically, it is about storing each database column separately so that the attributes belonging to the same column would be stored contiguously, compressed and densely-packed in the disk. This method has advantages in reading the records faster as compared to classical row-stores in which every row are stored one after another in the disk. In this paper, after understanding of what are column store technologies, its benefits, potential usages and applications; the following questions will be answered: What are the differences between column stores and row stores? How would be their performance compared both at storage and query execution level [1], [2]? Can a classical-row-based system achieve the performance of a column store system? While addressing our questions, we will go into the MonetDB [3], [4] open source column-store management system in detail.

***Index Terms*— Database Systems, C-store, Column-store, Column-oriented DBMS, invisible join, compression, materialization, MonetDB.**

I. Introduction

The roots of column-oriented database systems can be seen beginning from 1970s, but it was not until 2000s that some researches and applications started to be done. In the last recent years some column store databases namely MonetDB [3], [4] and C-Store [5] has been introduced by their authors, with the claim that their performance gains are quite noticeable against traditional approaches. These traditional approaches are for row-oriented database systems that have physical designs such that almost all tables in the database have one-to-one mappings to the tables in the logical schema. One of the important points that it will be looked in this paper is to figure out if the gains of column-stores are due to their own internal physical design or if the same can be achieved using some column-oriented design in a row-stores database.

MonetDB [3], [4] is a column-oriented open source database management system for high-performance applications in data mining, business intelligence, scientific databases, XML query and text and multimedia retrieval. It focuses on exploiting column-oriented data processing as a way to improve the computational efficiency of database engines. This system will be talked about more in Section II and VI.

To understand better the internal structure of column-stores, it can be observed that the attempt of simulating column-stores inside row-stores [2]. Star Schema Benchmark (SSBM) [6], [7] is a recently proposed data warehousing benchmark that has been implemented with column-oriented internal design as possible, in addition to some traditional designs. Column-oriented approach techniques which are used in SSBM (vertical partitioning, index-only plans and materialization) will be explained in Section III. This system is taken as an example and examined to address our questions.

After observing performance of the open source C-store database [5], which is a column-store, on SSBM; it will be shown that although the internal structure of a column store is emulated inside a row store, the query processing performance is quite poor. This observation raises the question of which kind of optimization techniques can be used to improve the performance of column-stores over row-stores on warehouse workloads. Four optimization techniques are defined which are specific to column-oriented database management systems [DBMS] as late materialization [4], [8], block iteration [8], compression [9] and invisible joins (a new technique); and they will be discussed in Section IV.

The rest of this paper is organized as follows: first, some background information, prior works, definitions related to column-stores and a brief description of an application system (Section II); then the physical database design methods used in row-oriented databases will be described (Section III); physical design and query execution techniques used in column-oriented databases will be covered (Section IV); and finally, the performance of row-stores and column-stores will be compared based on the experiments (Section V).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

II. BACKGROUND AND PRIOR WORK

In a column-store database, each column is stored contiguously on a separate location on a disk. The values stored in the columns are densely packed and compressed to improve read efficiency. Column-store databases perform faster than traditional database systems, since they are more I/O efficient for read-only queries. In that manner, column-scanners are different from row-scanners, since column-scanners translate value position information into disk locations and they combine and reconstruct the tuples from different columns.

Column-store systems include column-oriented physical design, and it is observed that due to superior CPU and cache performance (in addition to reduced I/O), they can perform better compared to commercial and open sources row-store databases on benchmarks. Additionally, they include optimizations for direct operation on compressed data [9].

As it is said before, SSBM [6], [7] will be used as an example to compare the performance of column-store and the commercial row-store.

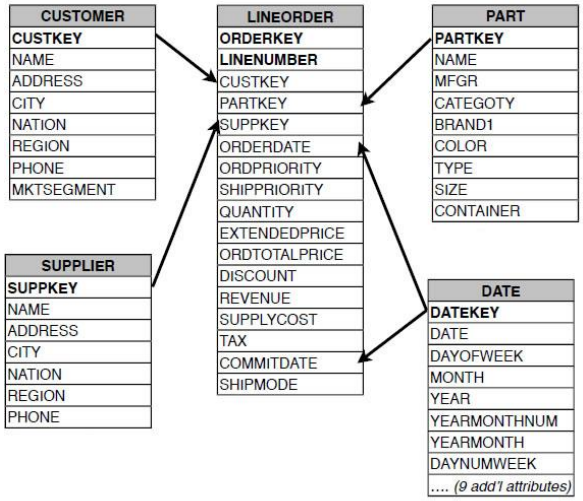


Figure 1: SSBM Schema Benchmark

Schema

As can be seen in the Figure 1, SSBM benchmark consists five tables in total. The fact table *LINEORDER* includes information about individual orders with *ORDERKEY* and *LINEORDER* composite primary keys. It also includes foreign keys from the other tables which are *CUSTOMER*, *SUPPLIER*, *PART* and *DATE*. The dimension tables contain information about their entities in the expected way.

Additionally, SSBM contains thirteen queries in four categories (flights) which we will explore some of them later in this paper to be used in the examples.

To have more idea about the applications of column-oriented approach, the system MonetDb is chosen as an example application. MonetDB was designed especially for data warehouses where large databases exist. It is observed that MonetDB is faster than traditional databases by the way of innovations at all layers, for instance, a storage based model on vertical fragmentation (column-stores). Basically, MonetDB exploits the large-main memories of computers in an efficient and effective way during the query processing, while the database is stored on the disk. It focuses on data workloads which are mostly read-dominated and where updates are mostly consisting large data-blocks to be added to the database at a time.

MonetDB is one of the first DBMSs which is publicly available and based on column-store technology.

MonetDB Physical Data Model

The storage model can be seen as a deviation of a traditional database system. The difference is that instead of storing the attributes as row-stores, MonetDB uses column-stores to represent the relational tables in the database. Each column is stored in a separate (surrogate, value) table, called BAT (binary association table). The column *surrogate* or *OID* (object identifier) is intended to identify the relational tuple which that attribute value belongs to. They are essantially indications of the positions, all attributes of the same tuple have the same OIDs. The column *value* is intended to keep the actual attribute values. In this way, for a relation R with k attributes, there would be k BATs; each BATs having the respective attributes as (surrogate, value) pairs. Positions are determined by insertion order of tuples and this technique helps that column-stores perform tuple reconstructions efficiently in case tuple order-preserving operators exist.

III. ROW-ORIENTED EXECUTION

In this section, the implementation of column-stores in a row-stored based system is being looked at. Three techniques are being introduced:

Vertical Partitioning

It is a basic technique to emulate a column-store in a row-store by partitioning each relation vertically [10]. Basically, an integer *position* is added to each column, to be used to connect the fields from the same row together. It ends up with more tables (one column for values, one column for positions) with fewer columns. In this way, only the necessary columns need to be read to answer a query.

Index-only Plans

Since vertical partitioning may cause wasting of space due to extra position attribute, index-only plans can be seen as alternatives. Indexes are added to each column of every table and collection of all the indices are created so that it is possible to answer a query without ever going to underlying row-oriented tables. The plan works by setting lists of (surrogate, value) pairs which satisfy the predicates on each table and keeping these lists in the memory in case there are multiple predicates on the same table.

Materialized Views

In this technique, there is a view with exactly the columns needed to answer to every query in the benchmark. In our example, the aim is to create an optimal set of materialized views for each flight (query category in SSBM schema) where each view is responsible for having the required columns to answer the queries in that flight.

IV. COLUMN_ORIENTED EXECUTION

In this section, three different optimization approaches to improve the performance of column-stores are being looked at and a new technique named *invisible join* is being introduced.

Compression

One of the most often cited advantages of column-stores can be said as data compression. Intuitively, column-stores' data are more compressible compared to row-stores. For instance, the attribute *phone number* of customers can be compressed using run-length encoding, where a sequence of repeated values is rewritten by a count and the value. Then, they can be stored together. Compression clearly reduces the space occupied in the disk; but additionally since the data is compressed, less time is being spent in I/O while the data is being read from disk to memory. The biggest difference between compression in column-stores and row-stores lies where a column is sorted and there are consecutive repeats of the same values in a column. In column-stores it is easier to summarize these value repeats whereas in row-stores it is a more complicated process. Consequently, compression has a huge impact on query performance if a high percentage of columns are being accessed by a query.

Late Materialization

In row-stores, information about the logical entities co-located in a single row of table in the memory. On the other hand, in column-stores, information about the entities is stored in different locations on the disk since different attribute values are stored in different columns. For instance, the attributes *name*, *e-mail*, *addres*, *phone number* etc. are all in the separate columns. However, queries usually need to retrieve information from more than one column of an entity. Thus, data from the multiple columns should be gathered together into rows' of information. That's why join-like materialization is needed in a column-store.

Naïve column-stores tend to use early materialization which leaves much of the potential benefits of column-stores as unrealized, whereas some recent column-stores named X100 and C-Store [5] prefers to use late materialization, by keeping the data in columns until much later into query plan and operation the data on these columns. In order to do it, *position* lists need to be constructed to establish the connection between the operations that being used on different columns. It is observed that the advantages of this technique are four-fold [2].

Block Iteration

Row-stores iterate through each tuple first, and then they extract the needed attributes. This leads to tuple-at-a-time processing where 1-2 function calls are needed to achieve wanted data. Some of per-tuple overhead can be reduced if blocks of tuples are available at once and called by a single function call [14], [15]. In all column-stores, data blocks of the same columns are sent to an operator in a single function call. This technique increases a column-store's ability to process the column values in a sequence in order to send large column blocks to a CPU at one time. Thus, no attribute extraction is needed and if the column is of a fixed width, the DBMS can treat the operation as a simple array lookup. It is observed that this technique improves the performance by fifty percent compared to a row-store which will be explained in Section V.

Invisible Join

This new technique will be introduced with the query (query 3.1 of SSBM benchmark) below, which basically finds total revenue from the customers who live in Asia and who buy a product from an Asian supplier between the years 1992 and 1997.

```
SELECT c.nation, s.nation, d.year,
       sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo,
     supplier AS s, dwdate AS d
WHERE lo.custkey = c.custkey
  AND lo.suppkey = s.suppkey
  AND lo.orderdate = d.datekey
  AND c.region =  ASIA
  AND s.region =  ASIA
  AND d.year >= 1992 and d.year <= 1997
GROUP BY c.nation, s.nation, d.year
ORDER BY d.year asc, revenue desc;
```

Besides from a traditional way to execute this query, which would be pipeline joins in order of predicate selectivity, a new method is introduced which works by rewriting joins into predicates, which can be evaluated by a hash lookup, on the foreign key columns in the fact table.

It has three phases. First, each predicate is performed on the relevant dimension table to retrieve the list of table keys. As being seen in Figure 2, we get relevant keys belonging to each dimension table applying the predicates.

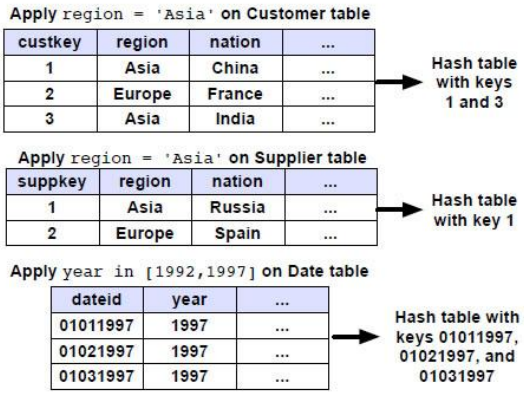


Figure 2: The first phase of Invisible Join for executing the query 3.1 of SSBM

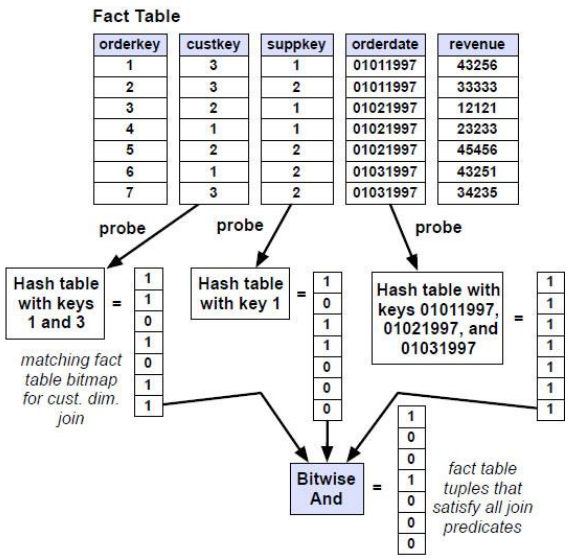


Figure 3: The second phase of Invisible Join for executing the query 3.1 of SSBM

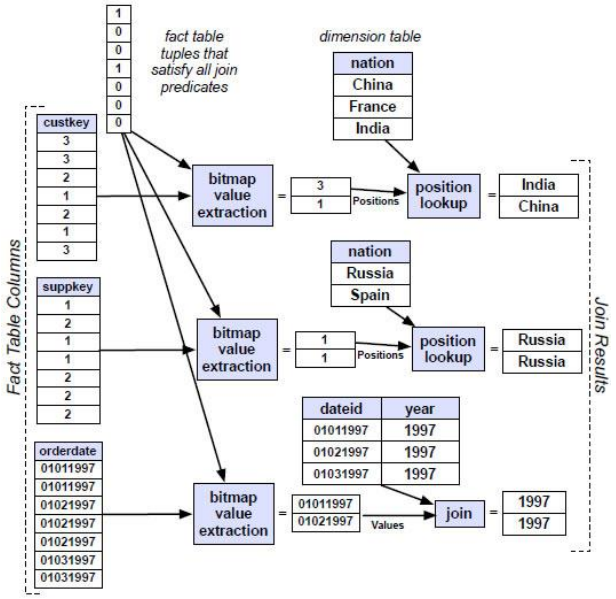


Figure 4: The third phase of Invisible Join for executing the query 3.1 of SSBM

In the second phase, each has table with the relevant keys is used to retrieve the positions of the records in the fact table satisfying the corresponding predicate. As it can be seen in Figure 3, hash table are probed with the foreign key-columns of the fact table and it returns the positions representing the fields satisfying the predicate. Then, it returns the intersection of our tables using bitwise and.

The final phase uses the final position table for each column of the fact tables which has a foreign key reference to the dimension tables, to get answer to the query. As can be seen in Figure 4, each foreign key value from fact tables are extracted using our position table values and then information is extracted from the needed dimension tables using these positions.

As a result, it can be said that since the number of positions in the position table is dependent on the selectivity of the whole query (not just a part of it), the necessary number of extraction of values is minimized and this improves the performance.

V. EXPERIMENTS

In this section, first, different attempts of emulating a column-store in a row-store comparing to C-Store's baseline performance is being examined. Then, it is being searched that whether there is a possibility for an unmodified row-store to get the benefits of column-oriented design. Finally, all the optimizations proposed for column-stores are being considered to find out which ones are the most significant.

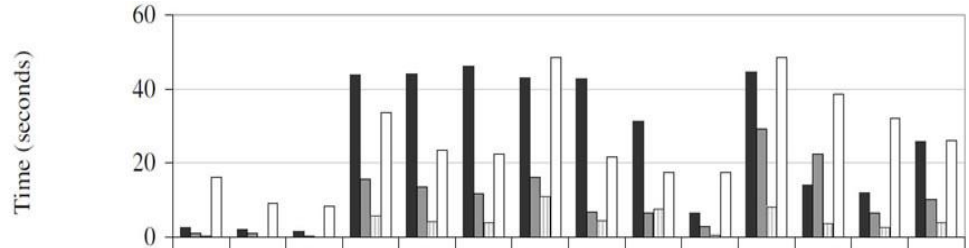
Motivation

Figure 5 compares the performances of C-store (column-store) and System X (row-store) on SSBM. (It should be said that beyond the basic difference of columns vs. rows, there are some implementation differences between these two systems that can affect the numbers.) In the figure, RS is being used for System X whereas CS is being used for C- store; and (-MV) indicates optimal collection of materialized views. We can see that C-store is six times better than System X in the base case; and three times better when System X is using materialized view. It shows that column-stores perform better than row-stores on data workloads. However, looking at the last row in the figure, the case where row-oriented materialized view data in C-store is stored, it is observed that System X numbers are faster than C-store numbers. This difference can be explained by that C-Store has not implemented some advanced performance features that are already available in System X, such as partitioning. System X can partition each material view optimally for the query flight that is designed for and partitioning improves the performance by reducing the data that needs to be scanned.

In order to understand the performance difference in these two systems, two additional experiments' results will be observed where a column-stored is simulated in a row-store and where column-oriented optimizations are removed from the column-store until it starts to simulate a row-store.

Column-Store Simulation in a Row-Store

By this experiment, the performance of different configurations of System X on SSBM is being described. For the base case, partitioning is used since it is known that it improves the performance in a row-store. Five different configurations of System X are set up as: traditional row-oriented representation, traditional bitmap approach (similar to traditional but with plans biased to use bitmaps), vertical partitioning approach, index-only representation and materialized views approach. As it can be seen in Figure 6,



	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2	3.3	3.4	4.1	4.2	4.3	AVG
RS	2.7	2.0	1.5	43.8	44.1	46.0	43.0	42.8	31.2	6.5	44.4	14.1	12.2	25.7
RS (MV)	1.0	1.0	0.2	15.5	13.5	11.8	16.1	6.9	6.4	3.0	29.2	22.4	6.4	10.2
CS	0.4	0.1	0.1	5.7	4.2	3.9	11.0	4.4	7.6	0.6	8.2	3.7	2.6	4.0
CS (Row-MV)	16.0	9.1	8.4	33.5	23.5	22.3	48.5	21.5	17.6	17.4	48.6	38.4	32.1	25.9

Figure 5: Baseline performance of C-store and System X.

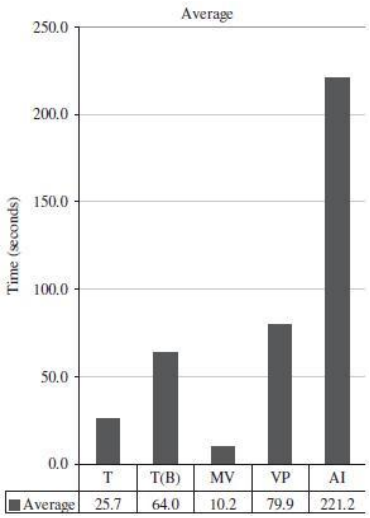
materialized views performs best since they read minimal amount of data needed by a query. Then, traditional approaches are the best ones. Especially the traditional row-oriented representation is almost three times faster than the attempt of emulating a column-oriented approach. The reasons why columnar approached are being limited can be explained by tuple overheads and inefficient tuple reconstruction. Tuple overheads can be quite large in a row-store with a fully vertically partitioned approach. Tuple reconstruction is also problematic since data belongs to one entity is stored in different locations on the disk, yet most queries need to access more than one attribute of an entity.

As a result, none of the attempts to emulate a column-store in a row-store is totally effective. Vertical partitioning could be effective if it is applied on a few columns, but otherwise due to tuple overheads and construction problems, it is not really competitive with traditional approaches. Index-only plans has a lower per-record overhead, but they force the system to use expensive hash joins with join columns of the fact table and this leads the reduce system performance.

Materialized views have good performance since they allow the System X to read just a subset of the fact table needed without merging the columns.

Column-Store Performance

As it was seen in Figure 5 and 6, the average query time in C-Store on SSBM (4.0 sec) is much faster than not only the column-store simulation in a row-store (80 to 220 sec), but also than the best scenario where the queries are known in advance and the row-store has created materialized views tailored for the query plans (10.2 sec). Now, an additional experiment will be discussed to find out why column-store are even faster than materialized view case or the CS Row-MV case. In this experiment, to learn about the performance effects of optimization techniques in column-stores, these techniques are being removed from the column-store until it starts to simulate a row-store.



	T	T(B)	MV	VP	AI
Average	25.7	64.0	10.2	79.9	221.2

Figure 6: Average performance across all the queries. Here, T is traditional, T(B) is traditional bitmap, MV is materialized views, VP is vertical partitioning and AI is all indexes.

It was already described that column-oriented optimizations (compression, late materialization, block iteration and invisible join) are used to improve the performance of column-oriented databases. Presumably, these optimizations make the performance difference between the column-store and the row-oriented materialized views cases from Figure 5. To verify this presumption, these optimizations are being removed from C-store and results are being observed at each step. Figure 7 shows the results of removing optimizations successively from C-Store.

Block-processing can improve performance anywhere from a factor from 5% to 50% depending on if the compression technique is already removed [2]. It can be seen that invisible joins can improve the performance from the factor 50% to 75%. Compression can improve the performance by almost the factor of two and late materialization can improve the

performance by almost the factor of three. Thus, it can be though that the most significant optimizations are compression and late materialization.

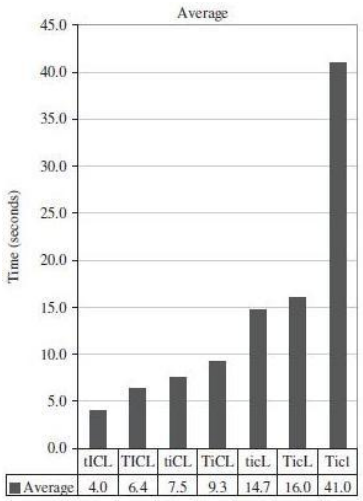


Figure 7: Average performance numbers for C-Store across all queries while various optimizations removed. Here, T= tuple-at-a-time processing, t= block processing; I=invisible join enabled, i= disabled; C= compression enabled, c= disabled; L= late materialization enabled, l= disabled.

It is being observed that after removing these optimizations from the column-store, it begins to behave like row-store (only with the difference that the necessary tuple-construction at the beginning of the query plans).

I. Conclusion

In the paper, column-oriented database approach has been mentioned. The performance of column-stores has been compared to row-stores. The techniques of emulating the column-stores in row-stores, such as vertical partitioning, index-only plans and materialized view, has been shown and it has been observed that these emulation plans did not end up with good performance results. The reasons of why column-stores execute the column-oriented data more efficiently have been look at. A new technique named invisible join which is proposed to be used to improve performance has been introduced.

A successful column-oriented system needs some system improvements such as record ids, fast merge joins of sorted data, run-length encoding and column-oriented query execution techniques (compression, late materialization, block iteration and invisible joins). Although some of these improvements are implemented in different row-stores [11], [12], [13], [14], to build a complete row-store that can transfer into column-store where column-stores perform well is still a problematic area needs to be discovered.

One of the most important examples of column-oriented database systems is MonetDB. It has been mentioned that its mainly target area is data intensive applications over massive amounts of data such as scientific databases and its physical structure has been explained.

References

[1] D. J. Abadi, P.A. Boncz, S. Harizopoulos. Column-oriented database systems. In *VLDB*, 2009.

[2] D.J. Abadi, S.R. Madden, N. Hachem. Column-stores vs. row-stores: how different are they really? In *Proc. SIGMOD*, 2008.

[3] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, M. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Artitectures. 2012.

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

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

[6] P. E. O'Neil, X. Chen, E. J. O'Neil. Adjoined Dimension Column Index (ADC Index) to Improve Star Schema Query Performance. In ICDE, 2008

[7] P. E. O'Neil, E. J. O'Neil, X. Chen. The Star Schema Benchmark (SSB). http: //www.cs.umb.edu/↑ poneil/StarSchemaB.PDF.

[8] M. Zukowski, P. A. Boncz, N. Nes, S. Heman. MonetDB/X100 - A DBMS in the CPU Cache. IEEE Data Engineering Bulletin, 28(2):17–22, June 2005.

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

[10] S. Khoshafian, G. Copeland, T. Jagodis, H. Boral, P. Valduriez. A query processing strategy for the decomposed storage model. In ICDE, pages 636–643, 1987.

[11] G. Graefe. Efficient columnar storage in b-trees. SIGMOD Rec., 36(1):3–6, 2007.

[12] A. Halverson, J. L. Beckmann, J. F. Naughton, D. J. Dewitt. A Comparison of C-Store and Row-Store in a Common Framework. Technical Report TR1570, University of Wisconsin-Madison, 2006.

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

[14] J. Zhou, K. A. Ross. Buffering databse operations for enhanced instruction cache performance. In SIGMOD, pages 191–202, 2004.

[15] S. Harizopoulos, V. Shkapenyuk and A. Ailamaki. QPipe: a simultaneously pipelined relational query engine. In SIGMOD, pages 383–394, 2005.