ISSN No. 0976-5697

Volume 3, No. 3, May-June 2012

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Acumen of Column-Store Architecture Approaches and Challenges

[1]Sanil.S.Nair
Assistant Professor (Department of MCA)
Prof. Ram Meghe Institute of Technology &Research,
Badnera- Amravati, India
sanils81@rediffmail.com

[2]Bhruthari G. Pund
Assistant Professor (Department of MCA)
Prof. Ram Meghe Institute of Technology &Research,
Badnera- Amravati, India
bgpund@yahoo.com

[3]Prajakta P. Deshmukh
Assistant Professor (Department of MCA)
Prof. Ram Meghe Institute of Technology &Research,
Badnera- Amravati, India
prajakt.deshmukh@gmail.com

Abstract: This paper provides (to the best of our knowledge) the detailed study of multiple implementation approaches of C-Store systems, categorizing the different approaches into three broad categories, and evaluating the tradeoffs between approaches. Here we investigate the challenges of building a column-oriented database system by exploring these three approaches in more detail. We implemented each of these three approaches and examined their relative performance on a data warehousing benchmark.

Keywords: Column oriented database, c-store, Query execution plans, and row stores

I. INTRODUCTION

In this paper, we have tried to mention the challenges of building a column-oriented database system by exploring these three approaches in more detail. We implement each of these three approaches and examine their relative performance on a data warehousing benchmark. Clearly, the more one tailors a database system for a particular data layout, the better one would expect that system to perform. Thus, we expect the third approach to outperform the second approach and the second approach to outperform the first approach. For this reason, we are more interested in the magnitude of difference between the three approaches rather than just the relative ordering. For example, if the first approach only slightly underperforms the other two approaches, then it would be the desirable solution for building a column-store since it can be built using currently available database systems without modification.

Consequently, we carefully investigated the first approach. We experiment with multiple schemes for implementing a column-store on top of a row-store, including:

a. Vertically partitioning the tables in the system into a collection of two-column tables consisting of (table key, attribute) pairs, so that only the necessary columns need to be read to answer a query;
b. Using index-only plans; by creating a collection of indices that cover all of the columns used in a query; it is possible for the database system to answer a query without ever going to the underlying (row-oriented) tables;
c. Using a collection of materialized views such that there is a view with exactly the columns needed to answer every query in the benchmark. Though this approach uses a lot of space, it is the 'best case' for a row-store, and provides a useful point of comparison to a column-store implementation.

We implement each of these schemes on top of a commercial row-store, and compare the schemes with baseline performance of the row-store. Overall, the results are surprisingly poor - in many cases the baseline row-store outperforms the column-store implementations. We analyse why this is the case, breaking down the fundamental from the implementation specific reasons for the poor performance.

II. ROW-ORIENTED EXECUTION

In this section, we discuss several different techniques that can be used to implement a column-database design in a commercial row-oriented DBMS (since we cannot name the system we used due to license restrictions, hereafter we will refer to it as System). We look at three different classes of physical design: a fully vertically partitioned design, an "index only" design, and a materialized view design. In our evaluation, we also compare against a "standard" row-store design with one physical table per relation [15].Vertical Partitioning: The most straightforward way to emulate a column-store approach in a row-store is to fully vertically partition each relation [12]. In a fully vertically partitioned approach, some mechanism is needed to connect fields from the same row together (column stores typically match up records implicitly by storing columns in the same order, but such optimizations are not available in a row store). To accomplish this, the simplest approach is to add an integer "position" column to every table - this is often preferable to using the primary key because primary keys can be large and are sometimes composite. This approach creates one physical table for each column in the logical schema, where the ith table has two columns, one with values from column i of the logical schema and one with the corresponding value in the position column. Queries are then rewritten to perform joins on the position attribute when fetching multiple columns from the same relation. In our implementation, by default, System chose to use hash joins for this purpose, which proved to be expensive. For that reason, we experimented with adding clustered indices on the position column of every table, and forced System to use index joins, but this did not improve performance - the additional I/Os incurred by index accesses made them slower than hash joins.

A. Index-only plans:

The vertical partitioning approach has two problems. First, it requires the position attribute to be stored in every column, which wastes space and disk bandwidth. Second, most row-stores store a relatively large header on every tuple, which further wastes space (column stores typically - or perhaps even by definition - store headers in separate columns to avoid these overheads). To ameliorate these concerns, the second approach we consider uses index-only plans, where base relations are stored using a standard, row-oriented design, but an additional unclustered B+Tree index is added on every column of every table. Index-only plans - which require special support from the database, but are implemented by System- work by building lists of (record-id,value) pairs that satisfy predicates on each table, and merging these rid-lists in memory when there are multiple predicates on the same table. When required fields have no predicates, a list of all (record-id, value) pairs from the column can be produced. Such plans never access the actual tuples on disk. Though indices still explicitly store rids, they do not store duplicate column values, and they typically have a lower per-tuple overhead than the headers in the vertical partitioning approach.

One problem with the index-only approach is that if a column has no predicate on it, the index-only approach requires the index to be scanned to extract the needed values, which can be slower than scanning a heap file (as would occur in the vertical partitioning approach.) Hence, an optimization to the index-only approach is to create indices with composite keys, where the secondary keys are from predicate-less columns. For example, consider the query SELECT AVG (salary) FROM emp WHERE age>40 - if we have a composite index with an (age, salary) key, then we can answer this query directly from this index. If we have separate indices on (age) and (salary), an index only plan will have to find record-ids corresponding to records with satisfying ages and then merge this with the complete list of (record-id, salary) pairs extracted from the (salary) index, which will be much slower. We use this optimization in our implementation by storing the primary key of each dimension table as a secondary sort attribute on the indices over the attributes of that dimension table. In this way, we can efficiently access the primary key values of the dimension that need to be joined with the fact table.

B. Materialized Views:

The third approach we consider uses materialized views. In this approach, we create an optimal set of materialized views for every query flight in the workload, where the optimal view for a givenflight has only the columns needed to answer queries in thatflight. We do not pre -join columns from different tables in these views. Our objective with this strategy is to allow System to access just the data it needs from disk, avoiding the overheads of explicitly storing record-id or positions, and storing tuple headers just once per tuple. Hence, we expect it to perform better than the other two approaches, although it does require the query workload to be known in advance, making it practical only in limited situations.

C. Tuple overheads:

As others have observed [12], one of the problems with a fully vertically partitioned approach in a row-store is that tuple overheads can be quite large. This is further aggravated by the requirement that the primary keys of each table be stored with each column to allow tuples to be reconstructed. We compared the sizes of column-tables in our vertical partitioning approach to the sizes of the traditional row store, and found that a single column-table from our SSBM scale 10 lineorder table (with 60 million tuples) requires between 0.7 and 1.1 GBytes of data after compression to store - this represents about 8 bytes of overhead per row, plus about 4 bytes each for the primary key and the column attribute, depending on the column and the extent to which compression is effective (16 bytes × 6 × 107 tuples = 960 MB). In contrast, the entire 17 column line order table in the traditional approach occupies about 6 GBytes decompressed, or 4 GBytes compressed, meaning that scanning just four of the columns in the vertical partitioning approach will take as long as scanning the entire fact table in the traditional approach [8, 9, 10].

D. Column Joins:

Merging two columns from the same table together requires a join operation. System favors using hash-joins for these operations, which is quite slow. We experimented with forcing System to use index nested loops and merge joins, but found that this did not improve performance because index accesses had high overhead and System was unable to skip the sort preceding the merge join.

III. EXPERIMENTS

Now that we have described the techniques we used to implement a column-database design inside System, we present our experimental results of the relative performance of these techniques [3]. We first begin by describing the benchmark we used for these experiments, and then present the results.

All of our experiments were run on a 2.8 GHz single processor, dual core Pentium(R) D workstation with 3 GB of RAM running RedHat Enterprise Linux 5. The machine has a 4-disk array, managed as a single logical volume with files striped across it. Typical I/O throughput is 40 - 50 MB/sec/disk, or 160 - 200 MB/sec in aggregate for striped files. The numbers we report are the average of several runs, and are based on a "warm" bufer pool (in practice, we found that this yielded about a 30% performance increase for the systems we experiment with; the gain is not particularly dramatic because the amount of data read by each query exceeds the size of the bufer pool).

IV. STAR SCHEMA BENCHMARK

For these experiments, we use the Star Schema Benchmark (SSBM) [13, 14] to compare the performance of of the various column-stores.

The SSBM is a data warehousing benchmark derived from TPC-H [7]. Unlike TPC-H, it is a pure, textbook star-schema (the "best practices" data organization for data warehouses). It also consists of fewer queries than TPC- H and has less stringent requirements on what forms of tuning are and are not allowed. We chose it because it is easier to implement than TPC-H and because we want to compare our results on the commercial row-store with our various hand-built column-stores which are unable to run the entire TPC-H benchmark.

A. Schema:

The benchmark consists of a single fact table, the LINEORDER table that combines the LINEITEM and ORDERS table of TPC-H. This is a 17 column table with information about individual orders, with a composite primary key consisting of the ORDERKEY and LINENUMBER attributes. Other attributes in the LINEORDER table include foreign key references to the CUSTOMER, PART, SUPPLIER, and DATE tables (for both the order date and commit date), as well as attributes of each order, including its priority, quantity, price, discount, and other attributes. The dimension tables contain information about their respective entities in the expected way. Figure 1 (adapted from Figure 2 of [14]) shows the schema of the tables. As with TPC-H, there is a base "scale factor" which can be used to scale the size of the benchmark. The sizes of each of the tables are defined relative to this scale factor. In this paper, we use a scale factor of 10.

B. Queries:

The SSBM consists of thirteen queries divided into four categories, or "flights". The four query flights are summarized here:

a. Flight 1 contains 3 queries. Queries have a restriction on 1 dimension attribute, as well as the DISCOUNT and QUANTITY columns of the LINEORDER table. Queries measure the gain in revenue (the product of EXTENDED- PRICE and DISCOUNT) that would be achieved if various levels of discount were eliminated for various order quantities in a given year. The LINEORDER selectivities (percentage of tuples that pass all predicates) for the three queries are 1.9 × 10–2, 6.5 × 10–4, and 7.5 × 10–5, respectively.
b. Flight 2 contains 3 queries. Queries have a restriction on 2 dimension attributes and compute the revenue for particular product classes in particular regions, grouped by product class and year. The LINEORDER selectivity’s for the three queries are 8.0 × 10–3, 1.6 × 10–3, and 2.0 × 10–4, respectively.
c. Flight 3 consists of 4 queries, with a restriction on 3 dimensions. Queries compute the revenue in a particular region over a time period, grouped by customer nation, supplier nation, and year. The LINEORDER selectivity’s for the four queries are 3.4 × 10–2, 1.4 × 10–3, 5.5 × 10−5, and 7.6 × 10−7 respectively.
d. Flight 4 consists of three queries. Queries restrict on three dimension columns, and compute profit (REVENUE - SUPPLYCOST) grouped by year, nation, and category for query 1; and for queries 2 and 3, region and category. The LINEORDER selectivity’s for the three queries are 1.6 × 10–2, 4.5 × 10–3, and 9.1 × 10–5, respectively [6].

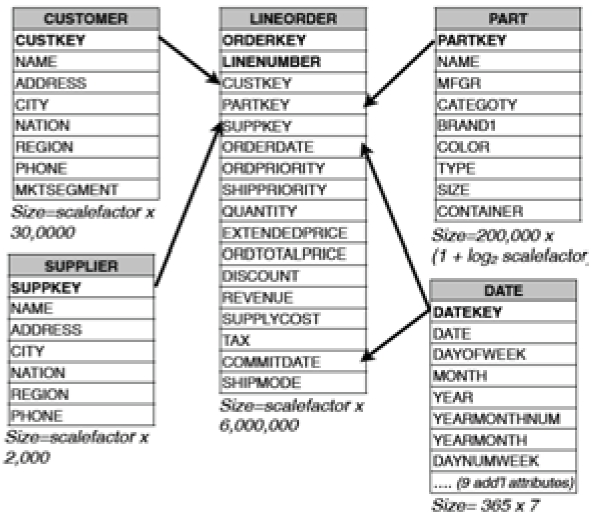


Figure 1. Schema of the SSBM Benchmark

V. IMPLEMENTING A COLUMN-STORE IN A ROW-STORE

We now describe the performance of the different configurations of System on the SSBM. We configured System to partition the lineorder table on orderdate by year (this means that a different physical partition is created for tuples from each year in the database). This partitioning substantially speeds up SSBM queries that involve a predicate on orderdate. Unfortunately, for the column-oriented representations, System doesn’t allow us to partition two-column vertical partitions on orderdate, which means that for those query flights that restrict on the orderdate column, the column-oriented approaches look particularly bad. Nevertheless, A "materialized views" approach with the optimal collection of materialized views for every query (no pre-joins were performed in these views).we decided to use partitioning for the base case because it is in fact the strategy that a database administrator would use when trying to improve the performance of these queries on a row-store, so is important for providing a fair comparison between System and other column-stores.

Other relevant configuration parameters for System include: 32 KB disk pages, a 1.5 GB maximum memory for sorts, joins, intermediate results, and a 500 MB bufer pool. We enabled compression and sequential scan pre-fetching.

We experimented with six configurations of System on SSBM:

a. A "traditional" row-oriented representation; here, we allow System to use bitmaps if its optimizer determines they are beneficial.
b. A "traditional (bitmap)" approach, similar to traditional,

but in this case, we biased plans to use bitmaps, sometimes causing them to produce inferior plans to the pure traditional approach.

c. A "vertical partitioning" approach, with each column in its own relation, along with the primary key of the original relation.
d. An "index-only" representation, using an unclustered B+tree on each column in the row-oriented approach, and then answering queries by reading values directly from the indexes.
e. A "materialized views" approach with the optimal collection of materialized views for every query (no pre-joins were performed in these views).

The average results across all queries are shown in Figure 2, with detailed results broken down by flight in Figure 3. Materialized views perform best in all cases, because they read the minimal amount of data required to process a query. After materialized views [2], the traditional approach or the traditional approach with bitmap indexing, is usually the best choice (on average, the traditional approach is about three times better than the best of our attempts to emulate a column-oriented approach). This is particularly true of queries that can exploit partitioning on orderdate, as discussed above. For query flight 2 (which does not benefit from partitioning), the vertical partitioning approach is competitive with the traditional approach; the index-only approach performs poorly for reasons we discuss below. Before looking at the performance of individual queries in more detail, we summarize the two high level issues that limit the approach of the columnar approaches: tuple overheads, and inefficient column reconstruction.

Tuple overheads: As others have observed [12], one of the problems with a fully vertically partitioned approach in a row-store is that tuple overheads can be quite large. This is further aggravated by the requirement that the primary keys of each table be stored with each column to allow tuples to be reconstructed. We compared the sizes of column-tables in our vertical partitioning approach to the sizes of the traditional row store, and found that a single column-table from our SSBM scale 10 lineorder table (with 60 million tuples) requires between 0.7 and 1.1 GBytes of data after compression to store - this represents about 8 bytes of overhead per row, plus about 4 bytes each for the primary key and the column attribute, depending on the column and the extent to which compression is effective (16 bytes × 6 × 107 tuples = 960 MB). In contrast, the entire 17 column lineorder table in the traditional approach occupies about 6 GBytes decompressed, or 4 GBytes compressed, meaning that scanning just four of the columns in the vertical partitioning approach will take as long as scanning the entire fact table in the traditional approach. Column Joins: Merging two columns from the same table together requires a join operation. System favours using hash-joins for these operations, which is quite slow. We experimented with forcing System to use index nested loops and merge joins, but found that this did not improve performance because index accesses had high overhead and System was unable to skip the sort preceding the merge join.

VI. DETAILED ROW-STORE PERFORMANCE BREAKDOWN

In this section, we look at the performance of the row-In this section we look at the performance of the store approaches, using the plans generated by System for query 2.1 from the SSBM as a guide (we chose this query because it is one of the few that does not benefit from orderdate partitioning, so provides a more equal comparison between the traditional and vertical partitioning approach.) [4, 5]. Though we do not dissect plans for other queries as carefully, their basic structure is the same. The SQL for this query is:

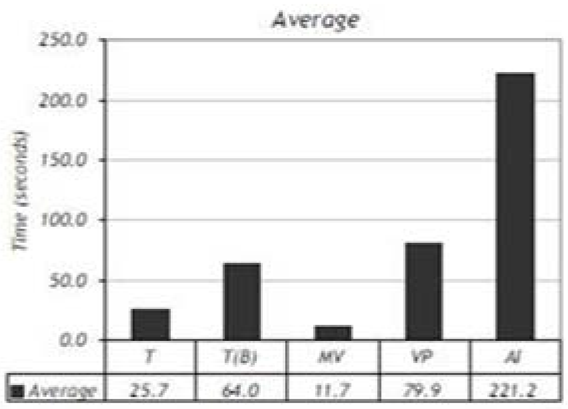


Figure 2. Average performance numbers across all queries in the SSBM for different variants of the row-store. Here, T is traditional, T(B) is traditional (bitmap), MV is materialized views, VP is vertical partitioning, and AI is all indexes.

```
SELECT sum(lo_revenue), d_year, p_brand1
FROM lineorder, dwdate, part, supplier
WHERE lo_orderdate = d_datekey
AND lo_partkey = p_partkey
AND lo_suppkey = s_suppkey
AND p_category = 'MFGR#12'
 AND s_region = 'AMERICA'
GROUP BY d_year, p_brand1 ORDER
 BY d_year, p_brand1
```

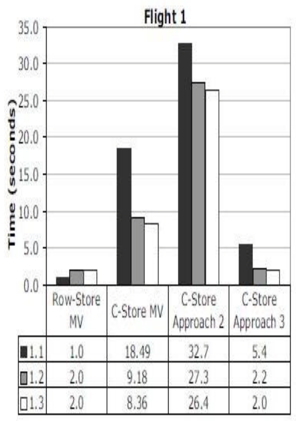
The selectivity of this query is 8.0 × 10–3. Here, the vertical partitioning approach performs about as well as the traditional approach (65 seconds versus 43 seconds), but the index-only approach performs substantially worse (360 seconds). We look at the reasons for this below.

A. *Traditional:*

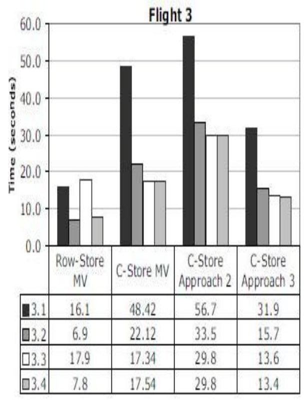
For this query, the traditional approach scans the entire lineorder table, using four hash joins to join it with the dwdate, part, and supplier table (in that order). It then performs a sort-based aggregate to compute the final answer. The cost is dominated by the time to scan the lineorder table, which in our system requires about 40 seconds. For this query, bitmap indices do not help because when we force System to use bitmaps it chooses to perform the bitmap merges before restricting on the region and category fields, which slows its performance considerably. Materialized views take just 15 seconds, because they have to read about 1/3rd of the data as the traditional approach.

B. Vertical partitioning:

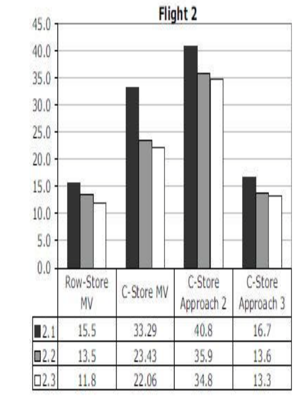
The vertical partitioning approach Hash-joins the partkey column with the filtered part table and the suppkey column with the filtered supplier table, and then hash-joins these two resultsets. This yields tuples with the primary key of the fact table and the p brand1 attribute of the part table that satisfy the query. System then hash joins this with the dwdate table to pick up d year, and finally uses an additional hash join to pick up the lo _revenue column from its column table.



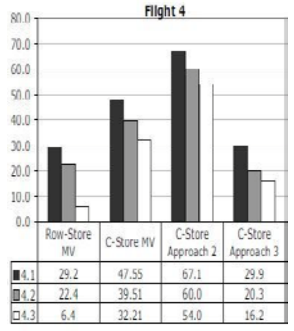
	Row-Store MV	C-Store MV	C-Store Approach 2	C-Store Approach 3
■1.1	1.0	18.49	32.7	5.4
▨1.2	2.0	9.18	27.3	2.2
□1.3	2.0	8.36	26.4	2.0



	Row-Store MV	C-Store MV	C-Store Approach 2	C-Store Approach 3
■3.1	16.1	48.42	56.7	31.9
▨3.2	6.9	22.12	33.5	15.7
□3.3	17.9	17.34	29.8	13.6
▨3.4	7.8	17.54	29.8	13.4



	Row-Store MV	C-Store MV	C-Store Approach 2	C-Store Approach 3
■2.1	15.5	33.29	40.8	16.7
▨2.2	13.5	23.43	35.9	13.6
□2.3	11.8	22.06	34.8	13.3



	Row-Store MV	C-Store MV	C-Store Approach 2	C-Store Approach 3
■4.1	29.2	47.55	67.1	29.9
▨4.2	22.4	39.51	60.0	20.3
□4.3	6.4	32.21	54.0	16.2

Figure 3. Figure3. Performance numbers for different variants of the row-store by queryflight. Here, T is traditional, T(B) is traditional (bitmap), MV is materialized views, VP is vertical partitioning, and AI is all indexes.

This approach requires four columns of the lineorder table to be read in their entirety (sequentially), which, as we said above, requires about as many bytes to be read from disk as the traditional approach, and this scan cost dominates the runtime of this query, yielding comparable performance as compared to the traditional approach. Hash joins in this case slow down performance by about 25%; we experimented with eliminating the hash joins by adding clustered B+trees on the key columns in each vertical partition, but System still chose to use hash joins in this case.

VII. CONCLUSIONS

The previous results show that none of our attempts to emulate a column-store in a row-store are particularly effective. The vertical partitioning approach can provide performance that is competitive with or slightly better than a row-store when selecting just a few columns. When selecting more than about 1/4 of the columns, however, the wasted space due to tuple headers and redundant copies of the primary key yield inferior performance to the traditional approach. This approach also requires relatively expensive hash joins to combine columns from the fact table together. It is possible that System could be tricked into storing the columns on disk in sorted order and then using a merge join (without a sort) to combine columns from the fact table but we were unable to coax this behaviour from the system.

Index-only plans avoid redundantly storing the primary key, and have a lower per-record overhead, but introduce another problem - namely, the system is forced to join columns of the fact table together using expensive hash joins before filtering the fact table using dimension columns [1]. It appears that System is unable to defer these joins until later in the plan (as the vertical partitioning approach does) because it cannot retain record-ids from the fact table after it has joined with another table. This giant hash joins lead to extremely slow performance.

With respect to the traditional plans, materialized views are an obvious win as they allow System to read just the subset of the fact table that is relevant, without merging columns together. Bitmap indices sometimes help - especially when the selectivity of queries is low - because they allow the system to skip over some pages of the fact table when scanning it. In other cases, they slow the system down as merging bitmaps adds some overhead to plan execution and bitmap scans can be slower than pure sequential scans. In any case, for the SSBM, their effect is relatively small, improving performance by at most about 25% [11].

As a final note, we observe that implementing these plans in System was quite painful. We were required to rewrite all of our queries to use the vertical partitioning approaches, and had to make extensive use of optimizer hints and other trickery to coax the system into doing what we desired.

In the forthcoming paper we will try to show how column-stores are designed using alternative approaches are able to circumvent these limitations.

VIII. REFERENCES

[1]. Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In CIDR, 2005.

[2]. Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. In SIGMOD '01, pages 271–282, 2001.

[3]. George Copeland and Setrag Khoshafian. A decomposition storage model. In SIGMOD, pages 268–279, 1985.

[4]. Alan Halverson, Jennifer L. Beckmann, Jeffrey F. Naughton, and David J. Dewitt. A Comparison of C-Store and Row-Store in a Common Framework. Technical Report TR1570, University ofWisconsin-Madison, 2006.

[5]. Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel R. Madden. Performance tradeoffs in readoptimized databases. In VLDB, pages 487–498, Seoul, Korea, 2006.

[6]. Setrag Khoshafian, George Copeland, Thomas Jagodis, Haran Boral, and Patrick Valduriez. A query processing strategy for the decomposed storage model. In ICDE, pages 636–643, 1987.

[7]. Carl Olofson. Worldwide RDBMS 2005 vendor shares. Technical Report 201692, IDC, May 2006.

[8]. Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond

[9]. Lau, Amerson Lin, Samuel R. Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alexander Rasin, Nga Tran, and Stan B. Zdonik. C-Store: A Column-Oriented DBMS. In VLDB, pages 553–564, Trondheim, Norway, 2005.

[10]. Michael Stonebraker, Chuck Bear, Ugur Cetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik. One size fits all? - Part 2: Benchmarking results.In Proceedings of the Third International Conference on Innovative Data Systems Research (CIDR), January 2007.

[11]. Dan Vesset. Worldwide data warehousing tools 2005 vendor shares. Technical Report 203229, IDC, August 2006.

[12]. Setrag Khoshafian, George Copeland, Thomas Jagodis, Haran Boral, and Patrick Valduriez. A query processing strategy for the decomposed storage model. In ICDE, pages 636-643, 1987.

[13]. Patrick E. O'Neil, Xuedong Chen, and Elizabeth J. O'Neil. Adjoined Dimension Column Index (ADC Index) to Improve Star Schema Query Performance. In Proc. of ICDE, 2008.

[14]. Patrick E. O'Neil, Elizabeth J. O'Neil, and Xuedong Chen. The Star Schema Benchmark (SSB). http://www.cs.umb.edu/~poneil/StarSchemaB.PDF.

[15]. Sanil.S.Nair, Bhruthari G. Pund. An Intuition of the Necessitate of Column-Oriented Database Systems (SSB).http://www.ijcaonline.org/proceedings/isdmisc/number4/3468-isdm091