OLAP Query Performance in Column-Oriented Databases (December 2012)

Sami Sorjonen

Abstract—Query response times are crucial for data warehousing systems. As these systems deal with large amounts of data, but commonly access only a small part of columns in single query, column-oriented databases have been researched as an alternative to traditional row-oriented databases. In contrast to row-oriented databases, which use “once-a-tuple” pipeline processing, column-oriented databases use “once-a-column” style processing, which is aimed at better I/O and cache efficiency. Special optimization techniques may be used with column stores and researchers have been developing some unique algorithms for column stores or column-oriented/row-oriented hybrid stores. The outcome of this research have been verified with practical experiments and with the help of different benchmarks, including Star Schema Benchmark, which is designed to measure efficiency of data warehouse environments.

Index Terms—column-oriented database, optimization, data warehousing, OLAP, multidimensional data model

I. INTRODUCTION

OLAP (online analytical processing) is a category of database processing, which was introduced in 1993 in contrast to OLTP (online transaction processing) [5]. The reason for this categorization is to point out the different nature of database processing used in data warehouses (OLAP) and operational systems (OLTP).

OLAP technology is widely used in data warehouses. Data warehouses contain large collections of corporate data, which is gathered from different locations, e.g. from the corporate’s operational systems. The data is typically used for decision support and for the development of corporate’s business processes.

There are some fundamental differences between OLTP-systems and OLAP systems. The typical features of OLAP-systems and data warehouses include [4]:

1) Multidimensional data model
2) Amount of data is large, usually from hundreds of gigabytes to terabytes.
3) The workloads are query intensive, i.e. they consist mostly of read operations and only seldom of update or delete operations.
4) The queries are mostly complex ad hoc queries, which may access a large number of records and perform a lot of scans, joins and aggregates.
5) Query response times are more important than transaction throughput.
6) Data is historical and contains data from long periods.

7) OLAP systems are targeted for decision support.
8) Data is gathered from multiple locations and it needs to be consolidated.

Features mentioned above distinguish the data warehousing OLAP systems from operational OLTP systems. These characteristics, combined with the fact that OLAP queries often access only a small part of columns, make the column-oriented database management systems (DBMS) an attractive alternative for OLAP. There are several scientific works, which show that column-oriented, “once-a-column” style processing, satisfies OLAP applications better than traditional “once-a-tuple” pipeline processing [2].

In this paper we take a closer look at some basic concepts of OLAP technology and column-oriented databases. In section 2 we start with introduction of multidimensional data model, which is very important for OLAP systems, and continue in section 3 with presentation of column-oriented storage and processing models. In section 4 and 5 optimization techniques used in columnar databases are explained with multidimensional schema example. The results of some practical experiments, which use multidimensional database schema, are presented in section 6. In section 7 we conclude and summarize the contents of this paper.

II. MULTIDIMENSIONAL DATA MODEL

In OLAP systems data is typically modeled dimensionally. Traditional relational multidimensional database schema consists of one (or few) fact table(s) and several dimension tables. The fact table is by far the largest table in schema, containing typically millions of rows. Dimension tables, on the contrary, usually contain only few thousand rows and they are linked to fact table with many to one relationship, using surrogate keys.

Fact-table holds the information about the business process transactions (e.g. sales) and to every transaction typically belongs some kind of numerical measures (e.g. price of a sale or profit of a sale). These numerical measures can then be viewed through one or more dimensions. For example through date dimension we can view the sum of all prices of sales in year 2010. Or through both, product and date dimensions, we could view the sum of all prices of sales of a particular product in 2010. Such views are implemented as SQL-joins between fact table and dimension tables. Database schema arranged dimensionally in this way is called star schema and its generalized form is called snowflake schema. An example of star schema is presented in figure 1. The fact table ‘Sale’,
which is the largest, is placed in the middle and has a grey background.

Fig. 1: An example of a star schema

III. COLUMN ORIENTED STORAGE AND PROCESSING MODEL

Complex queries against multidimensional data are the typical usage of OLAP systems and the most important performance metric for these systems is query response time. One OLAP query may access millions of records and contains usually a lot of table joins and aggregates. The aggregation commonly concerns only a small part of columns. Considering these facts, it is no surprise that column-oriented database systems are considered a promising approach for improving query performance and response times in data warehousing and OLAP technology.

Column-oriented SQL-engines handle data as attributes (columns) in contrast to row-oriented databases, which handle data as tuples (rows). These are sometimes also called as NSM (N-array Storage Model), which means row-wise model and DSM (Decomposed Storage Model), which means the column-wise model. The row/column-orientation can be implemented either at storage or at processing level, which leads us to categorizing the SQL-engines into four different groups [6]:
1) Row-oriented storage model with row processing model
2) Column-oriented storage model with row processing model
3) Column-oriented storage model with column processing model
4) Row-oriented storage model with column processing model

Categories 1 and 3 can be called as native query processing engines, as they use the same model for both storage and processing. The categories 2 and 4 can be regarded as enabled because they use processing model that differs from storage model [6].

A. Column-oriented storage model

Column-oriented storage model has some advantages over row-oriented storage. Row-oriented storage model organizes data physically as consecutive tuples and the length of the tuple attributes usually vary. Because of this, many pointers are needed to locate tuples efficiently, which leads to an additional cost of space. Columnar databases, however, may store fixed length values in attribute column. If the column contains various length attribute values, then the column only stores fixed length offset pointers to actual data. And if the column contains fixed length values, then column store model may locate the values directly by offset [6]. In addition, because the same attribute values may be stored together, more effective compression techniques can be used [3].

An interesting detail is, that traditional row-model storage may also emulate the column-oriented approach to some extent, by using techniques such as vertical partitioning, index-only plans and materialized views [1]. In vertical partitioning single table is created for every column in the table, in index-only plans an additional unclustered index is added on every column of the table and in materialized views the used view contains only the columns needed to answer the particular query (or part of it). Such techniques may improve the performance of a row-storage somewhat, but to fully benefit from columnar approach “changes must be made to both the storage layer and the query executor” [1], that is, use a DBMS that is designed from scratch with a columnar approach in mind.

There exist a number of column-oriented databases on the market, which implement the column-store model. These databases include commercial products such as Greenplum, Infobright, Paraccel, Sadas and Sybase IQ, as well as open source products such as C-store and ModetDB, just to name a few.

B. Column-oriented processing model

OLAP queries typically access only some of the column attributes in the relation. Because the traditional row-oriented database usually uses a query plan in which data is accessed one tuple at a time, the necessary attributes must be picked out from the tuple. This operation is very common in OLAP queries and because column-oriented processing model does not need to care about the unnecessary columns in any way, it is usually also more efficient. More specifically, the columnar processing model is more efficient in SELECT and PROJECT operations, but row-wise model seems to perform better in JOIN, GROUPING and AGGREGATE operations, depending however on scenario[6].

Also, column-oriented processing typically accesses column values as blocks, which diminishes the number of function calls and thus increases efficiency. In contrast, row-wise processing typically needs 1-2 function calls for every tuple, although this “per tuple overhead can be reduced in row-stores if block of tuples are available at once and operated on in a single function call”[1]. Some row-oriented database systems,
such as IBM DB2, implement this [1].

Column-oriented processing model may be used also with traditional row-storage by converting tuples from row-wise model (NSM) into column-wise model (DSM) on the fly. Such on-the-fly conversion may sometimes improve the performance of a row-oriented database as well as the opposite conversion from DSM to NSM may in certain scenarios improve the performance of a columnar database system [9].

IV. BASIC OPTIMIZATION TECHNIQUES

As mentioned earlier, column-oriented database may be more suitable for handling OLAP queries than traditional row-oriented database. That is because of the large amount of data in data warehouses, high column selectivity of the queries and large number of join, grouping and aggregations in the queries. In addition to the column-oriented approach there are also many optimization techniques available for column-oriented databases. Common techniques used to optimize the performance of columnar databases include compression, late materialization and block iteration.

A. Compression

Keeping data in compressed format improves the query performance and diminishes the space cost of data.

In columnar databases better compression ratio may be achieved, compared to row stores. This is due to the fact that “compression algorithms perform better on data with low information entropy” [1]. We can for example think of a relation containing customer information, such as name, phone number, address and email. The column containing only numerical phone number values is more compressible than tuples containing many different kinds of fields, such as textual name- and email-fields. This holds particularly true, if the column is sorted.

But the advantage of compression does not limit to reduced disk space, which with today’s disk cost applies only a minimal advantage. Compression also improves performance by diminishing the time of I/O, as data is read from disk to memory or from memory to CPU cache. There are also cases where query executor can operate directly on compressed data, avoiding decompression altogether, and thus improving the performance even more.

B. Block iteration

Fetching data records from data store is done by using function calls, which are relatively costly operations, compared for example to operations on data that has already been fetched [1]. In contrast to most row-stores, column-stores exploit this fact by fetching values of columns in blocks, that is, using only single function call to fetch multiple column values. The aim is to minimize costly function calls.

Furthermore, if column values are fixed-width, an array may be constructed from the values and operating on array is more efficient than operating one tuple at a time, as the rowstores do. Operating on array “also exploits potential for parallelism on modern CPUs, as loop pipelining techniques can be used [1]”. Naturally column-orientation also benefits from the fact, that attribute extraction from tuples is not needed.

C. Late materialization

Column-stores split the tuples to attributes and save them in multiple locations on disk. Because SQL-queries usually access more than one attribute, then in some point of query the columns must be combined into rows. This operation is called materialization.

Naive column-stores work as follows: read the needed columns from disk to memory (or from memory to CPU), construct tuples from columns and execute traditional row-processes on the rows. This is called early materialization.

More advanced column-oriented database management systems, such as C-store, try to postpone this materialization and operate on columns as long as possible. This is called late materialization.

As an example of early materialization and late materialization let us look the following (quite typical) OLAP query on database schema defined in figure 1:

```
SELECT c.customer_country, st.store_country,
       d.date_year, SUM(s.sale_price) AS sum_price
FROM customer c, sale s, store st, date d
WHERE s.customer_id = c.customer_id
    AND s.store_id = st.store_id
    AND s.date_id = d.date_id
    AND c.customer_country = 'Russia'
    AND st.store_country = 'Finland'
    AND d.date_year >= 1992
    AND d.date_year <= 1997
GROUP BY c.customer_country,
         st.store_country, d.date_year
ORDER BY d.date_year ASC, s.sale_price DESC;
```

The query finds the total sum of price that Russian customers paid for products they bought from stores located in Finland.

The traditional early materialization plan could work roughly as follows:

1) Execute the join on ‘customer’ and ‘sale’ tables, based on ‘customer_id’. This operation restricts ‘sale’ table rows to those sales that are carried out by Russians. As a result an intermediate table ‘customer-sale’ is created. Intermediate ‘customer-sale’ table holds the ‘sale_price’ data from the ‘sale’ table and the ‘customer_country’ data from the ‘customer’ table. This phase is described in figure 2.

2) Similarly perform a join between the intermediate table ‘customer-sale’ and the ‘store’ table. New intermediate table ‘customer-sale-store’ is created, which is the same table as ‘customer-sale’, but only holds those sales that are carried out at stores in Finland. Also the ‘store.store_country’ data is added to the ‘customer-sale-store’-table.

3) Perform a join between the intermediate table ‘customer-sale-store’ and the ‘date’-table. The result is limited to the sales in year 1992-1997 and the ‘date.date_year’ attribute is applied to the resulting ‘customer-sale-store-date’ in-
4) Execute the aggregation, grouping and ordering operations on ‘customer-sale-store-date’-table tuples.
5) Return the resultset.

Fig. 2. Early materialization of sale and customer tables

The early materialization suffers from many disadvantages including possible unnecessary tuple construction, early decompression and ineffective use of cache. The late materialization aims to improve these weaknesses and could work in a following way:

1) Filter the ‘customer.customer_country’ column with country="Russia"-condition and extract the ‘customer_id’ keys that match the conditions in the ‘customer.customer_id’-column.
2) Join the ‘customer_id’ keys with the ‘sale.customer_id’ column. The result of this join is one position list for both the ‘sale’ and the ‘customer’ tables. This is depicted in figure 3.
3) Filter and join the ‘store.store_country’ and the ‘date.date_year’ columns similarly with the ‘sale’ table. Note that the joins are executed only between the keys, the tuples with actual values are not yet constructed.
4) Extract the needed values from the ‘sale’, ‘customer’, ‘store’ and ‘date’ tables, construct the tuples and perform aggregation, grouping and ordering operations.
5) Return the resultset.

Fig. 3. Joining the columns of sale and customer tables in late materialization

V. ADVANCED OPTIMIZATION TECHNIQUES

In addition to compression, block iteration and late materialization, which may be considered as basic optimization techniques in column-oriented database, there also more special techniques, that have been researched. Advanced optimization techniques presented in this section include invisible join, DDTA-join and parallel porting.

A. Invisible join

Late materialization described in previous section suffers from the fact that in phase 4, the values from dimension tables are extracted in out-of-position order. Invisible-join, presented in [1] tries to improve this weakness. It is also a late materialization join designed on star schema style tables and for the query, that was presented in previous section, it would work roughly as follows:

1) Create a hash filter for ‘customer.customer_country’, ‘store.store_country’ and ‘date.date_year’ dimension columns, based on predicate selection of the query (figure 4).

Fig. 4. Invisible join, phase 1: creating filters (adapted from [6])

2) Generate a global bitmap vector using the hash filters and the corresponding foreign keys (‘sale.customer_id’, ‘sale.store_id’, ‘sale.date_id’) in fact table. The global bitmap vector indicates the position of all records in ‘sale’-table, which satisfy the predicate selection of dimension columns. This is illustrated in figure 5.
3) Extract the wanted attributes from dimension columns, using the global bitmap vector on foreign keys in ‘sale’ table and then the foreign keys on dimension columns. Because the key columns of customer and store tables are sorted, foreign key presents the position of desired value in dimension column and dimension values may be fetched with a fast array lookup. Note however, that the key column of the ‘date’ table is not sorted and thus full join must be executed. This phase is illustrated in figure 6.

4) Join the results from dimension columns and fact columns into tuples.
5) Perform aggregation, grouping, and ordering operations on tuples.
6) Return the resultset.

In invisible-join the number of values extracted from fact table is minimized compared to simple late materialization algorithm, because the selection is based on selectivity of whole query, not just on selectivity of one dimension at a time. Also the algorithm can use a technique called between predicate rewriting which basically means, that algorithm can take advantage of situation where set of keys in dimension (that satisfy the selection criteria) are contiguous by grouping these keys in hash filter.

B. DDTA-join

Invisible join also has performance problems: the foreign key columns in ‘sale’ table are scanned twice (for partial and for global join), the hash tables created in phase one may become large and join result bitmaps in phase 2 are produced on ‘sale’ table, that has a huge size. DDTA-join (Directly Dimensional Tuple Accessing) aims to improve these performance issues by using row-oriented processing for fact table and column-oriented processing for dimension tables [6] [8]. DDTA-join is depicted in figure 7 and it works this way:

1) Create predicate-vector bitmaps (not hash filters as in invisible join) for dimension tables ‘customer’, ‘store’ and ‘date’. Because the ‘date’ table’s key column is not sorted create a temporary surrogate key for it by calculating interval from current date.
2) Perform a full table scan on ‘sale’ table.
3) For each tuple in ‘sale’ table find the corresponding value from ‘customer.customer_country’, ‘store.store_country’ and ‘date.date_year’ dimension columns, using predicate-vector bitmaps created in phase one. This is a fast array lookup based on the fact, that foreign key in ‘sale’ table can be mapped to the actual position of dimension column. Create a join between ‘sale’-tuple and dimension column if it satisfies the query expression.
4) Perform aggregation, grouping and ordering operations on tuples.
5) Return the resultset.

The main improvement of DDTA-join compared to invisible-join is, that it eliminates the multi-pass scan on sale-table. In fact, creating a join with just one table scan on fact table, is an improvement to all materialization techniques presented in this paper, also to early materialization. This is due to the fact, that DDTA-algorithm inherits pipeline processing from a rowwise processing model combined with column-oriented processing model for dimension columns, which can be accessed with direct array lookup. DDTA-join needs to scan the large fact table only once, as for all other late materialization techniques need to scan large fact table, or at least one column of it, several times. Early materialization needs to perform at least one full scan, plus several partial scans against the large fact table.
C. Parallel porting

One more mechanism to improve response time of OLAP-queries is parallelism. Of course the use of parallelization is not restricted to column-oriented DBMS (database management system), but with columnar database the usage of parallel and distributed architectures is a natural opportunity [2]. We can divide parallel processing architectures, depending on usage of memory resources, into two categories:

1) Shared address space (shared memory system)
2) Distributed address space (shared nothing system)

In shared address space (figure 8) main memory (and disk) is shared with all processor units and each processor usually has a CPU-cache of its own.

In distributed address space (figure 9) the system consists of individual nodes and each node has its own processor, private memory and private disk(s). The communication between processors and memory is executed via shared bus or ethernet cable. This architecture is also called massively parallel processing or shared nothing system.

In order to use distributed architecture the software has to support parallelization. The shared memory parallelization can be implemented with the help of specialized software, which offers an API for implementation of shared memory parallelization. One such software is OpenMP. Distributed address space parallelization can be done through Message Passing Interface (MPI), which is a standard library specification for message passing between cluster nodes. Several MPI implementations exist, such as MPICH and Open MPI, and they allow library users to write parallel programs with different languages, for example in C or Python.

---

Fig. 7. Example for DDTA-JOIN (adapted from [6])

Fig. 8. Shared address space
It may seem a hard task to make the DBMS support shared address space or distributed address space parallelization. In [2], however, authors show “how the original sequential code can be extended with a minimum effort in order to support its execution on both shared and distributed memory architectures”. One parameter, that they consider relevant, is the number of changed or added code lines.

VI. EXPERIMENTS

There are a number of experiments done with columnar databases, which compare the performance of different optimization techniques. One way to compare the performance of database systems is the Star Schema Benchmark (SSB). It is a benchmark designed for data warehouses and uses a typical textbook star-schema. In this section we take a look at two experiments. The first one compares invisible join and DDTA-join in different column-oriented environments with Star Schema Benchmark. The second one is an experiment of parallelizing the query execution in shared and distributed memory systems, with column-oriented database SADAS as query engine.

A. CDDTA-JOIN in memory resident OLAP

CDDTA-join refers to the use of DDTA-join, which was presented in section V, with column-oriented model. In [6] the authors report of the experiments that were carried out by using CDDTA-join in different storage models. The experiments contained CDDTA-join in memory resident OLAP and CDDTA-join in disk resident OLAP. The results of both experiments showed considerable performance improvements when using CDDTA-join, compared to invisible join. We will take a closer look at the results on experiment with CDDTA-join in memory resident OLAP. The used platform consisted of Lenovo R630 G7 server with

1) CPU: four Intel Xeon CPUE7420@2.13GHz CPUs
2) Memory: 32GB DDR II SDRAM

The experiment was carried out by three different storage models for fact table:

1) NSM (row-oriented storage model)
2) DSM (column-oriented storage model)
3) hybrid storage model.

As we can remember, the DDTA-join uses row-wise model for the fact table, so the fact table attributes must to be converted on-the-fly into rows if DSM is used. In hybrid model only the foreign keys of the fact table were organized as row table and measure attributes were left as column arrays. The results of this experiment are shown in figure 10.

The Q2.1 – Q4.3 refer to different queries in star schema benchmark, so altogether 10 different queries were used during the test. As can be seen, the CDDTA-join performed remarkably well, sometimes even halving the response time compared to invisible join. CDDTA-join was always faster than invisible-join, regardless of the storage model used. Authors claim, that the time cost of invisible join is so much bigger because it needs to scan the fact-table columns twice [6].

B. Parallel porting and SADAS database

SADAS is a commercial, column-oriented database for data warehousing and it is optimized to perform fast in large and complex OLAP-queries. It is implemented by the research laboratory of Advanced Systems based in Naples and was partially sponsored by the Italian Ministry of Education.

In [2] the authors report about experimental work of changing the SADAS kernel to support shared memory and distributed memory parallelism in order to improve OLAP-query performance. As a case study example they take the following query [2]:

```
SELECT year, COUNT(year), AVG(payed),
       STDDEV(payed), AVG(tobepayed),
       STDDEV(tobepayed), correlation (payed, to-
       bepayed)
FROM taxes
WHERE installments < 5 AND level < 11
GROUP BY year
```

In the source code of SADAS database the execution of this query consists of four main loops. The authors claim that the case “represents a general example because OLAP functions can be represented in most cases through an SQL script” [2]. The original sequential code of first of these loops computes partial sums of values of the measure attributes. If we don’t count closing brackets or empty lines, the number of code lines in modified version is 27, which is more than double of the 11 code lines in original version. The modification of other
loops is more straightforward and the overall lines of code added or changed is about 15% of the original code [2].

The modifications for distributed memory parallelization, using the MPI technology, are more complex and the lines of code added or changed is about 35% compared to original sequential version [2]. Between loops there are also functional dependencies, which must be handled properly in both cases, when implementing the parallelization.

In [2] the authors also report the experimental results of their implementation, using OpenMP software for shared address space model and MPI technology for distributed address space model. The experiment was carried out in two different platforms. We will take a closer look at one of the experiments, that used a platform with four 1 GHz Pentium III nodes and with 512 MB RAM. The input data contained about 7 million records and it was a real case study data. The results of the experiment are presented in Table 1:

<table>
<thead>
<tr>
<th>Version</th>
<th>Number of nodes</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential (Original code)</td>
<td>1</td>
<td>9,15</td>
</tr>
<tr>
<td>Shared Memory (OpenMP)</td>
<td>2</td>
<td>4,71</td>
</tr>
<tr>
<td>Distributed Memory (MPI)</td>
<td>2</td>
<td>4,65</td>
</tr>
<tr>
<td>OpenMP + MPI</td>
<td>2</td>
<td>2,43</td>
</tr>
</tbody>
</table>

The best result was obtained by using both technologies, OpenMP and MPI, together. With 4 nodes (excluded from the table) the execution time dropped as low as to 1,25 with OpenMP/MPI hybrid solution. Furthermore, in MPI- and OpenMP/MPI hybrid versions the decrease of execution time was linear, when the number of nodes was increased from 1 to 4. This can be seen figure 11.

![Fig. 11: Speedup of parallel execution [2]](image)

The authors also carried out experiments on other platform and slightly different code modification. Overall good results in execution time and scalability were obtained in all parallelization experiments.

**VII. CONCLUSION**

The column-oriented databases show promising results when used with data warehouses, OLAP technology and multidimensionally modeled data. Column-oriented model may be implemented at storage level and/or at processing level and column-oriented databases may also use many optimization techniques, which are not available or not as efficient with row-oriented databases. These optimizations techniques include compression, block iteration and late materialization. More advanced optimization techniques include invisible join, DDTA-join and use of parallelism. The experiments with DDTA-join, which needs to scan only once the large fact table of multidimensional star schema, show promising performance improvements. The efficiency was verified with Star Schema Benchmark, which is designed for testing the performance of data warehouses and OLAP technology. Good results and query response times were also reported from parallelization experiments done with SADAS column store kernel.

**REFERENCES**


