Query Execution in Columnar Databases

Atte Hinkka
atte.hinkka@cs.helsinki.fi

***Abstract*—The traditional row-based query-processing, originally meant for online transaction processing (OLTP) optimizes for writing individual rows and fares less well on the read-intensive, aggregating workloads of today's online analytical processing (OLAP) systems. Column-oriented databases or column-stores for short, store data by the column. This new kind of storage scheme serves read-intensive operations better and enables a collection of optimizations that further speed up analytical queries.**

From a purely storage perspective, it is easy to see the benefits of storing data in columns. For example, aggregating values in a column only requires the system to read a single, continuous column from the disk rather than reading in rows of data, extracting the requested column values and calculating the result only after that. Storing data of the same type together is also useful because compression can be used to shift the load from I/O to CPU. By operating on columns of similarly typed data, data-dependent, poorly CPU-predictable machine code can be avoided and better CPU utilization achieved.

We will compare row- and column-oriented query execution and go through optimizations that are central to the performance benefits of column-oriented databases such as vectorized query-processing and compression.

***Index Terms*—Query processing, Relational databases**

I. Introduction

The semantic model of a query is the *query tree*. Query trees (or query operator trees) consist of *query operators* which in the traditional Volcano model [1] describe the practical relational algebra operations [2] used to answer queries. Examples of query operators are *scan*, *select*, *sort*, *join* and *projection* and *aggregation*.

Scan	Reads tuples from disk
Select	Selects tuples that fulfill a given condition
Sort	Sorts tuples according to specified condition
Join	Joins sets of tuples to each other
Projection	Include specified set of values from tuples
Aggregation	Computes aggregate values

TABLE I
SUMMARY OF TYPICAL QUERY OPERATORS

After receiving the query from the user, database system goes through multiple *query plans*, ways to execute a query. A query plan is the semantic model to use for executing a query. The choice of query tree used for the query plan is done by comparing the costs of different query trees [3]. Each of the query operators might have different costs depending on the order they are used in the plan.

The cost of a query plan is determined by inspecting the schema of the database and looking at different metrics collected from it. These metrics can be something very basic, such as row counts or usable indices, or even something as complicated as average values for columns [4]. By preferring indexed columns for tuple selection and by placing select-operators closer to the leaves of the tree the amount of processed tuples can be limited and the cost of the query lowered.

```
SELECT emp.last_name, emp.first_name, dep.name
    FROM employees AS emp, departments AS dep
    WHERE emp.created > date '2012-01-01'
        AND dep.id = emp.department_id
     ORDER BY emp.created,
```

Fig. 1. Example query

For example, a query such as described in figure 1 can be represented by a naive query tree such as depicted in figure 2.

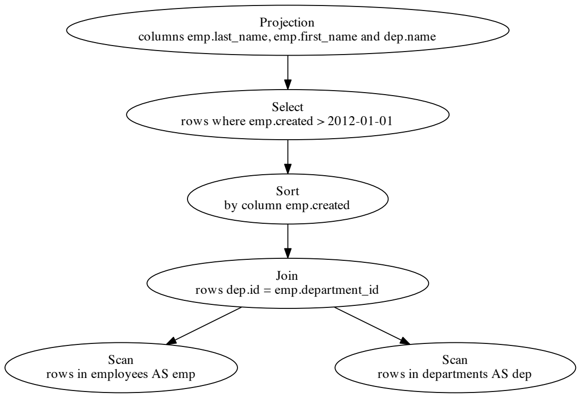
Projection
columns emp.last_name, emp.first_name and dep.name
Select
rows where emp.created > 2012-01-01
Sort
by column emp.created
Join
rows dep.id = emp.department_id
Scan
rows in employees AS emp
Scan
rows in departments AS dep

Fig. 2. Naive query operator tree for example query.

The naive query tree first scans both of the tables, then joins them and only after that starts dropping tuples from the execution. It ends up joining the tables at the earliest possible time which maximizes the join cost. For example, using a simple *join-algorithm* such as the *nested loop join* the time complexity of the join-operation is $O(N * M)$. In this case **N** would be the tuple count of *employees* and **M** the tuple count of departments.

After the join, tuples are sorted by the *created* column of *employees*. If the column is indexed, this is a fast operation. If the column isn't indexed, the operator ends up first sorting the tuples by the value, doing unnecessary work since some of the values might be dropped by the next operator in the tree. Naturally, intelligent enough select operator could use this known order to it's advantage and skip the comparison operation for part of the input tuples because it knows the input is ordered. By doing the select-operation before sorting the tuples, the sorting operation has to operate on less tuples. In general, the most efficient query plans utilize indices at the lowest possible level and avoid scanning the whole table from the disk, which is costly due to the I/O operations it causes. Scanning the whole table from the disk is called a *full table scan*.

Given that we have an index on the *created* column in the *employees* table, we can construct a more efficient query plan as shown in figure 3. The query plan utilizes the index on the *created* column and avoids a full table scan of employees. If the index is also ordered, it can be used by the sort-operator that is next in line after the select-operator. The join-operator is then left with only a subset of the employees tuples to join with the departments.

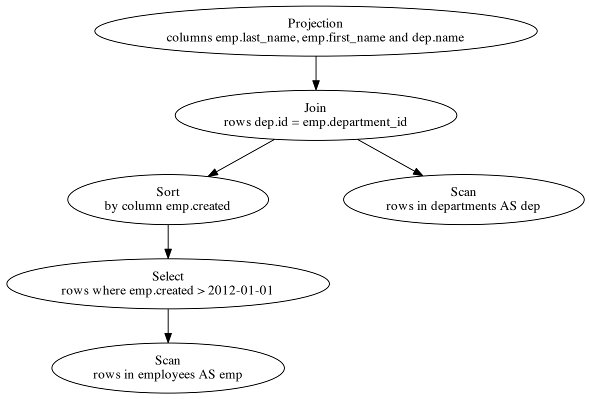
Projection
columns emp.last_name, emp.first_name and dep.name
Join
rows dep.id = emp.department_id
Sort
by column emp.created
Scan
rows in departments AS dep
Select
rows where emp.created > 2012-01-01
Scan
rows in employees AS emp

Fig. 3. Less naive query operator tree for example query.

The typical query in a column-store is different. Given a query such as in figure 1 with no aggregation operations yields less benefits in a column-store. A more typical OLAP query (modified from an example given by Bonchz et al [2]) is depicted in figure 4 is quite a different case. It can be realized into a query operator tree as pictured in figure 5. The tree consists of a single scan operator, followed by a select that implements the where-clause. After that aggregate values and direct projections are combined into a result. The query is characterized by the many aggregations it includes.

While typical queries for both OLTP and OLAP applications consist of common components, their usage contexts might be quite different. A business application is programmed by expert programmers, database and all its indices are tailor-made just for the application. An analytical application might be a massive data warehouse where data is stored for analysis and it might be used only for queries by people not well-versed in the intricacies of database systems [5]. Both systems can still support the *logical data model*, a common model after which relational database systems are built on. Logical data model specifies concepts such as tuples (or rows), columns, tables and databases and the operations on these concepts.

```
SELECT
        returnflag, linestatus,
        sum(quantity) AS sum_qty,
        sum(extendedprice) AS sum_base_price,
        avg(quantity) AS avg_qty,
        avg(extendedprice) AS avg_price,
        count(*) AS count_order
    FROM lineitem
    WHERE year = 2000
        AND region = 'ASIA'
    GROUP BY returnflag, linestatus
```

Fig. 4. Example analytical query

Logical data model serves well as the basis for a transactional database. Transactional use is write-heavy and involves full rows of data. For example, filling a *lineorder* table is made up of orders. When an order comes in, it is appended into the table. A column-store is not as well optimized for this kind of operation since it has to first destructure the row and then append the values into their respective columns separately.

Conversely, columnar database can aggregate the values in a lineorder table quicker since it already stores the values in a sequential column. This is beneficial also for performance and storage reasons. Operating on an array of values of same type can be optimized for modern processors [2] and compression is easier on values of the same type [5]. Making column-store fast, however, requires the implementor to take into account the new requirements posed. Basically, query execution and query operators need to change as well.

We will first take a view of history on the subject and look at column-store optimizations in brief in section II. After this we go through the introduced optimizations in depth in the following sections. We take a closer look at *block iteration* and other general query execution changes in section III, *vectorized processing* in section IV and finally go through *compression* issues in section V. One can find the conclusions from section VI.

II. Background

Storing database data in columns is nothing new. Systems such as TAXIR [6] and RAPID [7] date back as far as 1969 and 1976, respectively. Alternative data storage schemes have also been introduced before [8], but they haven't been as successful in attaining traction among the industry. There is a wealth of research on the workings of databases and especially on the optimization of the traditional transactional work load. Research on columnar databases, however, is quite recent and in the context of data warehousing a relatively new idea [5], even though the RAPID system was used by Statistics Canada for statistical calculations way back in the 70s.

Different kind of ways of optimizing row-stores for analytical work-loads have been introduced.

- *Vertical partitioning* is based on storing each column in its own table. Each of these vertical tables consist of the

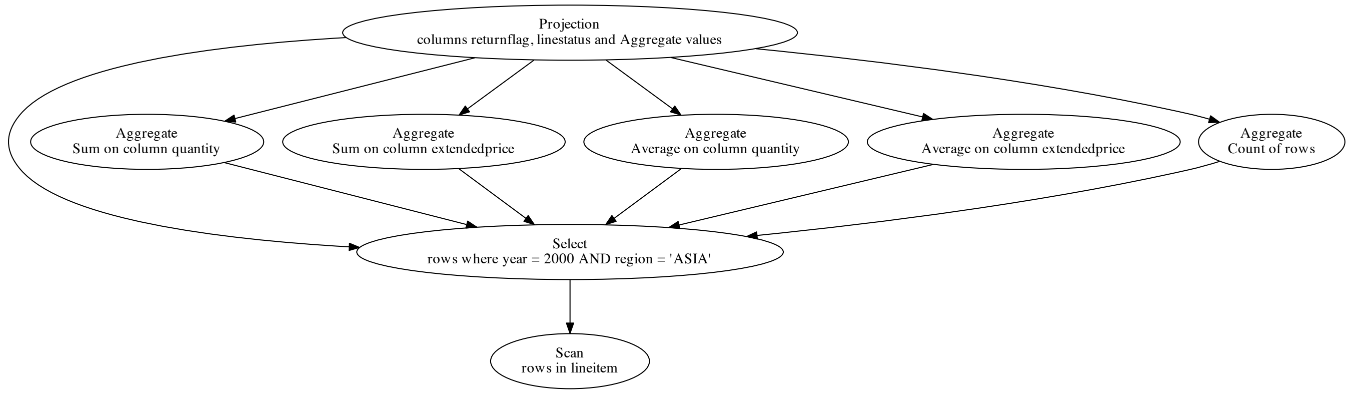
Projection
columns returnflag, linestatus and Aggregate values
Aggregate
Sum on column quantity
Aggregate
Sum on column extendedprice
Aggregate
Average on column quantity
Aggregate
Average on column extendedprice
Aggregate
Count of rows
Select
rows where year = 2000 AND region = 'ASIA'
Scan
rows in lineitem

Fig. 5. OLAP query operator tree.

actual value column and the row identifier column than is used to materialize the original row [9].

- *Materialized views* is another way of optimizing for analytical queries. Materialized views are views that are created in anticipation (and mainly used in benchmarks) of certain types of queries. The purpose of materialized views is to create views that contain only the relevant columns needed to answer a given query [9].
- *Index-only plans* are based on the idea of creating indices for all needed columns so that the the database system wouldn't need to do any table scans during query execution. [9].

Query processing and query plan optimization in row-stores naturally relies on the underlying well-defined semantics of the storage system, highly specialized data catalogs that define the type and number of data stored in each table and cost-functions based on relational algebra to come up with comparable values for query costs [3]. Column-stores use similar measures to discover how to execute queries. The differences between row- and column-stores are in the query execution and storage layers [9], not in the logical data model used.

The central optimizations that make column-stores faster than row-stores on analytical queries are based on the underlying implementation optimizations such as

- *vectorized query processing* [2], [10], which lets the CPU operate directly on typed data arrays.
- *Block iteration*, which is a technique to enable vectorized query processing by operating on more than one tuple or column value at a time.
- Column-specific compression techniques are yet another way to speed up column-stores. [11], [9].

Column-stores optimize away from the previously pervasive *Volcano*-like [1] systems that are based on the idea of reading (scanning) the tuples from disk and treating them as tuples from early on. This is called early *tuple materialization*. In the context of column-stores, the earliest (and most inefficient) tuple materialization strategy would be to construct tuples from each of the columns when the values are read from disk. Conversely, late tuple materialization makes it possible to implement the aforementioned optimizations [9]. Late tuple materialization is also the driver for the query operator changes used in column-stores [9].

System / optimization	Average query time
C-Store baseline	41 seconds
Late materialization	15 seconds
Compression	8 seconds
Invisible joins	4 seconds

TABLE II
PERFORMANCE BENEFITS OF OPTIMIZATIONS ACCORDING TO ABADI ET AL [9] ON THE *Star-Schema Benchmark*

Each of the optimization techniques presented improves the performance of a column-store. In the C-Store system [12] the late materialization changes (see section IV for details) provides biggest gains while compression and invisible joins benefit less (see II table for details).

While column-stores try to improve performance by using vertical data partitioning (columns are vertical structures in this sense), another option would be to partition the columns again horizontally to further improve compression possibilities and enable query processing on metadata collected from these newly constructed data clusters. In In Infobright's database system, these kind of data clusters are called *data packs* and they respectively contain *rough metrics*. Rough metrics of a data pack, such as min or max value, sum of values or number of nulls, can be used to help optimize queries or even answering queries directly[4]. For example, using the minimum and maximum values for a date field one can decide whether a given cluster is needed to answer a query or not.

III. QUERY EXECUTION IN COLUMN-STORES

As mentioned earlier, the typical Volcano query processing model is based on query operators that act as tuple iterators. A call to an unoptimized scan operator would always trigger a disk seek. A completely naive implementation of this model would then end up fetching every tuple from the disk individually. This kind of overhead is naturally partly mitigated by paging.

In a column-oriented database, to achieve speed gains in query processing it is beneficial to access more than a single tuple at a time to be able to vectorize processing. To achieve this, systems such as *C-Store* [12] and *MonetDB* [10] use a technique called block iteration. Block iteration at its heart is just about the iterators returning sets of items instead of single items.

Block iteration has also been investigated before in row-stores, mainly motivated by the need to amortize the iterator call. However, this isn't necessarily as beneficial for row-stores since the time saved may not be that significant because the per-tuple processing costs such as tuple deconstruction are relatively high [5].

Another query operator optimization that is made obvious in column-stores is pushing the predicate select operators down to the scan operator [5]. Going back to our OLAP query example in figure 4, if the select predicates were pushed to the scan operator, it could operate directly on the data read from the disk and possibly leverage optimizations developed specifically for that kind of comparison. It is also possible to work directly on compressed data if the predicate is provided with the compressed representation of the predicate value [11].

Invisible joins are a method for speeding up joins in a column-store. Invisible joins work by creating bit vectors for different columns and intersecting those bit-vectors to [9] provide the indices that are actually part of the resulting tuple. In our OLAP query example (in figure 4), this can be used for joining the *year* and *region* columns and for constructing the resulting tables. In other words, invisible joins are a way of rewriting joins as selection predicates. For one, it reduces the need to access columns out of order (reduces disk seeks) [9] and enables them to be pushed onto the scan operator which can then take advantage of vectorized processing (see section IV) to achieve performance benefits from it.

IV. Vectorized processing

The efficiency of a processor can be measured in many ways. The predominant one is the *instructions-per-cycle* (IPC) metric. Modern superscalar processors utilize deep instruction pipelines in order to maximize the throughput of instructions. In case of no conditional jumps it's relatively easy to fill the pipeline. When there are conditional jumps, it gets hairier. The processor basically has to guess, which branch to fill the pipeline with.

From the programming standpoint, a typical, Volcano-style [1] row-store is based on iterators. Each query operator offers an iterator programming interface, i.e. function or method such as *next()*, that enables the query operator to ask for the next tuple to be processed. These calls work in a cascading fashion so that for each call, the next operators might have to also call other iterators. Basically, from the scan operator up, the conditional jumps taken in query operator code are data-dependent [13]. For example, when a select operator goes through the tuples it gets from the previous operator, it uses a predicate to see if it should return the tuple. The branch-prediction performance, then, for each call to the select-operator depends wholly on the data. This results in poor branch-prediction performance [2].

Data-dependent branching also causes poor cache-locality. The CPU is unable to fill its caches with the right pages because it can't predict what the data needed might be [2]. Each cache miss then causes a lengthy fetch from the RAM.

Main benefit from vectorized processing comes from being able to work with concepts modern compilers produce efficient code for. Doing operations that can easily be parallelized (for example, summing the values in an array), is an easy target for optimization for a native code compiler. Examples of these kind of optimizations are SIMD instructions and parallel execution in general. This results in both higher IPC and better cache-locality [2] than what can be observed in a typical row-store.

Typically a Volcano-type query execution system operates on single tuples and is not able to pipeline arithmetical operations such as sum operations. For example, given a query such as

SELECT *SUM(price)* **FROM** *sales*,

when properly vectorized, can be written like in figure 6,

```
$sum \leftarrow 0$
$a \leftarrow sales.price$
for $i \leftarrow a$ do
    $sum \leftarrow sum + i$
end for
return $sum$
```

Fig. 6. Vectorized sum

but typically is represented more by the pseudocode in figure 7.

```
$sum \leftarrow 0$
$iter \leftarrow sales.columnIterator$
while $iter.hasNext()$ do
    $sum \leftarrow sum + iter.column[price]$
end while
return $sum$
```

Fig. 7. Iterator sum

The purpose of vectorized processing is avoiding the overhead of the query execution architecture. Every call of the *iterator.next()* is complicated and destroys the CPU's ability to manage its memory [14]. The iterator call may even end up going all the way to the hard disk, which results in system calls and more context switches to the system. Compiler must be able to utilize CPU cache and do loop inlining. Special instructions have been developed to further speed up operations on raw values such as arrays of numbers and vectorized processing enables them [2].

V. Compression in Column Stores

A naive implementation of compression works with database pages, uncompressing them as needed, when they are read from the disk. This simple scheme only needs changes to the storage layer [11]. At the other end of the spectrum is executing queries without decompressing part of the data [1] at all.

There has been much research on compression that has focused on compressing data more efficiently to a smaller size [5]. Using heavier compression schemes such as *Lempel-Ziv encoding* (LZ) [15] has only become practical now when the performance development of CPUs has outperformed the development of disk I/O [5].

The traditional ways to compress data in databases have been

- null suppression [16],
- dictionary encoding and
- run-length encoding [11].

Null-suppression works by omitting to store null- or empty values. Run-length encoding is a way to encode repetitive values by specifying the length of the run of the value in a sequence of values. Dictionary encoding is a common compression scheme used in row-stores. Dictionary encoding works by having a dictionary for common data values that maps short codes for the longer values [5].

The advantages of column-stores can gain from compression come from operating directly on compressed data and from compression-scheme specific optimizations in the scan operator [9]. For example, producing a bit vector for an invisible join from a column with run-length encoding the scan operator can skip parts of the column altogether.

The changes needed for join operators relate to late tuple materialization. For example, in the C-Store system [12], the join operators can either work on materialized tuples or they can alternatively defer materialization to a later time and only return tuples of indices of the column values that describe the join result [11].

For heavy-weight compression schemes, doing normal joins is too expensive. A specific kind of join operator, however, can be devised to circumvent this. *LZ-join* uses a windowed cache of uncompressed items. This complicates the join operator, but makes it feasible and faster on columns that cannot be efficiently compressed with a simpler compression scheme such as run-length encoding [17].

VI. Conclusion

Column-oriented databases provide multiple different ways of speeding up analytical processing. While the name refers to the underlying storage format, the optimizations that actually make columnar databases fast are not central to only columnar databases, but can speed up row-oriented databases aswell. Using vectorized processing to achieve better CPU utilization [2] and integrating compression into the query execution pipeline [11] column stores can execute analytical queries far quicker than row-stores. The biggest single benefit of column-stores is that the architectural foundations are fundamentally compatible with sequential operation that is the cornerstone of vectorized processor and what a single CPU is good at.

The research on query optimization has not been looking into the issues MonetDB [2] and C-Store [12] have brought up, but has instead been interested in coming up with more and more elaborate heuristic schemes for determining the optimal order to do Volcano-style [18], tuple-at-a-time query processing. Parting from operating tuple at a time, enabling native code compilers do a good job producing efficient loop-aware machine code and delaying tuple materilization can significantly speed up analytical query processing.

VII. References

References

[1] G. Graefe, “Volcano - An Extensible and Parallel Query Evaluation System,” *IEEE Trans. Knowl. Data Eng.*, vol. 6, no. 1, pp. 120–135, 1994.

[2] P. A. Boncz, M. Zukowski, and N. Nes, “MonetDB/X100: Hyper-Pipelining Query Execution,” in *CIDR*, 2005, pp. 225–237.

[3] R. Elmasri and S. Navathe, “Query Processing and Optimization,” in *Fundamentals of Database Systems*, 3rd ed. Addison-Wesley, 2000, ch. 18, pp. 585–588.

[4] D. Slezak and P. Synak, “A Rough-Columnar RDBMS Engine–A Case Study of Correlated Subqueries,” *Data* ..., 2012. [Online]. Available: ftp://ftp.research.microsoft.com/pub/debull/A12mar/infobright1.pdf

[5] D. J. Abadi, “Query Execution in Column-Oriented Database Systems,” Ph.D. dissertation, Massachusetts Institute of Technology, 2008.

[6] G. F. Estabrook and R. C. Brill, “The Theory of the TAXIR Accessioner,” *Mathematical Biosciences*, no. 5, pp. 327–340, 1969. [Online]. Available: http://home.comcast.net/~tolkin.family/taxir1.htm

[7] M. J. Turner, R. Hammond, and P. Cotton, “A DBMS for Large Statistical Databases,” in *VLDB*, 1979, pp. 319–327.

[8] G. P. Copeland and S. Khoshafian, “ A Decomposition Storage Model ,” in *SIGMOD Conference*, 1985, pp. 268–279.

[9] D. J. Abadi, S. Madden, and N. Hachem, “Column-stores vs. row-stores: how different are they really?” in *SIGMOD Conference*, 2008, pp. 967–980.

[10] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman, “MonetDB/X100 - A DBMS In The CPU Cache,” *IEEE Data Eng. Bull.*, vol. 28, no. 2, pp. 17–22, 2005.

[11] D. J. Abadi, S. Madden, and M. Ferreira, “Integrating compression and execution in column-oriented database systems,” in *SIGMOD Conference*, 2006, pp. 671–682.

[12] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O’Neil, P. E. O’Neil, A. Rasin, N. Tran, and S. B. Zdonik, “C-Store: A Column-oriented DBMS,” in *VLDB*, 2005, pp. 553–564. [Online]. Available: http://www.vldb2005.org/program/paper/thu/p553-stonebraker.pdf

[13] K. Ross, “Conjunctive selection conditions in main memory,” in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2002, pp. 109–120.

[14] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, “ DBMSs on a Modern Processor: Where Does Time Go? ,” in *VLDB*, 1999, pp. 266–277.

[15] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337–343, 1977.

[16] T. Westmann, D. Kossmann, S. Helmer, G. Moerkotte *et al.*, “The implementation and performance of compressed databases,” *Sigmod Record*, vol. 29, no. 3, pp. 55–67, 2000.

[17] L. Gan, R. Li, Y. Jia, and X. Jin, “Join Directly on Heavy-Weight Compressed Data in Column-Oriented Database,” in *WAIM*, 2010, pp. 357–362.

[18] G. Graefe and L. Shapiro, “Data compression and database performance,” in *Applied Computing, 1991.,[Proceedings of the 1991] Symposium on*. IEEE, 1991, pp. 22–27.