Position Caching in a Column-Store with Late Materialization: An Initial Study

Viacheslav Galaktionov[1, 2], Evgeniy Klyuchikov[1, 2], George Chernishev[1, 2, 3]
[1] Information Systems Engineering Lab, JetBrains Research
[2] Saint Petersburg State University, Russia
[3] National Research University Higher School of Economics
{viacheslav.galaktionov,evgeniy.klyuchikov,chernishev}@gmail.com

ABSTRACT

A common technique to speed up DBMS query processing is to cache parts of query results and reuse them later. In this paper we propose a novel approach which is aimed specifically at caching intermediates in a late-materialization-oriented column-store.

The idea of our approach is to cache positions (row numbers) instead of data values. The small size of positional representation is a valuable advantage: cache can accommodate more entries and consider intermediates that involve "heavy" operators, e.g. joins of large tables.

Position caching thrives in late materialization environments since position exchange is prevalent in them. In particular, expensive predicates and heavy joins are usually processed based on positions. Our approach is able to cache them efficiently, thus significantly reducing system load.

To assess the importance of intermediates our position caching technique features a cost model that is based on usage statistics and complexity estimations. Furthermore, to allow intermediate reuse for the queries that are not fully identical, we proposed an efficient query containment checking algorithm. Several policies for cache population and eviction were proposed. Finally, our approach is enhanced by lightweight compression schemes.

Experimental evaluation was performed using a stream of randomly generated Star-Schema-Benchmark-like queries. It showed up to 3 times improvement in query run times. Additionally, compressing the intermediates reduces the space requirements by up to 2 times without a noticeable performance overhead.

1 INTRODUCTION

Late materialization [1] is a query processing technique that aims to operate on positions as long as possible. Essentially, it defers tuple reconstruction to later stages of a query plan. The goal is to conserve disk bandwidth by: 1) reading individual columns only when they are necessary for processing at a given point, and 2) organizing column access in such an order that subsequent reads require fewer disk accesses due to data filtering.

This approach leads to unconventional query plans with novel opportunities [10] for efficient query processing. In such query plans operators: 1) usually ingest only necessary columns, 2) exchange not only data, but also positions. Positions are essentially row ids which will be later used to request data. An example of late materialization processing is presented in Figure 1.

Here, for column G, query processor reads only the values that have passed join predicate. For columns E and H it is even better: query processor will have to read values that have passed both join predicates. Of course, the benefit of such processing scheme depends on many parameters: join filtering power, data distribution, disk parameters such as seek times and bandwidth.

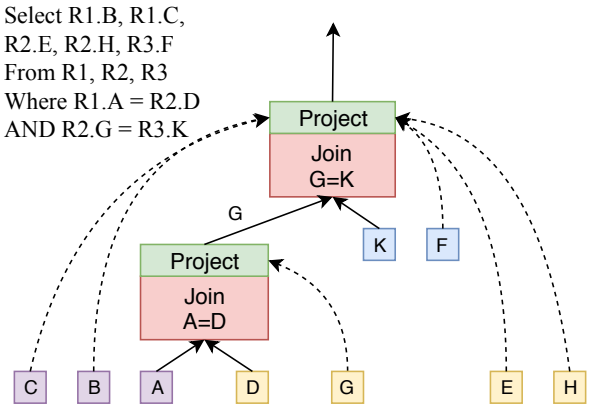


Figure 1: Late materialization example, adapted from [17]

Late materialization has formed the basis of several column-oriented DBMSes [1]. Its many aspects, such as the design of LM-enabled operators [3, 17] and the query processing schemes [6, 11], have been extensively studied. However, some aspects received less attention.

In this paper we further investigate one of them — intermediate result caching. Unlike many existing caching techniques, our approach does not cache the data itself. Instead, it aims to cache useful data in positional form which allows us to:

(1) reduce the memory footprint, since all table row data, which can be arbitrarily large, is replaced with a single position (a 32-bit number)
(2) use thusly saved memory to store intermediates that are closer to the bottom part of the plan. It comes more costly, but on the other hand, they have higher reuse chance.

Our caching approach also employs query containment checks which allows reusing intermediates from queries that are not fully identical, but one of them subsumes another.

To demonstrate the viability of the proposed position caching, we have implemented it inside PosDB — a distributed disk-based column-store with late materialization. Since PosDB currently lacks a full-fledged cost-based optimizer, we had to devise a number of simpler heuristic algorithms. While our approach is simpler than existing methods, it still allows us to obtain considerable performance gains.

Apart from that we have also implemented compression for cached positions. Keeping only sequences of integers favours compression which allows us to reduce memory footprint even further. For now, we have implemented only delta encoding and null suppression since our experiments indicated that these methods result in good space savings with a small overhead.

Overall, the contribution of this paper is the following:

(1) A novel approach to caching intermediates in a column-store that caches positions instead of data.
(2) A study of applicability of compression to caching inside a column-store.
(3) An experimental evaluation of several policies for cache population and eviction.

2 RELATED WORK

Caching intermediates. The idea of reusing intermediate results for query evaluation in databases has existed for a long time. It comes in two formulations. In the first one there is a number of simultaneously running queries and the goal is to generate query plans that would maximize result sharing. This is called multi-query optimization [15, 16]. The second option is caching: parts of query plans that were evaluated in the past are stored and reused when appropriate query arrives. In this paper we consider the second option.

The next two papers study caching of intermediates inside the MonetDB [11] system family, an in-memory column-store with late materialization. In reference [12] authors propose to exploit MonetDB operator-at-a-time processing model. The idea is to cache and reuse BAT by-products of query execution. The proposed approach featured plan subsumption and both inter- and intra- query reuse. The study [13] addresses the case of pipelined operator evaluation inside Vectorwise database engine.

In reference [9] another mechanism for in-memory result caching was proposed. The idea is to cache internal data structures of a database engine in order to lower memory pressure and preserve cache and register locality. In particular, hash tables that are produced by hash-joins and hash-aggregations.

Disk-based caching was considered in reference [4]. Apart from being able to cache data, this system can also cache positions. Also, authors proposed a cost-based optimizer to find the best plan. Experiments showed that position caching performs better than data caching when cache entries are stored on disk.

Caching and compression. Compression has been used in databases for a long time [19], however it is column-stores where it brought significant benefits. Compression in column-stores is different from row-stores: 1) columnar storage results in homogeneity of data which in turn leads to better compression rates and, 2) compression should be lightweight, i.e. it should not excessively load CPU.

We reference two papers which we believe to be most important, while many others can be found. Authors of reference [2] studied several lightweight compression algorithms in a disk-based column-store. They compared their behavior in case of eager decompression and in case when query engine operated on compressed data directly. Also, they have presented a decision tree for selecting compression method depending on data characteristics. Compression for in-memory column-stores was studied in reference [20]. Authors have proposed and evaluated three different compression schemes which aimed to efficiently use CPU resources.

To the best of our knowledge, no paper considered compression in the context of caching intermediates in column-stores.

3 PROPOSED APPROACH

Preliminaries and prerequisites. A common workflow of a DBMS caching subsystem is as follows. When a query arrives, its plan is generated and inspected for subtrees that a) can be reused later, b) can reuse current cache. If any are found, they interact with cache during query evaluation by either a) putting the resulting data into cache or b) reusing cached data instead of subtree evaluation.

In this study we assume that the target DBMS represents query plans as a tree of operators, each of which returns values or positions exclusively. As we target late materialization, most of the filtering and join operators are positional and (due to OLAP specifics) appear to be the most costly parts of a query plan.

Our caching approach intentionally targets only the positional data. As we stated in the introduction, "positional" cache entries are lightweight and are more susceptible to compression. Therefore, "heavy" operators from the bottom of the tree (usually costly filters and joins) can be cached with more ease. Thus, a significant decrease in system load can be achieved.

An essential drawback of position only caching is repeated data value reading. However, disk subsystem decreases the overhead. Caching is usually applied when the queries are somewhat similar in the terms of referenced disk pages. Therefore, the disk pages necessary for materialization are often already stored inside the buffer manager.

Caching: general overview. Our caching algorithm consists of pre- and post- processing of query plans. The former adds special operators to gather statistical information and substitutes subtrees with existing cache entries. The latter collects the final statistical information and updates the cache if needed.

The cache consists of query history and a set of cache entries (each being a list of position blocks). Query history is a list of subtrees of N most recently executed queries together with the subtrees that are still cached. Each history entry contains a pointer to the corresponding cache entry (if any), a high-level description of the subtree, its result's size and usage statistic. A usage statistic is a set of queries the entry can be potentially useful for. All this information is used in the cost models of cache population and eviction policies.

There might not be a perfectly matching cache entry for a subtree, but at the same time there may be an entry that provides a larger, subsuming, result. This is determined with a containment check. If the check is passed, the original subtree is replaced by this cache entry with a special filtering operator on top.

Caching: preprocessing. Preprocessing is the most sophisticated part, see Figure 2. The algorithm iterates through the query plan and for each subtree S: a) checks if it fits into cache b) adds a special operator `PBCounter` on top of it c) updates cache history d) tries to replace S with a (possibly filtered) cache entry.

The first two steps are related to size estimation of intermediates. An auxiliary `PBCounter` is used to count the number of blocks passed during query evaluation. Then, this number is used to: a) update the benefit measure of S (described later) b) filter future subtrees that are guaranteed not to fit into cache.

After size estimation has been taken care of, we add a history entry for S and update the usage statistics of other entries that can (potentially) be used by S. Usage statistics in their turn participate in the cost model of population and eviction policies. Therefore, we try to add the best yet uncached entry into cache.

Finally, after the cache was adjusted, we search for a cache entry that can replace S with minimum additional filtering. If one is found, we use a special operator `FetchCache` instead of direct S evaluation. This operator passes position blocks, decompressing them if needed.

Caching: postrocessing. This stage has two goals: gather the results of every `PBCounter` that is a part of the current query's

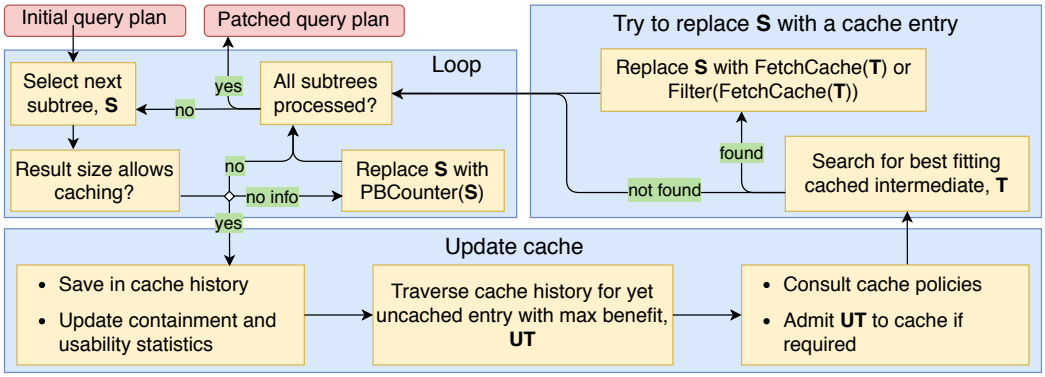


Figure 2: PosDB caching workflow: preprocessing

operator tree, and calculate the benefit estimates for each intermediate.

Policies for cache population and eviction. We have considered several different policies for adding and removing intermediates. We assume that for each history entry a numeric `benefit` can be calculated. We also define `total benefit` as a product of benefit and the usage frequency (number of queries in the the entry usage statistics).

For cache population we have considered:

- `MIN`: choose an uncached intermediate whose `total benefit` is larger than minimal among cached intermediates.
- `AVG`: choose an uncached intermediate whose `total benefit` is larger than average among cached intermediates.

Cache eviction has a larger number of options:

- LRU (Least Recently Used): evict intermediates that were not used for the longest time.
- LFU (Least Frequently Used): evict intermediates with the shortest history of usage.
- LCS (Largest Cache Space): evict intermediates that occupy the largest amount of space.
- BENEFIT: evict entries that have the lowest `benefit`.
- BENEFITHIST: evict entries that have the lowest `total benefit`.
- RANDOM: evict random intermediates.

Cost model. The `benefit` of a query history entry is defined as a function of:

- Subtree complexity: a weighted sum of the subtree operators. Each operator type corresponds to a constant weight which is a model parameter;
- Working set complexity: an estimated amount of data the query executor requests from the storage system to process the subtree;
- Actual result size: computed by the corresponding `PBCounter` operator;
- Usage statistics: a list of unique query numbers that could (or did) use it.

Note that we rely on two kinds of complexity: subtree and working set complexity. While the former favours the"structurally complex" subtrees, the latter favours operation on larger amounts of data. They allow to distinguish simple queries on a large data and complex queries an a small data.

Overall, the following formula is used to calculate `benefit`:

$$Benefit(I) = \frac{\text{Complexity}(I) \times \text{EstimatedSize}(I)}{\text{ActualSize}(I)}.$$

Query Containment. Implementing a caching subsystem implies solving the query containment problem. It arises when it is necessary to determine which cache entry corresponds to a newly arrived query. Given the sheer expressive power of SQL, it is hard to check even the simple equivalence of two queries. However, in order to build an efficient caching subsystem it is essential to find cached entries that can be used as base table to evaluate the given query. For example, consider two queries:

- `SELECT * FROM T1 WHERE T1.A > 100`
- `SELECT * FROM T1 WHERE T1.A > 110`

If the first query is cached, then the second one can use the stored answer with additional filtering. Depending on the predicate selectivity and hardware parameters this approach may result in savings.

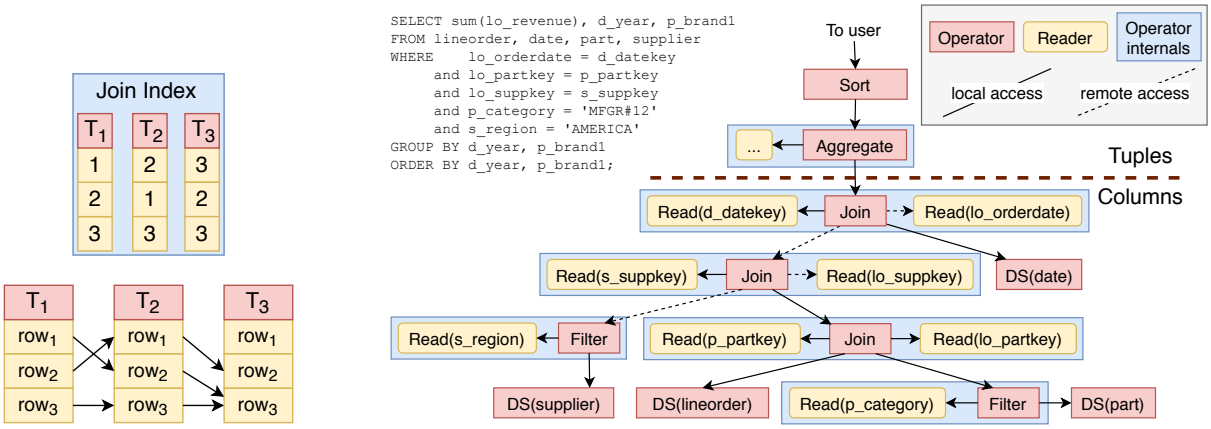
It is well-known that in the general case checking query containment is very demanding computationally. For example, it was shown [8] that the problem of conjunctive query containment is NP complete. Therefore, we had to restrict admissible query class to "mostly" conjunctive SPJ (Select-Project-Join) queries. "Mostly" means that `OR` can appear in `WHERE` clause only to list possible individual values of an attribute, e.g. `T1.A = 5 OR T1.A = 6 OR T1.A = 7` is allowed, while `T1.A < 5 OR T1.A = 10` is not. Therefore, the resulting class is limited, but big enough to cover all queries from the Star Schema Benchmark (SSB). Note, that since only positional data is involved, there is no need in checking containment of aggregation or sort.

In order to perform containment checks we have designed a special data structure to represent query metadata. Essentially, it records: 1) the set of attribute fragments covered, 2) all conducted joins, 3) all predicates that were used for filtering.

Obtaining such a structure is just a matter of traversing the corresponding operator tree.

Now, in order to perform a containment check given two such structures, all that is required is to make sure they share their sets of attribute fragments and joins, and every predicate from the first structure is implied by some predicate from the second structure. If that condition holds, then the second structure describes a containing query.

Compression. Cached position blocks appear to be a good target for compression. First, columnar representation guarantees data homogeneity and positions are just integers of fixed size. Secondly, the structure of a query plan explicitly restricts the possible position block structure. For example, large tables like SSB LINEORDER are usually treated in the way that preserves their initial order. Therefore, the corresponding position stream



(a) Example of join index **(b) Query plan example**

Figure 3: PosDB internals

is ordered and relatively dense. On the contrary, small tables which participate in joins often provide sparse and unordered sequences of positions.

Turning to concrete compression strategies, we have tried different encoding methods and found the *delta* and *null suppression* to be the most appropriate. The former allows to narrow down the range of absolute values inside each position block, especially when they have ascending or descending order. Then, the latter transforms 32-bit (signed or unsigned) numbers into 16-bit and 8-bit ones. Thus, up to four times space compression can be achieved with an almost negligible overhead of two `for` loops.

Two other apparently efficient compression strategies are RLE and Range. However, experiments demonstrated their limited use in practice. After the first filtering or join operator the consecutive positions are usually different or repeat just a little. In the latter case *delta + null suppression* give a comparable or better result.

We have also assessed and turned down some of the more complex compression methods, such as Dictionary compression and LZW. They either require significant time overhead or fail to provide noticeable space gains, encoding numbers with almost the same numbers.

4 POSDB

To evaluate our approach we have implemented it inside PosDB — a distributed column-store with a processing model that adheres to late materialization. Each PosDB query plan consists of two parts: position- and tuple-based (see Figure 3b). In the former (the bottom one) operators exchange only positions that are represented as a generalized join index [18] (Figure 3a). In our system this data structure is used to represent results of filter and join operators. For a filter, it is just a list of positions (row numbers) that satisfy a predicate. This structure can describe results of an arbitrary number of joins, e.g. consider the join index presented in Figure 3a. Here, the second row of T_1 was joined with the first row of T_2 and then the resulting tuple was joined with the second row of T_3.

The upper part of any plan consists of operators that work with tuples, similarly to classic row-stores. The lowest of these operators is responsible for materialization, i.e. it transforms join indexes into tuples of actual values. Such materializing operators perform aggregation and window functions.

Often, positional operators (e.g. `Join`, `Filter`) require data values in addition to positions. To provide such operators with data, a set of auxiliary entities called readers were introduced. For example, `ColumnReader` retrieves values of a specific attribute, while `SyncReader` provides values of several attributes synchronously.

PosDB is a both distributed and parallel column-store. It is distributed in terms of both data and query execution: a table may be fragmented and replicated across several nodes. A number of table-level fragmentation methods is supported: round-robin, hash and range partitioning strategies. Query distribution allows it to run a query on different nodes, possible with individual query plan parts residing on distinct nodes. Query distribution is implemented by two pairs of special operators: {`SendPos`, `ReceivePos`} and {`SendTuple`, `ReceiveTuple`}. Therefore, both positional and tuple operators can be executed on arbitrary nodes, regardless where their children reside.

Both inter- and intra-query parallelism [14] are supported. To implement intra-query parallelism two special operators were created. `Asynchronizer` allows to execute an operator tree in a separate thread and `UnionAll` is used to collect data from several subtrees that are executed in their own threads.

Recently [5], fully distributed operators like distributed join and aggregation were added. A detailed description of PosDB architecture can be found in papers [5, 7].

5 EXPERIMENTAL EVALUATION

We have evaluated the proposed approach with two experiments, designed to measure: a) the impact of different eviction policies on query execution performance and b) the impact of compression on the amount of space used for storing intermediates.

We fixed the size of cache history to 50 last queries and used only the `MIN` population policy. Cache size was varied from 2 to 32 thousands of position blocks, each of them being 32 KB large when uncompressed.

Finally, we relied on SSB with SF 10 and conducted a series of runs, a thousand of random "SSB-like" queries each. These queries differ from the usual SSB ones only in the way the filtering predicates are generated: in addition to randomly choosing the boundaries, we also randomly select the relation. For example, `DATE.D_YEAR = 1993` can become `DATE.D_YEAR < 1995`.

Experimental environment. We used a laptop with an Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz; 8 GB RAM; HDD

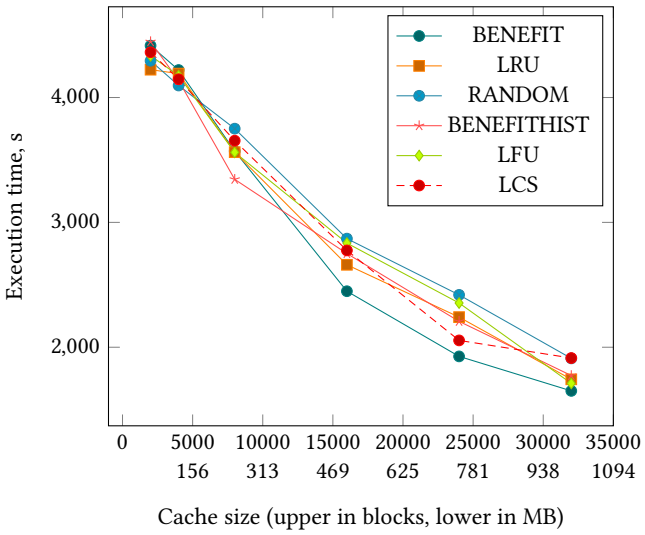


Figure 4: Query execution performance

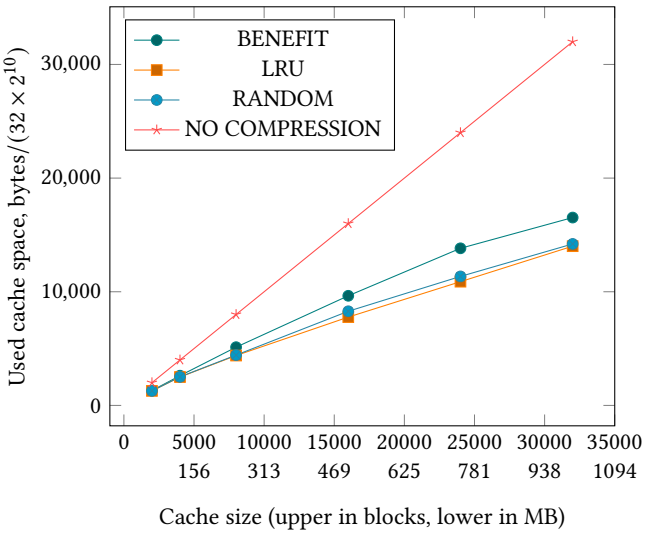


Figure 5: Memory usage before and after compression

Samsung ST1000LM024, 5400 rpm; Funtoo Linux, kernel version 5.2.7; gcc 9.2.0.

Query execution performance. First, we measured the performance of query execution when different eviction policies are used. The results are given in Figure 4 and show that with smaller cache sizes, the choice of a strategy had no significant impact on query performance. With larger cache sizes, however, `BENEFIT` turned out to be the best, `RANDOM` — the worst.

Another interesting outcome is that `BENEFITHIST`, which took history into account, resulted in worse performance than `BENEFIT`, which did not. We suppose that just relying on the usage frequency results in a longer eviction time for already-cold cache entries (which were hot some time ago). We believe the situation could be improved by also considering the times of several last uses.

Space consumption due to compression. Next, we enabled compression for intermediates and ran the same experiment using only the `BENEFIT`, `LRU`, and `RANDOM` eviction policies. Figure 5 shows the maximum amount of main memory used to store the intermediates during a particular run. The values were initially measured in bytes, and then divided by the standard size of a position block (32 KB).

It is worth noting that enabling compression has not had any noticeable effect on query execution times. Therefore, for the sake of brevity, the corresponding graph has been omitted.

6 CONCLUSION AND FUTURE WORK

In this paper we have proposed a novel approach to in-memory positional caching inside a column-store with late materialization. Compared to data caching, positional caching allows the system to save space by storing only integers which can be easily compressed. Experiments performed in PosDB demonstrated the viability of this approach. Also we have evaluated a number of policies for cache population and eviction, including our own. Future work will include evaluation of caching in a distributed environment.

REFERENCES

[1] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos. 2013. *The Design and Implementation of Modern Column-Oriented Database Systems.* Now Publishers Inc., Hanover, MA, USA.

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-oriented Database Systems *(SIGMOD '06)*.

[3] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. 2007. Materialization Strategies in a Column-Oriented DBMS. In *ICDE'07*. 466–475.

[4] Chungmin Melvin Chen and Nicholas Roussopoulos. 1994. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching *(EDBT '94)*. 323–336.

[5] George Chernishev, Viacheslav Galaktionov, Valentin Grigorev, Evgeniy Klyuchikov, Evgeniy Slobodkin, and Kirill Smirnov. [n.d.]. PosDB 2019: A Distributed Disk-Based Column-Store with Late Materialization (submitted). *Proc. VLDB Endow.* (Oct. [n. d.]).

[6] G. A. Chernishev, V. A. Galaktionov, V. D. Grigorev, E. S. Klyuchikov, and K. K. Smirnov. 2018. PosDB: An Architecture Overview. *Programming and Computer Software* 44, 1 (01 Jan 2018), 62–74.

[7] G. A. Chernishev, V. A. Galaktionov, V. D. Grigorev, E. S. Klyuchikov, and K. K. Smirnov. 2018. PosDB: An Architecture Overview. *Programming and Computer Software* 44, 1 (Jan. 2018), 62–74.

[8] Rada Chirkova. 2009. *Query Containment.* Springer US, 2249–2253.

[9] Kayhan Dursun, Carsten Binnig, Ugur Cetintemel, and Tim Kraska. 2017. Revisiting Reuse in Main Memory Database Systems *(SIGMOD '17)*. 15.

[10] Stavros Harizopoulos, Daniel Abadi, and Peter Boncz. 2009. Column-Oriented Database Systems, VLDB 2009 Tutorial. nms.csail.mit.edu/~stavros/pubs/tutorial2009-column_stores.pdf

[11] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012).

[12] Milena G. Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo A.P. Gonçalves. 2010. An Architecture for Recycling Intermediates in a Column-store. *ACM Trans. Database Syst.* 35, 4, Article 24 (Oct. 2010), 24:1–24:43 pages.

[13] Fabian Nagel, Peter Boncz, and Stratis D. Viglas. 2013. Recycling in Pipelined Query Evaluation *(ICDE '13)*. IEEE Computer Society, 338–349.

[14] M. Tamer Ozsu. 2007. *Principles of Distributed Database Systems* (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

[15] Prasan Roy and S. Sudarshan. 2009. *Multi-Query Optimization.* Springer US, Boston, MA, 1849–1852.

[16] Timos K. Sellis. 1988. Multiple-query Optimization. *ACM Trans. Database Syst.* 13, 1 (March 1988), 23–52.

[17] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. 2009. Query Processing Techniques for Solid State Drives *(SIGMOD '09)*. Association for Computing Machinery, 59–72.

[18] Patrick Valduriez. 1987. Join Indices. *ACM Trans. Database Syst.* 12, 2 (June 1987), 218–246.

[19] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The Implementation and Performance of Compressed Databases. *SIGMOD Rec.* 29, 3 (Sept. 2000), 55–67.

[20] M. Zukowski, S. Heman, N. Nes, and P. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE'06*. 59–59.