A Comprehensive Study of Late Materialization Strategies for a Disk-Based Column-Store

George Chernishev
Saint Petersburg State University
Saint-Petersburg, Russia
chernishev@gmail.com

Viacheslav Galaktionov
Saint Petersburg State University
Saint-Petersburg, Russia
viacheslav.galaktionov@gmail.com

Valentin Grigorev
Saint Petersburg State University
Saint-Petersburg, Russia
valentin.d.grigorev@gmail.com

Evgeniy Klyuchikov
Saint Petersburg State University
Saint-Petersburg, Russia
evgeniy.klyuchikov@gmail.com

Kirill Smirnov
Saint Petersburg State University
Saint-Petersburg, Russia
kirill.k.smirnov@gmail.com

ABSTRACT

By allowing operations on positions (row IDs, offsets), column-stores increase the overall number of admissible query plans. Their plans can be classified into a number of so-called materialization strategies, which describe the moment when positions are switched to tuples. Despite being a well-studied topic with several different implementations, there is still no formal definition for it, as well as no classification of existing approaches.

In this paper we review and classify these approaches. Our classification shows that, for disk-based systems, none of the existing implementation variants efficiently combines position manipulation inside both selections and joins. For this reason, we propose such an approach which we name "ultra-late materialization".

Further, we describe recent modifications of PosDB — a distributed, disk-based column-store. These modifications allowed us to implement a flexible query processing model. Relying on it, we have implemented a number of late materialization variants, including our approach.

Finally, we empirically evaluate the performance of ultra-late materialization and classic strategies. We also compare it with two industrial-grade disk-based systems: PostgreSQL and MariaDB Column Store. Experiments demonstrate that our variant of late materialization outperforms the closest competitor (MariaDB Column Store) by 50% which makes further investigation worthwhile.

1 INTRODUCTION

Unlike row-stores, column-stores can operate not only on data, but on positions as well (or row IDs, offsets, etc.) [3]. Usually, positions are employed on the lower levels of query plan, e.g., if there is a very selective predicate on some column, then it may be worthwhile to evaluate it first, and then perform positional lookup for values of other attributes.

Operating on positions opens a number of different query processing strategies centered around so-called "materialization". Roughly speaking, materialization is the process of switching positions back into values. Materialization eventually has to happen at some point during query execution, since the user needs tuples instead of positions.

The literature presents two main strategies: early (EM) and late materialization (LM). The basic idea of LM is to operate on positions and defer tuple reconstruction for as long as possible. This strategy is capable of significantly conserving I/O bandwidth and reducing the CPU processing load for appropriate queries. It was extensively studied in the earliest of the new-wave column-stores [21, 33] and was deemed useful in cases when a lot of tuples are discarded by plan operators. However, the majority of modern industrial column-stores do not employ this strategy and instead rely on EM. The idea of this approach can be described as not allowing positions past filter operators. The reasons for this are the complexity of implementing an optimizer and issues related to processing disk-spilling joins.

However, there is no formal definition of late materialization; it is more of a paradigm than a fixed set of rules. Over the years, there were multiple proposals which concerned position-based query processing with LM support, each involving different techniques, use-cases, and aims. Despite late materialization being a well-established topic, there is motivation to study the various approaches:

- More than ten years have passed since the last of materialization experiments. There are various accumulated changes in the hardware, such as the improvement of SSDs and appearance of novel types of storage, such as NVMe.
- Furthermore, our review demonstrates that the research landscape concerning the processing of joins in a late materialization environment is incomplete and needs expansion. The majority of reviewed studies addressed queries containing not more than a single join.
- There are novel applications of position enabled-processing such as systems for visual analytics.

Thus, there is a strong call for taking a second look into materialization.

In this paper we review existing approaches in order to compare and classify them. For this, we examine query plans and study position-related techniques that they employ. As the result, we come to the conclusion that there are two types of late materialization: late materialization in selection parts of the plan and late materialization in joins. We demonstrate that there are yet to be studies on how to efficiently combine these approaches for disk-based systems.

Following this, we propose the *ultra-late materialization* approach which efficiently combines both types of late materialization techniques.

Next, we describe the recent modifications of PosDB which allowed us to greatly expand its query processing model. It now offers substantial flexibility in terms of admissible query plans and makes it possible to implement different types of materialization strategies. We have implemented the proposed *ultra-late materialization* as well as several other materialization approaches.

Finally, we evaluate these strategies, comparing them with each other and running additional experiments with two industrial DBMSes, one of which is a column-store.

Overall, the contributions of this paper are the following:

(1) A discussion of LM approaches, examination of their supported query plans, their comparison and classification.
(2) A proposal for *ultra-late materialization*: a materialization strategy that allows to operate on positions in both selections and joins.
(3) A description of PosDB architecture and its recent modifications which made it possible to implement a number of different materialization strategies.
(4) An experimental evaluation of these strategies inside PosDB and additionally using two industrial systems — row- and column-store, using the Star Schema Benchmark.

2 BACKGROUND, RELATED WORK, AND MOTIVATION

2.1 What are Early and Late Materialization? A Classification of Approaches

Materialization is a process of combining data from individual columns into wide records [3]. There are several possible materialization strategies, each of which defines its own query plans and types of employed operators.

There are two types of approaches: early materialization and late materialization. The first one is adopted by many so-called [3] naive column-stores which only store data in columnar format and use this feature only to read a subset of columns which are necessary for a particular query. During query processing, they immediately decompress data, form rows and then proceed to evaluate query similarly to row-stores. Therefore, they do not process data in columnar form.

Although early materialization provides some benefits — lower volumes of data read from disk (compared to row-stores), it is possible to do even better. The idea of late materialization is to defer record reconstruction to a later moment. The goal is to find a better query plan which becomes available in the context of this strategy. Usually, implementing late materialization requires additional efforts such as devising novel query processing models and operators.

Unlike early materialization systems, late materialization ones not only store, but also process columnar data, aggressively using positions inside the query executor. The idea is to conserve disk bandwidth using knowledge of selectivities.

We can classify all studies by the place of query plan which employs positions. Query plans have a fairly regular structure: selections are placed on the bottom and joins usually reside above. The capability to employ positions inside each of these parts corresponds to a variant of late materialization. Thus, there are two places where late materialization can be implemented: inside selections and inside joins.

Late materialization has been studied for several decades. However, it was successfully implemented only in a handful of studies. We will review them below.

2.2 Early Days of Late Materialization: Works of G. Copeland et al.

One of the earliest studies that considered a variant of late materialization was a series of papers by G. Copeland et al. [14, 22], published in the 80's. The authors proposed a query execution approach that took into account positions in the table, which they called *surrogate attributes*. Data was stored in two-column tables made up from records containing <*surrogate attribute*, value> pairs.

To represent joins, the authors employed a join index [36]. However, unlike modern systems, they proposed to pre-compute and store the join indexes with the rest of the data on disk.

The central idea of the approach is to process sets consisting of two-column tables which may be either data columns with assigned pivot attributes or join indexes. The processing is based on a pivot surrogate — a column which other columns are joined to using an N-ary join.

The overall process consisted of several phases:

(1) The Selection phase, during which predicates are applied to individual tables. This process resulted in two types of intermediates: the ones which contained only *surrogate attribute* and <*surrogate attribute*, value> records.
(2) The Pivot phase, during which results from individual tables are "stitched" together. For this, intermediates obtained in the previous phase and stored join indexes are joined via an N-ary join. An optimal *pivot surrogate* is selected and all other itermediates are joined to it. This phase results in a set of two-column tables consisting of <*pivot surrogate*, *surrogate attribute*> records.
(3) The Materialization phase, which processes each table individually: each attribute necessary to answer the query is read from disk to construct a set of two-column tables containing <*pivot surrogate*, value of projected attribute> records.
(4) The Composition phase uses the N-ary join of intermediates obtained in the previous phase to construct records.

Thus, this approach may be considered as a variant of late materialization which operated on positions inside selections and joins. However, one may say that it was not pure late materialization, but hybrid instead, since after the value materialization phase the query executor continued to operate on both positions (*pivot surrogate*) and values.

The proposed processing scheme was rigid, offered little to no opportunities of query optimization, but more importantly, it did not explore available options of performing selections the way C-Store did, which is discussed further.

These drawbacks, at least partly, come from the fact the paper was published several years before the release of the famous Volcano [16] paper.

2.3 Late Materialization in Fractured Mirrors

The Fractured Mirrors paper [29] also explored late materialization. Its authors designed a system in which row- and column-store data layouts were combined. They used TPC-H [1] queries to benchmark and study plans produced by their approach.

The core of the approach is the DSMScan operator, which is a specialized variant of multi-way merge-join. To demonstrate the produced plans, the authors used several queries, two of which we will consider. They are presented in Figure 1.

The first one is Q1* which is essentially the TPC-H Q1 query, in which the predicate was inverted to make it more selective.

The leftmost shows the row-store mode plan. There, query executor performs an index scan and sends qualifying tuples into the aggregation operator. The columnar plan has a modified index scan operator that returns record ids, which are then fed into six columnar access operators. DSMScan orchestrates them

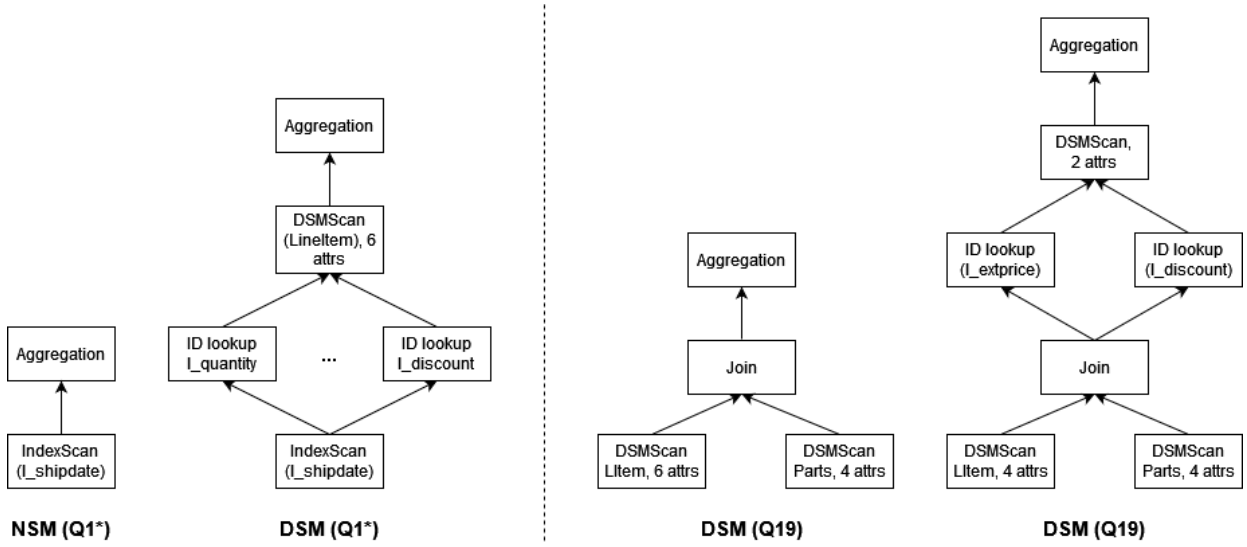


Figure 1: Query plans in Fractured Mirrors, adapted from original paper [29]

and creates records which are passed to the aggregation operator. Therefore, this query plan has late materialization in the selection part.

The right part of Figure 1 contains plans for the TPC-H Q19, which consists of selections and a single join. It shows two DSM plans — with early and late materialization. In case of early materialization, tuples are formed at the bottom levels of the plan and then joined with the tuple-based join. The late materialization plan is different: at the first step, only four out of six attributes belonging to LineItem (LItem in the figure) table are accessed. Then the tuple-based join is run and the remaining two attributes are read from disk. Therefore, one may say that for l_extprice and l_discount late materialization is performed. However, in order to perform reaccess, positions are kept together with values after the join. This way, it is more of a hybrid than late materialization technique.

Therefore, this variant includes late materialization in selections and partially in joins. However, similarly to the approach described in Section 2.2 it lacked C-Store-like scans and it did not deeply explore join performance. In this study, only a single considered query contained more than one join.

2.4 FlashJoin: late materialization inside PostgreSQL

In the previous study, joins "captured" some of attributes that were materialized earlier. An alternative approach was proposed by the HP Labs research group [34].

They proposed a new join operator called FlashJoin consisting of two components: *fetch kernel* and *join kernel*. The *join kernel* will process only the two necessary attributes, one for each table. The goal of the *fetch kernel* is to read attributes necessary for further query processing.

An example of this processing scheme is presented in Figure 2 (adapted from the original paper).

In this query, the first operator joins $R1$ and $R2$ using the $R1.A = R2.D$ condition. The result of its *join kernel* is the list of position pairs, one for each table. The next join requires the G attribute, therefore the *fetch kernel* reads it. In a similar manner, the next operator performs the join. However, this time the *fetch*

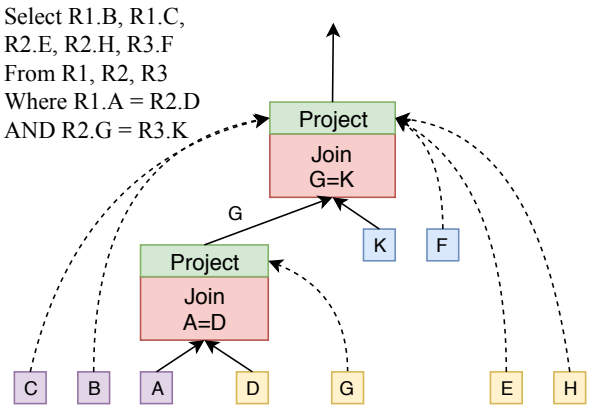


Figure 2: FlashJoin example

kernel reads five attributes, belonging to all three tables. Thus, this approach supports late materialization inside join sequences.

The approach was implemented inside the PostgreSQL system and evaluated in an SSD environment. However, their experiments used queries featuring no more than two joins. Finally, this study did not consider late materialization inside selections.

2.5 Late materialization in C-Store

In the C-Store [33] system, late materialization was implemented differently. It relied on a special data structure: a *multi-column* block. It describes a consequent horizontal fragment of a table, transferred between operators in a Volcano-style manner. Each block contains [3]:

- a position descriptor which indicates the status of each record — valid (i.e. satisfies the predicate) or not, and
- values of several columns, each one of which may be compressed by its own algorithm.

The *multi-column* block made it possible to implement late materialization inside selections [6]. Consider, for example, a query which scans the LineItem table (the SSB [2] one) and selects records consisting of *Linenum* and *Shipdate* attributes which satisfy $Shipdate < const1\ AND\ Linenum < const2$ condition.

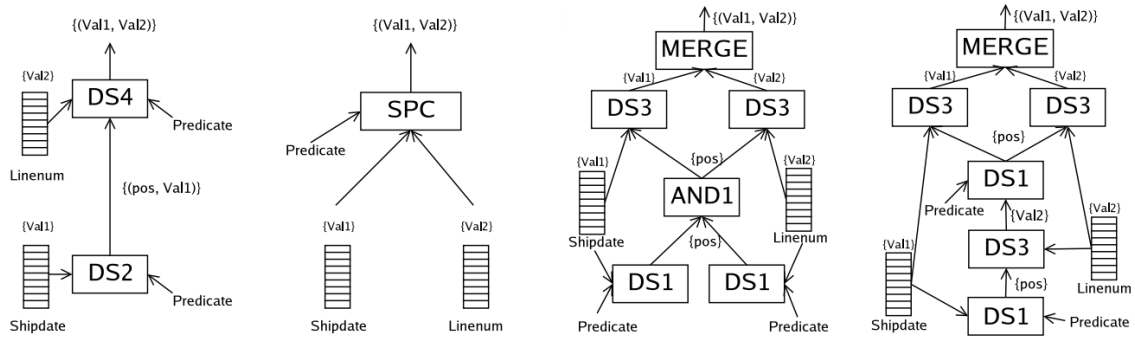


Figure 3: Possible query plans, figure taken from the original paper [6].

The paper discussed four approaches to evaluating this query, which are presented in Figure 3.

The first plan uses the DS2 operator to read the *Shipdate* column and filter it to produce positions with the corresponding values. Next, DS4 fetches values from the *Linenum* column using the position, applies the predicate and constructs records that satisfy it. The second plan does not use positions at all, it simply reads both columns synchronously and constructs the result. Thus, query executor reads each column exactly once.

The paper called the first two plans early materialization, but in fact the first one is a hybrid one since it operates both on positions and values.

The last two plans belong to the late materialization class. The third plan at first obtains positions satisfying each predicate, then intersects them and uses them to read the corresponding values. In the end, they are merged to construct records. The fourth plan filters *Shipdate* first, and then resulting positions are used to read *Linenum* and filter using the corresponding predicate. These refined positions are used to read values, and then a record is constructed similarly to the previous plan. The fourth plan is beneficial when the *Shipnum* predicate is very selective, which makes it possible to read a very small number of blocks from the *Linenum* column.

The C-Store was oriented towards projection-based processing. Projection is a pre-joined collection of columns that may involve multiple logical tables. For this reason C-Store preferred joinless queries, although it was possible to run joins between projections using pre-computed join indexes. Later studies [24] revealed that this idea was abandoned due to excessive computing costs.

A subsequent paper [6] evaluated late materialization in joins, but using only a single query and this query had only a single join. One of the reasons was the use of an unsuitable data structure (*multi-column*) to store post-join intermediates which restricted further experiments.

2.6 Late materialization in MonetDB family

The MonetDB [10, 21] system offers the largest number of options of materialization processing. Its idea is to operate on BATs (Binary Association Tables) — special tables that consist of one or two columns. Each column may contain either data itself (values) or positions. All MonetDB operators take BATs as input and also output BATs, thus forming an algebra.

In this query processing model, late materialization naturally appears both in selections and in joins [10]:

- After a selection, the result is a BAT containing positions.
- Join processing also results in BATs that contain positions.

Table 1: Classification of LM approaches

Study	Focus on LM in selections	Focus on LM in joins	Type
G. Copeland et al. [14, 22]	partial	full	disk
Fractured Mirrors [29]	partial	partial	disk
FlashJoin [34]	none	full	disk
C-Store [6, 33]	full	partial	disk
MonetDB family [10, 21]	full	full	mem
Hyrise [18]	full	none	mem

Moreover, MonetDB/X100 [9, 19] offered another late materialization option — selection vectors. The idea is to select a subset of data using a vector of positions inside some operator, without a dedicated filtration phase.

However, MonetDB is an in-memory system and its query processing model is defined by this fact. Likewise, its late materialization techniques are dependent on having all data in memory and thus have limited application in disk-based systems.

2.7 Late materialization in Hyrise

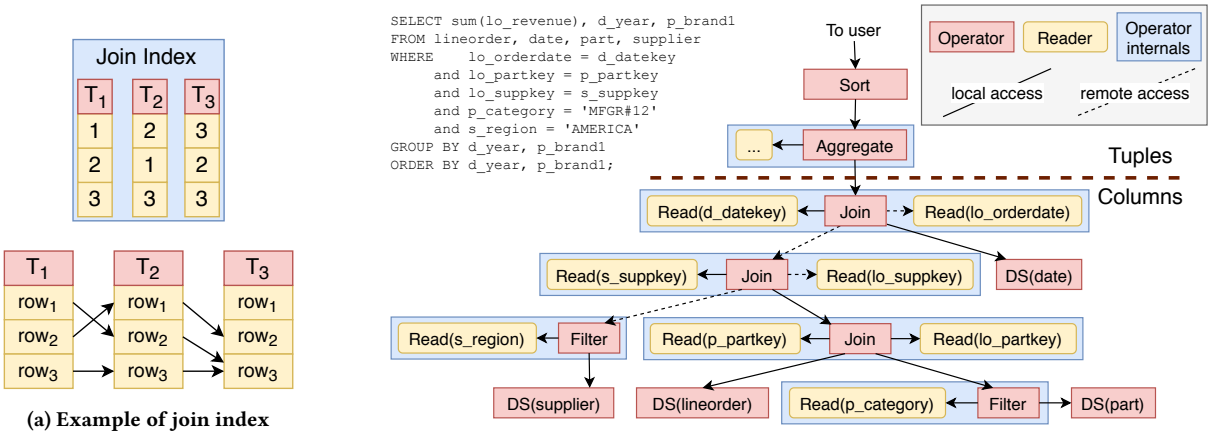
HYRISE [18] has also explored late materialization. The cited paper was an extension of study [6] (discussed in Section 2.5), whose authors studied the same query plans and adapted them for in-memory processing. Thus, they did not study its impact on joins and instead concentrated on scans.

2.8 Wrap-up and motivation for further research

Concluding this section, we can say that all existing late materialization models have some issues:

- They either support late materialization in joins only, but not in selections [29, 34], or they support a rich operator set for selections but fall behind in joins [6, 18, 33].
- They do not follow [14, 22] the now standard Volcano processing scheme, and thus are inapplicable in contemporary systems.
- They employ in-memory processing [10, 18, 21] and thus are inapplicable for disk-based DBMSes. Despite the boom of in-memory systems, there is still plenty of use-cases for the disk-based ones.

The summarized state of affairs in the area of late materialization is presented in Table 1. There, we describe how well late



(a) Example of join index

(b) Query plan example

Figure 4: PosDB'17 internals

materialization is elaborated inside selections and joins, and also system type (main memory or disk-based).

Therefore, a new disk-oriented late materialization model is needed, which will:

(1) support C-Store-style late materialization in selections,
(2) support late materialization inside joins and will allow handling arbitrary number of joins.

In this paper, we propose such an approach which we call *ultra-late materialization*. We implement it inside PosDB — a distributed disk-based column-store engine.

We run an extensive experimental study featuring several existing materialization approaches. The goal is to compare them with each other and study the resulting performance.

Finally, it is important to discuss three other "general" reasons for the re-evaluation of late materialization approaches:

- More than ten years passed since the last experiments with late materialization. Except a single paper [34], these experiments considered HDDs, state-of-the-art hardware of that time. Nowadays, SSDs are ubiquitous, and their characteristics, in particular lower random access time, may seriously improve the performance of late materialization inside joins. This may help overcome the out-of-order probing problem, which arises during late materialization inside joins. Modern SSDs have significantly reduced costs of disk seeks, which may provide efficient reading of values by positions even when position lists are not fully sorted.
- There is a resurgence of interest in late materialization-enabled systems, driven by the need of supporting provenance in systems for visual analytics [27, 28, 37]. A typical use-case is as follows: a user runs a query, visualizes it as a bar chart, and then wants to "zoom in" — that is, perform additional actions with a subset of visualized data which they select with a mouse. In such scenarios, the query executor needs to obtain records that contributed to the selected buckets and thus, position-oriented query processing is useful [37]. Therefore, an efficient late materialization model may improve performance of such applications.
- Position-enabled processing will be extremely useful for implementing functional dependency predicates inside queries [12]. A number of promising operators is described in a draft [8]. During their implementation, position-enabled query processing will play a vital role, since a lot of dependency discovery algorithms rely on partitions [7, 26], which are essentially positions.

3 SYSTEM ARCHITECTURE

The PosDB [13] project started in 2017 as a research effort to study late materialization-oriented query processing. We focused on query processing issues not studied in PosDB's predecessors, namely: LM and processing of aggregation queries (including window function processing), distributed processing in LM environments, subqueries and LM, etc. We were also interested in solving the problem of disk-spilling joins given the advancement of hardware (mostly SSDs) that happened since the time of the pioneers. In short, PosDB is a distributed and parallel column-store oriented at analytical processing in a disk-based environment.

As of 2017, query processing in PosDB was organized according to the pull-based Volcano model [16] with block-oriented processing. This model assumes that query plans are represented via trees that have operators as nodes and data flows as edges. Each operator supports an iterator interface and can produce either positions or tuples exclusively.

To represent intermediate positional data, PosDB uses a generalized join index [36]: a data structure that essentially encodes the result of a series of joins. The join index states that row r_1 of table T_1 was joined with row r_2 of table T_2 and so on. An example is presented in Figure 4a. Past the materialization point, PosDB represents data in tuples, similarly to row-stores.

The query plans in classic PosDB contain two parts: positional (columnar) data and tuples. Operators that manipulate positions (`joins`, `filters`, positional set operators) use join indexes to represent intermediate results, and the tuple part (`aggregation` and `sort` operators) is similar to row-stores. An example plan is presented in Figure 4b, where the dashed line denotes the materialization point (or the Tuples-Columns border).

Several operators such as the positional set `AND` and `OR` need only positional data, while others, i.e. `join`, `filter`, and `aggregation` also require the corresponding values. To fetch values using positions from the join index, we have introduced special entities named readers. For example, `ColumnReader` is designed

to retrieve values of a specific attribute, and `SyncReader` is designed to read values of several attributes synchronously.

PosDB is not only distributed, but also parallel: it includes the `Asynchronizer` and `UnionAll` operators. The former allows to start executing an operator subtree in a separate thread, similarly to the `Exchange` operator [17]. The latter enables the collection of data from several subtrees, all executed in their own threads, effectively providing data parallelism inside query plans.

A detailed description of the baseline architecture can be found in paper [13]. Since 2017, we have been exploring various aspects of LM-oriented processing: aggregation [35], window functions [25], compression [31], intermediate result caching [15], distributed processing [11].

Recently, the engine has been significantly extended. First of all, we have improved the disk-based capabilities by introducing a proper buffer manager and reworking access methods. This moved PosDB closer to industrial systems and allowed us to obtain higher-quality experimental data.

Next, we have extended the support of partitioning by adding hash- and range-based partitioning methods. We have also switched from a column-based partitioning scheme to a row-based one due to the former's excessive complexity of maintenance. On top of that, we have added compression support.

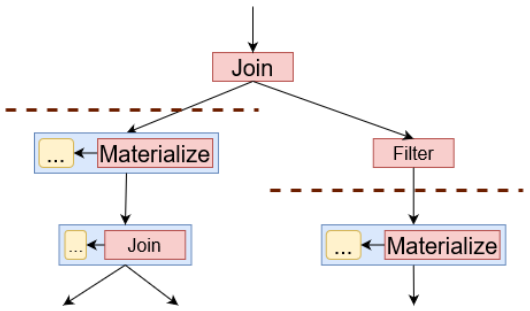


Figure 5: Novel query plans, possible in PosDB'21

Query processing faced the most extensive modifications. First of all, truly distributed operators like `distributed join` [23] and `distributed aggregation` [32] have been implemented. Next, we have implemented a number of novel tuple-based operators that significantly extended the number of allowed query plan types.

For the current study, the most important addition was the introduction of a tuple-based `join`, `filter`, `cross-product`, and `aggregation` operators. This allowed to greatly expand the number of admissible query plans and made possible an implementation of a several late materialization strategies. Unlike PosDB'17, now there may be multiple materialization points inside the query plan, as shown in Figure 5. In this figure, the top `join` and the `filter` are tuple-based operators. Thus, it is now possible to construct query plans that may have tuple data representation inside subplans. This may be beneficial in cases when running the whole query using late materialization is suboptimal.

Finally, we have laid foundations for hybrid materialization (HM) by introducing a special intermediate representation. For now, it is supported only in single-tabled parts of a plan.

4 QUERY EVALUATION STRATEGIES

From the query processing standpoint, the modifications that PosDB has undergone over the years have allowed us to implement several different approaches to materialization:

(1) early materialization,
(2) late materialization,
(3) hybrid materialization,
(4) *ultra-late materialization.*

The first two are the classic approaches, described in the early 10's [3]. They will be used as the baselines. Hybrid materialization is our recent development, aimed specifically at solving a particular issue that arises in the distributed case during position-based query processing. We do not include it in the benchmark since it does not fit into the evaluation scenario. Nevertheless, we believe it is important to mention it here, since it demonstrates an important issue that will inevitably arise in any position-based query processing model run in a distributed environment. Finally, *ultra-late materialization* is the core approach discussed in this paper.

Consider, for example, query Q.11 of the Star Schema Benchmark [2]:

```
SELECT
    SUM(lo_extendedprice*lo_discount) AS revenue
FROM
    lineorder, date
WHERE
    lo_orderdate = d_datekey AND
    lo_discount BETWEEN 1 AND 3
    d_year = 1993 AND
    AND lo_quantity < 25;
```

This query has a join, three predicates, and an aggregation. Query processing model available in PosDB allows to produce several different query plans for each of materialization strategies. Let us consider them.

4.1 Ultra-Late Materialization

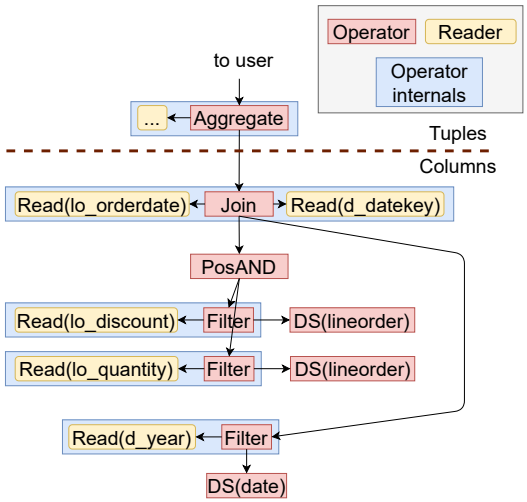


Figure 6: Query evaluation strategies for Q.11: ultra-late materialization

As stated in Section 2.8 the idea of the *ultra-late materialization* is to combine late materialization in selections and joins. This

makes possible operating on positions and thus deferring tuple reconstruction for as long as possible. Therefore, such plans should perform all filters and joins below the materialization point, thus using positional representation (join indexes).

The *ultra-late materialization* plan for the example query is presented in Fig. 6. At the first step, two position lists for the LINEORDER table are obtained by applying predicates to each of the involved columns. This is achieved via using `Filter` operators with readers corresponding to the involved columns.

Next, the `PosAND` operator is used to intersect position lists in order to obtain IDs of LINEORDER records that conform to both predicates from the query.

The DATE table is processed in the same way, but using only a single `Filter` operator which works with the d_year column.

The `Join` operator is located higher up in the query plan. Note that this is a positional join operator, i.e. it operates on join indexes. In this particular case, they are one-dimensional, i.e. plain position lists.

Despite being positional, this join still requires the data values themselves. Therefore, it needs readers for the corresponding columns which can be seen in the figure.

This operator produces a list of position pairs, each corresponding to LINEORDER and DATE respectively. The list is fed into the `aggregation` operator which also materializes the final result.

4.2 Early Materialization

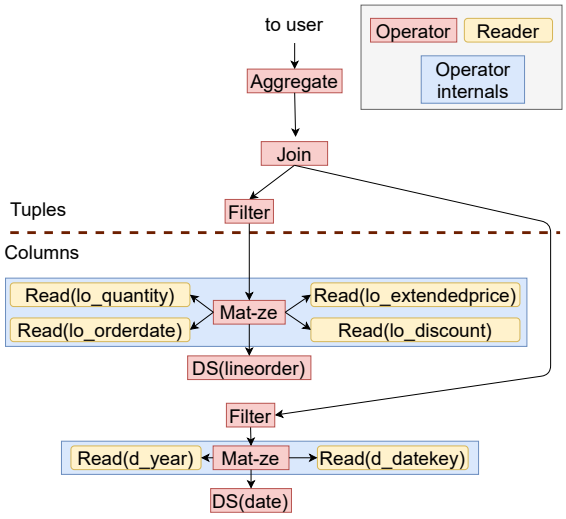


Figure 7: Query evaluation strategies for Q.11: early materialization

Despite being well-known [3], early materialization is a query evaluation strategy that has been implemented in PosDB since only recently. The idea of this strategy is to imitate a classic row-store: at first, a tuple representation is built using only the necessary attributes and then all required operators are run. Therefore, filters, joins and other operators are performed on the tuple-based representation.

Fig. 7 contains the EM plan for the considered query. It possesses several distinctive features:

- Tuple materialization happens at the bottom of the plan, i.e. materialization operators are essentially the first non-trivial ones.
- In a EM strategy, readers are present only in materialization operators — all other operators are tuple-based. The Tuples-Columns border resides on the lowest level among all considered strategies.

Finally, note that in this strategy only attributes necessary for the further query execution are materialized, i.e. not every one of the corresponding tables.

4.3 Late Materialization

Another baseline approach was the C-Store-style late materialization in selections. As described in Section 2.5, its core idea concerns operating on positions in the bottom parts of the plan only. That is, predicates are evaluated using join index representation, but costlier operators such as joins are evaluated using the classic tuple-based representation.

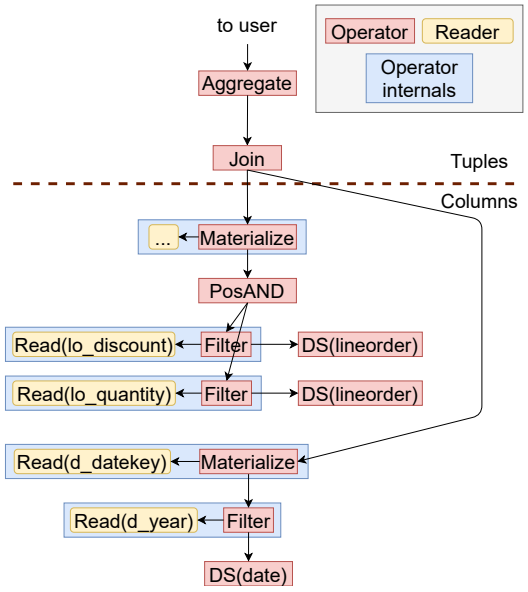


Figure 8: Query evaluation strategies for Q.11: late materialization

The LM plan of Q.11 is presented in Fig. 8. It can be seen that in this query evaluation strategy, predicates are evaluated in the positional part of the plan. Similarly to the *ultra-late materialization* plan, individual position lists for LINEORDER table are intersected. However, an operator of explicit materialization is run next. It transforms the join-index representation into a tuple-based one. Note that, similarly to the EM approach, this tuple representation contains only those attributes that would be requested by subsequent plan operators.

Unlike the EM strategy, which materialized four attributes, only three attributes will be materialized for the LINEORDER table: lo_orderdate, lo_discount, and lo_extendedprice. The lo_quantity attribute will not be materialized, since LINEORDER has already been filtered by the corresponding predicate and subsequent plan operators do not need this attribute.

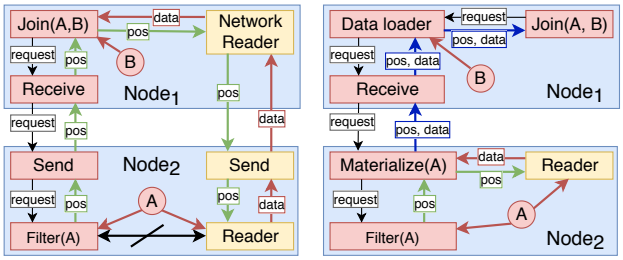
Concerning the DATE table, only the d_datekey attribute will be materialized. Therefore, higher plan levels will be reached only by two attributes of limited use: lo_orderdate and d_datekey. Both will be discarded immediately after the `join` operator since they do not belong to the answer. At the same time, they should be materialized, since they are needed to perform this join.

Finally, one can see that tuple-based part of the plan contains `aggregation` and `join` operators. Both of them are tuple-based and therefore contain no readers.

4.4 Hybrid Materialization

Intensive data reaccess within a query plan is an inherent characteristic of the late materialization approach. The existing papers usually consider the local case [3, 6, 30], where the induced overhead can be reduced by an efficient buffer manager. Then, the advantages of late materialization (compression [4], cache-consciousness [3]) outweigh its drawbacks.

However, in a distributed case, the overhead is significantly higher, because data has to be reaccessed via network. At the same time, a network cache is usually less robust than a buffer manager. Moreover, the common query processing workflow may require meaningless round-trips of position blocks between nodes.



(a) Late materialization **(b) Hybrid materialization**

Figure 9: PosDB: `Join-Filter` in a distributed case

Consider an execution of a `Filter-Join` operator sequence in PosDB using tables A and B, shown in Figure 9a. This operator sequence is commonly found in query plans. It also has the same workflow as a necessary step of the reshuffling process during a distributed join.

With the late materialization approach, the `Filter` operator processes table A and passes the resulting position blocks over the network to $Node_1$. Then the `Join` operator has to return the same blocks to $Node_2$ with a request for table A data. Thus, a meaningless network communication cycle exists.

Moreover, in the general case, the `Join` operator can process position blocks from a set of `Filter` operators executed both locally and on different remote nodes. Table A data may be requested by the next query plan operator again, immediately after the `Join`. Therefore, a specific optimization for a particular "`Filter-Join`" case is not sufficient. Instead, the whole query execution model should be improved.

PosDB solves the problem with two major changes: hybrid materialization support and moving out the data acquisition logic from operator internals into a separate optimizer-configurable entity.

First, hybrid materialization [3, 6] allows to pass both data and positions between operators. Thus, the required table A data is materialized immediately after the `Filter` operator and sent to `Join` together with position blocks, see Figure 9b. The meaningless position round-trip is eliminated, and the network communication overhead decreases nearly twice.

In existing articles, hybrid materialization is treated as an one-way step from purely position-based to tuple-based representation [3]. However, the distributed case, which we have just described, establishes a novel, completely different context. And in this context, reverting to late materialization just after the `Join` operator may be preferable.

Another major change is the encapsulation of the data acquisition process in an independent separate entity. This allows to combine both late and hybrid materialization when accessing several replicas located both locally and remotely. Moreover, it can be done seamlessly to an operator itself due to the unified interface and C++ template techniques.

Encapsulating the data acquisition strategy also increases opportunities for query plan tuning. A query optimizer can freely choose not only the particular nodes the data is received from, but also the order and the intensity of their usage. It can be especially important in the case of a distributed join. The dynamic reshuffling stage may consider several data streams from remote `Multiplexors` as well as local data from disk.

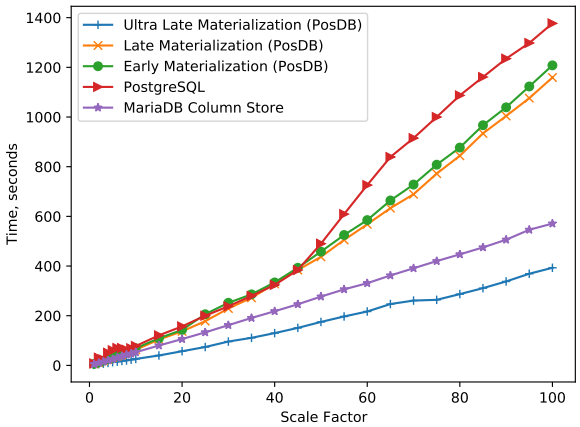


Figure 10: Total runtime of considered strategies

5 EXPERIMENTAL EVALUATION

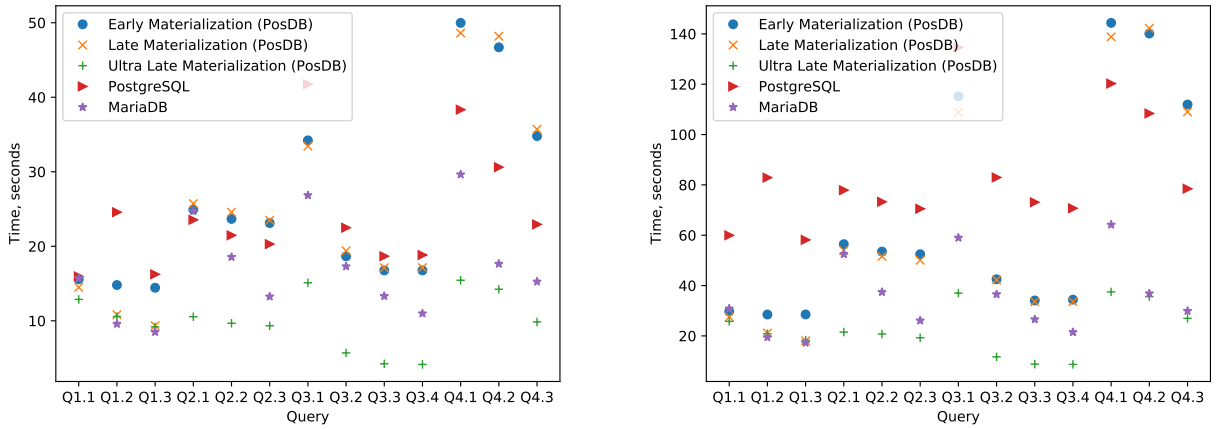
To study the considered strategies, we have performed a twofold experimental evaluation. First, we have compared them to each other within PosDB. Second, we have selected the two most similar industrial competitors: a disk-based row-store (PostgreSQL) and a column-store (MariaDB Column-Store). We centered our baseline around open-source-licensed systems due to the DeWitt Clause [20], which they are free of. At the same time, they are mature enough to be used in industrial applications.

Since our evaluation was performed in a centralized environment, we did not consider the HM strategy.

In order to run our experiments, we have used the Star Schema Benchmark [2], varying the Scale Factor (SF) in a $[1, 100]$ range. Thus, the total data volume ranged from 0, 8 to 80 GB. We have selected SSB because of its focus on engine comparison instead of optimizer output. This is achieved by the relative simplicity of queries: they do not allow much variability in terms of admissible query plans.

In order to provide an equal environment for all systems, the following was done during the test runs:

- Data compression, JIT-compilation, SIMD, and indexes were not used.
- DBMSes were not tuned, default parameters were used.
- Default data plans were used.



(a) Scale Factor 40

(b) Scale Factor 80

Figure 11: Per-query breakdown

- Hash-based versions of joins were used: the smaller table was put into a hash table.
- Intra- and inter- query parallelism was turned off. Distributed capabilities were not used.

All systems were run without warm-up, OS page cache was dropped before launching each experiment run. Since each run yielded almost the same numbers, we present results averaged over ten launches. Experiments were run on a desktop with the following hardware: AMD Ryzen 9 3900X, GIGABYTE X570 AORUS ELITE, Kingston HyperX FURY Black HX434C16FB3K2/32 32GB, 512 GB SSD M.2 Patriot Viper VPN100-512GM28H.

Software: Ubuntu 20.04 LTS, GCC 9.3.0, PostgreSQL 12.5, MariaDB Column-Store 1.5.2 on MariaDB Community Server 10.5.8.

First of all, we have measured the total runtime on the whole SSB workload (Fig 10). Here, it can be seen that PosDB's *ultra-late materialization* strategy performs the best with all SF values. The second best, MariaDB Column Store, is almost 50% slower. The next three, LM, EM, and PostgreSQL, show more interesting behavior. In the $[0, 40]$ range, all three of them have approximately the same run times. However, starting from SF 40 we can see the following:

(1) Late Materialization is consistently $\approx$ 5% faster than Early Materialization.
(2) PostgreSQL loses about 15% to both Early and Late Materialization. This happens due to PostgreSQL running out of memory to cache pages in its buffer manager.
(3) MariaDB Column Store beats PostgreSQL by more than two times.

The second experiment provides an in-depth view into the results by studying the performance of individual queries. We have selected two important points: SF 40 and SF 80. The first one is the point where all tuple-oriented strategies show approximately equal performance, and the second one is the point where the graphs fully diverge.

The results are presented in Fig. 11:

- For almost all queries, PosDB's *ultra-late materialization* yields the best result. The only exception are the queries of the first flight, on which MariaDB Column Store shows very close or better results.
- Fig. 11b shows that increasing the data size leads to PostgreSQL losing on the first three query flights out of total four.
- It is interesting to note that PosDB's LM and EM strategies are in-between in terms of performance on all but the fourth flight. This query flight is the most complex one, featuring four joins.

6 FUTURE PLANS

(1) The critical issue for adoption of late-materialization in the 00's were disk-spilling joins. LM-induced out-of-order probing [3, 5] killed the performance on HDDs and even SSDs of that time. Nowadays, hardware and software specifications have improved, and novel storage such as NVMe has appeared. All this calls for a second round of materialization evaluation that will employ joins with tables larger than main memory.
(2) We plan to further explore Hybrid Materialization not only because it looks promising for addressing the disk-spilling join problem, but it also offers query plans that may improve query performance further.
(3) Extending the system itself: implementing a parser, a query optimizer, supporting subqueries, indexes, implementing FD-manipulation predicates [8, 12], and so on.

7 CONCLUSION

In this paper, we reviewed existing approaches to implementing late materialization in order to compare and classify them. Then we demonstrated that there are two types of late materialization — one concerning selections and one concerning joins. We showed that there were no studies on how to efficiently combine these approaches for disk-based systems. Following this, we proposed the *ultra-late materialization* which makes it possible to implement both of these approaches inside a single query processing model.

Further, we have presented a new version of PosDB. Modifications that it underwent over the years allowed us to greatly expand its query processing model. Now it offers substantial flexibility in terms of possible query plans and allows to implement

different types of late materialization strategies. We have implemented the proposed *ultra-late materialization* as well as several other materialization approaches. Following the description of basic system architecture and recent modifications, we have discussed these strategies, using an SSB query as an example.

Next, in order to study materialization strategies we have performed experimental evaluation. We have compared them not only with each other, but with industrial-grade systems as well — PostgreSQL and MariaDB Column Store. Our experiments demonstrated that our *ultra-late materialization* provides 50% better performance than MariaDB Column Store, and is more than three times faster than PostgreSQL.

ACKNOWLEDGMENTS

We would like to thank Anna Smirnova for her help with the preparation of the paper.

REFERENCES

[1] [n.d.]. TPC Benchmark H. Decision Support. Version 2.17.1. http://www.tpc.org/tpch.

[2] 2009. P. E. O'Neil, E. J. O'Neil and X. Chen. The Star Schema Benchmark (SSB). http://www.cs.umb.edu/~poneil/StarSchemaB.PDF. Accessed: 10/09/2017.

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

[4] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. ACM, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548

[5] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. 2009. Column-oriented Database Systems. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1664–1665. https://doi.org/10.14778/1687553.1687625

[6] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. 2007. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis (Eds.). IEEE, 466–475.

[7] Tobias Bleifuß, Susanne Bülow, Johannes Frohnhofen, Julian Risch, Georg Wiese, Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. 2016. Approximate Discovery of Functional Dependencies for Large Datasets. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (CIKM '16)*. Association for Computing Machinery, New York, NY, USA, 1803–1812. https://doi.org/10.1145/2983323.2983781

[8] Nikita Bobrov, Kirill Smirnov, and George A. Chernishev. 2020. Extending Databases to Support Data Manipulation with Functional Dependencies: a Vision Paper. *CoRR* abs/2005.07992 (2020). arXiv:2005.07992 https://arxiv.org/abs/2005.07992

[9] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-pipelining query execution. In *In CIDR'05*.

[10] Peter A. Boncz and Martin L. Kersten. 1999. MIL Primitives for Querying a Fragmented World. *The VLDB Journal* 8, 2 (Oct. 1999), 101–119. https://doi.org/10.1007/s007780050076

[11] George Chernishev, Viacheslav Galaktionov, Valentin Grigorev, Evgeniy Klyuchikov, and Kirill Smirnov. 2017. A Study of PosDB Performance in a Distributed Environment. In *Proceedings of the 2017 Software Engineering and Information Management (CEUR Workshop Proceedings)*, Vol. 1864. CEUR-WS.org, Saint-Petersburg, Russia.

[12] George A. Chernishev. 2020. Making DBMSes Dependency-Aware. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/gongshow2020/gongshow/abstracts/cidr2020_abstract67.pdf

[13] 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. https://doi.org/10.1134/S0361768818010024

[14] George P. Copeland and Setrag N. Khoshafian. 1985. A decomposition storage model. *SIGMOD Rec.* 14, 4 (1985), 268–279. https://doi.org/10.1145/971699.318923

[15] Viacheslav Galaktionov, Evgeniy Klyuchikov, and George A. Chernishev. 2020. Position Caching in a Column-Store with Late Materialization: An Initial Study. In *Proceedings of DOLAP@EDBT/ICDT 2020 (CEUR Workshop Proceedings)*, Il-Yeol Song, Katja Hose, and Oscar Romero (Eds.), Vol. 2572. CEUR-WS.org, 89–93. http://ceur-ws.org/Vol-2572/short14.pdf

[16] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (June 1993), 73–169. https://doi.org/10.1145/152610.152611

[17] G. Graefe. 1994. Volcano — An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (Feb. 1994), 120–135. https://doi.org/10.1109/69.273032

[18] Martin Grund, Jens Krueger, Matthias Kleine, Alexander Zeier, and Hasso Plattner. 2011. Optimal query operator materialization strategy for hybrid databases. In *Proceedings of the 2011 Third International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA '11)*. IARIA, 169–174.

[19] 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

[20] Joseph M. Hellerstein and Michael Stonebraker. 2005. *Readings in Database Systems.* The MIT Press, pp. 96–.

[21] 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), 40–45. http://sites.computer.org/debull/A12mar/monetdb.pdf

[22] Setrag Khoshafian, George P. Copeland, Thomas Jagodis, Haran Boral, and Patrick Valduriez. 1987. A Query Processing Strategy for the Decomposed Storage Model. In *Proceedings of the Third International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 636–643. http://dl.acm.org/citation.cfm?id=645472.655555

[23] Donald Kossmann. 2000. The State of the Art in Distributed Query Processing. *ACM Comput. Surv.* 32, 4 (Dec. 2000), 422–469. https://doi.org/10.1145/371578.371598

[24] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1790–1801. https://doi.org/10.14778/2367502.2367518

[25] Nadezhda Mukhaleva, Valentin D. Grigorev, and George A. Chernishev. 2019. Implementing Window Functions in a Column-Store with Late Materialization. In *Model and Data Engineering - 9th International Conference, MEDI 2019, Toulouse, France, October 28-31, 2019, Proceedings (Lecture Notes in Computer Science)*, Klaus-Dieter Schewe and Neeraj Kumar Singh (Eds.), Vol. 11815. Springer, 303–313. https://doi.org/10.1007/978-3-030-32065-2_21

[26] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *Proc. VLDB Endow.* 8, 10 (jun 2015), 1082–1093. https://doi.org/10.14778/2794367.2794377

[27] Fotis Psallidas and Eugene Wu. 2018. Demonstration of Smoke: A Deep Breath of Data-Intensive Lineage Applications. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1781–1784. https://doi.org/10.1145/3183713.3193537

[28] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained Lineage at Interactive Speed. *Proc. VLDB Endow.* 11, 6 (2018), 719–732. https://doi.org/10.14778/3184470.3184475

[29] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. 2002. A case for fractured mirrors. In *Proceedings of the 28th international conference on Very Large Data Bases (VLDB '02)*. VLDB Endowment, 430–441. http://dl.acm.org/citation.cfm?id=1287369.1287407

[30] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. 2013. Materialization Strategies in the Vertica Analytic Database: Lessons Learned. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 1196–1207. https://doi.org/10.1109/ICDE.2013.6544909

[31] Alexander Slesarev, Evgeniy Klyuchikov, Kirill Smirnov, and George A. Chernishev. 2021. Revisiting Data Compression in Column-Stores. In *Model and Data Engineering - 10th International Conference, MEDI 2021, Tallinn, Estonia, June 21-23, 2021, Proceedings (Lecture Notes in Computer Science)*, J. Christian Attiogbé and Sadok Ben Yahia (Eds.), Vol. 12732. Springer, 279–292. https://doi.org/10.1007/978-3-030-78428-7_22

[32] Cluet Sophie and Moerkotte Guido. 1995. Efficient Evaluation of Aggregates on Bulk Types. In *Proceedings of the Fifth International Workshop on Database Programming Languages*. 8.

[33] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. VLDB Endowment, 553–564. http://dl.acm.org/citation.cfm?id=1083592.1083658

[34] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. 2009. Query Processing Techniques for Solid State Drives. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 59–72. https://doi.org/10.1145/1559845.1559854

[35] A. Tuchina, V. Grigorev, and G. Chernishev. 2018. On-the-fly filtering of aggregation results in column-stores. *CEUR Workshop Proceedings* 2135 (2018), 53–60.

[36] Patrick Valduriez. 1987. Join Indices. *ACM Trans. Database Syst.* 12, 2 (June 1987), 218–246. https://doi.org/10.1145/22952.22955

[37] Eugene Wu. 2021. Systems for Human Data Interaction (keynote). In *Proceedings of the 2nd Workshop on Search, Exploration, and Analysis in Heterogeneous Datastores (SEA-Data 2021) co-located with 47th International Conference on Very Large Data Bases (VLDB 2021), Copenhagen, Denmark, August 20, 2021*, Davide Mottin, Matteo Lissandrini, Senjuti Basu Roy, and Yannis Velegrakis (Eds.).