On-the-Fly Filtering of Aggregation Results in Column-Stores

Anastasia Tuchina
Saint Petersburg State University
Email: anastasia.i.tuchina@gmail.com

Valentin Grigorev
Saint Petersburg State University
Email: valentin.d.grigorev@gmail.com

George Chernishev
Saint Petersburg State University
JetBrains Research
Email: chernishev@gmail.com

***Abstract*—Aggregation is a database operation that aims to provide basic analytic capabilities by partitioning source data into several groups and computing some function on values belonging to the same group. Nowadays it is common in databases, and especially in the OLAP domain, which is a primary venue for column-stores.**

In this paper we propose a novel approach to the design of an aggregation operator inside a column-store system. The core of our approach is an analysis of predicates in the HAVING-clause that allows the runtime pruning of groups. We employ monotonicity and codomain analysis in order to detect groups in which predicates would never be satisfied. Eventually, we aim to save I/O and CPU costs by discarding groups as early as possible.

We start by providing a high-level overview of our approach and describe its use-cases. Then, we provide a short introduction into our system and describe a straightforward implementation of an aggregation operator. Next, we provide theoretical foundations for our approach and present an improved algorithm. Finally, we present an experimental validation of our approach inside PosDB — a distributed, disk-based column-store engine that features late materialization and block-oriented processing. Experiments using an SSD drive show that our approach can provide up to 5 times improvement over the naive version.

I. Introduction

Aggregation is rather common in databases and, in fact, it forms the basis of OLAP. For example, the TPC-H benchmark [1] does not contain a single query which does not involve aggregation. Therefore, in the early 80's the scientific community recognized the importance of efficient aggregation processing. While there are hundreds of studies concerning aggregation in row-stores, the column-store processing is less explored.

Column-stores are a relatively recent development [2]. Unlike classic approaches, where all attributes of every record are kept together, column-stores employ the opposite idea — they store each attribute separately. This leads to a number of challenges as well as opportunities in query processing.

In this paper we propose a novel technique intended for optimizing the processing of aggregation queries inside column-stores. Our approach concerns analysis of predicates in the `HAVING`-clause and relies on a simple idea: terminate processing of a group if, judging by the already processed data, the `HAVING`-predicate will never be satisfied. Thus, it should be possible to save I/O and CPU costs related to evaluation of aggregation functions located in the `SELECT`-clause.

In order to illustrate this, let us consider the following example (adapted from Q1 of TPC-H):

```
SELECT
  l_returnflag, l_linestatus,
  SUM(l_quantity) as sum_qty,
  SUM(l_extendedprice) as sum_base_price,
  SUM(l_extendedprice * (1 - l_discount))
                        as sum_disc_price,
  SUM(l_extendedprice * (1 - l_discount) *
             (1 + l_tax)) as sum_charge,
  AVG(l_quantity) as avg_qty,
  AVG(l_extendedprice) as avg_price,
  AVG(l_discount) as avg_disc,
  COUNT(*) as count_order
FROM lineitem
GROUP BY l_returnflag, l_linestatus
HAVING SUM(l_quantity) < 1000000
ORDER BY l_returnflag, l_linestatus
```

In this case naive processing can be organized as follows:

1) Rewrite query in the form

   SELECT *
   FROM (*original query without having*)
   WHERE *having-clause*

2) Move aggregation expressions from the `HAVING`-clause to the `SELECT`-clause of the original query if they are not already present there, and perform aggregation as usual (e.g. hash-based aggregation);
3) Filter aggregation results by the `HAVING`-predicate that uses columns introduced in the step 2 and then eliminate extra columns by projection.

However, if during the second step the partial sum of `l_quantity` for some group exceeds 1000000, it is possible to discard this group immediately. In the column-store case, this approach will allow us to save some I/O and CPU costs.

At first glance, it may seem that the benefits of our approach are rather limited, because joins incur more significant costs than aggregation. However, there are aggregation queries which do not involve joins, e.g. Q1 and Q6 in TPC-H. More importantly, all queries in this benchmark try to mimic real-life scenarios, and, thus, they represent an actual business need. This indicates that similar queries can be encountered in real-life workloads.

Overall, there are several types of use-cases when our approach would be of use:

1) Aggregation queries that do not involve joins: aggregation running on a denormalized table, a materialized view, solely on a fact table and so on;
2) A case where join operators produce roughly as much data as they receive or even more, so aggregation requires comparable time for processing.

Unlike row-stores, column-stores offer an opportunity to reap the fruits of such optimization with ease. In this paper we consider this problem in the context of PosDB [3]–[5] — a distributed disk-based column-store engine that features late materialization and block-oriented processing. However, we think that the proposed solution is sufficiently universal — almost any column-store system can benefit from similar optimization, regardless of its underlying architecture. Also, despite the current columnar focus of our study, we believe that our approach is applicable to hybrid systems and in-memory DBMSes as well.

Overall, the contribution of this paper is the following:

1) Theoretical basis for on-the-fly pruning of groups during the evaluation of the aggregation operator.
2) An experimental study of our approach using PosDB.

This paper is organized as follows. In Section II we present a short introduction into PosDB and the aggregation operator. Section III provides definitions and presents formal grounds for our analysis. In Section IV we describe an optimized version of the aggregation algorithm. Next, in Section V we present an experimental evaluation and discuss current results. Then, we survey existing aggregation studies in Section VI. Finally, in Section VII we conclude our study and present future work.

II. Query Processing in PosDB

In this section we will give a minimally sufficient description of PosDB internals, and present the naive aggregation algorithm. For a comprehensive description of the whole system, see [3], [5]. Several surveys of column-store technology are presented in references [2], [6]–[8].

A. Basics

Query processing in PosDB is built upon the pull-based Volcano model [9] with block-oriented processing. According to this model, each query is represented by a tree whose nodes are physical operators and edges are data flows. Each operator supports an iterator interface, and data is passed between operators in blocks.

Currently, PosDB supports late materialization only. This is a query execution strategy for column-stores that aims to defer the tuple reconstruction and materialization until as late as possible. In order to implement this strategy, a special intermediate representation, based on the join index from the study [10], was introduced. This representation links records of different tables together. It is used to pass blocks of positional data between operators.

Some operators (e.g. joins) require not only positions, but also attribute values. Therefore, in order to obtain attribute values from the join index we employ a special entity — a reader. There are two types of readers in PosDB that are essential for understanding of our current study:

- `ColumnReader` is a reader for retrieving values of an individual attribute. In fact, there are several subtypes of `ColumnReader`, since our system supports data partitioning and data distribution.
- `SyncReader` is a reader for retrieving values of several attributes related to the same join index in a row-by-row manner. It is implemented as a wrapper for several `ColumnReaders`.

B. Baseline version of aggregation algorithm

Consider the following SQL query:

```
SELECT T.A, T.B, COUNT(*) as count,
    MAX(T.C) - MIN(T.C) as diff
FROM T
GROUP BY T.A, T.B
HAVING count = 1
```

In this query, the three following components should be distinguished:

1) `GROUP BY` clause. It contains a list of attributes that are used to define groups. We will call these attributes *grouping parameters*.
2) `SELECT` clause. It contains a list of expressions that we will call *aggregation expressions*. In our study we assume that only grouping parameters are allowed on the top level. Other attributes should be wrapped in aggregate functions, such as `MAX`, `SUM`, etc. Currently, some database systems allow not only grouping parameters, but arbitrary attributes on the top level as well, e.g., MySQL and SQLite[1]. However, such understanding of aggregation is unambiguous only if there is a functional dependency between grouping parameters and "raw" attributes on the top-level of `SELECT` clause. Otherwise, it leads to uncertainty and, consequently, to implementation-dependent behavior. It should be noted that such understanding of aggregation is not prevalent in the database community since it contradicts the SQL'92 standard (see sections 6.5 and 7.9).
3) `HAVING` clause. It contains a predicate that is applied to the data regarding the whole group and is used to filter resulting rows.

Before describing the aggregation algorithm used in PosDB, we should formally describe the admissible aggregation expressions. They are inductively defined as follows.

Definition 1 A ***stateless expression*** is either:

- an identifier of an attribute belonging to a relation that was mentioned in a `FROM`-clause (*free variable*);
- a constant of a supported data type (*constant*);
- $A + B$, $A - B$, $A \cdot B$, $A \div B$, A^n, where A and B are admissible numeric stateless expressions and n is a non-negative integer (*arithmetic expressions*);

[1] https://stackoverflow.com/questions/1023347/mysql-selecting-a-column-not-in-group-by

Definition 2 An ***aggregation expression*** is either:

- an identifier of any attribute belonging to grouping parameters;
- a constant of a supported data type;
- $\mathrm{COUNT}(E)$, $\mathrm{MAX}(E)$, $\mathrm{MIN}(E)$, $\mathrm{SUM}(E)$, $\mathrm{AVG}(E)$, where E is an admissible stateless expression. In this case, we will call such expressions ***aggregates***.
- $A + B$, $A - B$, $A \cdot B$, $A \div B$, A^n, where A and B are admissible numeric aggregation expressions and n is a non-negative integer. In this case, we will call such expressions ***aggregation arithmetic expressions***.

Other types of expressions can be added later in a similar manner.

Both stateless and aggregation expressions may appear in the `SELECT` clause. Joint usage of these expressions on the top level of the `SELECT` clause is prohibited: stateless expressions should be used if there is no aggregation, and aggregation expressions otherwise. As can be seen from the definition, stateless expressions are also used as subexpressions for aggregates.

Let us now consider the semantics of stateless expressions. Here, we consider an abstract processing scheme rather than focusing purely on row- or column-stores. Consider row-by-row scanning of a relation. A stateless expression is a "pure function": it takes a row as an input and produces a result immediately. On the other hand, aggregation expression can produce a result only when the whole group is processed. During group processing, values of aggregates that compose an aggregation expression should be updated for each incoming row. These updates lead to change of the "current" value of the aggregation expression involving these aggregates. Thus, during processing aggregation expressions pass through a sequence of intermediate states. Monotonicity analysis of such sequences is a central part of the current study.

Concerning implementation, we must emphasize that in case of a disk-based column-store, expressions should use an interface which allows on-demand column reading. This requirement is essential in order to reduce the number of disk accesses. In PosDB, `SyncReader` provides this functionality.

Let us now turn to the aggregation operator itself. Its inputs are grouping parameters and a list of aggregation expressions. The local state [9] of the aggregation operator is a hash table with tuples composed of grouping parameters used as keys and lists of aggregates extracted from aggregation expressions used as values.

In a general case, aggregation is an operation that requires full materialization of the corresponding relation, i.e. it should process all data in order to produce the first result. Thus, it can be decomposed into two stages: aggregate evaluation and result generation. The aggregate evaluation stage is the core of the operator. It is organized as a loop over logical "rows" (provided by the corresponding reader in case of PosDB). On each iteration of the loop a new key-tuple is created and the hash table is probed. If a corresponding record is not found, then the aggregates are cloned (they have a state, therefore, an individual copy should be maintained for each group) and a new entry is added to the hash table. Otherwise, it already exists in the hash table and, thus, the corresponding aggregates should be updated.

In the end of the aggregate evaluation stage, each individual entry of the hash table contains computed aggregates for a particular group. Next, at the result generation stage, the hash table is iterated through. During this process, the algorithm computes aggregation expressions, constructs full tuples and returns them to a parent operator.

If the considered query contains a `HAVING`-clause then a parent operator performing filtering of results is added. In PosDB this operator is implemented as filtering on tuples.

III. Pruning possibility analysis

The core of the proposed approach is `HAVING`-predicate analysis. It consists of two components: monotonicity analysis and codomain analysis. Let us begin with an inductive definition of admissible (correct in the context of a `HAVING`-clause and currently supported) predicates.

Definition 3 The following predicates are admissible:

- $A = B$, $A \neq B$, $A > B$, $A < B$, $A \leq x \leq B$, where A, B and x are admissible aggregation expressions or constants (*atomic predicates*).
- $P \wedge Q$, $P \vee Q$, where P and Q are admissible predicates (*compound predicates*).

Other types of predicates can be added later in the same way. Currently, we support numeric types only.

A. Monotonicity analysis

Consider an aggregation expression from a fixed group. As we mentioned earlier, throughout the execution of the aggregation algorithm the "current" value of this expression changes several times and, thus, forms a sequence. Often we can guarantee monotonicity of these sequences by analyzing the properties of the corresponding aggregates. Therefore, we can talk about monotonicity of aggregation expressions itself.

Aggregates are monotonic in the following way:

- $\mathrm{COUNT}(E)$ is weakly increasing;
- $\mathrm{MAX}(E)$ is weakly increasing;
- $\mathrm{MIN}(E)$ is weakly decreasing;
- $\mathrm{SUM}(E)$ is
  - weakly increasing if $E \geq 0$,
  - weakly decreasing if $E \leq 0$,
  - constant, if $E \equiv 0$,
  - not monotonic otherwise;
- $\mathrm{AVG}(E)$ is not monotonic.

Note that during algorithm execution values of grouping parameters belonging to a particular group do not change. Values of constant expressions behave in a similar manner. Thus, the former should be considered as constants as well.

If we denote weakly increasing aggregation expressions as $E^{\uparrow}$ and weakly decreasing ones as $E^{\downarrow}$, then we can derive monotonicity of aggregation arithmetic expressions according to the following rules:

- $E_1^{\uparrow} + E_2^{\uparrow}$, $E_1^{\uparrow} - E_2^{\downarrow}$ are weakly increasing;
- $E_1^{\downarrow} + E_2^{\downarrow}$, $E_1^{\downarrow} - E_2^{\uparrow}$ are weakly decreasing;
- if $E_1 \geq 0$ and $E_2 \geq 0$:
  - $E_1^{\uparrow} \cdot E_2^{\uparrow}$, $E_1^{\uparrow} \div E_2^{\downarrow}$ are weakly increasing;
  - $E_1^{\downarrow} \cdot E_2^{\downarrow}$, $E_1^{\downarrow} \div E_2^{\uparrow}$ are weakly decreasing;
- if $E_1 \geq 0$ and $E_2 \leq 0$:
  - $E_1^{\uparrow} \div E_2^{\uparrow}$, $E_1^{\uparrow} \cdot E_2^{\downarrow}$ are weakly decreasing;
  - $E_1^{\downarrow} \div E_2^{\downarrow}$, $E_1^{\downarrow} \cdot E_2^{\uparrow}$ are weakly increasing;
- if $E_1 \leq 0$ and $E_2 \geq 0$:
  - $E_1^{\uparrow} \div E_2^{\uparrow}$, $E_1^{\uparrow} \cdot E_2^{\downarrow}$ are weakly increasing;
  - $E_1^{\downarrow} \div E_2^{\downarrow}$, $E_1^{\downarrow} \cdot E_2^{\uparrow}$ are weakly decreasing;
- if $E_1 \leq 0$ and $E_2 \leq 0$:
  - $E_1^{\uparrow} \cdot E_2^{\uparrow}$, $E_1^{\uparrow} \div E_2^{\downarrow}$ are weakly decreasing;
  - $E_1^{\downarrow} \cdot E_2^{\downarrow}$, $E_1^{\downarrow} \div E_2^{\uparrow}$ are weakly increasing;
- for other cases, we cannot guarantee monotonicity of the result.

Proof of these statements obviously follows from the properties of arithmetic operations on inequalities.

Let us now turn to `HAVING`-predicate analysis. We start with atomic predicates and then generalize our approach to the compound ones.

The following predicate types allow to terminate processing of a particular group earlier if the corresponding termination condition is satisfied:

Predicate type	Termination condition
$E_{\downarrow} > E_{\uparrow}$	$E_{\downarrow} \leq E_{\uparrow}$
$E_{\uparrow} < E_{\downarrow}$	$E_{\downarrow} \geq E_{\uparrow}$
$E_{\downarrow} = E_{\uparrow}$	$E_{\downarrow} < E_{\uparrow}$
$E_{\uparrow} = E_{\downarrow}$	$E_{\downarrow} > E_{\uparrow}$

The first column contains the predicate type, which is essentially a predicate with free variables. In contrast, the second column contains an "implementation" of the corresponding predicate where intermediate values of aggregates are substituted instead of being free variables. In our study we refer to both of these entities as predicates. Thus,

Definition 4 ***Potentially terminating predicate*** is a predicate that has a termination condition.

Definition 5 ***Terminating predicate*** is a predicate that has its termination condition fulfilled by a particular instance of data.

The following statements hold for compound predicates:

- $P_1 \wedge P_2$ is potentially terminating, if P_1 or P_2 is potentially terminating.
- $P_1 \wedge P_2$ is terminating, if P_1 or P_2 is terminating.
- $P_1 \vee P_2$ is potentially terminating, if both P_1 and P_2 are potentially terminating.
- $P_1 \vee P_2$ is terminating, if both P_1 and P_2 are terminating.

B. Codomain analysis

As shown earlier, there are several important cases — aggregate $\mathrm{SUM}(E)$, aggregation arithmetic expressions containing multiplication and division — that require additional analysis to assess their monotonicity. This analysis consists of checking whether $E \geq 0$, $E \leq 0$ or $E \equiv 0$. It can be carried out using codomain analysis of corresponding stateless numeric expressions.

Our codomain analysis is based on well-known interval analysis [11]. We support open, closed and half-closed intervals. Their endpoints can be infinite. Constants are represented by degenerate intervals.

The following operators of interval arithmetic are used to compute the codomain of an arithmetic expression:

- $(x_1, x_2) + (y_1, y_2) = (x_1 + y_1, x_2 + y_2)$
- $(x_1, x_2) - (y_1, y_2) = (x_1 - y_2, x_2 - y_1)$
- $(x_1, x_2) \cdot (y_1, y_2) = \big(\min(x_1y_1, x_1y_2, x_2y_1, x_2y_2), \max(x_1y_1, x_1y_2, x_2y_1, x_2y_2)\big)$
- $\frac{1}{(x_1,x_2)} = \begin{cases} (\frac{1}{x_2}, \frac{1}{x_1}), & \text{if } x_1x_2 > 0 \\ (-\infty, \frac{1}{x_1}), & \text{if } x_1 < 0,\ x_2 = 0 \\ (\frac{1}{x_2}, +\infty), & \text{if } x_1 = 0,\ x_2 > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$
- $(x_1, x_2) \div (y_1, y_2) = (x_1, x_2) \cdot \frac{1}{(y_1,y_2)}$
- $(x_1, x_2)^n = \begin{cases} [1, 1], & \text{if } n = 0 \\ (x_2^n, x_1^n), & \text{if } \begin{cases} n \text{ is even,} \\ x_1 \leq 0, x_2 \leq 0 \end{cases} \\ (x_2^n, x_1^n), & \text{if } \begin{cases} n \text{ is even,} \\ x_1 \leq 0, x_2 \leq 0 \end{cases} \\ \big[0, \max(x_1^n, x_2^n)\big), & \text{if } \begin{cases} n \text{ is even,} \\ x_1 \leq 0, x_2 \geq 0 \end{cases} \\ (x_1^n, x_2^n), & \text{if } \begin{cases} n \text{ is even,} \\ x_1 \geq 0, x_2 \geq 0 \end{cases} \\ (x_1^n, x_2^n), & \text{if } n \text{ is odd} \end{cases}$

Concerning endpoint inclusion, it should be noted that inclusion is kept if and only if operation is applied to included endpoints. Otherwise, an endpoint is excluded.

Note that there is an issue with the interval arithmetics. Estimates obtained by its application strongly depend on the form of an expression. For example, consider the expression $x/(1-x)$, for which two different intervals can be obtained — one for $x/(1-x)$ and another for $1/(1/x-1)$. However, it is guaranteed that all intervals would contain the range of the analyzed function.

Currently, it is unclear whether it would be useful to employ more complicated (and more resource-consuming) approaches to get better estimates in case of our algorithm.

Another aspect that should be discussed is the input of operators. Codomain may be described not by a single interval, but by a union of several disjunctive intervals. Thus, operators may also produce unions of several intervals. In our study we restrict ourselves to supporting only a single interval per attribute.

Information about possible values of attributes is taken from the meta-information based on the `CHECK`-constraints stored in the catalog.

IV. Optimized Aggregation Algorithm

We are now ready to present an optimized version of the aggregation algorithm described in the section II-B. Our algorithm combines both aggregation and filtering into a single operator. Therefore, the interface of the aggregation operator was slightly changed — now it also receives a `HAVING`-predicate as a parameter. If there is no `HAVING`-clause in the query, then a non-optimized version should be run.

The optimized algorithm tracks the state of the predicate for each group, so several changes should be introduced into the hash table. Now this table contains entries that are structures with the following fields:

- A list of aggregates that should be evaluated eventually (as in the baseline version). Here we will call such aggregates ***primary aggregates***.
- A copy of the `HAVING`-predicate to check whether it is necessary to further process the current group.
- A list of aggregates from the `HAVING`-predicate that are used to compute the state of the corresponding `HAVING`-predicate copy. We will call them ***auxiliary aggregates***.

The optimized algorithm requires a new stage — a predicate analysis stage. At this stage, the `HAVING`-predicate is analyzed using the approaches described in the previous section. If it is potentially terminating, then it will be possible to perform earlier termination of processing of any group when the corresponding termination condition is satisfied for this group. Otherwise, the baseline version of aggregation algorithm should be used.

Optimization itself is mostly applied at the aggregate evaluation stage. As in the baseline version of algorithm, we iterate through the logical "rows" (provided by `SyncReader` in PosDB). Inside the loop we check if the corresponding group already exists in the hash table. However, now we need to check the state of the predicate copy as well. If there is no such group, we clone the predicate, extract the aggregates (auxiliary) from it, update their values and evaluate the predicate. If the predicate is not terminating yet, then we also clone the primary aggregates and add a record to the hash table. If the predicate is already terminating, then we delete the cloned predicate and add a "tombstone" into the hash table instead of a normal record.

If the corresponding group is already in the hash table, and it was not "tombstoned", then we update the values of both primary and auxiliary aggregates from the predicate, and check the status of the predicate. If it becomes terminating, then we delete the found record and replace it with a "tombstone". Otherwise, we just proceed to the next logical "row" without fetching values needed for computation of both primary and auxiliary aggregates. It is this step of the algorithm that makes I/O and CPU savings possible.

At the result generation stage, we iterate over the hash table, skip groups that have been "tombstoned" and check the `HAVING`-predicate for the remaining groups. Next, tuples constructed from records that satisfy the predicate are passed to the parent operator.

V. Experiments

Experimental evaluation was performed on a PC with the following characteristics: 4-core Intel®Core™ i5-7300HQ CPU @ 2.50GHz, 8 Gb RAM, running Linux Ubuntu 16.04.1 LTS. 128GB KINGSTON RBU-SNS SSD was used as storage.

Test queries are based on Q1 from TPC-H. The first four of them have the following form:

```
SELECT
  l_returnflag, l_linestatus,
  SUM(l_quantity) as sum_qty,
  SUM(l_extendedprice) as sum_base_price,
  SUM(l_extendedprice * (1 - l_discount))
                        as sum_disc_price,
  SUM(l_extendedprice * (1 - l_discount) *
             (1 + l_tax)) as sum_charge,
  AVG(l_quantity) as avg_qty,
  AVG(l_extendedprice) as avg_price,
  AVG(l_discount) as avg_disc,
  COUNT(*) as count_order
FROM lineitem
GROUP BY l_returnflag, l_linestatus
HAVING having-clause
```

where ***having-clause*** is

- `l_linestatus = 'O'` for Q1;
- `having count(*) < 100000` for Q2;
- `(l_returnflag = 'A') OR (l_linestatus = 'O')) AND (MIN(l_tax) > MAX(l_discount)` for Q3;
- `SUM(l_quantity) < 1000000` for Q4.

These queries are designed for studying how optimization is affected by different kinds of predicates. All of them are supposed to demonstrate significant performance improvement due to avoidance of unnecessary I/O in the optimized algorithm.

We have also designed the Q5 query for a first, rough appraisal of the overhead introduced by our algorithm. Q5 is an example of a query where no performance improvement could be gained due to absence of I/O savings. In this query, all columns have to be read regardless of predicate values.

```
SELECT l_returnflag, l_linestatus,
FROM lineitem
GROUP BY l_returnflag, l_linestatus
HAVING l_returnflag = 'A'
```

For each combination of an algorithm and a scale factor (SF) we have run all the aforementioned queries 10 times and calculated the 95% confidence intervals. The full results of the experiments are presented in the Table I. We have also visualized the results for $\mathrm{SF} = 50$ in Fig. 1 to make it more illustrative.

As we can see from Table I, there is no considerable performance dependency on the predicate complexity — queries Q2–Q4 show very close results on all scale factors. On the other side, as expected, optimization efficiency significantly depends

TABLE I
EXPERIMENTS RESULTS (MILLISECONDS)

Query	Algorithm	SF = 1	SF = 5	SF = 10	SF = 25	SF = 50
Q1	Optimized	2087 ms ±2%	10515 ms ±2%	20743 ms ±1%	53145 ms ±1%	104899 ms ±1%
	Baseline	3305 ms ±2%	16589 ms ±3%	32615 ms ±1%	83419 ms ±2%	159588 ms ±2%
Q2	Optimized	841 ms ±1%	3830 ms ±2%	8217 ms ±1%	20150 ms ±1%	45833 ms ±2%
	Baseline	3293 ms ±3%	16564 ms ±3%	32389 ms ±1%	83336 ms ±2%	158751 ms ±2%
Q3	Optimized	687 ms ±1%	3634 ms ±1%	8163 ms ±1%	20086 ms ±1%	45734 ms ±1%
	Baseline	3458 ms ±2%	17276 ms ±3%	33843 ms ±1%	87288 ms ±2%	164593 ms ±2%
Q4	Optimized	766 ms ±1%	3717 ms ±1%	8217 ms ±1%	20078 ms ±1%	45754 ms ±1%
	Baseline	3511 ms ±2%	17651 ms ±3%	34626 ms ±1%	88972 ms ±2%	168037 ms ±2%
Q5	Optimized	510 ms ±1%	2500 ms ±1%	4980 ms ±1%	12693 ms ±1%	25994 ms ±1%
	Baseline	504 ms ±1%	2439 ms ±1%	4873 ms ±1%	12465 ms ±1%	25666 ms ±1%

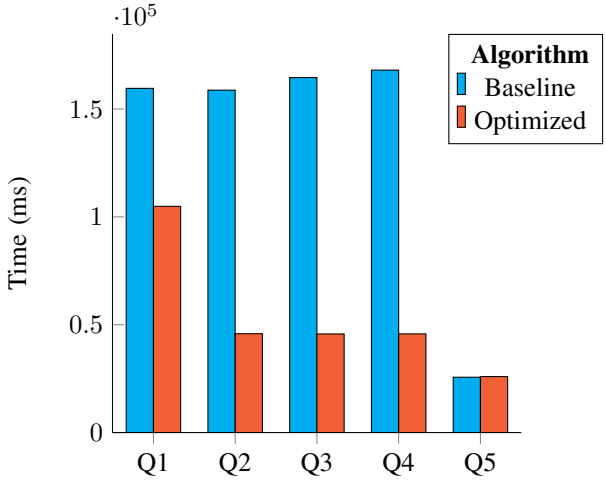

Fig. 1. Comparison of algorithm performance for SF=50

on the predicate selectivity and on the time when pruning can be performed. We suppose that the difference between Q1 and Q2–Q4 should be explained by this fact. Detailed analysis of these factors was postponed for the future.

Evaluation of Q5 shows that the overhead introduced by our algorithm is rather small (about 2–3%) and does not depend on the scale factor.

Concerning the dependency on the scale factor, it is looking close to linear for all the considered queries.

VI. RELATED WORK

According to reference [9], the processing of aggregation queries has been studied at least since the end of the 70's [12]. Nowadays, it is a mature area of research which features hundreds of papers [13].

These studies can be roughly classified into the following groups:

1) Optimization of aggregation queries. In these papers authors study how to efficiently process aggregation queries mostly by using various plan transformations. In the study [14], [15], the authors propose two transformations: eager aggregation (moving a `GROUP BY` down through join) and lazy aggregation (moving a `GROUP BY` up). The former allows to reduce the number of entries that need to be processed by the join operator, and, consequently, to improve the overall query processing performance. The latter is of interest when the query operates on a view containing a `GROUP BY`. A similar transformation is proposed in study [15]. The core of this approach is to pre-aggregate data using any column set that functionally determines the table being aggregated. In study [16] authors integrate subquery and aggregation processing techniques by proposing a set of shared primitives. Then these primitives are used to generate optimized plans. The problem of aggregation query optimization in the OLAP environment was considered in reference [17]. In this paper, an analysis of an approach called Hierarchical Pre-Grouping is performed and a number of transformations is proposed and analyzed.
2) Optimization of evaluation of an individual aggregation operator. These studies aim to organize processing in the most efficient way. There are two basic methods for performing an aggregation [9] — hashing and sorting. According to reference [9] sorting was considered in study [12]. Among contemporary studies it is essential to note Blink [18], where authors consider aggregating compressed data and reference [19] where hardware-efficient multi-threaded aggregation in column-store was presented. However, none of these studies analyzed aggregation predicates in order to improve individual operator performance.
3) A relatively recent approach called online aggregation. Its goal is to provide the database user with means to control the execution of aggregation. The proposed use-case is the following: while processing data, progressively refine an approximate answer and provide its running confidence intervals. Unlike plain aggregation, where the user passively waits for the answer, this approach allows the user to terminate a query early if they deem the approximate results acceptable. Originally, this approach was proposed in reference [20], and later, many studies have followed. For example, the study [21] extends this idea onto nested queries that contain aggregation and also proposes a multi-threaded model. Next, the paper [22] addresses parallel

aggregation on a cluster of computers.

4) Another group of papers studied approximate processing of aggregation queries. Unlike the previous approach, this type of studies does not imply user intervention during the processing. Approximate aggregation query answering using sampling was studied in reference [23]. The idea of this paper is to index outlying values in order to reduce the approximation error. The study [24] addresses the problem of approximate time-constrained query evaluation. The authors propose an algorithm for evaluating count-containing aggregation queries that can be stopped if the desired error range is obtained or the pre-specified time limit is exceeded.
5) Novel models of aggregation and novel aggregations operators. Horizontal aggregation is proposed in reference [25]. Its idea is to generate a new table, where a separate column for each unique value of columns belonging to aggregation expression is generated. At the same time, the rows of this table contain all unique values from the columns of `GROUP BY` list. The authors of this paper propose not only semantics of this operation, but also language extension and discuss query execution. Similarly, reference [26] describes two aggregation-like operators: PIVOT and UNPIVOT. The former transforms a series of rows into a series of fewer rows with additional columns. Data in one source column is used to generate the new column for a row, and another source column is used as the data for that new column. The latter performs the inverse operation by removing a number of columns and creating additional rows. In studies [27], [28] novel aggregation operators are proposed. The former study considers embedding of grouping variables [29] into SQL queries. Grouping variables is a tool that allows to specify additional conditions for the desired groups. It is much more expressive than `HAVING`-clause. The latter study proposes a generalization of an aggregation operator that allows formation of aggregation groups without requiring an ordering of the data relation.

VII. Conclusion

In this paper we have proposed a novel approach to evaluation of aggregation in column-stores. We employ monotonicity and codomain analysis in order to invoke early termination that allows to save CPU and I/O costs. To validate our idea, we have implemented the designed algorithms inside a disk-based column-store query engine. Preliminary experiments show that our approach can improve query performance up to 5 times over the naive algorithm.

In our future studies we plan to evaluate performance dependency on the number of groups, data distribution, selectivity and complexity of the predicate. We also going to assess the overhead introduced by our algorithm in more detail. Other future studies may include combining proposed operator with other approaches to aggregation (e.g. partial aggregation approach), applying it for in-memory systems, and exploring opportunities offered by novel hardware.

References

[1] Transaction Processing Performance Council, "TPC Benchmark™ H Standard Specification Revision 2.17.3." [Online]. Available: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf

[2] Stavros Harizopoulos, Daniel Abadi, and Peter Boncz, "Column-Oriented Database Systems," 2009. [Online]. Available: http://nms.csail.mit.edu/~stavros/pubs/tutorial2009-column_stores.pdf

[3] G. Chernishev, V. Galaktionov, V. Grigorev, E. Klyuchikov, and K. Smirnov, "PosDB: A Distributed Column-Store Engine," in *Perspectives of System Informatics*, A. K. Petrenko and A. Voronkov, Eds. Cham: Springer International Publishing, 2018, pp. 88–94.

[4] ——, "A study of PosDB Performance in a Distributed Environment," in *Proceedings of the 2017 Software Engineering and Information Management*, ser. SEIM '17, 2017.

[5] ——, "PosDB: An Architecture Overview," *Programming and Computer Software*, vol. 44, no. 1, pp. 62–74, Jan 2018. [Online]. Available: https://doi.org/10.1134/S0361768818010024

[6] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos, *The Design and Implementation of Modern Column-Oriented Database Systems*. Hanover, MA, USA: Now Publishers Inc., 2013. [Online]. Available: http://db.csail.mit.edu/pubs/abadi-column-stores.pdf

[7] G. Chernishev, "The design of an adaptive column-store system," *Journal of Big Data*, vol. 4, no. 1, p. 5, Mar 2017. [Online]. Available: https://doi.org/10.1186/s40537-017-0069-4

[8] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "MonetDB: Two Decades of Research in Column-oriented Database Architectures," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 40–45, 2012. [Online]. Available: http://sites.computer.org/debull/A12mar/monetdb.pdf

[9] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 73–169, Jun. 1993. [Online]. Available: http://doi.acm.org/10.1145/152610.152611

[10] Z. Li and K. A. Ross, "Fast Joins Using Join Indices," *The VLDB Journal*, vol. 8, no. 1, pp. 1–24, Apr. 1999. [Online]. Available: http://dx.doi.org/10.1007/s007780050071

[11] G. Alefeld and G. Mayer, "Interval analysis: theory and applications," *Journal of Computational and Applied Mathematics*, vol. 121, no. 1-2, pp. 421–464, Aug. 2000.

[12] R. Epstein, "Techniques for Processing of Aggregates in Relational Database Systems," Univ. of California, Berkeley, Tech. Rep., 01 1979.

[13] I. F. V. Lopez, R. T. Snodgrass, and B. Moon, "Spatiotemporal aggregate computation: a survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 2, pp. 271–286, Feb 2005.

[14] W. P. Yan and P.-A. Larson, "Eager Aggregation and Lazy Aggregation," in *Proceedings of the 21th International Conference on Very Large Data Bases*, ser. VLDB '95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 345–357. [Online]. Available: http://dl.acm.org/citation.cfm?id=645921.673154

[15] P. A. Larson, "Data reduction by partial preaggregation," in *Proceedings 18th International Conference on Data Engineering*, 2002, pp. 706–715.

[16] C. Galindo-Legaria and M. Joshi, "Orthogonal Optimization of Subqueries and Aggregation," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '01. New York, NY, USA: ACM, 2001, pp. 571–581. [Online]. Available: http://doi.acm.org/10.1145/375663.375748

[17] A. Tsois and T. Sellis, "The Generalized Pre-grouping Transformation: Aggregate-query Optimization in the Presence of Dependencies," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. VLDB Endowment, 2003, pp. 644–655. [Online]. Available: http://dl.acm.org/citation.cfm?id=1315451.1315507

[18] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle, "Constant-Time Query Processing," in *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ser. ICDE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 60–69. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2008.4497414

[19] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang, "DB2 with BLU Acceleration: So Much More Than Just a Column Store," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1080–1091, Aug. 2013. [Online]. Available: http://dx.doi.org/10.14778/2536222.2536233

[20] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online Aggregation," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '97. New York, NY, USA: ACM, 1997, pp. 171–182. [Online]. Available: http://doi.acm.org/10.1145/253260.253291

[21] K.-L. Tan, C. H. Goh, and B. C. Ooi, "Progressive Evaluation of Nested Aggregate Queries," *The VLDB Journal*, vol. 9, no. 3, pp. 261–278, Dec. 2000. [Online]. Available: http://dx.doi.org/10.1007/s007780000026

[22] C. Qin and F. Rusu, "Parallel Online Aggregation in Action," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, ser. SSDBM. New York, NY, USA: ACM, 2013, pp. 46:1–46:4. [Online]. Available: http://doi.acm.org/10.1145/2484838.2484874

[23] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya, "Overcoming limitations of sampling for aggregation queries," in *Proceedings 17th International Conference on Data Engineering*, 2001, pp. 534–542.

[24] W.-C. Hou, G. Ozsoyoglu, and B. K. Taneja, "Processing Aggregate Relational Queries with Hard Time Constraints," in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '89. New York, NY, USA: ACM, 1989, pp. 68–77. [Online]. Available: http://doi.acm.org/10.1145/67544.66933

[25] C. Ordonez, J. García-García, and Z. Chen, "Dynamic Optimization of Generalized SQL Queries with Horizontal Aggregations," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 637–640. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213919

[26] C. Cunningham, C. A. Galindo-Legaria, and G. Graefe, "PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, pp. 998–1009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1316689.1316775

[27] D. Chatziantoniou, "Using Grouping Variables to Express Complex Decision Support Queries," *Data Knowl. Eng.*, vol. 61, no. 1, pp. 114–136, Apr. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.datak.2006.05.001

[28] M. Akinde, M. H. Bhlen, D. Chatziantoniou, and J. Gamper, "θ–constrained multi-dimensional aggregation," *Information Systems*, vol. 36, no. 2, pp. 341 – 358, 2011, special Issue: Semantic Integration of Data, Multimedia, and Services. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306437910000748

[29] D. Chatziantoniou and K. A. Ross, "Querying Multiple Features of Groups in Relational Databases," in *Proceedings of the 22th International Conference on Very Large Data Bases*, ser. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 295–306. [Online]. Available: http://dl.acm.org/citation.cfm?id=645922.673628