Multi-view optimization in key-value stores

Jan Adler Kaiwen Zhang H.-A. Jacobsen

ABSTRACT

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

1. INTRODUCTION

Big-data infrastructures are growing rapidly in storage capacity and distribution.Modern databases, called key-value stores (Google's BigTable [1], Amazon's Dynamo [2], Yahoo's PNUTS [3], Apache's HBase [4] and Cassandra [5]) are capable of managing the enormous amounts of data. Thus, the analytical tasks come into foreground. Today's analytical frameworks have to keep pace with ever-growing amounts of stored and distributed data. Not only have they to execute fast, consistent and fault-tolerant, efficiency and cost are measures that gain importance, likewise. An analytical framework may be fast when executed in memory, but repeated scan over large chunks of data may trigger unnecessary read and waste processing time, thus, increase the cost of running the system.

In order to save storage and processing capacity, data bases have always used optimization algorithms. Multi-query optimization is used to improve the evaluation of analytical statements beforehand. In order to re-use computations results – instead of re-reading and re-evaluating a data set–, multi-query optimizations use techniques of data materialization. A common problem of multi-query optimization is the view selection problem. View selection, as is also described in [6, 7] is a problem, where a set of view candidates is defined, and the algorithm chooses which sub-set of the candidates should be materialized. It is a NP-hard problem that grows exponentially with the number of views (i.e. 2^n); mostly it is solved by using heuristics.

While multi-query optimization and view selection have been discussed in the literature, widely [8–12], multi-view optimization is a relatively under-represented problem. In multi-view optimization every view has its materialization. Even if there are methods to reduce the number of intermediate view tables (e.g. by merging them), a minimum of views has to be materialized. Thus, we can indeed apply some of the techniques known from multi-query optimization and view selection, the general approach (and optimization algorithm), however, is different.

Adler, Jergler and Jacobsen [13] showed that a distributed view maintenance framework (VMS) can be used to efficiently create and maintain sets of basic views expressions. The VMS supports real-time view updates and scales in and out with number of view maintained. To save processing power and avoid needless re-computations, the VMS takes an incremental maintenance approach. Results sets are materialized and updates on a base table propagate to the corresponding view table.

Knowing that these basic view types can form higher level analytical (i.e.SQL-like) statements, we propose, based on the VMS, an approach for multi-view optimization. We translate SQL-queries into maintenance plans, where each plan comprises a set of materialized views. Then, we show how resources can be shared locally and globally. We propose an algorithm for multi-view optimization that captures the logic of finding the best global plan for incremental maintenance of views.

Roadmap: The paper is structured as follows: first, we briefly discuss the basic architecture; i.e. the view maintenance system that is build on top of the key-value store. Then, we introduce the basic view expressions and show how a maintenance plans serves to execute a chain of these basic expressions. Further, we discuss the translation of a general SQL-statement into a proper maintenance plan. In the core part of the paper, we discuss optimization techniques and introduce our multi-view algorithm; Finally, we evaluate the algorithm in comparison to the non-optimized version.

2. BACKGROUND

In this section, we explain the infrastructure, necessary to manage base and view tables. We start with a general description of the KV-store and its data model. Further, we explain the View Maintenance System that creates and updates the materialized views (stored in KV-store).

2.1 KV store

Some KV-stores are based on a master-slave architecture, i.e. HBase; other KV-stores run without a master, i.e. Cassandra (a leader is elected to perform tasks controlling the KV-store). In both cases a *node* represents the unit of scalability – as arbitrary instances can be spawned in the network (see Figure **??**). The node persists the actual data. But in contrast to a traditional SQL-database, a node manages only part of the overall data (and request load). As load grows in the KV-store, more nodes can be added to the system; likewise, nodes can be removed as load declines. The KV-store will automatically adapt to the new situation and integrate, respectively drop the resource. KV-stores differ in how they accommodate; they also differ in how they perform load balancing and recovery (in face of node crashes). However, with regard to nodes, we can describe a set of universal events that occur in every KV-store (cf. Table 1).

A *table* in a KV-store does not follow a fixed schema. It stores a set of table records called *rows*. A row is uniquely identified by a *row key*. A row holds a variable number of *columns* (i.e., a set of column-value pairs). Columns can be further grouped into *column families*. Column families provide fast sequential access to a subset of columns. They are determined when a table is created and affect the way the KV-store organizes its table files.
Key ranges serve to partition a table into multiple parts that can be distribute over multiple nodes. Key ranges are defined as an interval with a start and an end row key. PNUTS refers to this partitioning mechanisms as tablets, while HBase refers to key ranges as regions. Multiple regions can be assigned to a node, often referred to as a region server. In general, a KV-store can split and move key ranges between nodes to balance system load or to achieve a uniform distribution of data. With regard to key ranges, we can also describe a set of universal events (cf. Table 1).

The data model of a KV-store differs from that of a relational DBMS. We describe a model that is representative for today's KV-stores. The model serves throughout the paper to help specify views and view update programs. Typically, KV-stores do not required fixed data schemas, but rather accommodate dynamic schema changes.

Thus, we formalize the data model of a KV-store as a map of key-value pairs $\{\langle k_1, v_1\rangle, .., \langle k_n, v_n\rangle\}$ described by a function $f : K \rightarrow V$. Systems like BigTable, HBase and Cassandra established data models that are multi-dimensional maps storing a row together with a variable number of columns per row. For example, the 2-dimensional case creates a structure $\{\langle(k_1, c_1), v_{1,1}\rangle, \langle(k_n, c_n), v_{n,n}\rangle\}$ with a composite key (k_n, c_n) that maps to a value $v_{n,n}$ described by $f : (K, C) \rightarrow V$. In the 3-dimensional case, another parameter, a timestamp, for example, is added to the key, which may serve versioning purposes. For the intentions in this paper, the 2-dimensional model suffices.[1]

We denote a table by $A = (K, F)$, where K represents the row key and F a *column family*. Column families are defined when a table is created. They are used in practice to group and access sets of column-value pairs. In terms of our data model, column families are optional. They can be dynamically assigned as the row is created. Let a base table row $a \in A$ be defined as $a = (k, \{\langle c_1, v_1\rangle..\langle c_n, v_n\rangle\})$. In this notation, the row key k comes first, followed by a set of column-value pairs $\{\langle c_1, v_1\rangle..\langle c_n, v_n\rangle\}$ belonging to the column family; this more closely resembles a database row and is used throughout the remainder of this paper. When using multiple column families, we define the table as $A = (K, F_1, ...F_n)$. Then the assignment of a column-value set to a column family F_x is denoted by $\{..\}_x$. The corresponding row would be defined as $a = (k, \{\langle c_1, v_1\rangle..\langle c_i, v_i\rangle\}_1..., \{\langle c_{i+1}, v_{i+1}\rangle..\langle c_n, v_n\rangle\}_n)$.

[1]Our approach also works with the 1-dimensional case, which is representative for simple key-blob stores. We use the 2-dimensional case here, as it is more expressive.

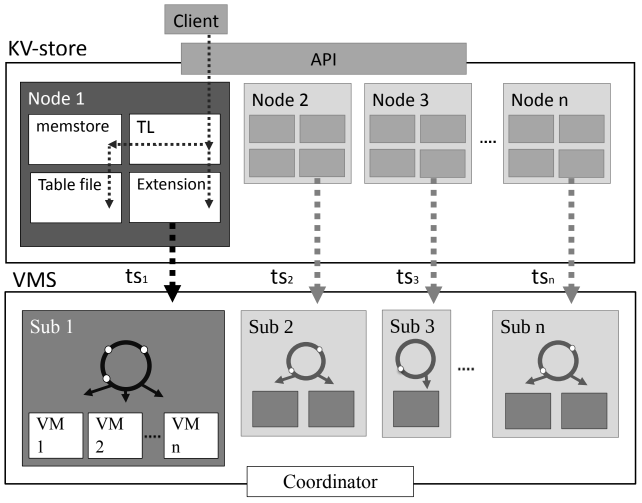

Figure 1: System Overview

The KV-store writes operations, that is client requests, to the TL, but not entire table rows. In contrast, a table row stores the row state, which may result from multiple client requests. Then, an operation $t \in T$ can easily be defined over table row $a \in A$, with $T = type(A)$ and $type \in \{put, delete\}$. A put operation in the transaction log is denoted as $t = put(k, \{\langle c_1, v_1\rangle..\langle c_n, v_n\rangle\})$. A put inserts or updates the column values at the specified row key. A delete operation $t \in T$ is defined as $t = delete(k, \{\langle c_1, \emptyset\rangle..\langle c_n, \emptyset\rangle\})$. Note that we are leaving the values empty; the client just specifies the row key and columns that are to be deleted. A stream – respectively, the output of one node's transaction log – is denoted as a sequence of operations $ts \in TS = (T_1, .., T_n)$. Finally, we can define the complete output of the KV-store as a set of operation streams as $ts_1, ..ts_n \in TS$.

2.2 VMS

The View Maintenance System receives updates from the KV-store in form of operation streams (see Figure 1). Every node of the KV-store produces a local transaction stream $(ts_1, .., ts_n)$;every stream of operations is handled by a subsystem of the VMS. The subsystem parallizes view update computation: it distributes the updates to a scalable number of view managers. A view manager actually applies the update operation to the view table. First, it looks up the view tables defined over the base, then it retrieves the corresponding view record, adds the delta to it, and writes back the result to the view.

We express view tables in the VMS with the help of relational algebra (e.g., we define a `SELECTION` as $S = \sigma_{c_1<x}(A)$ over a base table A). Defining a view table over a base table is equivalent to connection the output stream of base

table operations with the input stream of view table input operations. The VMS is also capable of defining view tables over view tables (e.g. we define a PROJECTION view as $P = \pi_{c_1,c_2}(S)$). Thus, we can concatenate multiple different view types; the VMS will update the view chain subsequently. It will receive the base table update, apply the update to the SELECTION view, and apply the result of the first update to the PROJECTION view.

3. VIEW EXPRESSIONS

In this section, we develop techniques for maintaining the following basic view expressions in our VMS: SELECTION, PROJECTION, INDEX, aggregation (i.e. COUNT, SUM, MIN, MAX, AVG) and join (i.e. INNER, LEFT, RIGHT, FULL). Internally, VMS provides a number of auxiliary views, which are the DELTA, PRE-AGGREGATION and REVERSE-JOIN view, described shortly. Figure **??** gives an overview of all view types and their dependencies.

3.1 Auxiliary views

Auxiliary views are internal to VMS and are not exposed to clients. They are maintained to enable, facilitate and speed up the correct maintenance of other view types. Some views could not be maintained consistently without the additional information provided by these auxiliaries, others simply benefit from their pre-computations. While auxiliaries introduce storage overhead, they support modularity in view maintenance. Logically, auxiliaries represent the basic elements of view maintenance, which can be shared within and between view definitions. Thus, their use amortizes as more complexer views are managed by VMS. Auxiliaries also speed up view maintenance significantly. The update programs that compute the different view types make up most of the view manger logic.

In what follows, we describe each auxiliary view type, including the problem it solves and how it is maintained given base data updates. In Table **??**, the definition, record format and update operation of the auxiliary views are depicted. In the first row the base table is shown. The records of the base table correspond to the general format r. A client put (insert or update) creates an update operation t_p, whereas a client delete creates a t_d operation. As discussed before, the KV-store updates the base table by itself and provides operations t_p and t_d through the TL.

Delta – The DELTA view is an auxiliary view that tracks base table changes between successive update operations. TL entries only contain the client operation. They do not characterize the base record state before or after the operation. For example, for a delete operation, the client only provides the row key, but not the associated value to be deleted, as input. Likewise, an update operation provides the row key and new values, but not the old values to be modified. In fact, a TL entry does not distinguish between an insert and update operation. However, for view maintenance, this information is vital. This motivated us to introduce the DELTA view. It records base table entry changes, tracking the states between entry updates, i.e., the "delta" between two successive operations for a given row. Views that derive, have this information available for their maintenance operations. Now, we show the computation steps of the DELTA view when updated (see Table **??**).

Computation: The DELTA view is defined over the base table as $D_A = \delta(A)$, meaning all operations on the base table are forwarded to the view. The format of the records in D_A corresponds to r' – the state of record r before update operation t_d or t_p is applied (e.g. r' holds the values that t_d deletes; or r' maybe $\emptyset$ if t_p is an insert operation). Once r' is retrieved from the DELTA view, r becomes the new r'. The VMS updates the view accordingly. If the client operation was a delete (t_d), the VMS also deletes the record in the view.

Pre-Aggregation – The PRE-AGGREGATION view is an auxiliary view that prepares the aggregation by already sorting and grouping the base table rows. Consecutive aggregation views only need to apply their aggregation function, then. Often, an application calculates different aggregations over the same aggregation key. To materialize these views, VMS must fetch the same record over and over. For MIN and MAX views, the deletion of the minimum (maximum) in the view requires an expensive base table scan to determine the new minimum (maximum). This motivated us to introduce the PRE-AGGREGATION view. This view type sorts the base table records according to the aggregation key, but stores the grouped rows in a map.

*Computation:*The PRE-AGGREGATION view P is defined over the DELTA view D (see Table **??**). The grouping of the view is based on a column name c_a from the base records; the table of the view is sorted according to the grouping key v_α. Every record in the view stores a map of key-value pairs (row key k and grouping value v_β) – representing the expanded list of grouping values that belong to a grouping key. The VMS can collapse this map in a second step to evaluate the count, sum, minimum, maximum or average of the grouped values. A pre-aggregation can only be defined over a DELTA view; an update may change the grouping key and, thus, requires touching the old and the new record in the view.

Reverse Join – A JOIN view is derived from at least two base tables. For an update to one of these tables, the VMS needs to query the other base table to determine the matching join rows. Only if the join-attribute is the row key of the queried base table, can the matching row be determined quickly, unless of course, an index is defined on the join-attribute for the table. Otherwise, a scan of the entire table is required, which has the following drawbacks: (i) Scans require a disproportional amount of time, slowing down view maintenance. (With increasing table size, the problem worsens.) (ii) Scans keep the nodes occupied, slowing down client requests. (iii) While a scan is in progress, underlying base tables may change, thus, destroying view data consistency for derived views. To address these issues, we introduce the REVERSE-JOIN view. It is an auxiliary view that supports the efficient and correct materialization of equi joins in VMS. [2]

Computation (2-tables): A REVERSE-JOIN view supports the maintenance of a join between two tables: $A \bowtie B_{A.c_\alpha = B.c_\alpha}$, with join key in column c_α of tables A and B. We define the REVERSE -JOIN view analogous to the PRE-AGGREGATION view (see Table **??**). This opens up potential for savings in storage and computation; we can just use the same view for both view types (if grouping and join key are the same).

To build the view, we use an aggregation function that collects all records of a specific join key; the join key, defined as c_a – full-filling the same purpose as the grouping key before – becomes the row key of the view. However, in contrast

[2]We are not discussing theta joins with a full join predicate, for it is outside the context of this work.

	view types	get record	on Insert	on Delete
1	A	$r = (k, \{\langle c_1, v_1\rangle, .., \langle c_n, v_n\rangle\})$	$t_p = put(r)$	$t_d = delete(k)$
2	$D = \delta(A)$	$r' = (k, \{\langle c_1, v'_1\rangle, .., \langle c_n, v'_n\rangle\})$	$put(r)$	$delete(k)$
3	$P = \gamma_{c_\alpha, \langle K, c_\beta\rangle}(D)$	$p = (v_\alpha, \{\langle k_1, v_1\rangle, .., \langle k_n, v_n\rangle\})$	$put(v_\alpha, \{\langle k, v_\beta\rangle\})$	$delete(v_\alpha, \{k\})$
4	$R = \gamma_{c_\alpha, \langle K, r\rangle, \langle L, s\rangle}$	$p = (v_\alpha, \{\langle k_1, r_1\rangle, .., \langle k_n, r_n\rangle\}_A$	$A : put(v_\alpha\{\langle k, v_1\rangle\}_A)$	$A : delete(v_\alpha, \{k\}_A)$
	(D_A, D_B)	$\{\langle l_1, s_1\rangle, .., \langle l_m, s_m\rangle\}_B)$	$B : put(v_\alpha, \{\langle k, v_1\rangle\}_B)$	$B : delete(v_\alpha, \{k\}_B)$
5	$\sigma_C(D)$	$r \vee \emptyset$	$(C) \rightarrow put(r)$	$(C) \rightarrow delete(k, \{c_1, .., c_n\})$
6	$\pi_{c_{\alpha_1}, ..c_{\alpha_n}}(D)$	$(k, \{\langle c, v\rangle \mid c \in c_{\alpha_1}, ..c_{\alpha_n}\})$	$put(k, \{\langle c, v\rangle \mid c \in c_{\alpha_1}, ..c_{\alpha_n}\})$	$delete(k, \{c_{\alpha_1}, .., c_{\alpha_n}\})$
7	$\gamma_{c_\alpha, Count(c_\beta)}(D)$	$(v_\alpha, \{\langle c, v_{count}\rangle\})$	$put(v_\alpha, \{\langle c, v_{count} + 1\rangle\})$	$put(v_\alpha, \{\langle c, v_{count} - 1\rangle\})$
8	$\gamma_{c_\alpha, Sum(c_\beta)}(D)$	$(v_\alpha, \{\langle c, v_{sum}\rangle\})$	$put(v_\alpha, \{\langle c, v_{sum} + v_\beta\rangle\})$	$put(v_\alpha, \{\langle c, v_{sum} - v_\beta\rangle\})$
9	$\gamma_{c_\alpha, Min(c_\beta)}(D)$	$(v_\alpha, \{\langle c, v_{min}\rangle\})$	$(v_\beta < v_{min}) \rightarrow put(v_\alpha, \{\langle c, v_\beta\rangle\})$	$(v_\beta = v_{min}) \rightarrow searchMin()$
10	$\gamma_{c_\alpha}(D)$	$v_\alpha, \{\langle k_\alpha, v_\alpha\rangle, ..\}_A$	$put(\{\langle k, v_1\rangle\}_A)$	$delete(x, \{k\}_A)$
11	$A \bowtie B(R)$	$((k_\alpha), \{\langle k_\alpha, v_\alpha\rangle, ..\}_A$	$put(\{\langle k, v_1\rangle\}_A)$	$delete(x, \{k\}_A)$

Table 1: Computation table

to the PRE-AGGREGATION view, the REVERSE-JOIN view consumes tuples from two different input tables, namely D_A and D_B. When updates are propagated, the REVERSE-JOIN view can be accessed rapidly from either side of the relation (the join key is always included in both tables' updates). If a record is inserted into one of the underlying base tables, it is stored into the view — whether it has a matching row in the join table or not. We are using column families $\{..\}A$ and $\{..\}B$ in the view to distinguish updates from different base tables. Later, this facilitates the computation of the matching join rows, for we only need to build the cross product between the records of the families. Thus, INNER, LEFT, RIGHT, and FULL JOIN can derive from the REVERSE-JOIN view without the need for base table scans, as we show below. We discussed the REVERSE JOIN with two join tables and a single join key. When increasing the number of join tables to n, we distinguish two cases:

(i) Join all tables on the same join key: we just need to extend the prior definition to n tables. For every new join table Φ with DELTA view D_Φ, a new column family is added to the view. This column family collects the updates of the base table. As long as we are joining on the same key, we can put all relations to the same REVERSE-JOIN view.

(ii) base tables are joined on different join keys: for example, $A \bowtie B \bowtie C$, where table A and B are joined on column c_1, whereas table B and C are joined on a column c_3. To enable this, we have to combine multiple REVERSE-JOIN views. We are free to choose, whether we build a REVERSE-JOIN for $A \bowtie B$ and combine the result with C or we build a REVERSE-JOIN for $B \bowtie C$ and combine it with A. For every pair of distinct join keys, we need a REVERSE-JOIN view. In the worst case, we have n join tables and $n-1$ distinct join keys, resulting in the same number of REVERSE-JOIN views. However, this compositional manner of deriving join views, leads to a number of possible optimizations. When computing join $A \bowtie B \bowtie C$ and $B \bowtie C \bowtie D$, relation $B \bowtie C$ only needs to be computed once, for instance.

3.2 Standard views

In this section, we describe how VMS maintains client-exposed views for a number of interesting standard view types. We also present alternative maintenance strategies, but defer a full-fledged cost analysis to future work.

Selection and Projection – A SELECTION view selects a set of records from a base table based on a *selection condition.* The row key of the base table serves as the row key of the view table and a single base record uniquely maps to a single view table record. A record delete is performed for every operation on the view. This is because, the selection condition cannot be evaluated on the operation, as it may not contain the value. For t_d, the VM does not know the deleted value and cannot determine, if there is a corresponding view record. For t_p, the VM is not able to distinguish between an insert vs. an update. Thus, in both cases, the VM pre-emptively deletes the record in the view.

A PROJECTION view selects a set of columns from a base table. Similar to the SELECTION view, the VM uses the row key of the base table as row key for the view table. The computation of the PROJECTION view changes analogously when derived from a DELTA view. To save storage and computation resources, we could combine DELTA, PROJECTION and SELECTION into one view. This would reduce the amount of records (due to selection), the amount of columns (due to projection), and still provide delta information to subsequent views. These considerations are important for multi-view optimization in VMS, which we defer to future work.

Count and Sum – The maintenance of COUNT and SUM views is very similar, so we treat them together. Generally speaking, in aggregation views, records identified by an *aggregation key* aggregate into a single view table record. The aggregation key becomes the row key of the view. Let a base table A and D be defined as before. Then, a SUM view is defined as $S = \gamma_{c_1, Sum(c_2)}(D)$. Note, the view is defined over a delta table and not a base table. The row key of table S is the aggregation key of the view (i.e. the value of c_1). Operations d_i, d_u and d_d are processed as follows: The view manager queries S to retrieve the last state of the aggregated value, e.g., $S(v'_1, \{\langle c_s, v'_s\rangle\})$. Then, the VM computes the new state of the aggregated value by adding the "delta", i.e., $v_s = v'_s + (v_2 - v'_2)$. In case the update operation changed the aggregation key (i.e., $v'_1 \neq v_1$), the update involves two records in the view table. Thus, the VM executes a delete on the old key v'_1 and an insert on the new key v_1. A COUNT view is a special case of a SUM view.

Updates for it results by setting $v_2 = 1$ and $v'_2 = 1$ in the following updates for `COUNT`:

Index – `INDEX` views are important to provide fast access to arbitrary columns of base tables. The view table uses the chosen column as row key of the view, storing the corresponding table row keys in the record value. If the client wishes to retrieve a base table row by the indexed column, it accesses the `INDEX` view first, then, it accesses the base table with the row keys retrieved from the record found in the view. This is a fast type of access, for the client is always accessing single rows by row key. The `INDEX` view is a special case of the `PRE-AGGREGATION` view. If the indexed key is the same as the aggregation key, we can combine both view types. Further, the `PRE-AGGREGATION` can be combined with a `REVERSE-JOIN` view (see description above). Thus, we receive three different view types at the cost of one.

Min and Max Views – `MIN` and `MAX` views are also aggregates. Both can be derived from a `DELTA` or a `PRE-AGGREGATION` view. When derived from a `DELTA` view, `MIN` and `MAX` are computed similar to a `SUM`. However, a special case is the deletion of a minimum (maximum) in a `MIN` (`MAX`). In that case, the new minimum (maximum) has to be determined. Without the assistance of auxiliary views, a table scan would have to be performed [?]. This motivated us to derive the `MIN` (`MAX`) from a `PRE-AGGREGATION`, which prevents the need for a scan.

3.3 Joins

Join – The `JOIN` view presents the results of n joined base tables. Since the matching of join partners has been already completed in the `REVERSE JOIN` view, the `JOIN` view is used to display results in a proper way. To get the join result, the VM takes the output operations of the `REVERSE JOIN` view and multiplies their column families. Thereby, the `INNER`, `LEFT`, `RIGHT`, `FULL` join can be constructed.

4. MAINTENANCE PLAN

In the previous section, we explained a number of basic view expressions (selection, projection, join, etc.) that our system can materialize and update incrementally, as base data changes. However, queries for data analysis are usually a lot more complex. They consist of multiple, nested view expressions; these can be applied in any order to build higher level constructs. The following query q combines a selection and a sum aggregation on a table A:

$$\text{Select Sum}(c_2) \text{ From } A \text{ Group By } c_1 \text{ Where } c_2 < 10$$

In order to build a *maintenance plan*, we decompose the given query into a number of basics view expressions. These basic view expressions are connected with each other. They form a chain that computes and maintains the result of the query. In the chain, each view expression computes an intermediate result; which is materialized into a view table and the updates on this view table are passed to the next view expression. This way, each single base operation propagates along the maintenance chain. The last materialized view in the chain represents the final result of the query.

Figure 2 shows the maintenance plan of query q. A maintenance plan always comprises a set of materialized tables (i.e. base tables and view tables), which Figure 2 depicts as boxes; as well as a set of view expressions, which Figure 2 depicts as small circles. The dotted lines represent the paths, along which the view updates flow to update the whole chain.

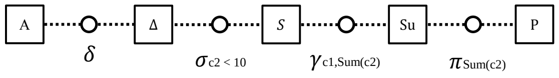

Figure 2: Maintenance plan

On the very left side of the figure, the base table A – over which q is defined – is shown. The base table is always the starting point of the maintenance chain. In case of a join operation, there can be multiple base tables. In example of q, the query is just defined over one base table A. To the right side of base table A, a delta expression (i.e. δ) follows to build the intermediate view table Δ. The connection between A and Δ implies the following: all client operations that are applied to base table A, are modified by the δ expression (i.e. their delta is evaluated) and stored to the Δ view. Next to the Δ view, a selection expression (i.e. σ) build a connection to the intermediate view S. The σ expression drops or passes (based on the selection criterion) the client operations from Δ to S. In the next to steps, a sum aggregation (γ) and a projection expression (i.e. π) succeed S to complete the maintenance chain. Expression π represents the last expression such that view P holds the final result of the query.

4.1 DAG

In order to describe a maintenance plan for a SQL-query (e.g. the plan in Figure 2), we use a so-called AND-OR-DAG (reference). The AND-OR-DAG was developed in the context of query optimizing. Since it is a perfect fit for our needs, we adapt the concept to describe and optimize the maintenance in our VMS. Every SQL-query has a corresponding maintenance plan, which can be described by a AND-OR-DAG. As the name says, the AND-OR-DAG is a directed acyclic graph. It alternates between view tables and view expressions; We denote the maintenance plan as $P = (V, E)$, where V is a set of vertices and E a set of edges. A vertex $v \in V$ is either a table vertex V_T with $T = T_0, ..T_n$ and or an expression vertex V_X with $X = \{\delta, \sigma, \gamma, etc..\}$. An edge $e \in E$ either connects a table vertex and an operation vertex or two expression vertices (concatenation).

Each query can be translated into a maintenance plan $p \in P$. For example, the query q from Figure 2 translates to a plan p where $V = \{A, \delta, T_0, \sigma, T_1, \gamma, T_2, \pi, T_3\}$ and $E = \{A \to \delta, \delta \to T_0, ...T_2 \to \pi, \pi \to T_3\}$. Through changing the order of operations or combining multiple view tables (i.e. executing different optimization), new maintenance plans can be constructed. These plans represent distinct DAGs, but they all compute the same result. We capture those maintenance plans in a set $P_q = \{p_1, ...p_n\}$.

4.2 Translation of SQL pattern

So far, we have shown how we a maintenance plan can be derived for a single query q. As we strive for supporting full SQL semantics, we now explain how a complete *SQL pattern* can be translated into a maintenance plan. This pattern is a blueprint to translate any type of SQL query. Though, the pattern can also be applied to describe nested

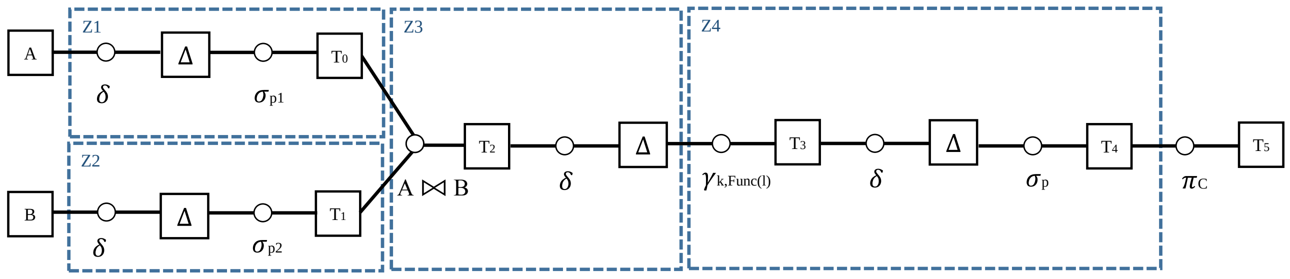

Figure 3: SQL pattern

SQL queries, we only discuss simple SQL queries (i.e. every view expression only occurs once). A usual SQL parser evaluates the clauses of a simple SQL query in the following order: from, where, group by, having, select, order by. We use these clauses and translate them into a general maintenance pattern (AND-OR-DAG). By removing elements that are not part of a query, every other simple query can be constructed.

As a first step, we map the clauses of a SQL query to the respective basic view expressions: from translates to one (or in case of a join) multiple base tables; the where clause translates to a Selection expression (σ); the group by translates to an aggregation (sum, count, min, max, depending on the aggregation function) expression; the having clause is just another selection expression; and, finally, the select clause translates to a projection expression.

Figure 3 shows the generalized maintenance plan of a simple SQL query. As before the figure depicts table vertices as boxes and expression vertices as small circles. The dotted areas are *zones*. The table vertices in a zone share the same row key (i.e. primary keys of the tables). This is insofar of interest as view tables within a zone can be merged to share the results of multiple view expressions at once. We will use this property later in the paper to reduce the amount of intermediate view tables.

On the very left side of Figure 3 you can see two base tables A and B. Both base tables are followed by a delta expression and the respective view table, as the changes in both tables need to be tracked. In the next step both delta tables are connected to a selection expression (σ) with different predicates (p_1 and p_2). Selection expressions (i.e. where clauses) are usually evaluated first in SQL queries; they are defined over a base tables only and they can reduce the amount of operations, passed to subsequent views, significantly. Applying a selection to a record doesn't change the row key. That is why Δ and Selection views are located in the same zone.

View tables T_0 and T_1 are combined to T_2 by a join expression. A join is usually evaluated in the from clause of a query. In Figure 3, two base tables A and B are joined. In general, the from clause of a query could contain a join on n-tables. For that case, we distinguish two cases: (1) All base tables are joined on the same join key. (2) Different pairs of join tables are joined on different join keys. Case (1) is trivial, as we still obtain one join view and one join expression with n input edges from the different delta tables. In case (2), we have to build a join view for each join pair and combine those join views again until just one join view remains.

After a join expression, the row key changes. There are joins (e.g. key/foreign-key) where the row key of one of the join tables can be used. However, as we control the process of view maintenance and we aim at generality, the row key of a join table is always the composite key of both base tables. Therefore, the join table T_2 also defines a new zone. A table in a new zone has to be followed by another Δ view to, again, track the changes of its records. At a first glance, the high amount of delta views may appears like a big overhead in the system; but during the optimization phase this overhead nearly vanishes, as most of the delta views can be combined with other expressions and don't need to be materialized separately.

After the second Δ view, an aggregation expression (e.g. a sum view) is applied to the maintenance plan. It is what we need when a group by clause in a query defines a set of aggregation keys and the select statement defines a set of aggregation functions and their respective values. Thus the aggregation view is, in general, defined as $\gamma A, Func(X)$. Like the join view, the aggregation expression enforces another row key in view T_3. Again, there are special cases, where the join key might be equal to the aggregation key. But as we aim at generality, we assume that the aggregation expression starts a new zone (i.e. Z_4). As before, the change of row keys in T_3 is followed by a delta expression. The last view expression in Z_4 is another selection (σp). It is derived from the optional having clause in a SQL query. This selection is treated exactly like the first selection, just that it operates upon the records of an aggregation and not of a base table.

The last view expression, the projection (πC) is present in most of the queries, as it is mapped from the select clause in a query. The projection doesn't necessarily has to be the last expression of the maintenance plan. It can also be materialized earlier. The projection only influences the number of columns. Thus, the resulting view uses whatever row key is taken in the preceding view. That is also the reason why view T_5 is not located within a special zone.

Now, we have mapped all possible query clauses (except the sort clause), building a maintenance plan from a query is straight forward. If a clause exists in the query the according view expression and its materialized result are added to the graph. The order, in which the vertices are added to the maintenance plan, is depicted in Figure 2. The maintenance plan that belongs to a query is also called query DAG.

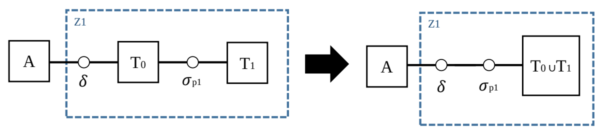

Figure 4: Horizontal optimization

4.3 Horizontal optimization

When evaluating a query the normal way (i.e. loading the record set from the base table and computing the results), intermediate steps doesn't need a materialization. In that case a materialization is only reasonable if it pays off over multiple subsequent uses. However, in the context of incremental view maintenance, intermediate steps have to be materialized. The higher storage cost is traded off against an easy and efficient update process – one base table operation just triggers some small view updates. However, not all intermediate steps have to be materialized, as this is quite storage consuming. We can already optimize a single query by looking at the different zones, depicted in Figure 3. As all the view tables within a zone share the same row keys, there is no need of materializing them separately; view expression can be combined and the zone just needs one materialization (cf. Figure 4).

So, how can multiple view expressions be combined to affect the same materialized view. From a logical point of view, it seems trivial. We delete the table nodes, concatenate the operations and build a new table node (cf. Figure 4). However, in practice we have to make two modifications: (1) the maintenance plan has to be updated, (2) the functionality of VMs needs to be adapted.

Updating the maintenance plan: To merge the table nodes of a zone, we need all expression nodes and table nodes of that zone in a sequence. Lets assume, the expression nodes of a zone are defined in correct order as $V_{X_{zone}} = \langle v_{x_1}, .., v_{x_n} \rangle \subset V_X$; the corresponding table nodes of the zone are defined as $V_{T_{zone}} = \langle v_{T_1}, ..., v_{T_n} \rangle \subset V_T$. Further, each table node refers to a table with the same row key (this is given as long as table nodes are in the same zone). Now, we can update the zone as depicted by algorithm.

Algorithm 1 Horizontal merging

```
   procedure $horizontalMerge(V_{X_{merge}}, V_{T_{merge}})$
       for all $v \in V_{T_{merge}}$ do
           $remove(v)$                                  ▷ queue marker
       end for
5:     for all $v \in V_{X_{merge}}$ do
           $remove(outgoingEdges(v))$
           for all $e \in ingoingEdges(v)$ do
               $setTarget(e, lastV)$
           end for
10:        $lastV \leftarrow v$
       end for
   end procedure
```

Adapting VMs functionality: So far, the maintenance plan only consists of alternating expression and table nodes. The VM receives a client operation. It looks up where the client operation comes from. Thus, it knows the next expression and table node. It simply applies the expression and inserts the result into the corresponding table node. After optimization, multiple expressions can be applied in a row. Just applying all expressions to the same table is not an option; this would reduce storage space, but the processing effort would be the same. Therefore, we also concatenate the execution of the expression. To do so, we take a look at execution of the maintenance plan at VM:

According to Jacobsens algorithm a view update algorithm consists of three parts: (1) retrieving the old view record, which, in a KV-store architecture, conforms to a *get*-operation; (2) computing the new view record, which is done locally by a function *compute* (3) adding the new, or deleting the old view record, which conforms to a *put*- or a *delete*-operation. If a view table is updated with a single basic view expression, those three steps are executed differently depending on the type of view expression (i.e. selection, projection, aggregation, etc.). However, if we update a view table with multiple view expressions at once, the three steps need to be combined. So, if we merge $V_{x_1}, .., V_{x_n}$, we likewise need to concatenate their get, compute and put functions.

$$get(x_1) \circ ... \circ get(x_n) \tag{1}$$

$$compute(x_1) \circ ... \circ compute(x_n) \tag{2}$$

$$put(x_1) \circ ... \circ put(x_n) \tag{3}$$

Example: Let a base table with a row key k and three columns be defined as $A(k, c_1, c_2, c_3)$. Let further a combination of view expressions be defined as δ, $\sigma_{c_1<10}$ and π_{c_2,c_3}. As soon as a base table operation (let it be an insert or an delete operation) is propagated all three steps of the algorithm have to be executed for a concatenation of all three expressions. Since Step 1 is equal for all three expressions we obtain the following equation: $get(k) \circ get(k) \circ get(k) = get(k)$. Instead of fetching the same record three times, we just fetch it once. Step 1 delivers the old view record r'. Step 2 computes the new view record, which is different for all three operations. Let the client operation be an insert into the base table. Then we obtain the following concatenation: $r \circ if(c_1 < 10) then r; else \emptyset \circ r$
$\{c_1\} = if(c_1 < 10) then r$
$\{c_1\}; else \emptyset$. The delta expression just writes back record r such that the old record is available if another operation is performed on the same row key. The selection expression writes back the record or nothing depending on the selection condition. The projection writes back only a selected number of columns. By combining those expressions, we derive a new expression, which is shown on the right side of the equation. In the last Step 3, we simply write the new record back to the view, which is similar for all views again: $put(k) \circ put(k) \circ put(k) = put(k)$. Instead of three puts, we just perform one.

In this section, we have discussed how multiple view expression in a query DAG can be combined to create one materialized view. This technique can be regarded as local optimization, as we perform it on a single query. It reduces the storage capacity needed, as well as the amount of total update operations in the VMS.

5. MULTI VIEW OPTIMIZATION

In the previous sections, we have shown how SQL queries can be translated into a query DAG (i.e. maintenance plan).

Now, we are ready to tackle the multi view problem. Imagine an application or warehousing system that encounters a large number and variety of different SQL queries. Using the naive approach, as described before, would mean that every query would translate to its own set of materialized views without sharing any intermediate results. To improve performance and reduce the storage overhead a lot, we need a query optimizer; the optimizer has to find a global maintenance plan which is optimal with regard to total throughput and storage usage (or some other cost parameters).

5.1 Cost model

The cost model has to trade-off the different aspects of incremental view maintenance. On the one side, there is the throughput of maintenance operations (which determines the freshness of the data in the view). A client operation triggers multiple updates along the maintenance plan. Depending on the amount and type of view expressions a specific number of view updates is generated. On the other side, there is the storage cost of hosting the intermediate and resulting view tables. The more intermediate tables are hosted, the more storage space is occupied. Therefore, we aim at reducing the number of intermediate views as much as possible. Whenever, we can possibly execute multiple operations on the same table, we combine them.

Both types of cost are expressed in out cost model: the processing or maintenance cost and the storage cost. The maintenance cost can be described by the number of operations (i.e. get, put and delete) per time interval t that the VMS needs to execute to keep the view records up-to-date. The rate of operations depends on the view expressions, thus, we compute it by iterating over the expression nodes in the maintenance plan. First, we define the cost over a single vertex $v \in V$ as $c(v)$. The equation is given as follows:

$$\begin{aligned} c(v) = & \underbrace{(i(v)+u(v)+d(v)) * w_g}_{\text{get}} \\ & + \underbrace{(i(v)+u(v)) * w_p}_{\text{put}} + \underbrace{(d(v)+u(v)) * w_d}_{\text{delete}} \end{aligned} \tag{4}$$

The equation computes the cost of all get, put and delete operations that are required to update the subsequent view table. The get part sums all get operations and multiplies them with w_g, which is a weight factor. For all incoming operations, a get-operation has to be performed on the view table. This way, the VM retrieves the old view record such that it can add the delta on top of it. In some cases, get operations are not needed at all (e.g. if a selection view follows a delta view). Then, the get-part of the equation can be set to zero. Since KV-stores are write optimized data bases – and, thus, execute writes faster than reads – we also use weight factors w_p and w_d for put and delete operations. Update operations can always be substituted by an insert plus a delete operation. Thus they are included in the put part, as well as the delete part of the equation.

The functions $i(v)$, $u(v)$ and $d(v)$ deliver the rate of insert, update and delete operations that the vertex receives from all its direct predecessors. Since the predecessors are table or expression nodes, whose input, again, may depend on preceding nodes, we define the input rate $i(v)$ recursively.

$$\begin{aligned} i(v) = & \sum_{v \in pred(v_x)} i(v) \quad if\ pred(v) \neq \emptyset \\ i(v) = & \quad \lambda_i(v) \quad if\ pred(v) = \emptyset \end{aligned} \tag{5}$$

If the set of predecessors is not empty, the sum of input rates over them is computed. If the set of predecessors is empty, the node must be a base table. Then, function $\lambda_i(v)$ represents the rate of operations that are applied to the base table. The input rate at the base tables is either provided to VMS as a meta parameter or has to be determined in a test run, before actual view maintenance is performed. The equations to compute update $u(v)$ and delete $d(v)$ rates can be formulated accordingly.

Having the processing cost evaluated we still need to compute the estimated maximum storage cost. The following general formula works for all view table nodes:

$$c(v) = r(v) * f(v) \tag{6}$$

The input is a table vertex $v \in V$. The function $r(v)$ evaluates the number of records in the table. Again, the number of records in tables can be defined recursively over the maintenance plan as:

$$\begin{aligned} r(v) = & \sum_{v \in pred(v)} r(v) \quad if\ pred(v) \neq \emptyset \\ r(v) = & \quad \mu(v) \quad if\ pred(v) = \emptyset \end{aligned} \tag{7}$$

The number of records is defined as the sum of records from the nodes predecessors, unless it is a base table. Then, the function $\mu(v)$ delivers the average number of records. Again, this number is either given or has to be estimated. Based on the number of records in the base table, the number of records in the view tables can be computed. The number has to be multiplied with a factor $f(v)$. The factor expresses that, depending on the view expression, a the number of view records can be larger or smaller than in the base table. A selection expression e.g., reduces the amount of records in the view table depending on the selection predicate. A join can possibly increase the number of records, e.g. when the cross product of two tables is built. Thus the function $f(v)$ is denoted as the product of factors from all view expressions that are direct predecessors of the node $predX(v)$. The factor, represented by function $\psi(v_x)$ has to be given or needs to be evaluated in a test run.

$$f(v) = \prod_{v_x \in predX(v)} \psi(v_x) \tag{8}$$

Now, we have defined the cost of table, as well as expression vertices, we are ready to compute the total cost of the maintenance plan. It is done as follows:

$$c(p) = \sum_{v_x \in V_x(p)} c(v_x) * w_x + \sum_{v_t \in V_T(p)} c(v_t) * w_t \tag{9}$$

In order to compute the cost of the graph, we need to evaluate the sum of the cost of all nodes. Because the cost is based on throughput of client operations, and nodes are depending on each other (i.e. one node my reduce the throughput of the next one), the maintenance plan needs to be traversed starting at the base tables. Thus, we apply a

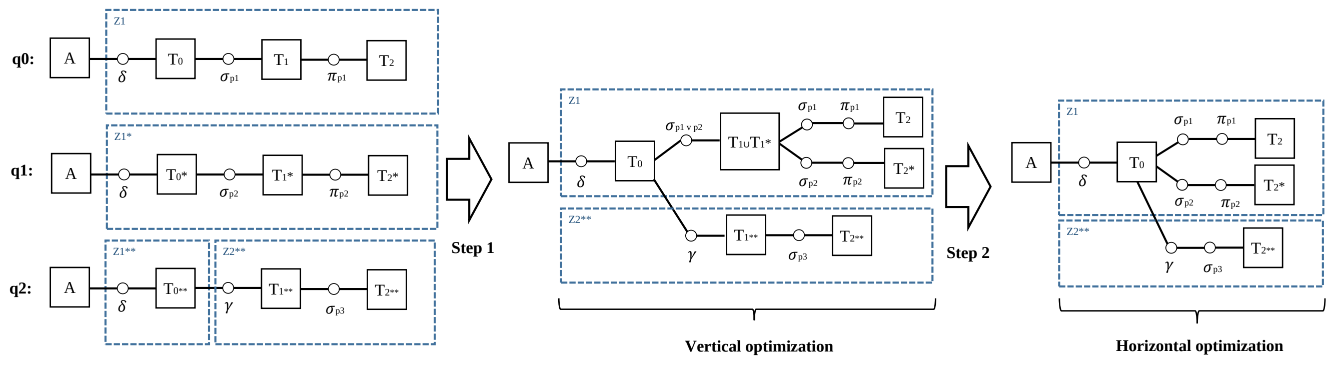

Figure 5: Multi-query optimization

graph traversing algorithm (e.g. Kahn's algorithm) to determine a topology of the graph (i.e. an ordered according to the placement in the graph). This can be done with linear complexity. Then, we follow the topology and compute the throughputs and storage cost of all nodes iteratively. Finally, we obtain the overall sum of the maintenance plan $c(p)$.

5.2 Vertical optimization

The optimization algorithm operates on a set of query DAGs. In section before, we merged table nodes within a zone of a single query pattern; we called it horizontal optimization. Now, we also consider merging table nodes of different maintenance plans from different queries; we call it vertical optimization.

Figure 5 depicts the process of vertical optimization (and subsequent horizontal optimization). On the left side of the figure, three separate maintenance plans of three different queries are shown (i.e. q_0, q_1 and q_2). These plans are directly derived from the corresponding SQL statements:

1. $q_0 \leftarrow$ Select c_1 from A where p_1
2. $q_1 \leftarrow$ Select c_2 from A where p_2
3. $q_2 \leftarrow$ Select Sum(c_2) from A group by c_1 having p_3

In Step 1, multiple vertical merges are executed on the plans, which leads to the graph in the center of the figure. The algorithm starts at the base tables (i.e. A) and merges in a zipper-like fashion along the view chain. As a first substep all base tables (i.e. all As) can be merged together. The nodes are identified by name; we take the first node of q_1 as base table node and remove the base table nodes of q_1 and q_2. Further, we redirect the outgoing edges of both maintenance plans. As a second sub-step, all delta views (i.e. T_0, T_0* and T_0**) are merged together. Since the delta expression for all three views is the same, we proceed exactly as before. We

Merging equal expressions

For the merging of two (or more) table nodes of two (or more) query DAGs, some preconditions have to be fulfilled: (1) the table node is either computed from the same expression (e.g. $\sigma_{c_1=10}(A)$) or it is computed from a similar expression of the same operator (e.g $\sigma_{c_1<10}(A)$ and $\sigma_{c_1>5}(A)$); thereby, the two expressions need at least a set of intersection records. Otherwise, it makes no sense to merge them. (2) the derivation of both tables has to be the same (aside from the merged expression). For example, if we merge two aggregation expressions, then the preceding expressions (e.g. delta, selection) have to match exactly.

Merging equal view tables is always advantageous. Instead of two tables we just use one, thus, we save storage and processing capacity.

Merging similar expressions

5.3 Problem definition

As a first step towards the solution, we formalize the problem as follows:

Given a set of n queries q_n and the corresponding set of maintenance plans $p_n \in P$: what is the global maintenance plan p_g that executes all plans p_n and is optimal with regard to overall cost.

5.4 Optimization algorithm

Now that we have defined the types of possible merge operations (i.e. horizontally and vertically) and also constrained the cost model, we discuss the optimization algorithm. The first version of our algorithm will work in three iterative steps: (1) the algorithm optimizes vertically; it finds all equal expressions and table nodes and merges them, without cost evaluation. (2) the algorithm optimizes vertically; it finds all similar expressions and table nodes and merges them based on cost estimation. (3) the algorithm optimizes horizontally; it uses the resulting query DAG from the steps before and condenses it further.

5.5 Complexity

6. RELATED WORK

Research of incremental view maintenance technologies started back in the nine-ties. The research was primarily conducted with regard to data warehouses [14–17]. While many principles of incremental maintenance still apply today (e.g. how deltas are computed), the underlying architectures have changed significantly. The storage capacity has grown, the degree of distribution increases continuously. Even though, there was research on maintenance of distributed sources [16], the warehouse itself was a stand-alone system that could be easily queried to retrieve information. With the amount of analytical data today multi-view optimization becomes a problem.

Cui and Widom [14, 15] already examined the usage of auxiliary views. Even if the auxiliary views in our context

Algorithm 2 Multi view optimization

```
    procedure $optimize(P_Q)$                         ▷ $P_Q$ is a set of plans
        $V_G \leftarrow findEqualGroups(P_Q)$                        ▷ Step 1
        for all $v_g \in V_G$ do
            $P_Q \leftarrow verticalMerge(v_g)$
 5:     end for
        $V_G \leftarrow findSimilarGroups()$                         ▷ Step 2
        for all $v_g \in V_G$ do
            $benefitial \leftarrow c(verticalMerge(v_g)) < c(v_g)$
            if $benefitial$ then
10:             $P_Q \leftarrow verticalMerge(v_g)$
            end if
        end for
        $V_G \leftarrow findZones()$
        for all $v_g \in V_G$ do                                     ▷ Step 3
15:         $P_Q \leftarrow horizontalMerge(v_g)$
        end for
    end procedure
```

serve a different purpose (they define a pre-step of the logical computation), they materialized intermediate steps to save processing effort. Already here the trade-off of storage vs. processing capacity can be observed. However, their research [14, 15] was more related to the problem of lineage tracing and view selection.

View selection as is also described in [6, 7] is problem, where an algorithm decides, which sub-set of a set of view candidates should be materialized. Especially in the context of multi-query optimization [8–12], it is widely used. However, view selection is a different problem than multi-view optimization (especially views that are maintained incrementally). In multi-view optimization every view has to be materialized. The problem is not the choice of the correct materialization set-up, but the reduction of processing and storage capacity by resource sharing.

Even if not the same problem, multi-query optimization (or view selection) can provide instruments to also approach the multi-view problem. For example, the maintenance plan (e.g. the AND-OR query DAG) or a cost model, similar to that of the Volcano optimizer [11, 12] can be used to find an optimal solution for the multi-view problem.

A lot of the modern algorithms for sharing results and optimizing in a multi-query environment are based on the map-reduce paradigm [18–21]. Map-reduce provides a very similar data model to our solution – records are processed in a key-value format. The translation of SQL-statements into basic expressions (like selection, aggregation, join) that can share elements of computation [20] is somewhat comparable to our solution. However, the processing style for map-reduce is a batch-oriented strategy. Self-contained sets of records are retrieved, computed and stored back to disk. When discussing the sharing of intermediate results, this problem also reduces to a view selection problem. As already stated, our model favours an incremental strategy. As we are dealing with streams of transactions (as opposed to sets of records), optimization algorithms are different.

7. EVALUATION

7.1 Set-up

7.2 Work load

7.3 Experiments

8. REFERENCES

[1] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[3] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[4] Lars George. *HBase: The Definitive Guide*. O'Reilly Media, Inc., 2011.

[5] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[6] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 307–318, New York, NY, USA, 2001. ACM.

[7] Dimitri Theodoratos and Wugang Xu. Constructing search spaces for materialized view selection. In *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP*, DOLAP '04, pages 112–121, New York, NY, USA, 2004. ACM.

[8] Anastasios Kementsietsidis, Frank Neven, Dieter Van de Craen, and Stijn Vansummeren. Scalable multi-query optimization for exploratory queries over federated scientific databases. *Proc. VLDB Endow.*, 1(1):16–27, August 2008.

[9] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in multi-query optimization. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 59–70, New York, NY, USA, 2001. ACM.

[10] Andinet Assefa and Fekade Getahun. Multi-query optimization for semantic news feed query. In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, MEDES '12, pages 150–157, New York, NY, USA, 2012. ACM.

[11] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, Washington, DC, USA, 1993. IEEE Computer Society.

[12] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 249–260, New York, NY, USA, 2000. ACM.

[13] J. Adler, M. Jergler, and H.-A. Jacobsen. Dynamic scalable view maintenance in kv-stores(extended). `https://drive.google.com/file/d/0B_xaKQjmla0nbE93VDVHeEFUeTA/view`.

[14] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal*, 12(1):41–58, May 2003.

[15] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, June 2000.

[16] Gianluca Moro and Claudio Sartori. Incremental maintenance of multi-source views. In *Proceedings of the 12th Australasian Database Conference*, ADC '01, pages 13–20, Washington, DC, USA, 2001. IEEE Computer Society.

[17] Lingli Ding, Xin Zhang, and Elke A. Rundensteiner. The mre wrapper approach: Enabling incremental view maintenance of data warehouses defined on multi-relation information sources. In *Proceedings of the 2Nd ACM International Workshop on Data Warehousing and OLAP*, DOLAP '99, pages 30–35, New York, NY, USA, 1999. ACM.

[18] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. Mrshare: Sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, 3(1-2):494–505, September 2010.

[19] Srinivas Vemuri, Maneesh Varshney, Krishna Puttaswamy, and Rui Liu. Execution primitives for scalable joins and aggregations in map reduce. *Proc. VLDB Endow.*, 7(13):1462–1473, August 2014.

[20] Guoping Wang and Chee-Yong Chan. Multi-query optimization in mapreduce framework. *Proc. VLDB Endow.*, 7(3):145–156, November 2013.

[21] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 99–110, New York, NY, USA, 2010. ACM.