A view maintenance concept for key-value stores

Ben Trovato*
Institute for Clarity in Documentation
1932 Wallamaloo Lane
Wallamaloo, New Zealand
trovato@corporation.com

G.K.M. Tobin†
Institute for Clarity in Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
webmaster@marysville-ohio.com

Lars Thørväld‡
The Thørväld Group
1 Thørväld Circle
Hekla, Iceland
larst@affiliation.org

Lawrence P. Leipuner
Brookhaven Laboratories
Brookhaven National Lab
P.O. Box 5000
lleipuner@researchlabs.org

Sean Fogarty
NASA Ames Research Center
Moffett Field
California 94035
fogartys@amesres.org

Charles Palmer
Palmer Research Laboratories
8600 Datapoint Drive
San Antonio, Texas 78229
cpalmer@prl.com

ABSTRACT

Executing SQL-like queries on distributed data sets in key-value stores(KV-stores) is a well known problem. While providing high availability, high write throughputs and mechanisms for fault tolerance, KV-stores sacrifice data semantics and powerful query languages. To reintroduce the SQL-like query capabilities, we propose a concept for materializing views in KV-stores. The different view types form – when combined with each other – the basis to build up a higher-level SQL-language. Further, the materialized views can be created and maintained efficiently to deliver fresh data for frequently accessed computations.

1. INTRODUCTION

KV-stores Nowadays, distributed KV-stores are powerful databases to handle large amounts of data. They spread data over the network, thereby allowing to scale the system horizontally. In case of scarce resources or heavy amounts of client request, the administrator can just add more nodes to relive the system. Spreading the data also minimizes access time to the clients and improves availability. A KV-Store is optimized for high write throughput and – due to its distributed nature – can serve thousands of client requests at a time. Since data is spread, computations can be executed in parallel on multiple parts of the data sets. Likewise, the distributed KV-store replicates data over multiple nodes and provides automated mechanisms for fault tolerance.

KV-store weakness Offering all the mentioned features the KV-store avoids a complex data model and sacrifices a powerful query language for a much simpler API (comprising of put, get and delete operations). Tables are schema-less, records are identified by a row key, data is stored into byte arrays. To perform analytical queries on tables, we perform scans over large data sets. This implies large run times for the queries and leads to performance gaps, during which clients suffer slow response times. Likewise, the logic for the SQL query is executed on application side. Often times code is application specific and cannot be reused later.

Known approaches One solution to the problem is loading the table snapshots of KV-store into an external data warehouse. Here, all kinds of analytical operations can be applied to the data set (in-memory technologies can be used. Since loading a complete data set is a long running task and we want to provide up-to-date results, this it not an option.

VMS

Our approach Materialized views are a well known concept to obtain results from a database and cache them for subsequent client requests. A materialized view can express any SQL-query and provides fast access to the query results. However, materialized views introduce the problem of view maintenance – once the base data changes the view needs to be updated. To update the views efficiently, we apply the principles of deferred and incremental view maintenance. We just update those part of the view that are affected by the corresponding base table update.

In paper XY, we designed a View Maintenance System that performs view maintenance at scale and tackles all the related consistency issues. Now, we use the View Maintenance System to implement all kinds of basic view types (e.g. selection, aggregation, join, etc..). Then, we provide a SQL layer to translate SQL queries into a view maintenance plan. We use a DAG (Directed acyclic graphs) to describe the query as a tree of connected materialized views (of basic view types). We describe the DAG of a general SQL pattern to match any kind of SQL query. Using VMS, we create and maintain the view tree in a hierarchical manner and provide

*Dr. Trovato insisted his name be first.
†The secretary disavows any knowledge of this author's actions.
‡This author is the one who did all the really hard work.

the latest results of the query. Since each materialized view adds maintenance effort, we introduce methods to optimize (i.e. merge intermediate views, reorder the DAG) and reduce storage, as well as computation cost.

The paper is structured as follows: first, we define a general data model for KV-stores by looking at the different available implementations.Then we briefly discuss the VMS designed in paper XY. In the core part of the paper, in Section view types, we define the basic view types that can be combined to form higher level abstractions. In Section 4, we show how a view maintenance plan can be derived from any SQL query. We also optimize this maintenance plan to reduce storage and computation cost. Finally, we evaluate our approach in an extensive experimental study, using HBase as a KV-store implementation.

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 2).

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 2).

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 $f : K \to V$. 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 $f : (K, C) \to V$. 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 $a = (k, \{\langle c_1, v_1\rangle..\langle c_i, v_i\rangle\}_1..., \{\langle c_{i+1}, v_{i+1}\rangle..\langle c_n, v_n\rangle\}_n)$.

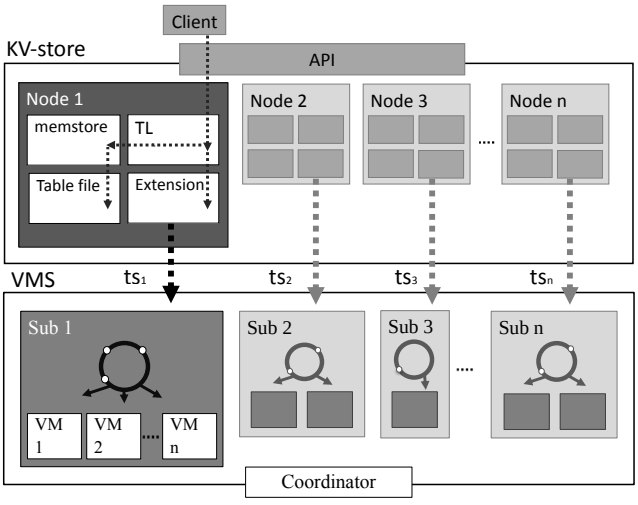

Figure 1: System Overview

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 $t = delete(k, \{\langle c_1, \emptyset\rangle..\langle c_n, \emptyset\rangle\})$.

[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.

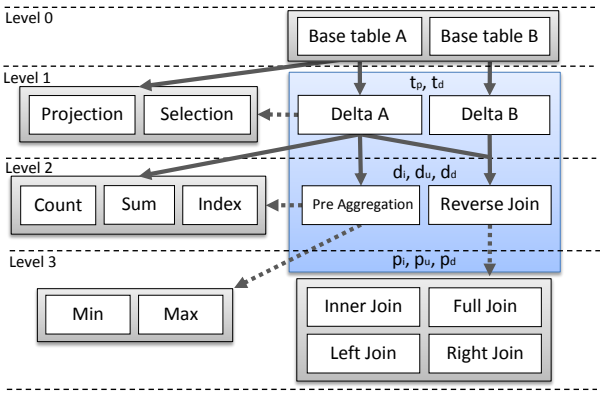

Figure 2: View types and their dependencies

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 View Maintenance System

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 TYPES

In this section, we develop techniques for maintaining the following basic view types 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 2 gives an overview of all view types and their dependencies.

Views at Level 1 directly derive from base tables residing at Level 0, as indicated by the directed edges in the figure. A path along edges from a base table to a view at a higher-numbered layer represents a complete update path through the system. More complex views on Level 2 and higher are derived from views on lower layers, which, from the perspective of these views, serve them as "base data". Note that there are alternatives in how certain view types can be derived. For example, `SELECTION` can be derived from a base table or from a `DELTA` view. Determining and quantifying the trade-offs among these view maintenance alternatives is outside the scope of the present work.

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 1, 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 1).

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.

View type	Def	record format	on Insert	on Delete
Base	A	$r = (k, \{\langle c_1, v_1\rangle, .., \langle c_n, v_n\rangle\})$	$t_p = put(r)$	$t_d = delete(k)$
Delta	$D = \delta(A)$	$r' = (k, \{\langle c_1, v_1'\rangle, .., \langle c_n, v_n'\rangle\}) \vee \emptyset$	$put(r)$	$delete(k)$
Pre Agg	$P = \gamma_{c_\alpha, \langle K, c_\beta\rangle}(D)$	$p = (v_\alpha, \{\langle k_1, v_1\rangle, .., \langle k_n, v_n\rangle\})$	$put(v_\alpha, \{\langle k, v_\beta\rangle\})$	$delete(v_\alpha, \{k\})$
Rev Join	$R = \gamma_{c_\alpha, \langle K, r\rangle, \langle L, s\rangle}(D_A, D_B)$	$p = (v_\alpha, \{\langle k_1, r_1\rangle, .., \langle k_n, r_n\rangle\}_A, ..$	$A : put(v_\alpha\{\langle k, v_1\rangle\}_A)$	$A : delete(v_\alpha, \{k\}_A)$
		$\{\langle l_1, s_1\rangle, .., \langle l_m, s_m\rangle\}_B)$	$B : put(v_\alpha, \{\langle k, v_1\rangle\}_B)$	$B : delete(v_\alpha, \{k\}_B)$

Table 1: Auxiliary computation

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 1). 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 1). 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).

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

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 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

View type	Def	record format	Insert	Delete
Selection	$\sigma_C(D)$	$r \vee \emptyset$	$(C) \rightarrow put(r)$	$(C) \rightarrow delete(k, \{c_1, .., c_n\})$
Projection	$\pi_{c_{\alpha_1},..c_{\alpha_n}}(D)$	$(k, \{\langle c, v\rangle \mid c \in c_{\alpha_1}, ..c_{\alpha_n}\})$	$put(k, \{\langle c, v\rangle \mid c \in c_{\alpha_1}, ..c_{\alpha_n}\})$	$delete(k, \{c_{\alpha_1}, .., c_{\alpha_n}\})$
Count	$\gamma_{c_\alpha, Count(c_\beta)}(D)$	$(v_\alpha, \{\langle c, v_{count}\rangle\})$	$put(v_\alpha, \{\langle c, v_{count} + 1\rangle\})$	$put(v_\alpha, \{\langle c, v_{count} - 1\rangle\})$
Sum	$\gamma_{c_\alpha, Sum(c_\beta)}(D)$	$(v_\alpha, \{\langle c, v_{sum}\rangle\})$	$put(v_\alpha, \{\langle c, v_{sum} + v_\beta\rangle\})$	$put(v_\alpha, \{\langle c, v_{sum} - v_\beta\rangle\})$
Min	$\gamma_{c_\alpha, Min(c_\beta)}(D)$	$(v_\alpha, \{\langle c, v_{min}\rangle\})$	$(v_\beta < v_{min}) \rightarrow put(v_\alpha, \{\langle c, v_\beta\rangle\})$	$(v_\beta = v_{min}) \rightarrow put(v_\alpha, \{\langle c, v_x\rangle\})$
Index	$\gamma_{c_\alpha}(D)$	$v_\alpha, \{\langle k_\alpha, v_\alpha\rangle, ..\}_A$	$put(\{\langle k, v_1\rangle\}_A)$	$delete(x, \{k\}_A)$
Join	$A \bowtie B(R)$	$((k_\alpha), \{\langle k_\alpha, v_\alpha\rangle, ..\}_A$	$put(\{\langle k, v_1\rangle\}_A)$	$delete(x, \{k\}_A)$

Table 2: Standard computation

single view table record.

Let a selection view be defined as $S = \sigma_{c_2 < v_x}(A)$, where the selection condition is $(c_2 < v_x)$ requiring that selected values in column c_2 are smaller than v_x. The view manager processes operations t_p and t_d as follows: 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. Pre-emptive deletes are executed for the same reasons as above.

An alternative maintenance strategy is to derive both views from a `DELTA` view. We create a `DELTA` view $D = \delta_{c_1,c_2}(A)$ and define the `SELECTION` view as $S = \sigma_{c_2 < v_x}(D)$ and likewise the `PROJECTION` view as $P = \pi_{c_2}(D)$. This alleviates the need for pre-emptive deletes, since the VM always receives the complete row state. The computation of the `SELECTION` view changes to: 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`: *Example 1:* Given tables A, D and S as before (i.e., view tables are defined as $D = \delta_{c_1,c_2}(A)$ and $S = \gamma_{c_1, Sum(c_2)}(D)$). Let the initial table states be given as depicted in Figure 3. Let the client perform an update operation on row key k_2, which is inserted into A resulting in the entry $t_1 \in T$ in the TL. Let $t_1 = put(A(k_2, \{\langle c_1, x_2\rangle, \langle c_2, 15\rangle\}))$. At this point, it is not know whether t_1 is an insert or an update operation (both operations could result from a put.) Once t_1 is processed on the delta view D by a view manager, another entry $t_2 \in T$ is created in the TL. Being induced by t_1, t_2 captures the changes to row k_2 as $t_2 = put(S(k_2, \{\langle c_1, x_1 \rightarrow x_2\rangle, \langle c_2, 30 \rightarrow 15\rangle\}))$. Note, t_2 now identifies the client operation as an update. Operation t_2 triggers an update on S. Because the operation changes the aggregation key, the VM generates an update statement for aggregation key x_1 and x_2. To execute the first statement, the VM queries the view table and retrieves $S(x_1, \{\langle c_s, 50\rangle\})$. It computes $50 - 30 = 20$ and puts $S(x_1, \{\langle c_s, 20\rangle\})$ to update view table S. To execute the second statement, the VM queries the view table and retrieves $S(x_2, \{\langle c_s, 85\rangle\})$. It computes $85 + 50 = 135$ and puts $S(x_2, \{\langle c_s, 135\rangle\})$ to update view table S.

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.

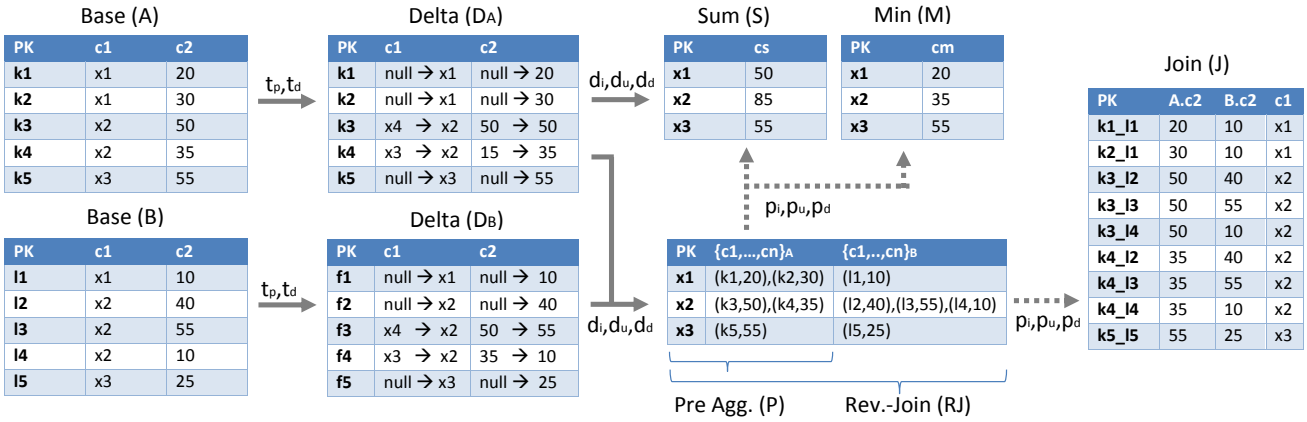

Figure 3: View table example

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.

The good news is, with only a handful of manual settings[3], the LaTeX document class file handles all of this for you.

The remainder of this document is concerned with showing, in the context of an "actual" document, the LaTeX commands specifically available for denoting the structure of a proceedings paper, rather than with giving rigorous descriptions or explanations of such commands.

4. MAINTENANCE PLAN

4.1 DAG

We define a maintenance plan as directed acyclic graph (DAG). We denote the graph as $G = (V, G)$, where every vertex $v \in V$ corresponds to a base table or a materialized view and every edge $e \in E$ corresponds to a computation, e.g. $\sigma(A)$. The client operations flow from the origin of the graph, i.e. the base table, over the edges to the materialized views. The edge transforms the input stream into updates for subsequent views. Vertices without outgoing edges represent views that deliver the result to the client.

Example – Consider a client issuing the following query to the View Maintenance System:

```
SELECT Sum(c2) FROM a GROUP BY c_1 WHERE c2 < 10
```

In the first naive approach the VMS translates every clause of the query into a materialized view. The materialized views are concatenated to build the final result of the query. The client can access the last view to fetch the results. To

4.2 Cost model

Cost model – Define node cost(storage) Define edge cost

[3]Two of these, the `\numberofauthors` and `\alignauthor` commands, you have already used; another, `\balancecolumns`, will be used in your very last run of LaTeX to ensure balanced column heights on the last page.

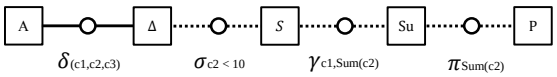

Figure 4: Maintenance plan

4.3 Optimization

Reorder operation –
Algorithm reordering
Merge operation –
Algorithm reordering

The *proceedings* are the records of a conference. ACM, as well as PVLDB, seeks to give these conference by-products a uniform, high-quality appearance. To do this, ACM / PVLDB has some rigid requirements for the format of the proceedings documents: there is a specified format (balanced double columns), a specified set of fonts (Arial or Helvetica and Times Roman) in certain specified sizes (for instance, 9 point for body copy), a specified live area (18 × 23.5 cm [7" × 9.25"]) centered on the page, specified size of margins (2.54cm [1"] top and bottom and 1.9cm [.75"] left and right; specified column width (8.45cm [3.33"]) and gutter size (.083cm [.33"]).

4.4 SQL pattern

The *proceedings* are the records of a conference. ACM, as well as PVLDB, seeks to give these conference by-products a uniform, high-quality appearance. To do this, ACM / PVLDB has some rigid requirements for the format of the proceedings documents: there is a specified format (balanced double columns), a specified set of fonts (Arial or Helvetica and Times Roman) in certain specified sizes (for instance, 9 point for body copy), a specified live area (18 × 23.5 cm [7" × 9.25"]) centered on the page, specified size of margins (2.54cm [1"] top and bottom and 1.9cm [.75"] left and right; specified column width (8.45cm [3.33"]) and gutter size (.083cm [.33"]).

4.5 Mapping

Clauses Evaluation order Mapping clauses -¿ view types

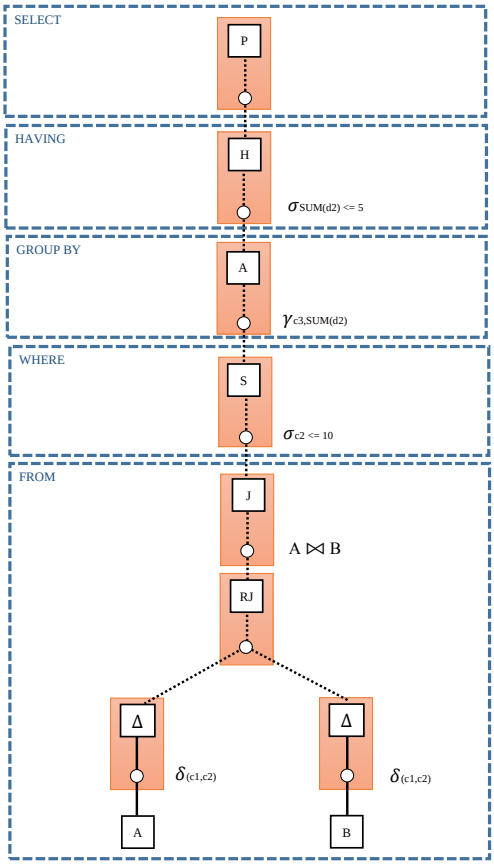

Figure 5: SQL pattern

4.6 Optimization

Reorder Merge

The good news is, with only a handful of manual settings[4], the LaTeX document class file handles all of this for you.

The remainder of this document is concerned with showing, in the context of an "actual" document, the LaTeX commands specifically available for denoting the structure of a proceedings paper, rather than with giving rigorous descriptions or explanations of such commands.

5. EVALUATION

5.1 Set-up

5.2 Work load

5.3 Experiments

view types simple maintenance plan

The *proceedings* are the records of a conference. ACM, as well as PVLDB, seeks to give these conference by-products a uniform, high-quality appearance. To do this, ACM / PVLDB has some rigid requirements for the format of the proceedings documents: there is a specified format (balanced double columns), a specified set of fonts (Arial or Helvetica and Times Roman) in certain specified sizes (for instance, 9 point for body copy), a specified live area (18 × 23.5 cm [7" × 9.25"]) centered on the page, specified size of margins (2.54cm [1"] top and bottom and 1.9cm [.75"] left and right; specified column width (8.45cm [3.33"]) and gutter size (.083cm [.33"]).

The good news is, with only a handful of manual settings[5], the LaTeX document class file handles all of this for you.

The remainder of this document is concerned with showing, in the context of an "actual" document, the LaTeX commands specifically available for denoting the structure of a proceedings paper, rather than with giving rigorous descriptions or explanations of such commands.

6. RELATED WORK

The *proceedings* are the records of a conference. ACM, as well as PVLDB, seeks to give these conference by-products a uniform, high-quality appearance. To do this, ACM / PVLDB has some rigid requirements for the format of the proceedings documents: there is a specified format (balanced double columns), a specified set of fonts (Arial or Helvetica and Times Roman) in certain specified sizes (for instance, 9 point for body copy), a specified live area (18 × 23.5 cm [7" × 9.25"]) centered on the page, specified size of margins (2.54cm [1"] top and bottom and 1.9cm [.75"] left and right; specified column width (8.45cm [3.33"]) and gutter size (.083cm [.33"]).

The good news is, with only a handful of manual settings[6], the LaTeX document class file handles all of this for you.

The remainder of this document is concerned with showing, in the context of an "actual" document, the LaTeX commands specifically available for denoting the structure of a proceedings paper, rather than with giving rigorous descriptions or explanations of such commands.

6.1 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references).

APPENDIX

You can use an appendix for optional proofs or details of your evaluation which are not absolutely necessary to the core understanding of your paper.

A. FINAL THOUGHTS ON GOOD LAYOUT

Please use readable font sizes in the figures and graphs. Avoid tempering with the correct border values, and the spacing (and format) of both text and captions of the PVLDB format (e.g. captions are bold).

[4]Two of these, the `\numberofauthors` and `\alignauthor` commands, you have already used; another, `\balancecolumns`, will be used in your very last run of LaTeX to ensure balanced column heights on the last page.

[5]Two of these, the `\numberofauthors` and `\alignauthor` commands, you have already used; another, `\balancecolumns`, will be used in your very last run of LaTeX to ensure balanced column heights on the last page.

[6]Two of these, the `\numberofauthors` and `\alignauthor` commands, you have already used; another, `\balancecolumns`, will be used in your very last run of LaTeX to ensure balanced column heights on the last page.

At the end, please check for an overall pleasant layout, e.g. by ensuring a readable and logical positioning of any floating figures and tables. Please also check for any line overflows, which are only allowed in extraordinary circumstances (such as wide formulas or URLs where a line wrap would be counterintuitive).

Use the `balance` package together with a `\balance` command at the end of your document to ensure that the last page has balanced (i.e. same length) columns.