Technical Report: Scalable Support for Incremental View Maintenance in Key-Value Stores

Jan Adler  Martin Jergler  Kaiwen Zhang  Arno Jacobsen
TU München
{adler, jergler, zhangk, jacobsen}@in.tum.de

ABSTRACT
Distributed key-value stores have become the solution of choice for warehousing large volumes of data. However, their architecture is not suitable for real-time analytics, as batch processing is a time-intensive task. To achieve the required velocity, materialized views can be used to provide summarized data for faster access. The main challenge is, the incremental, consistent maintenance of views at large scale. Thus, we introduce our View Maintenance System (VMS) to maintain SQL-like queries in a data-intensive real-time scenario. VMS can be scaled independently and at the same time provides guarantees for consistency, even under high update loads. We evaluate our full-fledged implementation of VMS on top of Apache’s HBase using a synthetic as well as a TPC-H workload. Exploiting parallel maintenance, VMS manages thousands of views in parallel, achieves up to 1M view updates per second and provides <5 ms access to view data.

1. INTRODUCTION
The web applications of major Internet players are backed by large-scale distributed computing frameworks, such as Hadoop or Spark. This approach works to the extent of an off-line batch analysis, but it does not provide the velocity required for real-time analytics of incrementally changing data [8]. To speed up the processing, there is a need to store aggregated data for faster access by the processing engine.

To this end, point solutions appeared raising the level of abstraction of a KV-store by either partially or fully materializing the desired application-level queries as views through the underlying store [9–11]. In this context, secondary indices have been added to KV-stores [9, 12], caching of application queries has been introduced [13], and massive processing of selection views (e.g., for following news feeds) has been enabled [10]. However, a generic solution that introduces view management for a wide variety of SQL query operators as views to KV-stores is still non-existent.

In this paper, we propose a View Maintenance System (VMS) to address the aforementioned challenges. As opposed to existing solutions, our design abstracts from a specific KV-store architecture and aims to support a broad spectrum of systems. VMS is based only on a few key features that KV-stores need to support. This concise set of base features facilitates the integration of view maintenance across different and heterogeneous KV-stores. We describe these features in detail in Section 2.

VMS provides mechanisms for the consistent materialization and incremental maintenance of views to KV-stores resulting in Select-Project-Join (SPJ) and aggregation querying for real-time applications. VMS operates as a separate module, which can be scaled independently of the underlying KV-store. VMS consumes streams of client base table operations and produces updates to view data records (see Figure 1). Views are, therefore, maintained incrementally: base table operations propagate through the system to affect only the derived view data records. Materialized views are standard tables stored by the KV-store and all properties such as concurrent access, availability, and fault-tolerance, apply to views as well.

To the best of our knowledge, our proposed solution, VMS, is the first system to provide incremental maintenance of materialized views in KV-store. The closest existing solution, Apache Phoenix, does not incrementally maintain SQL query results up to date, but rather generates entirely fresh results by executing base table scans periodically [11]. We argue that incremental maintenance is more scalable and provides better read latencies on the views, as shown in our baseline comparison results (see Section 6).
paper makes the following contributions:

1. We provide a thorough analysis of popular KV-stores to identify a small set of features forming the basis of a generic VMS model for correct view materialization (cf. Section 2).

2. We show how strong consistency can be achieved in a highly parallelizable view maintenance system to support Select-Project-Join (SPJ) semantics and aggregation.

3. We present the design of VMS as an external component, leveraging these common KV-store features to achieve horizontal scalability while maintaining strong consistency (cf. Section 4). In particular, VMS uses a novel group-based key hashing technique to distribute updates.

4. We introduce the concept of Pre-Processing Views to modularize and speed up the computation of similar aggregation and join views (cf. Section 5).

5. We validate VMS by extending HBase. Using a synthetic and a TPC-H workload, we show that VMS, compared to known baseline methods, can be scaled with linear performance gain and provides fast access to view data (cf. Section 6).

2. KV-STORE MODEL

In this section, we discuss KV-store internals that serve us in the remainder of the paper. We provide a general model which represents existing KV-stores such as [2–6]. We only consider highly distributed stores that have emerged over the last decade and disregard centralized KV-stores. Our objective is to distill a set of features our VMS requires from a KV-store.

The upper part of Figure 2 shows a general model for such KV-stores (the lower part of the figure, i.e., VMS, is explained in Section 4). Some designs explicitly designate a master node, e.g., HBase [5] or Bigtable [2], while others operate without explicit master, e.g., Cassandra [6], where a leader is elected to perform management tasks, or PNUTS [4], where mastership varies on a per-record basis. In all cases, a KV-store node represents the unit of scalability: KV-store nodes persist the data stored in the system. The number of nodes can vary to accommodate load change. In contrast to a centralized SQL-based DBMS, a node manages only part of the overall data (and a part of the request load).

KV-stores frequently employ a distributed lock-service (not shown in the figure), such as Chubby (Bigtable) or ZooKeeper (HBase and Cassandra), for coordination purposes (leader node election, centralized configuration, and root node storage).

A file system builds the persistence layer of a node in a KV-store. For example, HBase stores files in the Hadoop distributed file system (HDFS). Cassandra and PNUTS resort to node-local file systems for storage and do not rely on a distributed file system. In the file-system all KV-store relevant data is persisted and replicated, transaction logs as well as actual table files. Whereas, HBase relies on HDFS for redundancy of data, Cassandra relies on its own replication mechanism to keep data highly available in face of failures. If a node crashes, replicas serve to retrieve and restore data.

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 can hold 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 table files.

Key ranges serve to partition a table into multiple parts that can be distributed 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, 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.

Read/write path  - The KV-store API supports three client-side operations: put, which inserts a record, get, which retrieves a record, and delete, which removes a record.

In the read/write path, when reading or updating a table record, requests pass through as few nodes as possible to reduce access latency. Also, a KV-store aims to distribute client requests among all system nodes to spread load. For example, HBase routes client requests from a root node down to the serving KV-store node in a hierarchical manner. Cassandra, on the other hand, lets clients connect to an arbitrary node which forwards the request to the serving node. In either case, the client ends up at one particular KV-store node that is serving the key range the client wants to access.

Every node maintains a transaction log (TL), referred to as write-ahead log in HBase and commit log in Cassandra. When a client operation is received, it is first written into the TL of the KV-store node (cf. Figure 2). From then on, the operation is durably persisted. The purpose of TL, in either case, is not the long-term storage of data (this is managed by the table file, cf. Figure 2). TL serves as a first processing instance of the KV-store node to be able to quickly save and complete client operations. Upon node crash, they are recovered from the TL, which contains the operation sequence a node saw over time. During recovery, this sequence is replayed from the log.

Subsequently, the operation is inserted into a memstore. Memstores are volatile but they provide low latency access; this is needed to pre-sort the operations into a tree-like structure (cf. log-structured merge tree). Once a memstore exceeds a set capacity, it is flushed to disk. Continuous flushes produce a set of table files, which are periodically merged by a compaction process.

In our model, we rely on a set of ACID guarantees provided by the KV-store. All single-row writes are atomic with isolation. This guarantees single-row reads to always return an entire row which is consistent with the write history. This is a common model provided by popular KV-store such as HBase and Cassandra.

Extension points  - When a client updates a base table (e.g., put, delete) in KV-store, consequently, all derived view tables become stale. Thus, we design VMS to react to all KV-store client operations (on base tables) and update the affected view tables accordingly. At the same time, our goal is to not interfere with

---

1There are sometimes additional methods, e.g., a range scan or passing a selection predicate to the store. However, none of them extend beyond repeated single row access; none of them offer expressive semantics.
store-internal read/write paths for data processing. In this spirit, we determined a number of common extension points that can be used to stream all incoming client operations of KV-store.

We identified three realizations to stream client operations from the KV-store to VMS: (1) Access the store’s API and retrieve the current version, (2) intercept operations at the KV-store node (e.g., via update hooks, in HBase called coprocessors), (3) monitoring the TL from the KV-store node asynchronously.

Realization 1 may lead to inconsistent view states, as base data can change during retrieval of a previous update; none of the popular KV-stores offers snapshot isolation. In addition, this method incurs significant overhead (e.g., an update would trigger a read and one or more write to update derived views.)

Realization 2 could be used for incremental maintenance in a synchronous mode by directly inserting the view updates in the operation path (e.g., with post-operation coprocessor execution). This technique preserves freshness of the views but is only suitable if the number of maintained views is small, otherwise KV-store operations would be needlessly delayed. Another possibility is to use coprocessors for asynchronous maintenance, logging operations in another queue for further processing.

Realization 3 has several benefits: Reading TL is asynchronous and decouples processing. It neither interferes with update processing, i.e., no latency is added into the update path, nor imposes additional load. Moreover, maintaining numerous views at once means that every base table operation generates multiple view updates. Using TL, we decouple view update from client operation. Operations in TL are durably persisted and can be recovered by VMS. The TL contains operations, not table rows, which are suitable for incremental maintenance.

In VMS, we opted for Realization 3, which provides us the ability to scale the view maintenance system independently of the KV-store while providing the required level of consistency. Realization 2, when used in asynchronous mode, is also similar to our approach. However, we argue that Realization 3 re-uses a commonly used and reliable component of popular key-value stores, while Realization 2 would require building a redundant logging infrastructure.

**Data model** – As base and view table are all treated as standard tables in the KV-store, we present a notation for tables and records. We formalize the data model of a KV-store as a map of key-value pairs \( \{ (k_1,v_1), ..., (k_n,v_n) \} \) described by a function \( f: K \rightarrow V \). Systems like Bigtable, HBase and Cassandra established data models that are multi-dimensional maps and store a row together with a variable number of columns per row. For example, the 2-dimensional case creates a structure described by \( f: (K,C) \rightarrow V \) where \( K \) is the row-key and \( C \) is the column-key. This notation closely resembles a database row and is used throughout the remainder of this paper. Further, we define: (a) \( T = (K,F) \), a table (with \( K \) being the row key and \( F \) being a set of column-value pairs) \( b \), \( r = (k, (c_1,v_1), (c_2,v_2)) \), a record (c) \( p = \text{put}(k, (c_1,v_1), (c_2,v_2)) \), a put operation (d) \( d = \text{del}(k) \), a delete operation and (e) \( g = \text{get}(k) \), a get operation.

3. **CONSISTENCY MODEL**

In this section, we first review the notation used throughout the paper and the definition of incremental view update in the context of KV-stores. We also provide view data consistency models, based on prior works. We then propose a theorem which identifies three consistent view update hooks, in HBase called coprocessors, (3) monitoring the TL from the KV-store node asynchronously.

Realization 1 may lead to inconsistent view states, as base data can change during retrieval of a previous update; none of the popular KV-stores offers snapshot isolation. In addition, this method incurs significant overhead (e.g., an update would trigger a read and one or more write to update derived views.)

Realization 2 could be used for incremental maintenance in a synchronous mode by directly inserting the view updates in the operation path (e.g., with post-operation coprocessor execution). This technique preserves freshness of the views but is only suitable if the number of maintained views is small, otherwise KV-store operations would be needlessly delayed. Another possibility is to use coprocessors for asynchronous maintenance, logging operations in another queue for further processing.

Realization 3 has several benefits: Reading TL is asynchronous and decouples processing. It neither interferes with update processing, i.e., no latency is added into the update path, nor imposes additional load. Moreover, maintaining numerous views at once means that every base table operation generates multiple view updates. Using TL, we decouple view update from client operation. Operations in TL are durably persisted and can be recovered by VMS. The TL contains operations, not table rows, which are suitable for incremental maintenance.

In VMS, we opted for Realization 3, which provides us the ability to scale the view maintenance system independently of the KV-store while providing the required level of consistency. Realization 2, when used in asynchronous mode, is also similar to our approach. However, we argue that Realization 3 re-uses a commonly used and reliable component of popular key-value stores, while Realization 2 would require building a redundant logging infrastructure.

**Data model** – As base and view table are all treated as standard tables in the KV-store, we present a notation for tables and records. We formalize the data model of a KV-store as a map of key-value pairs \( \{ (k_1,v_1), ..., (k_n,v_n) \} \) described by a function \( f: K \rightarrow V \). Systems like Bigtable, HBase and Cassandra established data models that are multi-dimensional maps and store a row together with a variable number of columns per row. For example, the 2-dimensional case creates a structure described by \( f: (K,C) \rightarrow V \) where \( K \) is the row-key and \( C \) is the column-key. This notation closely resembles a database row and is used throughout the remainder of this paper. Further, we define: (a) \( T = (K,F) \), a table (with \( K \) being the row key and \( F \) being a set of column-value pairs) \( b \), \( r = (k, (c_1,v_1), (c_2,v_2)) \), a record (c) \( p = \text{put}(k, (c_1,v_1), (c_2,v_2)) \), a put operation (d) \( d = \text{del}(k) \), a delete operation and (e) \( g = \text{get}(k) \), a get operation.

As operations are performed on base tables, their states change: we depict the sequence of states with indices \( BS_0, BS_1, ..., BS_f \), where \( BS_0 \) is the initial state, \( BS_i \) is an intermediate state and \( BS_f \) is the final state. In a KV-store, every put or delete operation causes a single record to be modified and change its version (the sequence of versions of a record is called a record’s timeline) and, thus, affects the state of the corresponding base table (and \( BS_0 \) that contains it). Two states can be compared by the \( \leq \) operator. \( BS_i \leq BS_j \) means that the versions of all records in base tables \( BS_i \) are equal or newer than the versions of records in \( BS_j \).

If two states cannot be compared, which happens due to the concurrent execution of operations on different row-keys, their relationship is expressed by the \( || \) operator.

We define an incremental view update for a view \( V = \text{View}(BS) \) as follows. Given as input an operation \( t \) which involves a record in base table \( B \), where \( B \in BS \), a view update reads the current state \( V_t \) of the view table, processes the effect of a base table operation \( t \) according to the semantics of the view, and generates the state \( V_{t+1} \) for the view table. Note that each view update can therefore consist of several reads and writes or none at all, depending on the base table operation processed and the current state of the view. For instance, a view update for a projection view does not require any read on the view table since each base table operation completely determines the value to write. A view update for a selection view could also not produce any write if the base table operation processed does not satisfy the selection condition.

**View consistency** – A view data consistency model \(^2\) validates the correctness of a view table. Further, the model evaluates a view’s ability to follow a sequence of base table states and produce a corresponding sequence of valid view table states. The model as well as the different levels of consistency that we apply in our paper have been established and widely accepted in view research [14–18]. Depending on view types, view maintenance strategies, and view update programs, either none, or some, or all of the levels are attainable.

Once a base table changes, the view table – or rather the system that maintains the view – needs to react and incorporate the changes into the view. The accuracy of this maintenance is defined through the following levels:

**Convergence:** A view table converges, if after the system is idle for some time, the last view state \( V_f \) is computed correctly. This means it corresponds to the view expression over the final base state \( V_f = \text{View}(BS_f) \).

**View convergence** is a minimal requirement, as an incorrectly calculated view is of no use.

**Weak consistency:** Weak consistency is given if the view converges and all intermediate view states are valid, meaning that there exists a valid operation sequence such that every intermediate view state can be derived from a base table state. Weak consistency is violated, for instance, if non-idempotent view updates are applied, repeatedly.

**Strong consistency:** Weak consistency is achieved and the following condition is true. All pairs of view states \( V_i \) and \( V_j \) that are in a relation \( V_i \leq V_j \) are derived from base states \( BS_i \) and \( BS_j \) that are also in a relation \( BS_i \leq BS_j \). Strong consistency is violated, for instance, if concurrent processing of base updates leads to a different sequence of view updates.

**Complete consistency:** Strong consistency is achieved and every base state \( BS_i \) of a valid base state sequence is reflected in a view state \( V_i \). Valid base state sequence means \( BS_0 \leq BS_i \leq BS_{i+1} \leq BS_f \).

\(^2\) Our experiments confirmed that the penalty of reading from the file system are far smaller than intercepting events.

\(^3\) Not to be confused with consistency models for transaction processing (i.e., system-centric and client-centric models)
Example 1: Imagine a base table $B = (K,F)$ and a view table $V = \gamma_{c_1,\text{sum}(c_2)}(B)$ (see Figure 3). This view groups records by their value of $c_1$ and sums the values of $c_2$ for each group. The initial state of the base table is $B_0 = \{(k_1, (c_1, x_1),(c_2,15))\}$ and the corresponding state of the view table is $V_0 = \{(x_1,(c_{\text{sum}},15))\}$.

Now, the following update operations are applied to the base table:

1. $\text{put}(k_1, (c_1, x_1),(c_2,20))$
2. $\text{put}(k_2, (c_1, x_1),(c_2,10))$
3. $\text{del}(k_1)$

Since KV-store provide a consistent per-record ordering, but not for operations across records, several valid operation sequences exist for this example: (1),(2),(3), (1),(3),(2), or (2),(1),(3). Applying these operations in a different order generates different view states, as illustrated in the example.

To achieve convergence, the maintenance has to compute the final view state as: $V_f = \{(x_1,(c_{\text{sum}},10))\}$, which corresponds to the final base table state $B_f = \{(k_2, (c_1, x_1),(c_2,10))\}$. To achieve weak consistency, any intermediate views generated must belong to the same valid operation sequence. For instance, the view state sequence $V_0, V_1, V_2, V_V$ is valid since all view states belong to the same sequence, whereas $V_0, V_2, V_3, V_f$ is not because $V_2$ and $V_4$ belong to different operation sequences.

To achieve strong consistency, the intermediate states must be correctly ordered to match a single valid sequence. For instance, $V_0, V_1, V_2, V_f$ achieves strong consistency, but $V_0, V_2, V_3, V_f$ does not. The last level, complete consistency, can be achieved, representing all intermediate base states in the view. $V_0, V_1, V_2, V_f$ achieves complete consistency, but $V_0, V_2, V_3$ does not.

In the remainder of the paper, VMS is designed to provide strong consistency. We argue that convergence on its own is insufficient due to the online nature of our system. Since VMS is designed to incrementally maintain views, the targeted applications must be able to read correct intermediate states. Further, weak consistency is inadequate since a client can perform successive reads on the same view. If the view states are not correctly ordered, the client may enter an inconsistent state (e.g., having to roll-back on a previous state). On the other hand, complete consistency is too costly and can only be detected if the views repeatedly read from the base tables. Our views are expressive enough to satisfy the queries of our applications such that direct reads to the base tables are no longer required.

In order to maintain strong consistency, we identify a set of three properties which must be provided by the view maintenance system. The complete proof of Theorem 1 can be found in Appendix A.

**Theorem 1.** A view maintenance system which provides the following properties also guarantees that views are maintained strongly consistent.

1. View updates are applied exactly once
2. View updates are processed in isolation
3. (Base-)record timeline is always preserved

We now provide a brief explanation of Theorem 1. If we employ Property 1 of the theorem, we ensure that all update operations are delivered and applied exactly once. However, Property 1 alone does not guarantee convergence of the view. When using parallel execution, e.g., multiple update operations might be applied to the same view record concurrently and affect its correctness. We illustrate this observation in Example 2.

**Example 2:** We reconsider the table of Example 1 and its update operations (1) - (3). When two update programs process update operations (1) and (2) in parallel, the following can happen: Update program 1 and update program 2 retrieve the initial view record $(x_1,(c_{\text{sum}},15))$ simultaneously from the view. Update program 1 applies its incremental update, leading to a new view record $(x_1,(c_{\text{sum}},20))$ and writes it back to the view; update program 2 also applies its incremental update to the initial value, leading to $(x_1,(c_{\text{sum}},25))$ and writes back to the view. The result is an incorrect value, as both incremental updates should be applied subsequently, leading to a view record $(x_1,(c_{\text{sum}},30))$.

Property 2, would have avoided the wrong execution in Example 2, since read isolation would prevent the lost update. If Property 1 and Property 2 of the theorem are applied, convergence is still not guaranteed. Asynchronous processing and high parallelism can lead to reorderings of update operations.

**Example 3:** Again, we consider table set-up of Example 1 and operations (1) - (3). Operation (3) and operation (1) affect the same base table key $k_1$; if their order is changed, meaning (3) is processed before (1), the end result is incorrect. Exchanging the order of a put and a delete operation, causes the value of this particular record to be present in the view computation, where it shouldn’t.

Therefore, we also apply Property 3 and enforce the preservation of a record’s timeline. All three properties together guarantee that convergence, weak, and even strong consistency (correct ordering is established) can be achieved. By complying to the requirements of the theorem, we show that our approach can achieve strong consistency for the views it maintains.

### 4. VIEW MAINTENANCE SYSTEM

In this section, we present and discuss the design of VMS, its components and the path of operation processing. The design relies on a group-based key hashing technique and maintains the three consistency properties of Theorem 1 while providing scalability.

#### 4.1. Design overview

The lower half of Figure 2 gives an overview of VMS, which is comprised of a number of n distributors and a number of m view servers (VSS). The input to VMS is a set of operation streams $(o_{S1}, o_{S2}, o_{S3},...o_{SN})$ each generated by the KV-store clients and emitted by a KV-store node. Every KV-store node is connected to its own distributor. The distributor notifies base table operations processed by its attached KV-store node by reading the correspondingTL. Since new operations are always appended, the TL can be read sequentially for fast access. Upon retrieval, the distributor distributes the operation stream to all registered VSS. The number m of VSS is configurable (and independent of n); VSS can be dynamically assigned to or removed from the VMS.

A VSS is designed to be lightweight and can be elastically deployed to accommodate a changing workload. It computes view updates, based on base table operations received as input, and performs operations on the appropriate view tables, which are stored in the KV-store, via the KV-store API. VSSs are kept stateless to be exchangeable at any time and to minimize dependency. Given a number of view definitions and a sequence of operations,
a VS is always able to execute any view table update from any host. Our design provides the following four benefits:

1. Seamless scalability: Multiple views may have to be updated as a consequence of a single base table operation. As VMS exceeds its service levels, additional VSs can be spawned.

2. Operational flexibility: VSs introduce flexibility to the system architecture. All VSs can be hosted together on the same physical node or on different nodes.

3. Elasticity: VSs can be reassigned as base table operation load changes.

4. Fault-tolerance: If a VS crashes, another VS can take over and continue processing the operation stream.

4.2. Update distribution

A distributor reads the stream of arriving base operations, resolves the required view updates and distributes the updates to a number of VSs. Figure 4 depicts the whole working process of a distributor.

On top of the distributor, the reader component can be found. The reader accesses the KV-store node’s file system (e.g., HDFS) and reads incoming operations in sequence from the TL. The reader then forwards the operations to the resolver component. The resolver knows all queries that are defined over base tables. (In our implementation we simply store the query list in a KV-store table.) Using the list, the resolver evaluates the matching queries for a base operation. Each query consists of a chain of one or more (intermediate or final) view tables. The resolver determines the view tables that need to be updated in consequence of the base operation. It creates a separate view update for every view table and forwards them to the assigner component.

The assigner is responsible for selecting a VS that will apply the view update. Before the assigner starts its work, it will create evenly sized groups (cf. Figure 4) of VSs. Each group of VSs is working on one of the view tables. In the figure, Group 1 maintains View Table 1 and Group 2 maintains View Table 2.

The hashes of all active VSs are contained in a global hash-ring (which is synchronized over the distributed lock service, cf. Figure 2). The assigner is capable of selecting a specific group on the hash-ring, while ignoring the remaining VSs. In Figure 4, the grey dots on the circle are hashes of VS 1, 2, 3, and belong to Group 1; the white dots on the circle are hashes of VS 4, 5, 6 and belong to Group 2.

To assign an arriving view update, the assigner determines the responsible group. Distribution of operations within a group is then performed using consistent hashing [19]. The assigner hashes the view update’s row-key and associates the hash value in clockwise direction to one of the VSs in the group. In this manner, a distributor assigns operations uniformly across the available groups and VSs. In a final step, the distributor sends the updates to the assigned VSs and completes the distribution process.

The hash-ring is synchronized across all distributors. Thus, the same VS will be picked to treat the same update everywhere in the system. This is relevant when regions (of base tables) move from one KV-store node to another and consistency still has to be guaranteed.

For simplicity, all VSs could be arranged in one large group. A VS is capable of performing every view update on every view table. However, limiting the size of each group reduces the number of view tables a VS has to access, which improves the performance of the KV-store. A VS benefits greatly from repeated accesses to the same table (cf. Section 6), since region lookups are cached and updates buffered to be applied in batches. However, when the number of view tables surpasses the number of VSs, it becomes necessary for a group to process updates on multiple view tables. Using groups (to manage view tables) in combination with consistent hashing (to manage key-ranges) has proven to be the best combination for our case. This mechanism ensures maximal degree of concurrency for distributing updates and best leverages the performance of the underlying KV-store. Further, it guarantees the ordered propagation of base table operations to view tables, setting the basis for view table consistency.

Instead of using consistent hashing, VSs could be simply assigned a fixed key-range within a view table. Here is a number of reasons, why we picked consistent hashing over this method: (1) Key-regions can change, new keys can be added to the system. (2) Providing elasticity, VSs have to be efficiently inserted and removed from the system (without relocating key-ranges). (3) A more balanced load distribution achieves higher throughput in the system.

Consistent hashing can manage each of these cases without further adaptations in the implementation. It supports changes of resources during the assignment process by nature. A balanced load distribution can be easily achieved by using virtual nodes (multiple representations of the same VS on hash-ring).

In general, there are two methods for hashing an operation: Method 1: The assigner component reads the operation and uses the hash of the base table record to determine the VS (row-key k, cf. Figure 4).

Method 2: The assigner component computes the view table row-key from the operation data and uses the hash of the view table row-key to determine the VS (row-key x, cf. Figure 4).

If the base table and view table have the same row-key (e.g., a selection view), both methods lead to the same result. But if row-keys are different (e.g., an aggregation or join view), both methods differ. The trade-off between both approaches is discussed in Section 4.3. Once the operation has been assigned to the hash-ring, it is inserted into a queue and sent to the appropriate VS.

Every VS maintains its own transaction log, referred to as VS-log. When receiving an operation, a VS directly writes it to the VS-log. Similar to the transaction log, the VS-log is kept available by the underlying file system, employing recovery mechanisms in face of VS crashes (e.g., in the case of HBase, the file system redundantly replicates file blocks via HDFS.)

To access and update view tables, a VS acts as a client to the KV-store, using its standard client API. Given a base table operation (e.g., a put on a base table A), the VS retrieves and caches the view definitions of the derived views (e.g., a selection and count view S and C, both derived from A). Then, VS computes
and submits the resulting view table updates (to S and C) via
the client API. For some of the view types maintained, the VS
has to first query the view table as part of the update logic. For
example, in a count view, the VS reads the current count from
the view before applying the delta of the base table operation
(e.g., incrementing or decrementing the count).

4.3. Achieving consistency

We show how VMS achieves a high degree of concurrency while
providing consistency via the three properties of Theorem 1 found
in Section 3. Throughout this section, we will discuss the advan-
tages and disadvantages of both hashing methods (described in
Section 4.2) and their impact on the design.

4.3.1. Exactly-once property

Property 1, updating a view exactly once for every client operation
is critical, as views can be non-idempotent. There are two possible
scenarios that violate the exactly-once requirement: an operation
is lost (due to node crash, or transmission errors); an operation
is applied more than once (due to a log replay after a node crash).
In either case, the view is incorrect (i.e., does not converge).

One architectural decision of VMS is to read the operations from
the TL of the KV-store node. Operations in the TL are persisted
safely and replicated by the file system. When operations are lost
due to a node or VS failure, they can always be recovered from
the TL of the KV-store node. This follows the normal procedure for
recovering failures in the KV-store itself. As soon as a node crash
is detected, the KV-store replays all transactions that have not been
flushed to the table file before and would have been lost otherwise.

As operations can not be lost, the VMS guarantees at-least-once
semantic. Still, we need to ensure that operations in VMS are
not duplicated. When a crash occurs, the reader may re-process
some of the operations and send them to the VSs a second time. Thus,
the VS needs to identify duplicates and drop them. We achieve the
identification of duplicates with the help of a global ID (i.e., signature); this ID is obtained from the KV-store. A global
ID is created by combining the operation sequence number and
the KV-store node ID, and is attached to every operation. Each
VS keeps track of the highest maintained operation ID of each
node, separately (this information is stored into ZooKeeper for
fault-tolerance). If an operation with a lower ID is sent to the
VS, it identifies the duplicate and drops it.

4.3.2. Isolated view update

According to Property 2, every view update has to be executed
with isolation. We rely on the semantics that are provided by
the KV-store to achieve this. We assume (as it is the case for
HBase and Cassandra) that for a single put or delete operation,
e.g., put(k1, (c1,x1), (c2,10)), the execution is atomic and isolated
at the record level.

In addition to single row operation isolation, the complete
update process, which can also include a get on the old view
record followed by a put/delete operation to the new view record,
has to be also isolated. Some view types define a mapping from
multiple base table records to a single view table record (e.g.,
aggregation). Different base table records may be propagated
to different VSs. Multiple VSs can concurrently update the same
view record. Depending on the hashing method employed (from
Section 4.2), this problem has to be solved differently:
Method 1: Distributing the operation according to the base
view table row-key means, there could be multiple VSs, updating
one view table row-key. For example, put operations put(k1, (c1,x1),
(c2,10)) and put(k2, (c1,x1), (c2,5)) could be forwarded to different
VSs, yet, both VSs have to access the same row-key x1. To solve
the problem, we use test-and-set methods to avoid interruption
during view updates. The VS retrieves the old view record, e.g.,
x1 = (c1+sum(50)) and extracts one of the values (here, it is the
aggregation value 20). When updating and writing back the new
value, e.g., 25, the VS sends a test-and-set request to the server
with a test-value (here, 20).
Method 2: Distributing the operations according to the hash of
the view table row-key completely eliminates the need of further
synchronization mechanisms. Every VS is responsible for an equal
amount of view table records and, thus, for a part of the view
table. As there is no ambiguity, VSs can just use regular get and
put/delete operations.

4.3.3. Record timeline

Record timeline (Property 3) means that sequences of opera-
tions on the same row-key are not re-ordered when processed by
VMS. The two methods outlined above have different properties
concerning record timeline.
Method 1: By distributing the operations according to the base
table row-key, the timeline of a record cannot be broken. All
operations that touch a specific row-key will always be directed
to the same VS; they will be retrieved in order, sent to the VS
in order, and updated in the view in order.
Method 2: Distributing the operations according to the hash
of the view table row-key also creates a record timeline. But in
contrast to Method 1, it is the timeline of the view table, which
bears the following consequences: As long as the view table record
doesn’t change, operations to the same base table row-key are
also forwarded to the same VS. As soon as an update modifies
the view table row-key, which means the update touches two records
in one view table, the timeline of the base table row-key could
be broken. For example, put(k1, (c1,x2), (c2,5)) on a record which
previously had the value (k2, (c1,x1), (c2,5)) causes an aggregation
view V = γ(c1, sum(c2)) to subtract 5 from the view row-key x1, and
add 5 to x2.

For that reason, we need a mechanism to preserve record time-
line in this view table row-key change scenario. We employ a buffer
which stores operations where the view table key is modified: in-
sert and delete operations, as well as update operations that do not
change the key are processed as normal. During a key change, the
operation is inserted into the buffer and first sent to the VS manag-
ing the old key (in our previous example, the VS of x1). The
operation is then deleted from the buffer and passed to the VS which
is in charge of the new key (e.g., the VS of x2). During that process,
any new operations on the affected base table row-key are also
buffered and will be processed after the key change is completed.

4.3.4. Comparison of methods

The key-hashing methods described in Section 4.2 have different
characteristics. Method 1 preserves the base record timeline na-	ively, whereas Method 2 supports concurrent access to the view
table. In both cases, additional mechanisms are required in order
to satisfy our strong consistency model: Method 1 requires the
use of test-and-set methods, and Method 2 requires an operation
buffer for updates with view row-key changes.

For our implementation, we adopt Method 2 over Method 1 to
avoid the use of locking mechanisms. Even though test-and-set is
an optimistic locking mechanism (i.e., view records are always ac-
cessible), it is not suitable for our case. If there is high contention,
a large volume of view updates have to be repeatedly applied. In
addition, test-and-set is required for every view update, regardless
of the type (insert, update, or delete). In contrast, the operation
buffer used by Method 2 is only needed for a small fraction of
update operations and does not impact insert or delete operations
at all.

5. VIEW MAINTENANCE CONCEPT

In this section, we develop techniques for maintaining different
view types in VMS. We first define a set of Pre-Processing View
types. Based on that, we describe a set of standard view types
(e.g., selection, projection and join) and explain how they make
of the Pre-Processing Views. Finally, we explain the composition of multiple view types to form a SPJ query with aggregation.

Figure 5 illustrates the maintained view types and the relation between them. On the left side of the figure, the base tables (i.e., A and B) can be found. If a client updates a row in one of the base tables, the update propagates along the dark grey arrows: first, VMS updates the intermediate Pre-Processing Views, and second, VMS updates the final views on the right side of the figure (those which are accessed by clients to fetch results).

5.1 Pre-processing views

Pre-Processing Views are internal to VMS in a sense that they are neither specified by clients nor exposed to them. Their primary intent is to enable or optimize correct maintenance of client-level view types. Pre-Processing Views introduce storage overhead, but they can serve as a basis for different view types defined in the system; e.g., a single relation of a multi-table-join can be reused in different join views (that embed the same relation). Logically, Pre-Processing Views represent the basic elements of view maintenance and their use amortizes as more complex views are managed by VMS. Pre-Processing Views also speed up view maintenance significantly as discussed in Section 6. We now describe the three key Pre-Processing View types we leverage in VMS.

Delta – A DELTA view is a Pre-Processing View that tracks base table changes between successive operations. TL entries only contain the client operation but 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 actual value which is to be deleted. Likewise, a put 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; hence, we capture it in a DELTA view. The DELTA view records base table entry changes, tracking the states between row updates, i.e., the delta between two successive operations for a given key. Derived views leverage this information for their maintenance operations. A delta view is defined using the \( \delta(T) \) operator. Per definition it includes all columns of base table \( T \).

Example 4: In Figure 5, table Delta A tracks all changes of base table A. The column names of both tables are equal. Now a client issues a put operation to base table A (the dark grey box): it changes the value of row \( k_1 \), column \( c_2 \) from 50 to 10. The change is propagated and represented in Delta A as \( 50 \rightarrow 10 \).

Pre-aggregation – The PRE-AGGREGATION is a Pre-Processing View that prepares for aggregation by sorting and grouping base table rows. Subsequently, aggregation views only need to apply their aggregation function to the pre-computed state. This majorly benefits applications that calculate different aggregations over the same aggregation key.

To materialize these aggregates without our pre-aggregation, VMS would have to fetch the same record multiple times. In addition, for min and max views, the deletion of the minimum (maximum) in the view would require an expensive base table scan to determine the new minimum (maximum), introducing consistency issues that result from the sought after value changing while a scan is in progress, shown by the analysis in [18]. This motivated us to introduce the PRE-AGGREGATION view. This view type sorts the base table records according to the aggregation key, storing the grouped rows in a map. Aggregation functions like count, sum, min, max or average, can then be applied to the map. Thus, aggregation results become available instantaneously.

A pre-aggregation is defined using the \( \rho_c(T) \) symbol, indicating that it pre aggregates records of table \( T \) using column \( c \) as key.

Example 5: In Figure 5, table Pre-Agg A serves as a pre-processing step for table Sum A and Min A. The put operation in table Delta A changed the value of \( c_1 \) from 50 to 10. The change propagates to the PRE-AGGREGATION view as follows: VMS evaluates the aggregation key (i.e., \( x_2 \)) and updates the row in the table Pre-Agg A, column \( c_A \) to \((k_1,10)\) (see grey box). In a next step VMS can easily compute the sum and the minimum of row \( x_2 \) and update the corresponding rows in tables Sum A and Min A.

Reverse join – A REVERSE-JOIN view is a Pre-Processing View that supports the efficient and correct materialization of join views in VMS. A join view is derived from at least two base tables. For an update to one of these tables, the VS needs to query the other base table to determine the matching join rows. A matching join row can only be determined quickly if the join-attribute is the row-key of the queried base table, or if an index is defined on the join-attribute for the table. Otherwise, a scan of the entire table is required, which reveals the following drawbacks: (i) Scans require a disproportional amount of time, slowing down view maintenance and with increasing table size the problem gets even worse. (ii) Scans keep 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. We take the join key \( (jx) \) for the two base tables as row-key of the REVERSE-JOIN view. When operations are propagated, the REVERSE-JOIN view can be accessed from either side of the relation with the help of the join key (always included in both tables’ operations). If a record is inserted into one of the underlying base tables, it is stored in the REVERSE-JOIN — whether or not it has a matching row in the other base table.
This technique enables inner, left, right, and full joins to derive from the REVERSE-JOIN view without the need for base table scans, as we show below. Further, we can parametrize, subsequent join views. The view operator is defined as $\rho_{C}(T_{1,..,T_{n}})$ with $C = \{c_{1},..,c_{m}\}$ indicating that it pre aggregates records from table $T_{1}$ to table $T_{n}$ using columns $c_{1}$ to $c_{m}$.

Conceptually, a REVERSE-JOIN view is constructed as a multi table PRE-AGGREGATION view. Each column of the REVERSE-JOIN view keeps a number of records from one of the base tables. If aggregation and join key are equal, both view types can be merged into one; e.g., PRE-AGGREGATION $\rho_{a}$ can be maintained together with REVERSE-JOIN $\rho_{c}$ if $c \in C$. Likewise, REVERSE-JOIN views can be merged if one of the view’s columns is a subset of the other view’s columns or even, if both REVERSE-JOIN views share a single column. E.g., $\rho_{c_{1}}$ can be maintained together with $\rho_{c_{2}}$ if $C_{1}\cap C_{2} \neq \emptyset$.

Example 6: In Figure 5, Reverse Join $A . B$ is a construction of view table $Pre$-Agg $A$ and view table $Pre$-Agg $B$. Because aggregation and join keys are equal, Reverse Join $A . B$ serves multiple purposes: it is a pre-processing step for aggregation tables $\sum A$, $\min A$, but also for Join $A . B$. Still, the reverse join view needs to be updated only once – which saves two of three of the processing steps for VMS. The update process executes as follows. The two columns $c_{A}$ and $c_{B}$ store records of the ingoing base tables $A$ and $B$, sorted by the join key. We reconsider the update of row-key $x_{2}$ to $(k_{3},10)$. To update the table Join $A . B$, VMS takes row $k_{3}$ and builds the cross product with entries of row $tab_{B}$, row $x_{2}$. This leads to an update of rows $k_{3}J_{2}, k_{3}J_{3}$ and $k_{3}J_{3}$.  

5.2. Standard views

We now describe how VMS maintains client-level views for a number of interesting standard view types. We also present different maintenance strategies, but defer a full-fledged analytical 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. A projection view selects a set of columns from a base table. Similar to the selection view, the VS uses the row key of the base table as row-key for the view table. To save storage and computation resources, we can combine DELTA, projection and selection into a single 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 optimizations with VMS, which we defer to future work.

Aggregation – The maintenance of count and sum views is similar, so we treat them together. In general, 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.

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 view. 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 PRE-Processing Views, a table scan would have to be performed [18]. This gave the motivation to derive the min (max) from a PRE-AGGREGATION preventing the need for a scan.

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 a client wants 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.

![Figure 6: Maintenance of composite view.](image)

**Join** – A join view constitutes the result of joining $n$ base tables. Since the matching of join partners is already accomplished by the associated REVERSE-JOIN, the actual join simply serves to combine the results in the correct way. To obtain the join result, the update program takes the output operations of the REVERSE-JOIN and multiplies their column families. In this manner, the inner, left, right, and full join can be easily maintained.

5.3. View composition

So far, we have only discussed the simple case, with one base and one view table. As we strive for support of more complex constructs (i.e., SPJ queries with aggregation) in VMS, we need to extend our design.

Usually, a query consists of multiple clauses (e.g., select, from, where, group by). In order to materialize the corresponding view, we build a maintenance plan represented as a directed acyclic graph. Each node in the graph represents a materialized view, each edge represents a view operator (i.e., $\sigma, \delta, \pi$, etc.). Starting from the base table, all views are inter-connected and each pair of view tables resembles a base/view table-relation. When a base table is updated, the operation propagates to all views and updates them incrementally. The result of the query can be obtained from the last view.

Figure 6 shows the maintenance process of the following query: 

```
SELECT SUM(c2) FROM bt1 WHERE c2 < 10 GROUP BY c1
```

Each clause of the query translates into one view type (i.e., select into $\pi$, where into $\sigma$, etc.). It can be observed that the update path is divided into two zones. A zone describes a part of the update path where all views possess the same row-key. Once a zone ends, a new zone begins. We call the row-key – changes, updates need to be re-distributed in order to sustain consistency (cf. Figure 4). In the figure two zones can be identified. The maintenance of a zone is always processed in a cycle of three steps:

1. Operation is provided to distributor
2. Operation is assigned and sent to VS
3. Update is applied to view tables of zone

In the first step, the operation is provided to the distributor (cf. Figure 6). When maintaining the first zone the operation is directly derived from the operation of the client on base table $A$. It is retrieved from the TL by the reader component, passed to the resolver component and finally to the assigner component. In the second step, the assigner determines a responsible VS and sends the operation to it. In the third step, the VS goes along the path of each view table of a zone (i.e., view table $V_{0}$ and $V_{1}$ of Zone 1), retrieves the old view record, updates it incrementally and stores the new version back into the view.

Then, the cycle is repeated for the Zone 2. When maintaining subsequent zones (i.e., not the first zone), the operation is sent to the distributor by the responsible VS. The reader and the resolver component of the distributor are bypassed, the operation
is directly handed to the assigner component. The assigner, then again, determines a responsible VS (i.e., $V_{S_1}$) and sends the operation. The VS treats the arriving operation exactly like a base table operation – just that the base table is the view table of the previous zone. I continues and updates all remaining view tables (i.e., $V_2$, $V_3$ and $V_4$ of Zone 2).

Queries can be a lot more complex, they can be nested and they can embrace many base tables at once. Using the Pre-Processing Views and the standard view types that we introduced, every kind of SQL-query can be constructed. Every SQL-clause is mapped to one of the view types, every nested part builds its own sub graph in the maintenance plan. Constructing a maintenance plan may result in numerous zones that are maintained one after the other, starting from the first one. Still, as all the operations in a zone are distributed according to the cardinality of the row-key, a high parallelism of updates can be achieved. Each zone can be maintained by hundreds of VS in parallel. Even the barriers between zones can be overcome quickly. Many VSs can process and sent many operations from one zone to the next.

One of the downsides of our approach is the long path and the high number of intermediate steps that the incremental maintenance demands. This may lead to a high storage consumption and to a high processing cost when maintaining complex queries. However, we present a widely unoptimized solution. We consider the optimization in a multi view environment as out of the scope of this work. Still, we point out that the maintenance of queries can be greatly accelerated by merging different view types (cf. REVERSE-JOIN view) and also by materializing only parts of the maintenance plan. In the best case, only one view table of a zone needs a materialization, as all the view tables within a zone share the same row-key.

6. EVALUATION

In this section, we report on the results of an extensive experimental evaluation of our approach. We fully implemented VMS in Java and integrated it with Apache HBase. Before we discuss our results, we review the experimental set-up and the workload.

Experimental set-up – All experiments were performed on a virtualized environment (OpenStack) of a cluster comprised of 42 physical machines (each equipped with 2x Intel Xeon (8 cores), 128 GB RAM, 600 GB SSD, 1 TB HDD and inter-connected with a 10 GBit/s network).

The set-up employs up to 500 virtual machines (running Ubuntu 14.04). A group of 33 VMs is dedicated to the Apache Hadoop (version 2.1.1) installation: one as name node (HDFS master) and 32 as data nodes (HDFS file system). Co-located with HDFS, HBase (version 0.98.6.1) is installed on these same VMs with one master and 32 region servers, respectively. For performance reasons, the data node/region server machines are properly sized (4 cores, 4 GB RAM, SSDs for fast read/write performance). Three additional VMs are used to host the ZooKeeper service that is used by HBase and also VMS. The VMS distributors are deployed on 32 VMs to match exactly the number of region servers. They are deployed on separate VMs to not interfere with the processing of HDFS/HBase. The VSs are deployed on 50, 100, 200 and 400 VMs (each possessing 1 core and 1 GB RAM). The small set-up (i.e., 50 VMs) is used for micro- benchmarking the basic view types. The large set-ups (i.e., 100, 200 and 400 VMs) are used to evaluate multi view and maximal throughput of the system. Each VM possesses 1 core and 1 GB RAM and hosts a single VS.

Synthetic workload – The synthetic workload we generate consists of insert, update and delete operations that are issued to HBase using its client API. Operations are generated uniformly over the key space. Our synthetic workload ranges from 2M to 8M operations.

For aggregation views, we create base tables that contain one column $c_1$ (aggregation key) and another column $c_2$ (aggregation value). We choose a random number $r_a$ between 1 and some upper bound $U$ to generate aggregation keys. We control the number of base table records that affect one particular view table record. For a selection view, we use the same base table layout and apply the selection condition to column $c_2$. Join views require two base tables with different row-keys. The row-key of the right table is stored in a column in the left table, referred to as foreign key.

TPC-H workload – We use a workload generated by the dbgen tool of the TPC-H benchmark\footnote{Our view maintenance performance metrics are not comparable to query evaluation performance metrics. Consequently, our performance results differ from published TPC-H results.}. TPC-H contains a record set of typical business data, containing many columns of various data types (addresses, text fields, float values). Our objective is to use VMS to incrementally maintain views over TPC-H data, for fast access to summarized data used in online analytics.

Out of the TPC-H entities, we use two of the largest tables, which are the orders table (1.5M records) and the customer table (150k records). The orders table is used to compute projection, selection and aggregation queries. The grouping of the aggregations is done by customer key ($o\_custkey$) and clerk number ($o\_clerk$). For the join query, the orders table is joined with customer table based on the customer key. In contrast to the synthetic workload, the TPC-H workload is insert-only. All 1.5M orders in the orders table have a distinct primary key.

Queries – Our evaluation uses the following four queries:

1. \( q_1 \) : select \( c_1 \) from \( bt_1 \)
2. \( q_2 \) : select \( c_1 \) from \( bt_1 \)
3. \( q_3 \) : select \( c_1 \), \( function(bt_1, c_2) \) from \( bt_2 \) group by \( c_1 \)
4. \( q_4 \) : select \( bt_1, c_1, bt_2, d_2 \) from \( bt_1, bt_2 \) where \( bt_1, c_1 = bt_2, d_1 \)

The Pre-Processing Views and standard views used on the VMS update path are noted below each query. For example, \( q_4 \) requires a composition of two view tables: each base table operation triggers the update of both view tables. For \( q_4 \), the aggregation operation function can be sum, min, max, average, or count. For \( q_4 \), VMS creates four view tables (two DELTA, one REVERSE_Join and one projection). Nonetheless, each base table operation triggers only three view table updates (i.e., the depth of update path). This is because base table operations are streaming in from two different base tables.

Metrics – We evaluate the VMS performance by logging the current, average, minimum and maximum throughput every second at every VS. The throughput is always determined in terms of base table operations per second (i.e., \( bto/s \), cf. y-axis Figure 7). One base table operation can trigger multiple view updates or none at all (e.g., when a record does not match a selection). The throughput of VMS is computed by summing up the average throughput of all VSs. Depending on the depth of the update path and the maintained view types, the throughput of a VS can vary.

To test the scalability of VMS, we perform each experiment with a varying number of VSs, ranging from 1-50 for the small set-up, and 1-200 for the large set-up. We also provide a set of experiments evaluating the view update latency and access time. Using these metrics, we can assess the staleness of the view data, which provides us with a point of reference with pure HBase, where queries have to be evaluated by a client directly reading the base table data via the KV-store API.

Experimental execution – Prior to each experiment, we create an empty base table and define a set of view tables. View definitions (i.e., queries) are maintained as meta-data in a separate view definition table. By default, HBase stores all base table records in one region. We configured HBase to pre-split every table into 50 regions. This choice allows HBase to balance regions with high granularity and ensures a uniform distribution of keys among available region servers. Splitting the regions of view tables is crucial for VMS to distribute the load.
10 to 50 parallel client threads are used to send base table operations to the KV-store. Once the entire workload has been processed by KV-store, VMS then starts to maintain the views using the transaction logs, which are now filled with the entire volume of operations. This two-phase process ensures that we evaluate the maximal throughput of VMS without being bound by the incoming rate of base table operations. In other words, we control the experiments to avoid a situation where the transaction logs are read faster than they are being written.

**Projection performance** — The performance of query q1 is depicted in Figure 7. We use the small cluster set-up and apply 2M records (synthetic workload) and 1.5M records (TPC-H workload, orders table), respectively. Likewise, for the synthetic workload, we vary the number of row-keys (1k, 10k, 100k). A small number of row-keys implies a larger number of operations per row-key (insert, update, delete). The TPC-H workload is a pure insert workload, every row-key is used only once.

We observe a nearly perfect linear scaling for all four configurations tested. With every VS that we add to the VMS, the throughput of the entire system increases proportionally. Each VS is able to process from 902 to 1026 base table operations/second, regardless of the total number of VSs in the system. Moreover, we observe that the throughput with 1k row-keys is the lowest, with a 12% reduction compared to 100k keys. This is because the distribution of keys in the hash-ring improves when using a higher number of row-keys. Although we are using virtual nodes to improve the performance of consistent hashing, there has to be a sufficient number of keys. For example, maintaining 100 row-keys with 50 VSs means that only 2 keys can be assigned to one VS. Experimentally, we assess that at least 10-20 row-keys per VS are needed to guarantee a uniform distribution of operations in the system.

**Selection performance** — The performance of query q2 is depicted in Figure 8. The synthetic workload produces records with a column value $c_2$ between 0 and 100. Based on this column, we adjust the selection criterion $x$ in q2 to achieve 20%, 50% and 80% selectivity. For the TPC-H workload, we select records from the orders table based on the field $o_{totalprice}$. We set the variable $x$ to 10000.0 which has a selectivity of 50%.

We observe linear scalability for the selection query evaluation in all tested configurations. Further, VMS achieves 46% higher throughput with lower selectivity (50%) compared to high selectivity (80%), as most of the base table operations are dropped during maintenance without causing view updates since they affect records which are outside of the selection range.

The throughput of the 50% TPC-H workload is 9.6% lower than the throughput of the 50% synthetic workload. This is related to the fact that the distribution of values in the TPC-H workload is not as uniform as for the synthetic workload; the actual selectivity is slightly higher than 50% (ca. 52%).

**Aggregation performance** — The performance of query q3 is depicted in Figure 9. We evaluate the synthetic workload by varying the aggregation-ratio: 10k:1k means that 10k base table row-keys are grouped down to 1k aggregation row-keys. For the TPC-H workload, we evaluate aggregation of the column $o_{custkey}$ (i.e., 1M:150k) and the column $o_{clerk}$ (i.e., 1M:1k). In each case, we are maintaining four different views using different columns: sum, min, max, and count. The update path for aggregation queries requires an additional update step using a PRE_AGGREGATION view table, which pre-aggregates the records based on the aggregation key (see Section 5.1).

We achieve linear scalability for aggregation in most cases, except for aggregation of $o_{clerk}$ (1M:1k) in the TPC-H workload. We note that VMS is sensitive to the aggregation-ratio. A higher ratio indicates that more base table row-keys are aggregated into one view row-key, which increases the size of the PRE_AGGREGATION view. At a large scale, this Pre-Processing View becomes a bottleneck, since VMS has to load and separate those pre-aggregated records to apply the aggregation function for every operation.

However, this intermediate step allows multiple aggregation operators, over different columns, to be maintained in the same step. The PRE_AGGREGATION view can be amortized over multiple queries on the same base table. We also evaluated the aggregation experiments using only a single operator (instead of five) and found no significant impact on performance (1%).

**Join performance** — The performance of query q4 is depicted in Figure 10. We create 2M records for the synthetic workload (1M per base table, respectively). For the TPC-H workload, we join the table orders (1.5M records) and customers on columns $o_{custkey}$ and $c_{custkey}$. We evaluate the synthetic workload by varying the number of join keys (1k, 10k, 100k). A lower number of join keys produces a higher output for each join operation (i.e., cross product). The TPC-H workload creates a join with 150k join keys, where the join is a key/foreign-key join (i.e., the ratio is 1:n not n:m).
Query q₄ uses a REVERSE_Join view: join records from each table are collected and pre-joined. The actual join is then built in a successive step. Similar to the PRE_AGGREGATION case discussed previously, the REVERSE_Join view can be re-used by multiple join views to amortize the overhead of this extra maintenance step. Thus, the performance results for join are comparable to aggregation, with linear scalability achieved. Note that, the performance decreases by 13% for 1k join keys compared to 100k join keys: this is again due to the low number of keys used for hashing, which affects load distribution to the VSs.

Multi-query performance – Figure 11 shows scalability results for the simultaneous maintenance of multiple queries using the large set-up, 2M records, and 10 base tables. We use a fix number number of 100 queries and vary the number of base tables over which the queries are defined. All queries are of type q₄, and use a different selection range. Each query requires 2 view tables: correspondingly, VMS has to maintain 200 views.

We observe that the system scales linearly even above 50 VSs. Further, we notice that the number of base table operations per second (bto/s), VMS achieves, is depending on the ratio base tables/queries. E.g., in a system, where all 100 queries are defined over the same base table, VMS has to process 100 updates per base operation. In contrast, when using 20 base tables, VMS only has to update only 5 queries per base operation (assuming uniform distribution of queries). For that reason, we see the throughput double in Figure 11 when specifying twice the amount of base tables.

Maximum performance – Figure 11 shows the maximal performance of the system in terms of view updates per second (vu/s). On the x-axis the number of maintained view tables are shown. We scale the system to maintain 12800 view tables using a set of 100, 200 and 400 view server. Then we measure the average throughput, depicted on the y-axis.

We observe that VMS is able to achieve a peak throughput of 1.15M vu updates per second. Considering that each view update consists of a read and write operation this yields 2.2M HBase operations per second. Further we observe that the system is capable of maintaining 12.8K views in parallel; even though, throughput is declining to one third of the original seize. The decline is related to the fact that a single VS has to access multiple different tables when maintaining a high number of views. E.g., when maintaining 12K views with 400 view server, each view server has to access 30 different view tables (each comprising of 10 key-ranges). Thus, it needs to access and cache a lot of different locations. An optimal threshold can be described as using not more than 4 view tables per view server. If the number of view tables surpasses this threshold, the number of VSs should be increased to avoid performance hits (though maintenance is still working correctly).

Baseline comparison – We compare the following strategies for computing a count aggregation query: (i) Client table scan: To obtain the most recent count, a client scans the base table and aggregates all values on its own. (ii) Server table scan: The client sends a request to HBase, which internally computes the view and returns it as output. In our implementation, all region servers scan their part of the table in parallel. Intermediate results are then collected and merged. (iii) Materialized views: Our proposed approach, views are maintained incrementally by VMS configured using 40 VSs to materialize the count view in parallel. For (i) and (ii), we disregard inconsistencies that may result from concurrent table operations while scans are in progress ⁵. Our objective with this experiment is to assess the benefits of VMS compared to pure KV-store approaches from the perspective of a user using access time to the desired data as a metric.

Figure 13 shows access time results, where we measure the latency a client perceives (i.e., the time until results are available) as the scanned key range increases. Client table scans are sequential by nature and require a large number of RPCs to HBase, even if requests are batched. This approach does not scale with an increasing table size. Server table scan performs better because HBase is able to exploit the data distribution, which results in a linear speed-up. Nonetheless, the time to obtain the most recent values for a count aggregation reached 5 s for a table with 1M rows. These results can be cached, in which case frequent recalculations are needed to keep them up-to-date in high churn scenarios. Moreover, concurrent table scans may interfere with the write performance of region servers. Also note that HBase provides no ACID guarantees for server table scans: the view results may therefore be inconsistent.

Meanwhile, our proposed approach VMS offers constant read times to the view data, even when the table size is increased. This is because the read path to the view data is decoupled from the view update path. Moreover, the client only accesses the aggregated values and not the base table. Therefore, the latency remains below 1 ms, even for a base table with 10 million rows.

Maintenance cost – We assess this cost as the performance impact on client operations per second (here, defined as the throughput while inserting 2M tuples into the base table). Figure 14 shows the throughput with and without concurrent view maintenance. We track the client throughput as the number of VSs increases from 5 to 50. We observe that the client throughput remains unaffected up to an amount of 25 VSs. From this point on a small impact on throughput of ca. 7% can be observed. Thus, we assess the cost of maintenance to be small. Both, clients and VSs, access HBase through the same API and use the same region servers (base table and view table regions can be located at the same node). As their number increases, HBase needs to provide sufficient resources (i.e., amount of region servers), such that they can operate unhindered from each other.

View update latency – Figure 15 depicts the view update latency of different query types. The update latency is defined as the time needed by VMS to update all the views affected by a base table operation. The update latency includes the time taken to read the operation from the TL, to assign and send it to a VS which updates the corresponding views. In this experiment, we do not take into account any queuing delay incurred at overloaded VSs.

On average, VMS needs 0.85 ms to process one base table operation for query q₁, 0.85 ms to 1.5 ms for query q₂ and 2.5 ms to 3.5 ms for query q₃ and q₄. Again, aggregation and join query achieve the same results due to their similar design.

⁵In practice, this would not be an option; thus, these two approaches are merely serving as a baseline for our solution.
We use the view update latency results to analytically derive several metrics. The first equation evaluates the maximal throughput of VMS given a number of VSs and the update latency of the query:

$$\text{max throughput} = \frac{\text{numVS}}{\text{latency}}$$

(1)

For example, 50 VSs and a latency of 1.2ms yields a maximal throughput of 41.666666/second. The maximal throughput indicator can be used to correctly provision the HBase region servers and VMS VSs. When the rate of base table operations exceeds the maximal throughput (e.g., during a peak-time), the asynchronous nature of VMS allows the system to continue to operate by queuing incoming operations at the VSs. The clients must wait a certain amount of time until their operations take effect on the view table, which we call view staleness:

$$\text{staleness} = \left(\frac{\text{queue size}}{\text{numVS}}\right) \times \text{latency}$$

(2)

The parameter queue size describes the sum of the size of the queues over all deployed VSs. Using 25 VSs with a total of 100k operations queued, VMS experiences a view staleness of 5 seconds. Depending on the input rate of operations to the system, we can increase the number of VSs to reduce view staleness. By employing 200 VSs, the staleness already drops below 1s.

The asynchronous nature of VMS proves to be advantageous with regard to maximal throughputs: the system can operate above its maximal throughput for a while (e.g., during a peak-time). When doing this, more operations are received than processed and queued up at the VSs.

7. RELATED WORK

Research on view maintenance started in the 80s [14,15,20–22]. Blakeley et al. [20] developed an algorithm that reduces the number of base table queries during updates of join views. Colby et al. [22] introduced deferred view maintenance based on differential tables that keeps around a precomputed delta of the view table. Much attention has been given to preventing update anomalies when applying incremental view maintenance [14,21,23]. All these approaches originated from the databases at the time of their inception, i.e., storage was centralized and a fully transactional single-node database served as starting point, which is greatly different from the highly distributed nature of the KV-store we consider in this work.

Zhuge et al. [14] consider view maintenance in multi-source data warehousing scenarios. Similar to KV-store nodes in our approach, sources are view-agnostic but capable of informing the data warehouse (VMS in our approach) of update events. An update event causes the data warehouse to query affected source systems to calculate the view relation for the updated record. Because this approach leads to update anomalies, VMS completely dispenses with the use of base table scans.

An approach for view maintenance in KV-stores together with different consistency models has been presented in [18]. We leverage these consistency models in our work, apply them to attain consistency for different view types and greatly extend the system scope. Unlike their work, for instance, our approach is capable of consistently maintaining join views, which is made possible through our proposed Pre-Processing Views. In addition, we provide an industry-strength scalable implementation through our design choices, namely monitoring of the TLs and the key hashing technique.

Some interesting materialization approaches are presented in [12,13]. Pequod [13] serves as front-end application cache that materializes application-level computations. It supports a write-through policy to propagate updates to the back-end store, while serving reads from the cached data. Unlike our work, the approach does not focus on global consistency and provides at most the client-level read-your-writes consistency model for certain views. SLIK [12] provides strong consistency, however, it is limited to the materialization of secondary index views in KV-stores.

Cui et al. [24] introduced the concept of auxiliary views as an optimization for the dual purpose of view maintenance and data lineage tracing. However, auxiliary views are the materialization of an intermediate result that is stored to compute the lineage of a view record.

In our approach, we defined the concept of Pre-Processing Views. In contrast to the auxiliary view [24], the Pre-Processing View represents a preliminary step, which serves to facilitate and speed up the processing of subsequent views. The Pre-Processing View allows to connect (and also parametrize) multiple following views, e.g., a reverse join can serve to feed multiple join or aggregation views. It is a decomposition of different view types and cannot be used for data lineage.

In recent years, there has been a rising interest in developing support for the materialization of views in a KV-store, both in open source projects and products [8,11,25,26] and in academia [9,10,12,13,18,27,28]. Percolator [8] is a system specifically designed to incrementally update a Web search index as new content is found. Naiad [25] is a system for incremental processing of source streams over potentially iterative computations expressed as a dataflow. Both systems are designed for large scale environments involving thousands of nodes, but are not addressing the incremental materialization of the kind of views considered in this work.

The Apache Phoenix project [11] develops a relational database layer over HBase, also supporting the definition of views. Few implementation details about the views are revealed, except for the fact that view definitions are limited to selection views and view results are generated by periodically scanning the base tables. Also, a long list of view limitations is given. For example, “A VIEW may be defined over only a single table through a simple SELECT * query. You may not create a VIEW over multiple, joined tables nor over aggregations.” [11] The baseline comparison in Section 6 evaluates the performance of periodic base scans against our incremental maintenance solution.

8. CONCLUSIONS

In this paper, we present VMS, a scalable view maintenance system which operates with a distributed KV-store as an independent component. VMS leverages TLs, a common feature provided by KV-store, to asynchronously monitor new operations in a durable manner while remaining decoupled from the underlying system. VMS is designed around a novel group-based key hashing technique to distribute new updates which maintains strong consistency while providing a high degree of parallelism. Thus, our VMS architecture can be scaled incrementally, independently of the underlying KV-store, and accommodates the addition of view servers as maintenance load increases.

VMS provide efficient, incremental, and deferred view materialization. Our proposed Pre-Processing View types are leveraged to facilitate and speed up maintenance and avoid expensive table scans. Building on the view types, VMS can consistently compose and maintain SPJ queries with aggregation.

As a proof of concept, VMS has been implemented on top of HBase. Our experimental evaluation with synthetic and TPC-H workloads shows that VMS scales linearly in view update load and number of view servers running. We demonstrate the system’s ability to perform real-time analysis even in the presence of hundreds of queries (views, respectively). Further, we quantify the benefits and drawbacks of the approach, compare it to baseline solutions, and measure view update latency and staleness.

Future work on VMS include exploring optimizations for the maintenance of multiple, overlapping views. The Pre-Processing View facilitates and speeds up maintenance and avoids expensive table scans. The integration of batch processing...
9. REFERENCES


[10] Adam Silberstein, Jeff Terrace, Brian F. Cooper, and Raghu Ramakrishnan. Feeding frenzy: Selectively materializing users’ event feeds. SIGMOD ’10. ACM.


APPENDIX

A. PROOF OF CONSISTENCY

As stated in Section 4.3, Theorem 1 states that a view maintenance system fulfilling all three of the following requirements achieves strong consistency:

1. P1: View updates are applied exactly once.
2. P2: View updates are processed in isolation.
3. P3: Record timeline is always preserved.

To prove this theorem, we need to rely on weaker models of consistency which form the basis for strong consistency. The full spectrum of our consistency model is as follows:

Convergence: A view table converges, if after the system quiesces, the last view state \(V_f\) is computed correctly. This means it corresponds to the evaluation of the view expression over the final state of base tables \(V_f = View(BS_f)\). View convergence is a minimal requirement, as an incorrectly calculated view is of no use.

Weak consistency: Weak consistency is given if the view converges and all intermediate view states are valid, meaning that there a valid operation sequence from which every intermediate view state can be derived from a base state of \(BS\).

Strong consistency: Weak consistency is achieved and the following condition is true. All pairs of view states \(V_i\) and \(V_j\) that are in a relation \(V_i \leq V_j\) are derived from base states \(BS_i\) and \(BS_j\) that are also in a relation \(BS_i \leq BS_j\).

Complete consistency: Strong consistency is achieved and every base state \(BS_i\) of a valid base state sequence is reflected in a view state \(V_i\). Valid base state sequence means \(BS_0 \leq BS_1 \leq ... \leq BS_n\).

Our proof is organized in three stages: we start with proving convergence and then present extensions to also prove weak consistency and finally strong consistency.

A.1. Notation

First, we define the following notation for keys, operations on keys, and the ordering of operations. Let \(k_x\) denote a key in a base table \(A\), where \(x \in X = \{1,...,n\}\), and \(X\) is the table’s key range. \(X\) can be qualified with the name of the table when multiple tables are involved (e.g., \(X_A\) for table \(A\), \(X_C\) for table \(C\)). Further, let an operation on key \(k_x\) be defined as \(\{t[i]k_x\}\). A totally ordered sequence of operations \(S\) on a single table of length \(N\) is denoted by \(\{t[1]k_{x_1}, t[2]k_{x_2}, t[3]k_{x_3}..., t[N]k_{x_N}\}\), where \(vi \in \{1,...,n\}, x_i \in X\). In other words, this sequence contains...
operations over an arbitrary key in the base table, where it is possible that a key is updated several times or not at all.

The index $i$ in $i^{(t)}[k_i]$ is used to express a sequence of operations on a single row-key (i.e., the record timeline). For example, a sequence of operations on row-key $k_i$ is denoted as $(i^{(t)}[k_i],i^{(2)}[k_i],...,i^{(w)}[k_i])$. The last operation on a particular row-key is always denoted with $\omega$. Note that each operation in the timeline also exists in the overall sequence for the table containing the record: $\forall i \in \{1,\ldots,n\}, \exists j \in \{1,\ldots,n\}, i^{(t)}[k_i] = i^{(t)}[k_j]$. A sequence of operations $S = \{t_1[k_i],...,t_n[k_i]\}$ over table $A$ produces a sequence of base table states $(B_0,...,B(T(\text{V}[k_i])))$. $B_0$ is the initial state of $A$, and $B(t_n)$ is the state of the table after applying some transaction $t_n$. We also call the final state of the table $B_f$, where $B_f = B(t_s[k_i])$. $B_f(k_i)$ for $x \in X$ denotes the final base table state of a specific key $k_i$.

**Lemma 1.** For any two operation sequences on $k_i$: $S_1 = (i^{(t)}[k_i],i^{(2)}[k_i],...,i^{(w)}[k_i])$ and $S_2 = (i^{(t)}[k_i],i^{(2)}[k_i],...,i^{(\omega)}[k_i])$. Let $B_f(k_i), B'_f(k_i)$ be the final state of $k_i$ after applying $S$ or $S'$, respectively. Then, $\forall \omega \in S_1: B_f(k_i) = B'_f(k_i)$.

**Proof.** The lemma states that the final state of a given key is completely determined by the last operation on that key. This follows due to the idempotence of the key-value store write operations (put, delete). The operations are repeatable with no consequence and do not read the previous stored state of the key (stateless).

According to Lemma 1, the state of a key in a base table is only dependent on the last operation applied on that key. Therefore, the notation $B(t_s[k_i])$ refers to the state of some key $k_i$ after applying an operation $t_s$ irrespective of the sequence of operations preceding $t_s$.

**A.2. Convergence**

Using the above notation, the property of convergence is defined as follows:

**Convergence:** Given a finite sequence of operations $(t_1,t_2,t_3,...,t_N)$ for some base table $A$, $B_f$ is the final state of $A$. The final state $V_f$ of a view table over $A$ converges if $V_f = \text{View}(B_f)$, where $\text{View}(B_f)$ is the evaluation of the view expression of the final state of $A$, $B_f$.

We prove convergence on a case-by-case basis for each type of view expression.

**One-to-one mapping:** **SELECTION, PROJECTION** views define a one-to-one mapping between base and view table. We first prove the following lemma:

**Lemma 2.** Given a sequence of operations applied for a view (using **SELECTION, PROJECTION**, only) on key $k_i$: $S = (i^{(t)}[k_i],...,i^{(\omega)}[k_i])$. Let $V_f(k_i)$ be the final state of the view on $k_i$ after applying $S$. Then, $\forall \omega \in S: \text{View}(\omega) = V_f(k_i)$.

**Proof.** Since **SELECTION** and **PROJECTION** are both idempotent stateless operations, the final state of the view maintained on $k_i$ is equivalent to applying the view operator on the final operation in the sequence.

We now prove convergence by contradiction. Suppose that $V_f \neq \text{View}(B_f)$. Then, $\exists \omega \in X, \text{View}(B_f(k_i)) \neq V_f(k_i)$. According to Lemmas 1 and 2, $i^{(t)}[k_i] \neq s^{(\omega)}[k_i]$, where $s^{(\omega)}[k_i]$ is the last operation processed by the view for $k_i$ and $i^{(t)}[k_i]$ is the last operation processed by the base table for $k_i$. However, according to properties P1 and P3, the last operation processed by the view and the base table must be the same since both sequences contain the same operations and in the same order. Therefore, $i^{(t)}[k_i] = s^{(\omega)}[k_i]$, $\text{View}(B_f(i^{(t)}[k_i])) = \text{View}(s^{(\omega)}[k_i])$, which contradicts $\text{View}(B_f(k_i)) \neq V_f(k_i)$.

We use a similar proof for the **DELTA** operator.

**Lemma 3.** Given a sequence of operations applied for a view (using **DELTA** only) on key $k_i$: $S = (i^{(t)}[k_i],...,i^{(\omega)}[k_i],t^{(\omega)}[k_i])$. Let $V_f(k_i)$ be the final state of the view on $k_i$ after applying $S$. Then, $\forall \omega \in S: \text{View}(t^{(\omega)}[k_i]) = V_f(k_i)$.

**Proof.** **DELTA** is an operator which computes the difference of state between two operations. Therefore, the final state of the view depends on the last two operations in $S$, which is the same as applying **DELTA** on a base table key which processes these two operations.

Suppose that $V_f \neq \text{View}(B_f)$ for **DELTA**. Then, $\exists \omega \in X, \text{View}(B_f(k_i)) \neq V_f(k_i)$. According to Lemma 3, $i^{(t)}[k_i] \neq s^{(\omega)}[k_i]$ and $\text{View}(B_f(i^{(t)}[k_i],t^{(\omega)}[k_i])) = \text{View}(s^{(\omega)}[k_i],t^{(\omega)}[k_i])$, which contradicts $\text{View}(B_f(k_i)) \neq V_f(k_i)$.

**Many-to-one mapping:** **(PRE-)AGGREGATION** and **INDEX** views define a many-to-one mapping between base and view table. The row-key of the view table is the aggregation key. Multiple row-keys in the base table can relate to a particular aggregation key. However, a base table row has always only one aggregation key. A correct view record with aggregation key $x$ is defined as the combination of multiple base records $k_1,k_2,...$ related to the particular key $x$. We prove the following lemma:

**Lemma 4.** Given a sequence of operations applied for a view (using **(PRE-)AGGREGATION** and **INDEX**) on table $A$. Let $V_f$ be the final state of the view after applying the sequence. Let $S$ be an arbitrary sequence containing only the last operation on each key $k_i, x \in X: S = (i^{(t)}[k_i],...,i^{(\omega)}[k_i])$. Then, $\text{View}(S) = V_f(k_i)$.

**Proof.** **(PRE-)AGGREGATION** and **INDEX** are stateless operations which depend only on the last operation of each key involved. Therefore, applying the view expression on the state of a base table after it has processed the last operation of every key returns the same state as the final view state.

Suppose that $V_f \neq \text{View}(B_f)$ for **(PRE-)AGGREGATION** or **INDEX**. According to Lemma 4, $\exists \omega \in X, i^{(t)}[k_i] \neq s^{(\omega)}[k_i]$, where $s^{(\omega)}[k_i]$ is the last operation processed by the view for $k_i$ and $i^{(t)}[k_i]$ is the last operation processed by the base table for $k_i$. According to properties P1 and P3, the last operation processed by the view and the base table for key $k_i$ must be identical. Therefore, $i^{(t)}[k_i] = s^{(\omega)}[k_i]$, $\text{View}(B_f(i^{(t)}[k_i],...,i^{(\omega)}[k_i])) = V_f(s^{(\omega)}[k_i],...,s^{(\omega)}[k_i]))$, which contradicts $\text{View}(B_f(k_i)) \neq V_f(k_i)$.

**Many-to-many mapping:** **(REVERSE-)JOIN** views define a many-to-many mapping between base and view table. The row-key of the view table is a composite key of both join tables’ row-keys. Multiple records of both base tables form a set of multiple view records in the view table. Since the joining of tables takes place in the **REVERSE JOIN** view, we prove convergence only for this view type. A **REVERSE JOIN** view has a structure that is similar to an aggregation view. The row-key of the **REVERSE JOIN** view is the join key of both tables. All base table records are grouped according to this join key. But in contrast to an aggregation view, the **REVERSE JOIN** view combines two base tables to create one view table. A correct view record with join key $x$ is defined as a combination of operations on keys $k_1,k_2$ from join table $A$ and operations on keys $k_1,k_2$ from join table $B$. This property provides the basis for the following lemma:

**Lemma 5.** Given a sequence $S$ of operations applied for a view (using **(REVERSE-)JOIN**) on tables $A$ and $C$. Let $V_f$ be the final state of the view after applying $S$. Let $S$ be an arbitrary
sequence containing only the last operation on each key \( k_x, x \in X_A = 1, ..., m \) in \( A \) and each key \( y_x, x \in X_B = 1, ..., m' \) in \( C \) : 
\[
S = \{ t^ω[k_1], ..., t^ω[k_n], t'[ω'[y_{n'}], ..., t'[ω'[y_{n''}]] \}.
\]
Then, \( \text{View}(S) = V_j \).

**Proof.** (REVERSE-)JOIN are stateless operations which depend only on the last operation of each key involved (from both tables). Therefore, applying the view expression on the state of base tables after each has processed the last operation of every key returns the same state as the final view state.

For (REVERSE-)JOIN, convergence is achieved if \( V_j = \text{View}(B_f, B_f') \) where \( B_f, B_f' \) are the final view states for tables \( A, C \) involved in the join, respectively. Suppose that \( V_j \neq \text{View}(B_f, B_f') \).

According to Lemma 5, \( \exists x \in X_A \cup X_C, t^ω[k_x] \neq s^ω[k_x], \) where \( s^ω[k_x] \) is the last operation processed by the view for \( k_x \) and \( t^ω[k_x] \) is the last operation processed by a base table for \( k_x \).

According to properties P1 and P3, the last operation processed by the view and the base table containing key \( k_x \) must be identical. Therefore, \( t^ω[k_x] = s^ω[k_x], \) which contradicts \( \text{View}(B_f, B_f') \neq V_j \).

### A.3. Weak consistency

Weak consistency has been defined as follows: Weak consistency is already proven, we only need to prove the statement that all the intermediary view states are valid, meaning there exists a valid sequence of operations from which they can be derived from the state of one or more base tables (\( V_j = \text{View}(B_1, ..., B_k) \)). As we already proved convergence, we need to show that all the intermediary view states are correct.

For view expressions which depend on one key in table \( A \) (excluding DELTA), suppose that for some intermediate view state \( V_j, \forall i \in 1, ..., N, V_j \neq \text{View}(B_i) \). Let \( T = \{ t_1[k_1], t_2[k_2], ..., t_n[k_n] \} \) be the sequence of operations applied on \( A \). Let \( s_j \) be the operation that produced \( V_j : V(s_j) = V_j \). By Lemmas 2, \( \forall i \in 1, ..., N, s_j \neq t_i \). In other words, the view processed an operation which was never processed by the originating base table. However, property P1 and P2 ensure that each operation corresponds to a base table operation and is fully processed. By contradiction, \( V_j = \text{View}(B_i) \).

For DELTA on table \( A \), suppose that for some intermediate view state \( V_j, \forall i \in 1, ..., N, V_j \neq \text{View}(B_i) \). The proof is similar to the above, with the addition that property P3 ensures each pair of consecutive operations processed by the view must exist and have been processed as a consecutive pair by the base table \( A \).

For view expressions which depend on multiple keys in table \( A \), suppose that for some intermediate view state \( V_j, \forall i \in 1, ..., N, V_j \neq \text{View}(B_i) \). Let \( T = \{ t_1[k_1], t_2[k_2], ..., t_n[k_n] \} \) be the sequence of operations applied on \( A \). According to Lemma 4, Let \( S \) be an arbitrary sequence containing only one operation on each key \( k_x, x \in X_A = 1, ..., m \) such that \( V(S) = V_j \). According to the definition of intermediate view states, \( \exists k_x, x \in X_A, t(k_x) \in S, t(k_x) \notin T \). In other words, there exist at least one operation on some key in \( A \) which was processed by the view, but not by the base table. However, property P1 and P2 ensure that each operation corresponds to a base table operation and is fully processed. By contradiction, \( V_j = \text{View}(B_i) \).

For (REVERSE-)JOIN on tables \( A, C \), the argument is similar than for SELECTION and PROJECTION. According to Lemma 5, every state \( V_j \) is created on a pair of operations on \( A \) and \( C \).

Using property P1, it is guaranteed that both operations have been previously processed in the originating tables.

### A.4. Strong consistency

Strong consistency has been defined as follows: Weak consistency is achieved and the following conditions hold true. All pairs of view states \( V_i \) and \( V_j \) that are in a relation \( V_i \leq V_j \) are derived from base states \( B_i \) and \( B_j \) that are also in a relation \( B_i \leq B_j \). Since weak consistency is already proven, we only need to prove the statement \( V_i \leq V_j \Rightarrow B_i \leq B_j \). If this statement is false, then only two of the following cases can occur: Either \( V_i \leq V_j \Rightarrow B_i \parallel B_j \) or \( V_i \leq V_j \Rightarrow B_i \geq B_j \). Both cases can only be constructed by breaking the record timeline. To be precise: At least one record has to exists, whose timeline is broken. Formally, we demand \( \exists t \in B_i (\forall t_k \in B_j : (r(t_l) = r(t_k)) \land (l > k)) \). Because property 3 prevents the breaking of timelines, we conclude that both cases are not possible. Thus, we have proven strong consistency by contradiction.