Dynamic Scalable View Maintenance in KV-stores

Jan Adler
Technische Universität München
adler@in.tum.de

Martin Jergler
Technische Universität München
jergler@in.tum.de

Arno Jacobsen
Technische Universität München
jacobsen@in.tum.de

***Abstract*—Distributed *key-value stores* (KV-stores) have become the solution of choice for many data-intensive scenarios. However, their limited query languages, their loose constraints on the data model and their highly distributed architectures impose challenges for applications that require sophisticated analytical capabilities. Likewise, computed results need to be maintained up-to-date, which implies repeated scans of millions of rows on distributed and frequently changing data sets. To address this problem, in this paper, we develop a *view maintenance system* (VMS) based on deferred and incremental view maintenance strategies. We build the VMS on top of a generalized KV-store model, extending the typical, distributed KV-store architecture. Our VMS supports the maintenance of hundreds of views in parallel, while simultaneously providing guarantees for consistency and fault tolerance. To evaluate our concept, we deliver a full-fledged implementation of the VMS with Apache's HBase and conduct an extensive experimental study.**

I. Introduction

Today's large scale internet services (e.g. Google Maps, Facebook, Amazon, etc.) handle millions of client requests, producing terabytes of data on a daily basis [1]. To handle this load, major internet companies have developed distributed databases called KV-stores such as Google's BigTable [2], Amazon's Dynamo [3], Yahoo's PNUTS [4], Apache's HBase [5] and Cassandra [6] (originally Facebook).

KV-stores are highly available to the user and provide advanced features like automated load balancing and fault tolerance. KV-stores scale horizontally – by partitioning data and request load across a configurable number of nodes. To achieve these properties at a large-scale, KV-stores sacrifice an expressive query language and data model, only offering a simple API, comprising of *put*, *get* and *delete* operations. While this API provides efficient access to single row entries, the processing of more complex (SQL-like) queries, e.g. selection, aggregation, and join, require costly application-level operations. Although, some KV-stores provide additional features to support higher-level query processing, those features are often rudimentary and impose bottlenecks.

Many existing approaches separate transactional and analytical processing. A complete snapshot of the data base is copied or loaded to an *external* data warehouse and then, processed in a batch-wise fashion. Therefore, numerous existing frameworks with varying abstraction levels are available, e.g. Map Reduce [7], Apache Spark [8], Apache Hive [9], etc. While this approach exploits the benefits of high performance parallel processing, it always requires an initial load overhead. Further it is not capable of providing up-to-date results – as frequent changes in the base data occur.

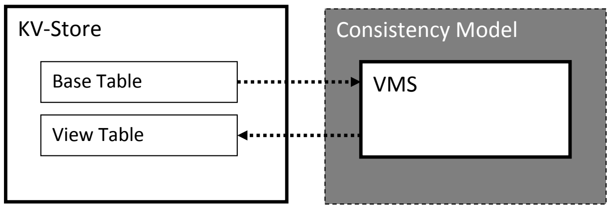

Fig. 1: System overview

Another way of solving the problem is the use of *internal* KV-store mechanisms that directly operate on the KV-store data. Apache Pheonix [10] enables rich SQL semantics by using the coprocessor functionality (little code snippets, deployed on the KV-store nodes) of HBase. While this approach is implementation bound to a specific KV-store, we strive for a more general solution.

Our approach introduces mechanisms for the materialization and incremental maintenance of views to KV-stores. The user attains sophisticated query capabilities, simply through definition of view expressions. The KV-store keeps results highly available and enables access of many clients in parallel – as materialized views are simple tables managed in KV-store. Likewise, views can be maintained incrementally; only those view records are updated whose base records have been changed. The challenge then, is not the scanning and computation of base data any more, but the efficient and correct maintenance of views. To achieve this at scale, we develop the *Distributed View Maintenance System* (VMS) as shown in Figure 1. The VMS consumes streams of client operations (of a base table) and produces the corresponding updates (to the view table).

First, we examine a set of KV-store implementations [2], [4]–[6] and derive the common key characteristics from their architectures and their data models (Section II). As is common in the literature on view maintenance, we use a consistency model (Section III) to ensure materialized views remain consistent with base data. But unlike the existing models [11]–[13] — which where mainly applied to centralized data warehouse environments — we design our own consistency model to match the needs of a highly distributed environment (i.e. KV-stores). Based on the KV-store model and the requirements of the consistency model, we describe the design of the scalable VMS (Section IV). We design our VMS to scale in view update load and number of views maintained. Our design does not interfere with the read/write processing against tables in the KV-store, thus, leaving base table processing latencies unaffected. Finally, we conduct a comprehensive experimental study (Section V).

II. KV-store Model

In this section, we provide background information about KV-store internals that serve us in the remainder of this paper. We abstract from any particular store, focusing on the main concepts, grounding them in existing systems such as [2]–[6]. The section is organized as follows: we discuss the design of the KV-Store and its components; we depict the processing of a client operation through the KV-store architecture; we define a common data model for KV-stores; and we identify extension points that will later serve to connect KV-store to VMS.

A. Design

Some KV-stores are based on a master-slave architecture, i.e. HBase; other KV-stores run without a master, i.e. Cassandra (a leader is elected to perform tasks controlling the KV-store). In both cases a *node* represents the unit of scalability – as arbitrary instances can be spawned in the network (see Figure 2). The node persists the actual data. But in contrast to a traditional SQL-database, a node manages only part of the overall data (and request load). As load grows in the KV-store, more nodes can be added to the system; likewise, nodes can be removed as load declines. The KV-store will automatically adapt to the new situation and integrate, respectively drop the resource. KV-stores differ in how they accommodate; they also differ in how they perform load balancing and recovery (in face of node crashes). However, with regard to nodes, we can describe a set of universal events that occur in every KV-store (cf. Table I).

A *file system* builds the persistence layer of a node in a KV-store. For example, HBase stores its files in the Hadoop distributed file system (HDFS). Cassandra does not employ a distributed file system. Instead, its nodes access the local file system to store commit log and table files. Nevertheless, Cassandra also relies on its own replication mechanism. Replication enables the KV-store to keep its files available and safe in the presence of failures. If a node (and even its local file system) crashes, a copy on one of the replica nodes will be used to restore the 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 holds a variable number of *columns* (i.e., a set of column-value pairs). Columns can be further grouped into *column families*. Column families provide fast sequential access to a subset of columns. They are determined when a table is created and affect the way the KV-store organizes its table files.
Key ranges serve to partition a table into multiple parts that can be distribute over multiple nodes. Key ranges are defined as an interval with a start and an end row key. PNUTS refers to this partitioning mechanisms as tablets, while HBase refers to key ranges as regions. Multiple regions can be assigned to a node, often referred to as a region server. In general, a KV-store can split and move key ranges between nodes to balance system load or to achieve a uniform distribution of data. With regard to key ranges, we can also describe a set of universal events (cf. Table I).

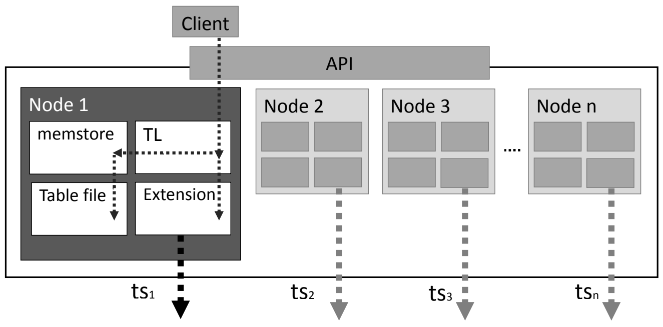

Fig. 2: KV-model

B. Operation processing

The KV-store API allows for three simple *client operations*: put, which inserts a record; get, which retrieves a record; and delete, which removes a record[1]. Now, we describe the path of a client operation, manipulating the data via put or delete statement, through the KV-store architecture.

The system routes the client request to the serving node. When updating a table record, clients only access a single node, i.e., the one that serves the accessed record. This reduces access time and distributes client requests among all system nodes. For example, HBase routes client requests from a root node down to the serving node in a hierarchical manner. Cassandra, on the other hand, lets the clients connect to an arbitrary node. This node forwards the request to the serving node. In both cases, the client eventually transmits its request to the serving node.

Every node maintains a *transaction log* (TL). In HBase it is called write-ahead log, whereas Cassandra refers to it as commit log. When a client operation arrives, it is first written into the transaction log of the node (cf. Figure 2). From then on, the operation is persisted durably – as the TL is stored and replicated at the file system that we discussed before.

Then, the operation is inserted into a *memstore*. Memstores are volatile; upon a node crash, they can be recovered from the TL, which contains the operation sequence a node saw over time. During recovery, this sequence is replayed from the log. 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. After a flush, the corresponding entries in the TL can be purged.

C. Data model

The data model of a KV-store differs from that of a relational DBMS. We describe a model that is representative

[1]There are additional methods, e.g. a range scan. However, none of them extends beyond repeated single row access; none of them offers expressive semantics.

for today's KV-stores. The model serves throughout the paper to help specify views and view update programs. Typically, KV-stores do not required fixed data schemas, but rather accommodate dynamic schema changes.

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

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

D. Extension points

Extension points are interfaces that can be used to interact with the KV-store. Since our VMS only reacts to the KV-store – as we don't interfere with its data processing – we are interested in events exclusively. There are two different kinds of events our VMS needs to react to: Administrative events and data events.

1) Administrative events: These events (Table I) occur during a state change in the KV-stores infrastructure, e.g. a new node is added to the KV-store (and starts processing client operations). The KV-store implementations provide different ways to react to administrative events. HBase lets developers use various kinds of Coprocessors. The latter represent little pieces of code that can be deployed on the KV-store nodes. Thus, we can modify a Coprocessor to notify our VMS, as soon as the new node is added. The VMS reacts accordingly and allocates resources to maintain the view tables that depend on the new node.

2) Data events: When the client updates a base table (put, get, delete) a data event occurs. The view tables become stale and the VMS needs to update them correspondingly. We identified three different methods to stream updates on base data from KV-store: (1) Access KV-API, (2) intercept KV-store operations (e.g. Coprocessors), (3) read the transaction log. Method (1) is not feasible, because we do not receive consistent snap-shots of the data – as records may change during the scanning process. Moreover, we create a lot of performance overhead. Method (2) is promising with regard to freshness of the view (as it is synchronous), but only suitable if the number of maintained views is small. As it grows large, operations of the KV-store are delayed to a great extent.

Component	Event	Method
Node	added	*onNodeAdded()*
	removed	*onNodeRemoved()*
Key-range	opened	*onKeyRangeOpened()*
	closed	*onKeyRangeClosed()*
	split	*onKeyRangeSplit()*
	moved	*onKeyRangeMoved()*

TABLE I: Administrative events

Method (3) as opposed to (2), has several benefits: (i) Reading the TL is asynchronous. It neither interferes nor imposes additional load on the KV-store processing. [3](ii) Maintaining numerous views at once means that every base table operation generates multiple view updates. Using the transaction log, we decouple the view update from the client operation (iii) Operations in the TL are durably persisted and can be recovered, not only by the KV-store, but also by VMS. (iv) The transaction log contains client operations, not table rows, which is fine for incremental and deferred view processing.

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

III. Consistency Model

Since consistency is a major concern during incremental view maintenance – additionally, we introduce a high degree of concurrency – we define a consistency model before building the VMS. In data warehouse environments, for batch-based view update processing, consistency models have been extensively explored (e.g., [11]–[13], [17]. For view maintenance

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

[3]Our experiment results showed that the penalty of reading from the file system are far smaller than intercepting events.

in KV-stores a model has been proposed in [18]. First, we adapt this model and the corresponding consistency levels to match the characteristics of KV-stores (defined in the previous section). Second, we define a theorem, capturing the requirements against our VMS, in order to achieve strong consistency.

A. Definition

A view data consistency model for incremental, deferred view maintenance explains whether the view state eventually converges, whether all intermediate view states are valid, and whether the sequence of view states produced by base table operations corresponds to the sequence of operations applied to base data [11]–[13], [17], [18]. Generally speaking, depending on view types, view maintenance strategies, and view update programs, none, some, or all of the listed properties are attainable [11]–[13], [17], [18].

In our context, we stream operations from multiple sources (i.e. nodes) and apply them to different parts of the view table, located on different nodes. We want to parallelize view maintenance and improve performance to a great extent, we further relax the consistency model. We do not claim the stream of a local source to be in sequence. Instead, we define consistency on a record base level. Every sequence of operations, that is applied to a specific row key has to be in sequence. A base state B_j is said to be greater than B_i if all operations to a specific row key in B_i are included in B_j plus a number of subsequent operations. Formally we write $(\exists t_k \in B_j)(\forall t_l \in B_i)\ k > l\ \wedge\ r(t_k) = r(t_l)$. Function $r()$ delivers, when applied to an operation, the row key of that operation. Now, we define consistency levels in agreement to the model. Formally, the consistency levels of the model are defined 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 base state $V_f = View(B_f)$. 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 base table state in the sequence of base table states produced by the operation sequence from which they can be derived

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 B_i and B_j that are also in a relation $B_i \leq B_j$.

Complete consistency: Strong consistency is achieved and every base state B_i of a valid base state sequence is reflected in a view state V_i. Valid base state sequence means $B_0 \leq B_i \leq B_{i+1} \leq B_f$.

B. VMS requirements

A system that is to achieve a given consistency level has to offer certain guarantees based on which the consistent and correct materialization of views can be based. We show that our approach can attain strong consistency for the views it maintains, which we capture in the following theorem.

Theorem 1: The following requirements are sufficient to guarantee that views are maintained *strongly* consistent in a VMS.

1) View updates are applied *exactly once*
2) view record updates are *atomic*
3) record *timeline* is always preserved

Due to the space constraints, we present the theorem without proof. We refer the reader to for the complete proof. Now, we explain every requirements and its implications in detail.

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

2) Atomic view record update: Most view types define a mapping from multiple base table records to a single view table record. Different base table records may be propagated to different VMs. When a view is updated it is possible that multiple VMs are concurrently updating the same view record The following example illustrates this situation.
Example 1: Given base table $A(\underline{K}, F)$ and a `SUM` view $S(\underline{X}, F)$, defined as $S = \gamma_{c_1, Sum(c_2)}(A)$. Assume a client inserts two records into base table A. The KV-store writes the following operations into the transaction log: $t_1 = put(k_1, \{\langle c_1, x_1\rangle, \langle c_2, a\rangle\})$ and $t_2 = put(k_2, \{\langle c_1, x_1\rangle, \langle c_2, b\rangle\})$. Let the VMS receive both operations and process them in parallel. To perform view maintenance, the VMS retrieves the corresponding old view records from the view table. Since S is a `SUM` view – and t_1 and t_2 refer to the same aggregation key x_1 – the VMS retrieves the same view record twice, e.g. $(x_1, \{\langle c_s, v_s\rangle\})$. In case of t_1 VMS adds the delta a to the view record; in case of t_2 VMS adds the delta b. Then, the VMS writes the updated view records back to the view table. Depending on which record is written first, the VMS overwrites one update with the other. Say, VMS writes $(x_1, \{\langle c_s, v_s + a\rangle\})$ to S; then it overwrites the result with $(x_1, \{\langle c_s, v_c + b\rangle\})$. In consequence the delta a is missing in the view. The correct result should be $(x_1, \{\langle c_s, v_s + a + b\rangle\})$. Therefore, atomic view updates are essential.

3) Record timeline: Record timeline means that sequences of operations on the same row key are not re-ordered when processed by VMS. Again, a little example demonstrates the importance of record timeline semantics.
Example 2: Given base table $A(\underline{K}, F)$ and a `SELECTION` view $S(\underline{K}, F)$, defined as $S = \sigma_{c_2 < x}(A)$. A client ex-

ecutes two operations on base table A. Operation t_1 inserts key k_1 to the table; operation t_2 updates key k_1. Since both operations touch the same key, KV-store writes them into the TL in sequence: $t_1 = put(k_1, \{\langle c_1, x_1\rangle, \langle c_2, a\rangle\})$ and $t_2 = put(k_1, \{\langle c_1, x_1\rangle, \langle c_2, b\rangle\})$. Assume that both operations match the selection condition (i.e. $(a < x)$ and $(b < x)$) such that both operations trigger an update on the view table. Let the VMS receive both operations and process them in parallel. Due to concurrency, the order of both operations could be reversed. First, the view record k_1 is updated with t_2, second with t_1. Then final state of the view record will be $(k_1, \{\langle c_1, x_1\rangle, \langle c_2, a\rangle\})$ while the base table contains $(k_1, \{\langle c_1, x_1\rangle, \langle c_2, b\rangle\})$. As we observe, breaking the record timeline may also violate convergence – which is a minimal requirement for view maintenance.

C. Proof of Consistency

Theorem 1 states that a VMS system fulfilling all three requirements achieves at least strong consistency. In the following, we will present a proof for this theorem, which is organized in three stages: we start with proving convergence and then present extensions to also prove weak consistency and finally strong consistency.

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, where $x \in X = \{1, \ldots, m\}$, and X is the table's key range. Further, let an operation on key k_x be defined as $t[k_x]$, and a totally ordered sequence of such operations be denoted by $\langle t_1, t_2, t_3, \ldots, t_N\rangle$, where N defines the total number of operations on the table in a given timespan. Hence, a generalized sequence of operations on a base table is represented by $\langle t_1[k_{x_1}], t_2[k_{x_2}], \ldots, t_n[k_{x_m}]\rangle$, where $k_{x_1}, \ldots, k_{x_m} \in X$ can be arbitrarily chosen from the base table's key range for every timestamp. Based on this generalized sequence every other sequence of operations can be derived, e.g. $\langle t_1[k_1], t_2[k_2], t_3[k_1]\rangle$. In some cases, we also want to keep the ordering of operations variable. For that reason, we introduce an index s_i with $i \in \{1, \ldots, n\}$. Then we can write the arbitrary sequence as $\langle t_{s_1}[k_{x_1}], t_{s_2}[k_{x_2}], \ldots, t_{s_n}[k_{x_n}]\rangle$ with $s_1 \neq \cdots \neq s_n$. Using this notion, we are capable of representing every possible sequence of update operations in the system.

The index $t^{(i)}[k_x]$ is used to express a sequence of operations on a single row key (i.e. the time-line). For example, a sequence of operations on row key k_x is denoted as $\langle t^{(1)}[k_x], t^{(2)}[k_x] ..., t^{(\omega)}[k_x]\rangle$. The last operation on a particular row key is always denoted with ω. (see Section III).

For the proof, we assume such an arbitrary sequence of base table operations and then we show that — given the requirements in the theorem — a VMS system will produce correct view results. Formally, we show that $V_f = View(B_f)$.

2) Convergence: As mentioned earlier, every view type defines its own mapping from base table to view table records. Thus, we prove the different cases separately.
One-to-one mapping: `SELECTION`, `PROJECTION` views define a one-to-one mapping between base and view table. The row key of the base table is also the row key of the view table. Operations for both view types are idempotent, meaning an operation could be applied several times without changing the result. The view record is always a representation of the last base table operation applied. A correct view record with row key k is defined as the last operation on the row key in the base table key, e.g. $k \leftarrow View(t^{(\omega)}[k])$. The function $View$ calculates the view record (by using the appropriate view definition) and applies the result to the correct view key. A view table converges, if all view records are computed correctly in the last view state.

We start defining an arbitrary sequence of operations on the base table, shown in Step 1a. Clients can update different row keys in the base table, using put or delete operations. Likewise, they can update the same row key multiple times. The update operations of the clients form a particular global order, expressed through $t_1..t_n$. They take the base table from its initial state B_0 to its final state B_f. In the next step, all update operations get forwarded, causing the ordering of operations to be lost. If we would continue working with an unordered set, convergence of the view will be violated at some point. For this reason, we apply requirement (iii) (,i.e. the time-line requirement) to our equation, as depicted in Step 1b. As updates operations do not influence each other (see requirement (ii)), we are able to create a set of sub-sequences. These sub-sequences only consist of updates that have been applied to the same row key. Likewise the sub-sequences contain all operations from the previous step (see requirement (i)).

$$S_b = \langle t_1[k_{x_1}], ..., t_n[k_{x_n}]\rangle; \ \left(B_0 \overset{S_b}{\to} B_f\right) \tag{1a}$$

$$S_1 = \langle t^{(1)}[k_1], .., t^{(\omega_1)}[k_1]\rangle, .., S_m = \langle t^{(1)}[k_m], .., t^{(\omega_m)}[k_m]\rangle \tag{1b}$$

$$S_1 = \langle t^{(\omega_1)}[k_1]\rangle, .., S_m = \langle t^{(\omega_m)}[k_m]\rangle \tag{1c}$$

$$S_v = \langle t^{(\omega_1)}_{s_1}[k_1], ..t^{(\omega_m)}_{s_n}[k_m]\rangle; \ \left(V_0 \overset{S_v}{\to} V_f\right) \tag{1d}$$

$$V_f = k_x \leftarrow View(t^{(\omega_x)}[k_x]) = View(B_f) \tag{1e}$$

In Step 1c, we pick the last element of all sub-sequences and eliminate the rest. As stated above, only the last operation on a particular row key has an influence on the final view state. Again, we observe that the time-line of a row key is vital. If it is broken, e.g. for row key k_1, then an operation $t^{(\omega-1)}[k_1]$ can be incorporated into the final result and render it incorrect. After the elimination, we unite the operations again in Step 1d. The reunion allows the remaining operations to be executed in every possible order (i.e. every operation could be executed in parallel). Finally, the view definition is applied to every operation in R. This leads to the correct final view records and hence, to convergence. The `DELTA` view also defines a one-to-one mapping between base and view records. In contrast to the aforementioned views, the results of the `DELTA` view do not only relate to the last, but to the two last operations. Therefore, we need to change the last three steps of the proof as follows:

$$S_1 = \langle t^{(\omega_1 - 1)}[k_1], t^{(\omega_1)}[k_1]\rangle, .., S_m = \langle t^{(\omega_m - 1)}[k_m], t^{(\omega_m)}[k_m]\rangle \tag{2a}$$

$$S_v = \langle t_{s_1}^{(\omega_1 - 1)}[k_1], t_{s_2}^{(\omega_1)}[k_1], ..t_{s_{n-1}}^{(\omega_m - 1)}[k_m], t_{s_n}^{(\omega_m)}[k_m]\rangle \tag{2b}$$

$$(\forall t_{s_i}^{(\omega_y - 1)} \in S_v)(\exists t_{s_j}^{(\omega_y)} \in S_v)\ s_i < s_j;\ \left(V_0 \overset{S_v}{\rightarrow} V_f\right)$$

$$V_f = k_x \leftarrow View(t^{(\omega_x - 1)}[k_x], t^{(\omega_x)}[k_x]) = View(B_f) \tag{2c}$$

As can be observed in Step 2b, the two last operations of a time-line are included into the final result. However, the sequence can be arbitrarily ordered, but needs to preserve the time-line of both last operations (i.e. $\omega - 1$ must always precede ω). Computing V_f leads to the valid final state and the DELTA view converges.

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 $k_{x_1}..k_{x_j}$, related to the particular key. In terms of incremental view maintenance, the correct view record can be defined as a number of last operations, that have been applied to this combination of base records: $x \leftarrow View(t^{(\omega_1)}[k_{x_1}], .., t^{(\omega_j)}[k_{x_j}])$. In case of a SUM view e.g., this resolves to $x \leftarrow f(t^{(\omega_1)}[k_1]) + .. + f(t^{(\omega_j)}[k_j])$. We start again, defining an arbitrary sequence of base table operations in Step 3a. In contrast to the previously handled views, we are now processing δ-operations. We construct a number of m sub-sequences, each containing the δ-operations of one particular base record key. In Step 3b, we merge the δ-operations together. All δ- operations add up to form the last transaction as the end result (i.e. $\delta(t^{(1)}[k_1]) + .. + \delta(t^{(\omega_1)}[k_1]) = t^{(\omega_1)}[k_1]$).

$$S_b = \langle t_1[k_{x_1}], ..., t_n[k_{x_n}]\rangle;\ \left(B_0 \overset{S_b}{\rightarrow} B_f\right) \tag{3a}$$

$$S_1 = \langle \delta(t^{(1)}[k_1]), ., \delta(t^{(\omega_1)}[k_1])\rangle, .., \tag{3b}$$

$$S_m = \langle \delta(t^{(1)}[k_m]), ., \delta(t^{(\omega_m)}[k_m])\rangle$$

$$S_1 = \langle t^{(\omega_1)}[k_1]\rangle, .., S_m = \langle t^{(\omega_m)}[k_m]\rangle \tag{3c}$$

$$S_v = \langle t_{s_1}^{(\omega_1)}[k_1], ..t_{s_n}^{(\omega_m)}[k_m]\rangle;\ \left(V_0 \overset{S_v}{\rightarrow} V_f\right) \tag{3d}$$

$$V_f = x \leftarrow View(t^{(\omega_{x_1})}[k_{x_1}], .., t^{(\omega_{x_j})}[k_{x_j}]) = View(B_f) \tag{3e}$$

Now, we can unite the single sequences as done before. We retrieve a final sequence as shown in Step 3d. The operations of this sequence are then applied to the view — simultaneously they are grouped and stored according to their aggregation key. The final view records are calculated correctly, as depicted in Step 3e, which causes the aggregation view to converge.

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 key. 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 $k_1..k_n$ from join table A and operations on keys $l_1..l_p$ from join table B. In order to represent both keys we introduce an additional variable $z_1, .., z_n \in \{k_1, .., k_m, l_1, .., l_p\}$. Then, the correct view record is defined as: $x \leftarrow View(t^{(\omega_1)}[z_1], .., t^{(\omega_j)}[z_j])$. We start with a sequence of arbitrary client updates to both base tables, as depicted in Step 4a. Then, the order of updates is lost and the time-line requirement is realized in Step 4b.

$$S_b = \langle t_1[z_1], .., t_n[z_n]\rangle;\ \left(B_0 \overset{S_b}{\rightarrow} B_f\right) \tag{4a}$$

$$S_1 = \langle \delta(t^{(1)}[k_1]), .., \delta(t^{(\omega_1)}[k_1])\rangle, .., S_m, .., S_{m+1}, .. \tag{4b}$$

$$S_{m+p} = \langle \delta(t^{(1)}[l_p]), .., \delta(t^{(\omega_p)}[l_p])\rangle$$

$$S_1 = \langle t^{(\omega_{k_1})}[k_1]\rangle, .., S_m = \langle t^{(\omega_{k_m})}[k_m]\rangle, .., \tag{4c}$$

$$S_{m+1} = \langle t^{(\omega_{l_1})}[l_1]\rangle, .., S_{m+p} = \langle t^{(\omega_{l_p})}[l_p]\rangle$$

$$S_v = \langle t^{(\omega_{k_1})}[k_1], ..t^{(\omega_{k_m})}[k_m], \tag{4d}$$

$$t^{(\omega_{l_1})}[l_1], .., t^{(\omega_{l_p})}[l_p]\rangle;\ \left(V_0 \overset{S_v}{\rightarrow} V_f\right) \tag{4e}$$

$$V_f = x \leftarrow View(t^{(\omega_{z_1})}[z_1], .., t^{(\omega_{z_j})}[z_j]) = View(B_f) \tag{4f}$$

We eliminate all but the last operations ω and reunite the operations in Step 4e. This leads to the final Step 4f, where the operations are applied to the view record. Since the view records are calculated correctly (i.e. only the last operations of the row keys are included) we conclude that the view converges.

3) Weak consistency: Weak consistency has been defined as follows: Weak consistency is given if the view converges and all intermediary view states are valid, meaning they can be derived from one of the base states with $V_j = View(B_i)$. As we already proved convergence, we need show that all the intermediary view states are correct likewise. We start again with an arbitrary sequence of operations in Step 5a. In order to generate an intermediate base state, we cut the sequence at any point before an operation t_a, with $1 < a < n$. After the ordering is lost, we apply the time-line consistency. But in contrast to before, we are not capable of processing the complete time-line (i.e. $(1)..(\omega)$). Instead, we process the timeline until an intermediary element $\alpha_x \leq \omega_x$.

$$S_b = \langle t_1[k_{x_1}], ..., t_n[k_{x_n}] \rangle; \; \left(B_0 \overset{S_b}{\rightarrow} B_a \right) \tag{5a}$$

$$S_1 = \langle t^{(1)}[k_1], .., t^{(\alpha_1)}[k_1] \rangle, .., S_m = \langle t^{(1)}[k_m], .., t^{(\alpha_m)}[k_m] \rangle \tag{5b}$$

$$S_v = \langle t^{(1)}_{s_1}[k_1], ..t^{(\alpha_1)}_{s_i}[k_1], .., t^{(1)}_{s_j}[k_m], .., t^{(\alpha_m)}_{s_a}[k_m] \rangle; \; \left(V_0 \overset{S_v}{\rightarrow} V_a \right) \tag{5c}$$

$$(\forall t^{(\lambda_1)}_{s_1}[k_{x_1}] \in S_v)(\forall t^{(\lambda_2)}_{s_2}[k_{x_2}] \in S_v) : (x_1 = x_2) \; \wedge \; (\lambda_1 < \lambda_2) \Rightarrow s_1 < s_2$$

$$V_a = k_x \leftarrow View(t^{(\alpha_x)}[k_x]) = View(B_\alpha) \tag{5d}$$

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 Bi 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 negated, 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 time-line. To be precise: At least one record has to exists, whose time-line is broken. Formally, we demand $(\exists t_l \in B_i)(\forall t_k \in B_j) : (r(t_l) = r(t_k)) \wedge (l > k)$. Because requirement (iv) prevents the breaking of time-lines, we conclude that both cases are not possible. Thus, we have proven strong consistency by contradiction.

IV. View Maintenance System

In this section, we build and integrate the VMS with KV-store – always keeping consistency requirements in mind. The section is organized as follows: we discuss the design of the VMS and its components. We depict the processing of an operation through the VMS architecture. We analyse consistency of VMS in a dynamic context. Finally, we discuss fault tolerance mechanisms provided by VMS.

A. Design

Figure 3 provides a system overview. The VMS consists of a *coordinator* and an arbitrary number of *subsystems* (Sub). The coordinator controls load balancing and recovery mechanisms, whereas the subsystems – more specifically the *view managers* (VMs) in the subsystems – update the view tables. The input of the VMS is a set of operation streams (ts_1,ts_2..ts_n); each resulting from a KV-store node (cf. Figure 2). Thereby, one subsystem of VMS manages one stream of operations exclusively. The subsystem distributes incoming operations to the view managers. The number of view managers in a subsystem is scalable. New view managers can be *assigned* to or *removed* from the subsystem. The coordinator can also re-assign view managers from one subsystem to another.

The **view manager** is light-weight and deployed in large numbers to accommodate the view update load. It applies base table operations to view tables. A view manager can only belong to a single subsystem. The subsystem feeds the view manager with operations; the view manager processes the operations in order. A view manager is the unit of scalability for VMS. We keep VMs stateless – they should be exchangeable at any time – To minimize dependency. Given a number of view definitions and a sequence of operations, the view manager is always able to execute any view table update at any host.

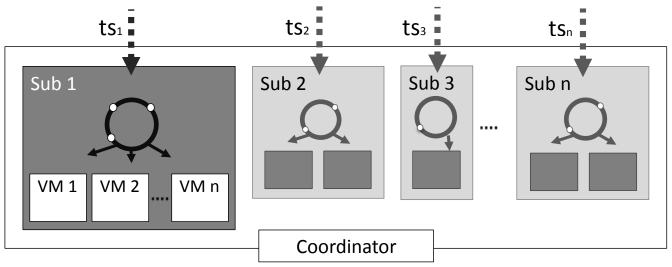

Fig. 3: View Maintenance System

Our design bares the following benefits: (i) Hundreds of views may be updated in consequence of one base table operation. As VMS exceeds its performance limits, additional VMs can be assigned (below, we show this experimentally). (ii) VMs introduce flexibility to the system architecture: VMs of a subsystem can be hosted together on the same node; or each of them can be hosted at a different node. (iii) Also, VMs can be reassigned from one subsystem to another in times of changing base table update load. (iv) If a VM crashes, another VM can take over and continue processing the operation stream.

The *coordinator* manages the component configuration of VMS. It sends commands to the remaining components and induces certain actions. For example, if a VM has to be assigned to a node, the coordinator sends an assignment command. Likewise, the coordinator reacts to system events and takes care of the system state. It monitors the VMS state by relying on the distributed configuration service *Zookeeper*. Every VM instance is represented in Zookeeper by an *ephemeral node*. In case of a VM crash, the ephemeral node is destroyed and the coordinator receives an event. It then performs different recovery steps to guarantee error recovery.

B. Operation processing

When receiving operations from the stream, the subsystem distributes them to the set of assigned view managers. Thereby, it applies the principles of *consistent hashing* to route the operations. Routing the operations allows for maximal concurrency, while simultaneously guaranteeing timeline consistency (cf. Section III-B3). The subsystem will distribute operations uniformly – but it will send multiple operations on the same row key always to the same view manager.

Consistent hashing is realized as follows: the subsystem maintains a hash-ring (cf. Figure 4) where it places a node for every assigned view manager (it applies a hash-function to the view managers name to determine the place). As the base table operations stream in, the subsystem extracts their row keys. Now, the subsystem applies the hash-function to the row key and determines the place on the hash-ring. Starting at this point, the subsystem travels clockwise until it hits the first node of a view manager – meaning this view manager is responsible for processing. The subsystem sends the operation to this view manager and continues its work.

Similar to the KV-store, the view manager runs its own transaction log, called *VM log*. Before processing the operations, the view manager writes them to the VM log, located at the file system. Here the VM log is replicated and can be used for recovery as soon as the view manager crashes.

To access and update the view table, the view manager acts as a client to the KV-store, using the standard KV-API. Given a client operation, the VM retrieves the view definitions of the views defined over the updated base table, runs the update programs, and submits view table updates via the KV-API. All view table definitions are stored in a meta-table in the KV-store keyed on base table name. View table definitions are cached by a VM. Once the VM knows the affected view tables and their definitions, it runs the view update logic and submits view table changes via the KV-API. Depending on the view type maintained, the VM has to query the view table as part of the update logic, which we detail below.

To full fill the atomic update requirement (cf. Section III-B2), we use a test-and-set mechanism. [4] When updating a table record, a caller sees (tests) whether a record has been concurrently modified between a read and an update. Revisiting Example 2, let VM_2 retrieve value $(x1, \{(col_1, a)\})$. Then, it computes $(x1, \{(col_1, a + c)\})$ trying to put the new value, while testing for the old value a. The test-and-set fails because the old record value changed concurrently to $a + b$. Thus, VM_2 fetches the updated value again and re-computes $(x1, \{(col_1, a + b + c)\})$. This time the test-and-set succeeds and the record is written.

After a view table has been updated, the VM persists the current state of work – that is, the *sequence number* of the last processed client operation and the name of the updated view table. The state is stored into a field in Zookeeper, such that the coordinator can restore the information in case of a crash. Then, a new view manager can resume the work from this point.

C. Dynamic Consistency

By now, consistency is guaranteed through the design of VMS. Consistent hashing ensures the record timeline during maintenance. But the VMS achieves these guarantees only in a static set-up – a fixed number of nodes and view managers is assumed. Switching to a dynamic context (and allowing the VMS to assign and withdraw view managers; or allowing the KV-store to add and remove nodes, move key-ranges, etc.) makes securing consistency difficult again. The following example shows why:

Example 3: Consider a subsystem set-up as shown in Figure 4. Two view managers VM_1 and VM_2 are already assigned to the subsystem: they are responsible for a certain range on the hash-ring; queues are feeding them with incoming client operations. Assume a client performing a put operation $t_1(A(k_1, \{..\}))$ to a base table that is managed by the view managers. Further assume the subsystem selects VM_2 as the next responsible view manager for t_1. Then, t_1 is put to the queue of VM_2. Now assume, a view manager VM_3 is assigned to the subsystem and that VM_3 acquires the responsibility for key k_1. It is inserted into the hash-ring and a new queue is created. In a next step, a client sends an operation $t_2(A(k_1, \{..\}))$ to the same key. Because responsibility has changed, the operation is now added to the queue of VM_3. Considering, that VM_3 has just been added, it's queue is comparatively empty and hence processing very fast. It is likely to happen that VM_3 processes t_2 before VM_2 can process t_1. Because both operations refer to the same base record, the timeline of the record is broken. As explained before, this has to be prevented in order to preserve convergence of views.

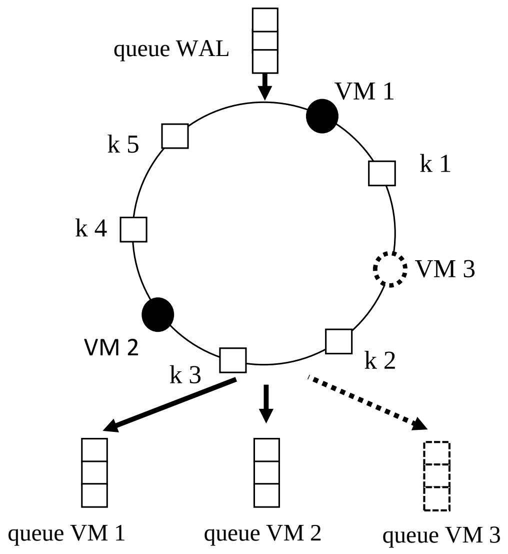

Fig. 4: Subsystem set-up

Thus, we now refine the behaviour of VMS in a dynamic context. First, we discuss the dynamics related to VMS actions (Table II); second, we discuss the dynamics related to KV-store events (Table I).

1) Dynamics VMS: In Table II all VMS actions are shown. Some of them are not relevant to the consistency discussion. Adding a view manager to VMS (or removing it from VMS) does not affect the view maintenance process. The reassign action is a logical combination of a withdraw (from the old subsystem) and an assign (to the new subsystem). Thus, we now refine the assign and withdraw action of VMS.

The solution we suggest, is using so called *markers*. Markers identify whether a view manager has completed the processing of all operations (that were sent before) and, thus, allow the secure assignment and removal of view managers. Markers are sent just like client operations to the view manager: they are inserted into the queue of the view manager and

[4] In HBase a `checkAndPut` method is provided to realize this mechanism. Most KV-stores offer a similar abstraction.

Component	action	method
View Manager	add	*addViewManager()*
	remove	*removeViewManager()*
	assign	*assignViewManager()*
	withdraw	*withdrawViewManager()*
	re-assign	*reassignViewManager()*

TABLE II: VMS actions

become a part of the operation stream. When the view manager reads an operation and recognizes a marker, it replies with an acknowledgement back to the subsystem.

To realize the marker mechanism, we adapted parts of the subsystem and VM implementation. The logic on the subsystem side is slightly more subtle and in the following we base our description on a set of primitive operations. These include methods that are needed by the subsystem to assign (or withdraw) VMs to (or from) itself: (i) $createQueue(vm)$ creates a new queue for a particular view manager and (ii) $deleteQueue(vm)$ removes an existing queue for a particular view manager at the subsystem. The queue for a VM can only be deleted, if it is empty and no operation is queued. (iii) $queue(vm, operation/marker)$ inserts, either a base table operation, or a marker into the queue of a particular view manager. The methods (iv) $activateQueue(vm)$ and (v) $deactivateQueue(vm)$ start or stop the sending thread that keeps transferring the operations in the queue for a particular VM. A view manager can be added to the hash ring by (vi) $insertHash(vm)$, or removed from the hash ring by (vii) $removeHash(vm)$.

Assign view manager – When a view manager is assigned, it is added to the hash-ring of the subsystem. Unless the hash-ring is empty, the new view manager is assigned a key region that is, at the same time, withdrawn from another view manager. In Figure 4, a view maintenance process of a subsystem is shown. Recall, that the subsystem selects the responsible view manager by applying the consistent hash function. The operation is then inserted into the corresponding queue.

We change the assignment procedure by adding the marker-based acknowledgement mechanism (cf. Algorithm 1). The algorithm is executed synchronously and if another assignment procedure is called on the subsystem, it has to wait, until the first operation has terminated.

The procedure $assignVm$ takes two parameters: vm_a, the view manager that should be assigned to the subsystem, and VM_{sub}, a set of view managers that are already assigned to the subsystem. The algorithm creates a queue for vm_a and inserts it into the hash-ring. Then, it queues a marker m_a to all assigned VMs (VM_{sub}). After the subsystem has received acknowledgements from all VMs, it is guaranteed that no operation in the key range of the newly added view manager vm_a is still pending. Referring back to Example 3: The timeline of k_1 can not be changed any more. Operation t_2 has to wait in the queue of VM_3, until VM_2 acknowledges the processing of t_1 because the queue of VM_3 is activated

Algorithm 1 Secure assignment procedure at subsystem

```
    procedure $assignVm(vm_a, VM_{sub})$
        $createQueue(vm_a)$                        ▷ create queue
        $insertHash(vm_a)$                         ▷ add VM to hashring
        for all $vm \in VM_{sub}$ do
5:          $queue(vm, m_a)$                       ▷ queue markers
        end for
        for all $vm \in VM_{sub}$ do               ▷ wait for acks
            $receiveMessage(vm)$
        end for
10:     $activateQueue(vm_a)$                      ▷ activate queue
    end procedure
```

only after the marker has been acknowledged and this, in turn, implies that operation t_1 has been processed.

Withdraw view manager – During a withdraw procedure, consistency can be violated, likewise. At the moment where a view manager is withdrawn, i.e. removed from the hash-ring, its queue might still contain operations. If another view manager that acquires the key range is fast enough, it might processes operations before the withdrawn view manager has finished. Again, the timeline of base records is changed. In order to prevent inconsistency, we design Algorithm 2 analogous to Algorithm 1. It takes two parameters: vm_w, the view manager that should be withdrawn from the subsystem, and VM_{sub}, the set of view managers that is assigned to the subsystem.

Algorithm 2 Secure withdraw procedure at subsystem

```
    procedure $withdrawVm(vm_w, VM_{sub})$
        for all $vm \in VM_{sub} \wedge vm \neq vm_w$ do
            $deactivateQueue(vm)$                  ▷ deactivate queues
        end for
5:      $removeHash(vm_w)$                         ▷ redirect operations
        $queueMarker(vm_w, m_w)$
        for all $vm \in VM_{sub}$ do
            $receiveMessage(vm)$                   ▷ wait for acks
        end for
10:     $removeQueue(vm_w)$
        for all $vm \in VM$ do                     ▷ activate queues again
            $activateQueue(vm)$
        end for
    end procedure
```

First, the queues of the view managers that possibly increase their key range on the hash-ring (i.e., all other VMs) are deactivated. Then, a marker m_w is queued to the view manager that is withdrawn. If the view manager has acknowledged the marker, the region server knows, that all operations have been processed. It removes the queue of the view manager and re-activates the queues of all view managers.

2) Dynamics KV-Store: In Table II all KV-store events are depicted. Again we can exclude some of them from the consistency discussion. Adding a node to KV-store is not critical. The VMS just creates a subsystem for the new node and assigns view managers to it – which start processing the client operations of the new node. Removing a node from KV-store is also not critical, as the key ranges on that node are either closed or moved to another node. The VMS continues reading client operations until the end of the node's transaction log (as the file is still available) and then, securely reassigns the view managers to another subsystem.

Moving regions – Once in a while, the KV-store moves key ranges from one node to another node. This can be for load balancing or for recovery reasons. Independent of the reason, consistency might be violated in such cases. The operations of a key range can be processed by multiple view managers. If a key range is moved, operations can still linger in the queues of the old node's view managers. Now, the key range is opened again and the new node's view managers start their work. In consequence new operations may overtake old operations and the timeline of individual records may be broken.

Like the preceding scenarios, also this problem can be addressed by using markers. However, in contrast to before, the transfer of a key range is executed by the KV-store itself. The KV-store *closes* the key range on the old node, *moves* the key range to the new node, and *opens* it again. The VMS needs to be informed about these steps by the corresponding events. If a *key range closed* event is called on the subsystem of the old node, it reacts as follows: it creates a flag in Zookeeper with the (hash-)name of the key range and sets its value to false. Then, the subsystem sends markers to all its view managers. After receiving the acknowledgements, it sets the value of the flag to true.

The subsystem on the new node receives a *key range opened* event and checks Zookeeper for the appropriate flag. If the flag does not exist, the key range is opened for the first time. However, if the flag exists, the subsystem inserts a deactivate operation into the queue (instead of deactivating it directly). All existing operations are processed, until the sending thread encounters the deactivate operation. If the subsystem would deactivate the view manager queues directly, there would be the prospect of a deadlock. By moving two key ranges from old to new subsystem and v.v., the VMS would deactivate all sending queues and prevent the markers from being sent. The subsystem continues checking the flag until it evaluates to true (or a time-out expires). Then, it re-activates the queues and resumes normal operation mode.

Deleting transaction logs – Another dynamic that needs to be discussed is the deletion of log files. Usually, the KV-store node runs a background process to merge or delete log files. Although, possibility is small, there is a chance of a transaction log getting deleted before the VMS can read all the operations from it. Then, convergence fails, as some updates are missing. To solve the problem, the deletion mechanism of KV-store has to be overwritten (HBase e.g., offers plug-ins for that purpose). Then, the log deletion can be delayed to a point where all operations have been retrieved from transaction log.

D. Fault tolerance

Failure detection and recovery play a critical role, especially in large-scale distributed systems. In this section, we analyze the behavior of VMS under VM and node crashes to ensure that after appropriate recovery measures, views still converge.
VM crash – A VM maintains a queue with operations dispatched to it. During a VM crash, these operations are lost, which may result in non-converging views. Our recovery measures described here, ensure view convergence under VM crash. A VM crash triggers an event via Zookeeper, notifying the VMS coordinator.

First, the coordinator sends a withdraw command to the concerned subsystem. The subsystem withdraws the crashed VM from the hash-ring and stops dispatching operations to it. This way no updates, that were in-fight while the VM crashed, are lost. Next, the coordinator starts a new view manager instance. Upon start-up, it tells the new VM to replay the VM log of the crashed view manager. The new VM contacts Zookeeper and retrieves the last processed sequence number of the crashed view manager (cf. Section IV-B). The new VM accesses the VM log of the crashed VM and – Starting from the sequence number – replays all the entries (and updates the views correspondingly). We conclude that our VMS is able to prevent loss and duplication of operations during crashes. Thus (and because we assume reliable communication) the VMS full fills the exactly-once requirement (cf. Section III-B1).
Node crash – A node crash is handled by the recovery mechanism of the KV-store. The KV-store moves all key ranges of the crashed node to other nodes. In case, client operations exist that were just residing in the memstore (and had not been written to the table file), the KV-store replays the transaction log. During replay, all the operations are inserted into the memstore and flushed to disk, directly.

The transaction log of a crashed node is still available (due to replication). Thus, the subsystem that is streaming the operations from the crashed node's TL continues reading until the end of file. As the KV-store starts moving key ranges to different nodes, the VMS reacts as described in Section IV-C2 – consistency is guaranteed during the process. Now, that the stream (of the crashed node's TL) runs dry, the VMS re-assigns all view managers to a different subsystem.

E. View types

Our approach supports the maintenance of the following basic view types: `SELECTION`, `PROJECTION`, `INDEX`, aggregation (e.g., `COUNT`, `SUM`, `MIN` and `MAX`), and general `JOIN` views. More complex queries are translated into a DAG of view expressions, where the output of one view is connected to the input of the next one.

We also introduce a number of auxiliary view types, such as the `DELTA`, `PRE-AGGREGATION` and `REVERSE-JOIN` view, which are designed to support fast and efficient view maintenance. The `DELTA` view tracks base table changes between successive update operations. The `PRE-AGGREGATION` view allows for materialization of multiple aggregation views without further cost; and it avoids table scans for MIN and MAX views (in case a minimum/maximum gets deleted). The REVERSE-JOIN view pre-joins table entries by join-key and allows for materialization of multiple join views(INNER, LEFT, RIGHT, FULL) without further cost; likewise, it avoids table scans to find join partners.

Our approach avoids table scans at all cost (even at the expense of higher storage cost), as they bare the following drawbacks: (i) Scans require a disproportional amount of time, slowing down throughput of view maintenance. (With increasing table size, the problem worsens.) (ii) Scans keep the nodes occupied, slowing down client requests. (iii) While a scan is in progress, underlying base tables may change, thus, destroying view data consistency for derived views.

V. 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 experiments, we briefly review the experimental set-up and the generated workload.

A. Experimental set-up

All experiments were performed on a cluster comprised of 40 nodes (running Ubuntu 14.04). Out of these, 11 machines were dedicated to the Apache Hadoop (v1.2.1) installation, one machine as name node (HDFS master) and 10 machines as data nodes (HDFS file system). On HDFS, we installed HBase (v0.98.6.1) with one master and 10 region servers. On every region server, we deployed one of our extensions (cf. Figure 2). View managers (VMs) were deployed on 20 separate machines to be able to scale them without interfering with the core system (i.e., the 12 HBase nodes.) Also, running multiple VMs on the same machine is possible, since, as we observed, the limiting factor for a VM is the access latency to HBase (i.e., request processing latency). Finally, another 8 nodes were reserved for HBase clients to simulate the load on the base tables by constantly issuing updates.

B. Workload

Prior to each experiment, we created an empty base table and a set of view tables. Concrete view definitions as well as the assignment of view tables to base tables 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 split every table into 50 parts – which lets HBase balance the regions with high granularity and ensures an uniform distribution of keys among available region servers.

For COUNT, SUM, MIN, and MAX 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. Thereby, it is possible to vary the number of base table records that affect one particular view table record. For the SELECTION view, we use the same base table layout and apply the selection condition to column c_2. The JOIN view requires 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.

Basically, the workload we generate for our experiments consists of *insert*, *update* and *delete* operations that are issued to HBase using its client API. Operations are generated according to different distributions over the key space (we use Zipf and Uniform). A Zipf distribution, for instance, simulates a "hot data" scenario, where only few base table records are updated very frequently.

C. Experiments

With our experiments, we primarily evaluate the performance of the system with regards to throughput and view maintenance latency.

1) Impact of view type on throughput: First, we evaluate the performance for every view type, separately. We measure the throughput during view maintenance, i.e., the number of updates a view can sustain per second. We configure a system with a fixed number of 10 region servers to host all tables. The number of VMs varies between 10 to 50, and all VMs are assigned evenly to region servers. Furthermore, 40 clients generate a total of 1 million operations per experiment. Updates are concurrently processed by all VMs. The experiments are completed after all clients sent their updates and all VMs emptied their queues.

For the SELECTION view, we experimented with three different selectivity values(i.e., selection of 10, 50 and 90 percent of base records), while varying the number of VMs that process the update load. Figure 5 shows the results. The SELECTION view is not realized with an auxiliary view. Compared to the other views, its maintenance results in the highest throughput. The performance depends on the amount of records selected. Interesting is that the absolute throughput is limited by the throughput clients exert on the system. In Figure 5, the performance is monotonically decreasing for a selectivity of 10%. This means, VMs propagate updates as fast as they are applied to HBase by clients. Increasing the number of view managers only slows down the system as more components are running concurrently.

Figure 6 shows the performance of the aggregation view types: COUNT, SUM, MIN, MAX, and INDEX. Again, we use a fixed configuration comprised of 10 region servers and 40 clients that generated 1M operations. Aggregation views derive from an auxiliary view (here a DELTA view.) Therefore, the throughput for aggregation views is lower than for the SELECTION view. However, in contrast to the SELECTION view, the throughput significantly benefits from increasing the number of VMs. Not surprisingly, COUNT and SUM views show similar performance, as their maintenance is nearly identical. INDEX view maintenance exhibits a lower throughput than COUNT and SUM, which only store one attribute for the aggregated value, whereas the INDEX view stores a variable number of primary keys associated with the index column value of the indexed table. The performance of both MIN and MAX is worse compared to the INDEX view. In a MIN (max) view, we also store base table records, together with

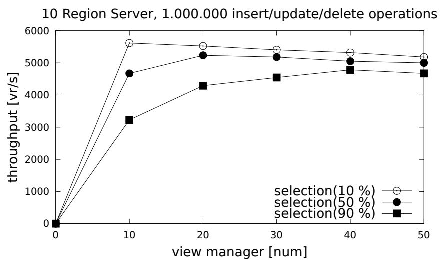

Fig. 5: Selection performance

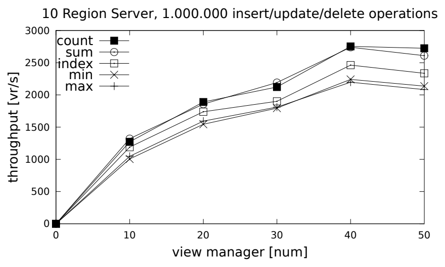

Fig. 6: Aggregation performance

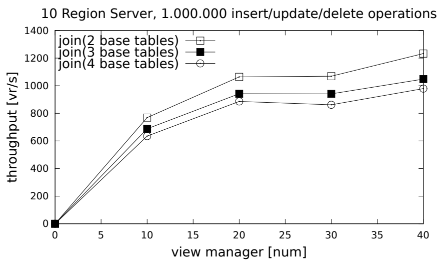

Fig. 7: Join performance

the aggregated value. The storage overhead is equal to INDEX, but sometimes the values of an entire row needs to be queried to recalculate the new minimum (maximum).

Figure 7 shows the evaluation results for the JOIN view under the same conditions as above. Compared to the other view types and not surprising, the JOIN shows lower throughput. The JOIN view requires a more complex internal auxiliary view table constellation and maintenance. However, we suspect that throughput is still higher than as if full table scans would have to be used to find matching rows (not considering consistency issues, if scans would be used.) Similar to aggregation views, JOIN benefits from an increased number of view managers. Also, here as well, auxiliary tables for JOIN can be amortized as more views are defined.

2) Costs and benefit of view maintenance: To determine the benefits, i.e., the latency improvement a client experiences when accessing a view, and the costs, i.e., essentially, the decrease in overall system throughput that results, we conducted an experiment with three different view maintenance strategies: (i) *Client table scan:* To obtain the most recent count view values, a client scans the base table and aggregates all values on its own. (ii) *Server table scan:* To obtain the most recent count view values, the client sends a request to HBase, which internally computes the view in a custom manner and returns it as output. In the implementation, we use HBase's ability to parallelize table access. All region servers scan their part of the table and, at the end, intermediate results are collected and merged. (iii) *Materialized view:* Views are maintain incrementally by VMS configured with 40 VMs used to materialize the count view in parallel. For (i) and (ii), we disregard inconsistencies that may result from concurrent table updates while scans are in progress. Our primary objective is to obtain some understanding of how VMS fares relative to other potential approaches.

The results are given in Figure 8, where we measure the latency a client perceives (i.e., the time to wait for the result) as the scanned key range increases. The first strategy performed worst. *Client table scans* are sequential by nature and require a large number of RPCs to HBase, even if requests are batched. Especially with increasing table size, this approach becomes more and more impracticable.

Server table scan is promising 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 aggregate reached 5 s for a table with 1M rows. While this result could now be cached in materialized form, in scenarios with frequently changing base data, recalculations would have to be frequently repeated to keep the results up-to-date. Moreover, concurrent table scans may interfere with the write performance of region servers.

Because views are materialized and updates are done incrementally, latencies for the third strategy are exactly the same as for every HBase table. 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.

Materializing views comes at a cost. We assess this cost as the performance impact on base table creation time (here, defines as the time to insert 1M tuples into the base table.) Figure 11 shows the base table creating time with and without concurrent view maintenance. We track creation time as the number of clients increases. Figure 11 shows view maintenance with 20, 30 and 40 VMs. While view maintenance increases the base table creation time by approximately 40%, a further increase of VMs (by ten) only increases base table creation time by approximately 2%.

3) System scalability: In Figure 5 and Figure 6, we evaluated the performance of different view types. In both experiments, we increased the number of VMs. Now, we examine system performance when scaling up region servers. In Figure 9, the number of region servers is varied from 4 to 10. We run the experiment with a selection, a count and an index view and measure throughput. We observe an almost linear increase of throughput independent of the view type processed.

The effect can be explained as follows: Each region server runs on a separate node. Adding another region server results in additional I/O channels (i.e., separate disk). Thus, the overall performance of HBase increases. Clients requests are completed faster due to the additional I/O capacity. Likewise, VMs can perform faster view updates. We draw the following conclusion: Scaling up the number of VMs as in Figure 5 and Figure 6 improves the view maintenance throughput up to a point where VMs are saturated with updates that they can push through the available I/O channels. Scaling up the region server improves overall performance, also for VMs, due to the additional I/O capacity.

Figure 12 shows system throughput as multiple views are maintained by a varying number of VMs. We note a performance leap as load changes from maintaining one to

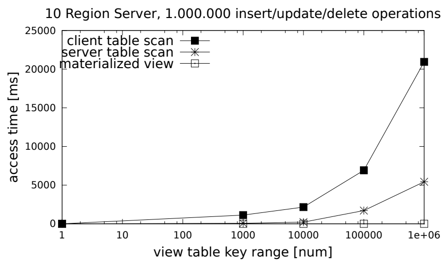

Fig. 8: Benefit of view maintenance

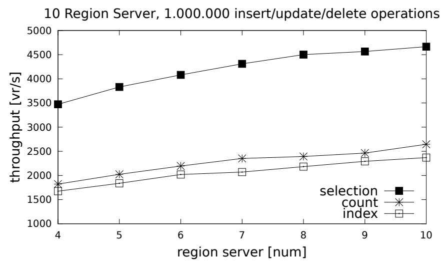

Fig. 9: Scale region servers

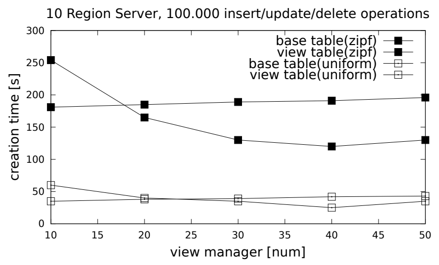

Fig. 10: Zipf distribution

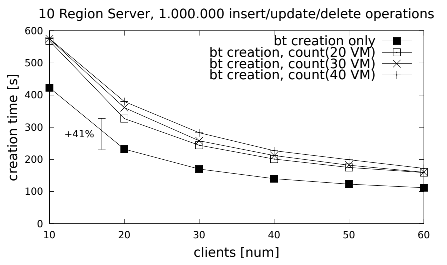

Fig. 11: Cost of view maintenance

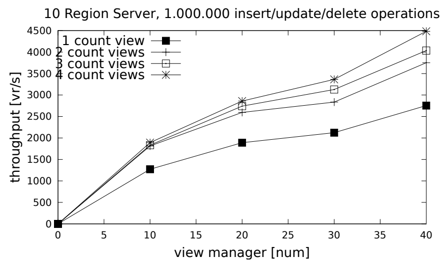

Fig. 12: Scale views

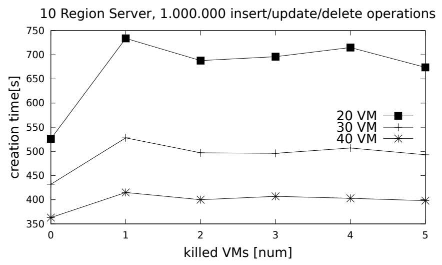

Fig. 13: Fault tolerance - Impact of crash

two count views. In our workload the count views derive from the same auxiliary table, which must be maintained as well, but only once. In further increasing the number of views, the effect diminishes. All aggregation views, especially join views, benefit from the sharing of auxiliary views.

However, increasing the number of views in the system, also increases the lag between base table states and derived view table states. While, as we show in Section III, our view states are consistent, they do lag behind in time. This lag can be reduced by increasing the number of VMs to speed up view maintenance. Thus, the more views we maintain, the more important is the number of VMs allocated.

4) Impact of data distribution: Figure 10 evaluates the effect of different distributions on system performance. We scale the number of VMs and track the latency for creating base tables and derived views. Keys of update operations are drawn from a Zipf and a Uniform distribution.

For the Uniform distribution, creation latencies are independent of the number of VMs assigned, even small numbers of VMs can handle the load in our set-up. For the Zipf distribution, latencies are much higher. We also see that creation and maintenance latency increase, yet are positively effected by increasing the number of VMs that propagate updates. Thus, especially for a skewed workload, the system greatly benefits from being able to dynamically assign VMs. VMs can be assigned to hot spots in the key range, away from ranges where they are not needed. Nevertheless, in this case, HBase may constitute a bottleneck for clients. Clients issue updates to a set number of region servers that handle 90% of the load, thus, resulting in a slowdown. HBase developers suggest that keys of updates should be salted. That is a prefix is assigned to keys, such that their distribution becomes uniform again. In this case, VMs propagate updates as we saw above.

5) Impact of VM crash: Figure 13 shows the impact of a view manager crash on system performance. In this experiment, we track view maintenance latency, as the number of VMs available in the system crashes. We ran experiments with 20, 30 and 40 VMs, respectively. Twenty seconds after view maintenance started, we terminate a number of VMs. In this experiment, the VMs are distributed evenly to region servers. In a set-up with 20 VMs and 10 region servers, we have a ratio of 2 VMs per region server. Terminating both VMs of the same region server stops view maintenance. The operations arriving at that region server cannot be forwarded until a new VM is assigned to the region server. In the experiment, we terminate VMs of different region servers.

In our set-up, a single crashing VM reduces the view maintenance latency, because it takes additional time for recovery to replay the TL and because the remaining VMs have to absorb additional load. The failing of further VMs does not further impact the situation. As long as every region server loses only one of its two VMs, the overall processing time remain the same. Overall, the impact of a VM crash lessens, the more VMs are deployed. Thus, increasing the number of VMs, increases the fault resilience of the system.

VI. Related Work

Over the past decades, there has been much research on view maintenance, for example, [11], [12], [14], [19], [20]. J. Blakeley et al. [19] developed an algorithm that reduces the number of base table queries during the update of a join view. Y. Zhuge et al. [11] developed the ECA algorithm, which prevents update anomalies. L. Colby et al. [20] introduced deferred view maintenance based on differential tables that keeps around a precomputed delta of the view table. Multiple algorithms are developed to perform deferred view maintenance. Much attention has been given to preventing update anomalies when applying incremental view maintenance [11], [14], [16]. These approaches originate from the databases of that time, i.e. data storage was centralized and data sets were of manageable size.

A lot of the mentioned papers, focus on pessimistic mechanisms, i.e. they prevent update anomalies in the moment or even after they have occurred. Instead, our approach tries to establish consistency by a more optimistic concept. Correct ordering of base table operations is ensured by consistent hashing right through the maintenance process. Test-And-Set methods are used to retain the commit order, no explicit locking or transactional mechanisms are need. In the end, this results in a higher throughput and more up-to-date view computations.

In the context of data warehouses, view maintenance has also been considered extensively. Early approaches aimed at integrating incremental, deferred view maintenance based on distributed data sources [12], [17], [21]. Y. Zhuge et al. [17] suggested the strobe algorithm, a refinement of the ECA algorithm that handles update anomalies, occurring when multiple base table updates of a join view arrive from different data sources. Agrawal et al. [21] defined the sweep algorithm, which like ECA and Strobe, compensates via error terms for interfering updates. In these approaches, a data warehouse is considered as a central component, where the flow of base table updates is joined and a global time-stamp can be set. Thus, operations can be serialized in order to obtain a global order. In the context of a KV-store, update propagation is distributed and no global order exists, thus, the above approaches cannot be applied.

Versioning and time-stamp-based approaches have been used to defer view maintenance and ensure data consistency [16], [22]. For example, Zhou et al. [22] use a SQL version store to categorize updates. That is, the database can be queried in the state, it has been in, when the base table update was applied. Also, Zhou et al. [22] introduced the scheduling of view maintenance tasks. There, tasks can be run, when the update load in the system is low, yet, the approach applies tasks always in the correct update order. Unfortunately, as illustrated above, in a KV-store, we cannot rely on time synchronization. While some KV-store provide mechanisms for row versioning, the local time on each node could differ due to clock skew and deriving a unique global time-based update order is not possible. Experiments with the versioning feature yielded a big storage overhead and complicated the maintenance process.

Jacobsen et al. [18] discuss view maintenance in the context of Web Data Platforms, i.e., large-scale storage systems, such as HBase, where view managers are independent components for updating views. Updates, issued to storage servers, are propagated to the view managers. View managers relieve the storage servers that run view update programs to update view tables. The authors identify and analyze data consistency issues in this context and define a number of consistency levels. These levels provide theoretical guarantees for different classes of views in the context of the abstract WDP assumed in the paper. Here, we use some of the foundations on view table consistency developed in [18].

Agrawal et al. [23] apply incremental, deferred view maintenance to PNUTS [4] supporting equi-joins, selection, and group-by-aggregation views. Similar to Jacobsen et al. [18], Agrawal et al. can only rely on record time-line guarantees in view maintenance. Updates to a view record may involve one or more reads on base tables. View tables are replicated in the underlying PNUTS architecture. Moreover, it does not only provide a solution for special types of views and implementations, but a simple generic system of view modules, that are built on top of a generalized KV-store model.

VII. Conclusions

In this paper, we developed a scalable view maintenance system, fully integrated with a distributed KV-store. We demonstrated the efficient, incremental, and deferred materialization of selection, index, aggregation, and join views with HBase. Our approach is capable to consistently maintain multiple views that may depend on each other. In the spirit of KV-stores, our view maintenance architecture is incrementally scalable by adding additional view managers as maintenance load increases. In our approach, a stream of base table updates is propagated to view tables by a bank of view managers operating in parallel. To establish view table consistency, we resort to the application of a number of known techniques that are combined in novel ways to materialize views consistently. We also address fault tolerance and recovery to react to failing view managers in our approach. Our experimental evaluation quantified the benefits and cost of the approach and shows that it scales linearly in view update load and number of view managers running.

In future work, we aim at exploring optimizations for the maintenance of multiple, overlapping view expressions and explore automatic means for reacting to view maintenance load variations in our architecture.

References

[1] Jay Parikh. Data Infrastructure at Web Scale. http://www.vldb.org/2013/keynotes.html.

[2] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 2008.

[3] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. SOSP, 2007.

[4] B. F. Cooper et al. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 2008.

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

[6] Eben Hewitt. *Cassandra: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[7] Jeffrey et al. Dean. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 2008.

[8] Matei et al. Zaharia. Spark: Cluster computing with working sets. Berkeley, CA, USA. USENIX Association.

[9] Ashish et al. Thusoo. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*

[10] Apache Phoenix. https://phoenix.apache.org/.

[11] Y. Zhuge et al. View Maintenance in a Warehousing Environment. *SIGMOD Rec.*, 1995.

[12] H. Wang and M. Orlowska. Efficient Refreshment of Materialized Views with Multiple Sources. CIKM, 1999.

[13] X. Zhang et al. Parallel Multisource View Maintenance. *The VLDB Journal*, 2004.

[14] A. Gupta et al. Maintaining Views Incrementally. *SIGMOD Rec.*, 1993.

[15] Ki Yong Lee et al. Efficient Incremental View Maintenance in Data Warehouses. CIKM, 2001.

[16] K. Salem et al. How to Roll a Join: Asynchronous Incremental View Maintenance. SIGMOD, 2000.

[17] Y. Zhuge et al. The Strobe Algorithms for Multi-source Warehouse Consistency. DIS, 1996.
[18] Ramana Yerneni Hans-Arno Jacobsen, Patric Lee. View Maintenance in Web Data Platforms. *ACM Trans. Program. Lang. Syst.*, 2009.
[19] J. Blakeley et al. Efficiently Updating Materialized Views. *SIGMOD Rec.*, 1986.
[20] L. Colby et al. Algorithms for Deferred View Maintenance. *SIGMOD Rec.*, 1996.
[21] D. Agrawal et al. Efficient View Maintenance at Data Warehouses. *SIGMOD Rec.*, 1997.
[22] J. Zhou et al. Lazy Maintenance of Materialized Views. VLDB, 2007.
[23] P. Agrawal et al. Asynchronous View Maintenance for VLSD Databases. SIGMOD, 2009.