Dynamic Scalable View Maintenance in KV-stores

Jan Adler
TU München

Martin Jergler
TU München

Arno Jacobsen
TU München

ABSTRACT

Distributed *key-value stores* have become the solution of choice for many data-intensive applications. However, their limited query language support imposes challenges for applications that require sophisticated query capabilities. To address this problem, in this paper, we develop the *View Maintenance System* (VMS) to incrementally maintain selection, projection, aggregation, and join views on behalf of applications. We design VMS given a generic KV-store model based on a small number of features available in many popular store architectures. VMS supports the maintenance of hundreds of views in parallel, while simultaneously providing guarantees for view consistency, even under node crash scenarios. To evaluate our concepts, we deliver a full-fledged implementation of VMS through Apache's HBase and conduct an extensive experimental study. Exploiting parallel maintenance, VMS achieve throughputs up to 60k view updates per second (120k table updates).

1. INTRODUCTION

The properties of major Internet players are backed by what has become known as *key-value stores* (KV-stores), handling millions of client requests and producing terabytes of data on a daily basis [1]. Examples include Google's Bigtable [2], Amazon's Dynamo [3], Yahoo's PNUTS [4], Apache's HBase [5], and Cassandra [6] which originated at Facebook.

As opposed to earlier generation key-value stores, such as BerkeleyDB, which were more intended as main-memory databases to persist application configurations, among others, the KV-stores we consider in this paper are highly distributed and large-scale systems designed to back today's massive-scale Internet services.

These KV-stores are highly available and provide advanced features like load balancing and fault-tolerance [2–6]. For example, KV-stores scale horizontally by partitioning data and request load across a configurable number of nodes. To achieve these properties at scale, KV-stores sacrifice, among others, an expressive query language and data model, only offering a simple API, comprising *get*, *put*, and *delete* operations. While this design offers efficient access to single row entries, the processing of more complex SQL-like queries, such as, selection, projection, aggregation, and join, require costly application-level operations. For example, to compute a join, the tables to be joined have to be read by the application, joined, and written back to the store, unless result storage is not required.

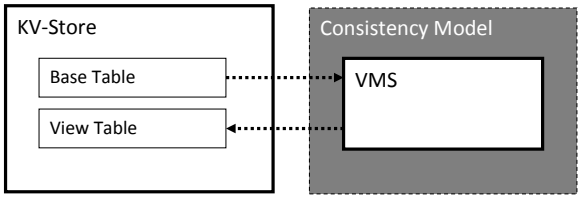

Figure 1: System overview

Point solutions have appeared that raise the level of abstraction of a given KV-store by partially or fully materializing desired application-level queries as views through the underlying store [7–9]. In this manner, secondary indices have been added to KV-stores [7, 10], caching of application queries has been enabled [11], massive processing of selection views for feed following applications has been enabled [8], however, a generic solution that introduces view management for a wide variety of SQL query operators as views to KV-store is still at large.

It is this problem that this paper addresses. As opposed to existing solutions, our design abstracts from a specific KV-store, aiming to support a whole gamut of KV-stores that offer a small set of key features our design leverages.

Our approach introduces mechanisms for the materialization and incremental maintenance of views to KV-stores resulting in sophisticated query capabilities for applications, simply through definition of view expressions. Materialized views are tables managed by the KV-store and all properties such as concurrent access, availability, and fault-tolerance, apply to views as well. Views are maintained incrementally: base data updates propagate through the system to only effect the derived view data records. The challenge is the design of the *View Maintenance System* (VMS) that efficiently and correctly maintains views at the desired scale. We developed such a system, as shown in Figure 1. VMS consumes streams of client base data operations and produces updates to view data records.

This paper makes the following contributions: (1) We provide an analysis of popular KV-stores to identify a small set of common features, a generic VMS design requires to correctly materialize views (cf. Section 2.) (2) We design VMS in the spirit of existing KV-stores to offer horizontal scalability and fault-tolerance by simply leveraging the existing mechanisms of the stores (cf. Section 3.) (3) We introduce novel concepts, namely auxiliary

views, as basis for correctly and consistently materializing views in KV-stores (cf. Section 4.) (4) We validate VMS by extending HBase with view maintenance capabilities and demonstrate that the extended architecture is at par in terms of performance with plain HBase at negligible overhead (cf. Section 5.)

2. BACKGROUND

In this section, we discuss KV-store internals that serve us in the remainder of the paper. We abstract from any particular store, focus on main concepts, and ground concepts in existing KV-stores such as [2–6]. We only consider the highly distributed stores that have emerged over the last decade and disregard centralized KV-stores. Our objective is to distil a set of features our VMS requires from a KV-store. In the second part of the section we discuss the consistency model. We establish a notion to describe tables and records in the system. Further, we constrain the relations between tables in order to express a certain level of consistency.

2.1 KV-store Design Overview

In the upper half of Figure 2 you can see our generalized KV-store model. Some KV-stores explicitly designate a master node, e.g., HBase or Bigtable, while others operate without explicit master, e.g., Cassandra, where a leader is elected to perform management tasks, or PNUTS, where mastership varies on a per-record basis. In all cases, a system *node* represents the unit of scalability: The number of nodes can vary to accommodate load change. Nodes persist the data stored in the system. In contrast to a centralized SQL-based DBMS, a node manages only part of the overall data (and part of the request load).

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.

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 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, 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[1].

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 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 clients ends up at one particular node that is serving the key range the client wants to access.

[1]Sometimes there are additional methods, e.g., a range scan or passing a selection predicate to the store. However, none of them extends beyond repeated single row access; none of them offers expressive semantics.

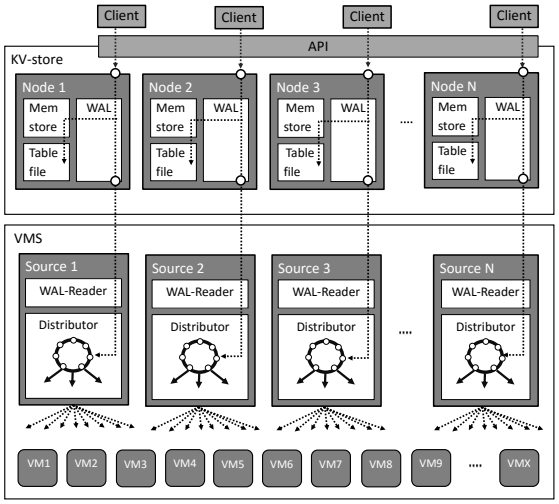

Figure 2: KV-Store and VMS

Every node maintains a *transaction log* (TL), referred to a write-ahead log in HBase and commit log in Cassandra. When a client operation arrives, it is first written into the TL of the node (cf. Figure 2). From then on, the operation is durably persisted. Subsequently, the operation is inserted into a *memstore*. Memstores are volatile; 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. 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 are purged.

Extension points – We designed VMS to react to KV-store events and to not interfere with store-internal read/write paths for data processing. In this spirit, we determined a number of common "extension points," all considered KV-stores exhibit. We characterize extension points by events, which the VMS reacts to in maintaining views. There are two different kinds of events that VMS needs to react to: *administrative events* and *data events*.

Administrative events occur during a state change in the KV-store infrastructure, e.g., a new node is added (and starts processing client operations). KV-stores provide different ways to react to administrative events. For example, HBase lets developers use various kinds of *coprocessors*, which refer to application-developer provided code pieces that can be deployed and run on system nodes before or after certain events. We leverage this mechanisms to notify VMS, as soon as certain events of interest occur (e.g., the addition of a node.) VMS reacts accordingly and allocates resources to maintain view tables that depend on the newly added node.

Data events occur, when a client updates a base table (e.g., put, delete). Consequently, base table derived view tables become stale and VMS needs to update them. Generally speaking, there are three methods to stream updates on base data from the KV-store to VMS: (1) Access the store's API, (2) intercept operations (e.g., via coprocessors), (3) read the TL.

Method 1 may lead to inconsistent view states, as base data change, before a prior update can be retrieved; none of the popular KV-stores offers snapshot isolation. Also, this method would incur a lot of overhead (e.g., an update would trigger a read and one or more write to update derived views.) Method 2 looks promising, especially, with regard to freshness of the view (coprocessor execution is synchronous in the update path), but it is only suitable if the number of maintained views is small, otherwise KV-store operations would be needlessly delayed, counter-acting the asynchronous operation behind many design decisions. Thus, Method 3 is the preferred choice.

Method 3 has several benefits: (i) 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.[2] (ii) 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. (iii) Operations in TL are durably persisted and can be recovered by VMS. (iv) The TL contains operations, not table rows, which is fine for incremental maintenance.

2.2 Consistency model

A view data consistency model[3] validates the correctness of a view table. Furthermore, the model evaluates a view table's ability to follow a sequence of base table states an produce a corresponding number of valid view states. It does so by defining different levels of consistency of a view [12–16]. Depending on view types, view maintenance strategies, and view update programs, none, some, or all of the levels are attainable [**?**,12–15].

As base and view table are nothing more than standard tables in the KV-store, we create a notion of tables and records first. The data model of a KV-store differs from that of a relational DBMS. We propose a model that is representative for today's KV-stores. 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 described by $f:(K,C)\rightarrow V$ where K is the row key and C is the column key. In the 3-dimensional case, another parameter, a timestamp, for example, is added to the key, which may serve versioning purposes. For this paper, the 2-dimensional model suffices. This notation more closely resembles a database row and is used throughout the remainder of this paper. We use the following notation to define: (a) a table (with K being the row key and F being a set of column-value pairs), (b) a record, (c) a put operation and (d) a delete operation.

$$(a)\ T=(K,F)$$
$$(b)\ r=(k,\langle c_1,v_1\rangle..\langle c_n,v_n\rangle)$$
$$(c)\ p=put(k,\langle c_1,v_1\rangle..\langle c_n,v_n\rangle)$$
$$(d)\ d=del(k)$$

Lets now define a base table as $B=(K,F)$ and a view table as $V=(K,F)$. The relation between both tables can be expressed as $View(B)=V$, where function $View$ can be any term described by relational algebra (e.g. view table is a projection of base table, $V=\pi_{c_1}(B)$). If the clients of the KV-store start doing their updates, the state of the base table changes; we depict the sequence of states with indices $B_0..,B_i,..,B_f$, where B_0 is the initial state, B_i is an arbitrary intermediate state and B_f is the final state. Every client put or delete operation causes a record to change its version (all versions of a record are called a record's timeline) and thus, the base table to change its state. Two states can be compared by the operator $\leq$. $B_i \leq B_j$ means that the versions of all records in B_j are equal or newer than the versions of records in B_i.

Once the base table changes, the view table – or rather the system that maintains the view – needs to react and in-cooperate the changes into the view. How accurate this maintenance is done, is defined by the following levels:

> *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 $B_0 \leq B_i \leq B_{i+1} \leq B_f$.

Example 1: Imagine a base table $B=(K,F)$ and a view table $V=\gamma_{c_1,sum(c_2)}(B)$. The initial state of the base table is $B_0=\{(k_1,\langle c_1,x_1\rangle,\langle c_2,15\rangle)\}$ and the corresponding state of the view table is $V_0=\{(x_1,\langle c_{sum},15\rangle)\}$. Now, the following update operations are applied to the base table:

$$(1)\ put(k_1,\langle c_1,x_1\rangle,\langle c_2,20\rangle)$$
$$(2)\ put(k_2,\langle c_1,x_1\rangle,\langle c_2,10\rangle)$$
$$(3)\ del(k_1)$$
$$(4)\ put(k_3,\langle c_1,x_1\rangle,\langle c_2,45\rangle)$$

The base table reaches a final state $B_f=\{(k_2,\langle c_1,x_1\rangle,\langle c_2,10\rangle),(k_3,\langle c_1,x_1\rangle,\langle c_2,45\rangle)\}$. Accordingly, to achieve consistency level convergence, the maintenance system has to compute the final view state as: $V_f=\{(x_1,\langle c_{sum},55\rangle)\}$. To achieve weak consistency the maintenance system could (there are various possibilities) produce two intermediate view states $V_1=\{(x_1,\langle c_{sum},30\rangle)\}$ and $V_2=\{(x_1,\langle c_{sum},20\rangle)\}$. Note that, though both states are valid, they are not in correct order. Thus, to achieve strong consistency the order of view states V_1 and V_2 has to be changed. The last (and usually too costly) level, complete consistency, can be achieved, representing all intermediate base states in the view: $V_1=\{(x_1,\langle c_{sum},20\rangle)\}$, $V_2=\{(x_1,\langle c_{sum},30\rangle)\}$ and $V_3=\{(x_1,\langle c_{sum},10\rangle)\}$

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.

[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)

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

1. *View updates are applied exactly once*
2. *View updates are atomic*
3. *(Base-)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.) Instead we provide a brief explanation of Theorem 1. If we make use of Rule 1 of the theorem, we make sure that all update operations are delivered and applied exactly once. Rule 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 change its correctness.
Example 2: We reconsider the table set-up of Example 1 and its update operations (1) - (4). When two update programs process update operation (1), and (2) in parallel, the following can happen: Update program 1 and Update program 2 retrieve the initial view record (x_1,$\langle c_{sum}$,15$\rangle$) simultaneously from the view. Update program 1 applies its incremental update, leading to a new view record (x_1,$\langle c_{sum}$,20$\rangle$) and writes it back to the view; update program 2 also applies its incremental update to the initial value, leading to (x_1,$\langle c_{sum}$,25$\rangle$) 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,$\langle c_{sum}$,30$\rangle$).

Rule 2 – more specifically the isolation of transactions – would have prevented the wrong execution in Example 2. If Rule 1 and Rule 2 of the theorem are applied, convergence is still not guaranteed. Asynchronous processing and high parallelism can lead to reordering of update operations.
Example 3: Again, we consider table set-up of Example 1 and operations (1) - (4). Operation (3) and operation (1) touch 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 row-key to be present in the view computation, where it shouldn't.

Therefore, we also apply Rule 3 and demand the preservation of a record's timeline. All three rules together make sure 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 attain strong consistency for the views it maintains.

3. VIEW MAINTENANCE SYSTEM

In this section, we present and discuss the design of VMS. By illustrating how a base table operation may effect a view table, we provide the intuition for the resulting view consistency established by VMS. Finally, we discuss fault-tolerance.

3.1 Design Overview

The lower half of Figure 2 gives an overview of VMS, which is comprised of a *master*, a number of n *source systems* (always the same number as KV-store nodes) and a number of m view manager. The input to VMS is a set of operation streams; each emitted by a KV-store node (cf. Figure 2). Technically, the *WAL reader* component of a source system connects to the underlying HDFS and reads the WAL of its assigned node. Because operations are always appended to the WAL, the reading happens fast and in-order. After retrieving the operations from WAL, the WAL reader passes them to the distributor. This component distributes the incoming operation stream to all registered VMs. The number of VMs is configurable. VMs can be dynamically *assigned* to or *removed* from the VMS.

A VM is designed to be light-weight and can be deployed in large numbers to accommodate a changing view update load. It computes view updates, based on base table operations it receives as input via the KV-store API, for view tables. A VM only belongs to a single sub-system. The sub-system feeds the VM with operations which processes them in order. A VM is the unit of scalability of VMS. VMs 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 VM is always able to execute any view table update from any host.

Our design exhibits at least the following four benefits: (i) Seamless scalability: Hundreds of views may have to be updated as a consequence of a single base table operation. As VMS exceeds its service levels, additional VMs can be spawned (below, we show this experimentally). (ii) Operational flexibility: VMs introduce flexibility to the system architecture. All VMs of a given sub-system can be hosted together on the same node or each VM can be hosted at a different node. (iii) Accommodate load variations: VMs can be reassigned from one sub-system to another as base table update load changes. (iv) Fault-tolerance: If a VM crashes, another VM can take over and continue processing the operation stream.

3.2 Update Propagation

A source distributes the arriving base table updates (i.e. WAL stream) to the VMs via consistent hashing by maintaining a hash-ring (cf. Figure 3), where active VMs are registered. Row keys of arriving updates are hashed into the ring and associated in clock-wise direction with active VMs. In this way, a source-system distributes operations uniformly across the available VMs and ensures that base table operations on the same row key are always handled by the same VM. On the one hand, this mechanism ensures maximal degree of concurrency for update propagation, on the other hand it guarantees the ordered propagation of base table updates to view tables, setting the basis for view table consistency.

There are two alternatives to compute a the hash value of an update operation. *Alternative 1:* the distributor component uses the hash of the base table row-key to determine a VM (row-key k, cf. Figure 3). *Alternative 2:* the distributor component uses the hash of the view table row-key to determine a VM (row-key x, cf. Figure 3). If base table and view table have the same row-key (e.g. a selection view), both alternatives will match the same result. But if row-keys are different (e.g. an aggregation or join view), both alternatives will lead to different up- and downsides, which will be discussed in Subsection 3.3.

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

To access and update view tables, a VM 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 VM retrieves and caches the view definitions of the derived views (e.g., a `SELEC-`

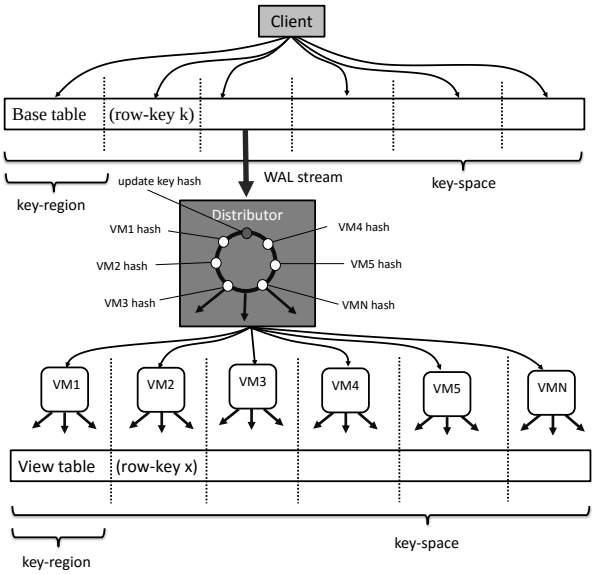

Figure 3: Update distribution through Consistent Hashing

TION and COUNT view S and C, both derived from A). Then, VM runs the update program, and submits view table updates (to S and C) via the client API. For some of the view types maintained, the VM has to query the view table first, as part of the update logic; in a COUNT view, the VM reads the current count from the view before applying the delta of the base table operation. These view queries are always get operations (i.e., single row accesses) and can be evaluated quickly.

3.3 Consistency

Since consistency is a major concern during incremental view maintenance – additionally, we introduce a high degree of concurrency – we now show, how the fundamentals of Section 2.2 influence the VMS design. As a first step, we discuss the three rules of Theorem 1, separately.

3.3.1 Exactly once property

Like stated in our theorem, Rule 1, 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).

One architectural decision of VMS was to read the update operations from the WAL of the KV-store node. Updates in the WAL are persisted safely and replicated via HDFS. Even though a node or a VMgoes down and update operations are lost, they can be always recovered from the WAL of the KV-store node. Actually, the WAL serves the same purpose for the KV-store itself. As soon as a node crashes, the KV-storereplays all transactions that have not been flushed to the table file before and would have been lost otherwise.

As updates cannot be lost, the VMS guarantees an at-most-once semantic. Still, we need to assure that updates in VMS are not duplicated. In case a crash occurs, the WAL reader may re-process some of the updates and send them to the VMs a second time. Then, the VM needs to identify the duplicate and drop it. We achieve the identification of duplicates with the help of a global ID (i.e. signature); this ID can only be obtained from the KV-store, as the update operations originate from here. Our implementation builds on top of HBase, a global ID can be created by combining an operations sequence number and the node ID. This ID is delivered with every update operation. The VM just keeps track of the highest maintained operation ID (this information can be stored into Zookeeper, as the system is fault-tolerant). If ever an operation with a lower ID is sent to the VM, it identifies the duplicate and drops it.

3.3.2 Atomic view record update

Like stated in our theorem, Rule 2, every view update has to be executed atomically. Regarding a view update, we rely on the semantics that are provided by KV-store. We assume (as it is the case for HBase and Cassandra) that a single put or delete operation, e.g. $put(k_1,\langle c_1,x_1\rangle,\langle c_2,10\rangle)$, is executed ACID compliant. Thus, a row update has to be atomic; the KV-store is not allowed to execute let's say $put(k_1,\langle c_1,x_1\rangle)$ and then $put(k_1,\langle c_2,10\rangle)$ later.

Despite the single update operation (i.e. get/put), also the complete update process – comprising of a get-operation to the old view record and a put/delete- operation to the new view record – has to be ACID compliant. Most view types define a mapping from multiple base table records to a single view table record (e.g. aggregation). As shown in Example 2 different base table records may be propagated to different VMs, multiple VMs can concurrently update the same view record. Remembering the execution alternatives from Subsection 3.2, this problem has to be solved, differently.

Alternative 1: Distributing the update operation according to the base table row-key means, there could be multiple VM, updating one view table row-key. For example put operations $put(k_1,\langle c_1,x_1\rangle,\langle c_2,10\rangle)$ and $put(k_5,\langle c_1,x_1\rangle,\langle c_2,5\rangle)$ could be forwarded to different VMs, still both VMshave to access row key x_1. To solve the problem, we use test-And-set methods to avoid interruption during updates. The VM retrieves the old view record, e.g. $(x_1,\langle c_{sum},20\rangle)$ 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 VM sends a test-and-set request to the server with a test-value (here 20).

Alternative 2: Distributing the update operation according to the hash of the view table row-key completely eliminates the need of further synchronization mechanisms. Every VM is responsible for an equal amount of view table records and, thus, for a part of the view table. As there is no ambiguity, VMs can just use regular get and put/delete operations.

3.3.3 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.

Alternative 1: Distributing the update operation according to the base table row-key means, time-line of a record cannot be broken. All operations that touch a specific row-key will always be directed to the same VM; they will be retrieved in-order, sent to the VM in-order and updated in the view in-order.

Alternative 2: Distributing the update operation according to the hash of the view table row-key also creates a record timeline. But in contrast to Alternative 1, it is the timeline of the view table, which bears the following consequences: As long as the view table record doesn't change, updates to the same base table row-key are also forwarded to the same VM. As soon as

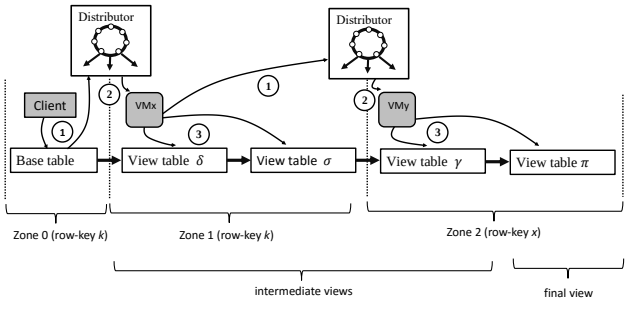

Figure 4: Executing a query with views

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 that reason, we need a mechanism which prevents this scenario. We employ a buffer for update operations. But only for those updates, that change the view table key in question – insert and delete operations, as well as update operations that don't change the key are processed just like before; matching this criterion, the update is inserted into the buffer, send to a VM, where it causes the old entry to be deleted; it is, then, deleted from the buffer and passed to another VM, where it causes the new entry to be inserted. If during that process, however, more updates flow in

3.3.4 Conclusion

In the previous sections, we described two alternatives and their implications. Both alternatives have their up- and downsides. Alternative 1 supports the preservation of base record time-line, whereas Alternative 2 supports the concurrent access to the view table. Both alternatives are convenient consistency concepts, in both cases the implementation needs to be complemented with additional mechanisms (i.e. test-and-Set methods, update buffer) to achieve strong consistency.

Still, we favor Alternative 2 over Alternative 1. One reason is the fact that Alternative 1 uses Test-And-Set methods. Even though it is a pessimistic locking mechanism – in contrast to optimistic locking, a view record is always accessible – it is not suited well in our case [4] for the following reasons: (1) If there is high contention, a lot of view updates have to be repeatedly applied, (2) we have to use test-and-set for every view update, despite of it being an insert, update or delete, which introduces overhead. In contrast, we choose Alternative 2 for the following reasons: (1) it is a locking-free alternative (most state-of-the-art transaction systems use locking-free mechanisms). (2) The update buffer is only needed for a small fraction of update operations (i.e. not for inserts, not for deletes, etc..). Thus performance overhead is minimal.

3.3.5 Nested constructions

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. SQL-like queries) 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 translate the query, we need to build a maintenance plan. This can be done with the help of a directed acyclic graph (DAG); each node in the DAG represents one materialized view. Starting from the base table, all views are inter-connected and each pair of view tables resembles a base/view table-relation. The moment a base table gets updated, the operation travels subsequently to all views and updates them incrementally. The result of the query can be obtained from the last view.

In Figure 4, the maintenance process of the following query is depicted: Select $sum(c2)$ from $bt1$ group by $c1$ where $c2 < 10$ 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. To improve performance, a VM can update the view tables of a zone in one run. Once a zone – and therefore also the row key – changes, updates need to be re-distributed in order to sustain consistency (cf. Figure 3). In the figure two zones can be identified. The maintenance of a zone is always processed in a cycle of three steps:

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

In the first step, the update operation is provided to the distributor component of the source system. During maintenance of the first zone the update is provided by the WAL reader of the source system itself, during maintenance of subsequent zones, it is provided by the last responsible VM. In the second step, the distributor assigns and distributes updates as described above (cf. Figure 4). In the third step, the VM goes along the path of each view table in the zone, retrieves the old view record, updates it incrementally and stores the new version back into the view. Then the cycle is repeated for every zone. As all the updates of a zone can are distributed according to the cardinality of the row key, a high parallelisation of updates can be achieved.

4. VIEW MAINTENANCE CONCEPT

In this section, we develop techniques for maintaining a set of basic view types in VMS. This view types can be combined to build SQL-like query constructs. Because of the characteristics of KV-store – especially the single key row access – we define a set of auxiliary view types first. Based on that we describe the standard view types (i.e. `SELECTION`, `PROJECTION`, `JOIN`), briefly and explain how they make use of the auxiliaries. Finally, we show an example of a view composition.

4.1 Auxiliary views

Auxiliary views are internal to VMS and are not exposed to clients. They are maintained to enable, facilitate and speed up the correct maintenance of the other view types. Some view types could not be maintained consistently without the additional information provided by auxiliaries, others simply benefit from their pre-computations. While auxiliaries introduce storage overhead, they support modularity in view maintenance; e.g., a single relation of a multi-table-join can be reused in different join views (that embed the same relation). Logically, auxiliaries represent the basic elements of view maintenance and their use amortizes as more complex views are managed by VMS. Auxiliaries also speed up view maintenance significantly, as we show in Section 5. In what follows, we describe each auxiliary view type.

Delta – The `DELTA` view is an auxiliary view that tracks base table changes between successive update operations. TL entries only contain the client operation. They do not characterize the

[4] Actually, modern system try to avoid locking completely

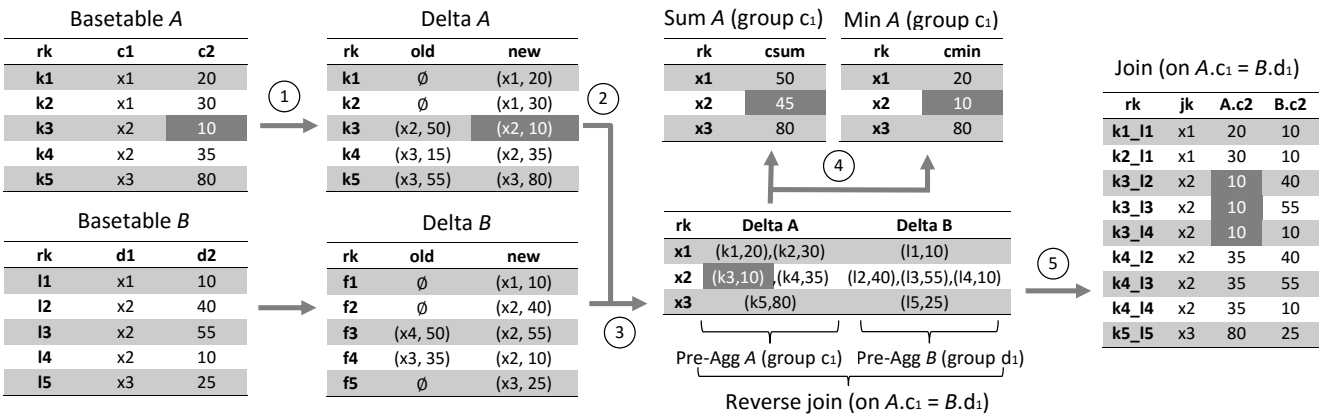

Figure 5: View table example

base record state before or after the operation. For example, for a delete operation, the client only provides the row key, but not the associated value to be deleted, as input. Likewise, an update operation provides the row key and new values, but not the old values to be modified. In fact, a TL entry does not distinguish between an insert and update operation. However, for view maintenance, this information is vital. This motivated us to introduce the `DELTA` view. It records base table entry changes, tracking the states between entry updates, i.e., the "delta" between two successive operations for a given row. Views that derive, have this information available for their maintenance operations.

Pre-Aggregation – The `PRE-AGGREGATION` view is an auxiliary 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 over and over. Moreover, 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 [16]. 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 `AVG`, can then be applied to the map. Thus, aggregation results become available instantaneously.

Reverse Join – A `REVERSE-JOIN` view is an auxiliary 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 VM needs to query the other base table to determine the matching join rows. Only if the join-attribute is the row key of the queried base table, can the matching row be determined quickly, unless of course, an index is defined on the join-attribute for the table. Otherwise, a scan of the entire table is required, which has the following drawbacks: (i) Scans require a disproportional amount of time, slowing down view maintenance. Also, with increasing table size, the problem worsens. (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* (jk) of the two base tables as row key of the `REVERSE-JOIN` view. When updates are propagated, the `REVERSE-JOIN` view can be accessed from either side of the relation with the help of the join key (it is always included in both tables' updates). 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.

4.2 Standard views

In this section, we describe how VMS maintains client-exposed views for a number of interesting standard view types. We also present alternative maintenance strategies, but defer a full-fledged 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 VM 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. Generally speaking, in aggregation views, records identified by an *aggregation key* aggregate into a single view table record. The aggregation key becomes the row key of the view.

`MIN` and `MAX` views are also aggregates. Both can be derived from a `DELTA` or a `PRE-AGGREGATION` view. When derived from a `DELTA` view, `MIN` and `MAX` are computed similar to a `SUM`. However, a special case is the deletion of a minimum (maximum) in a `MIN` (`MAX`). In that case, the new minimum (maximum) has to be determined. Without auxiliary views, a table scan would have to be performed [16]. This motivated us to derive the `MIN` (`MAX`) from a `PRE-AGGREGATION`, which prevents 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 the client wishes to retrieve a base table row by the indexed column, it accesses the `INDEX` view first, then, it accesses the base table with the row keys retrieved from the record found in the view. This is a fast type of access, for the client is always accessing single rows by row key.
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 maintained, easily. So far, we are only providing equi joins. Theta joins, with more complex join predicates are deferred to future work.

4.3 View composition

Figure 5 gives a comprehensive example of the introduced view types (standard and auxiliary). On the left side, two base tables A and B are shown. Both base tables consist of a row key (rk) and two columns (c_1,c_2). Either base table is connected to a delta table (Delta A and B) that tracks the changes of base table columns. The delta tables are connected to a `PRE-AGGREGATION` and a `REVERSE-JOIN`, respectively. These views reorder and group the base table records according to a secondary key (found in columns $A.c_1$ and $B.c_1$)[5].

In the set-up of Figure 5, aggregation key (of `SUM` and `MIN`) and join key (of `JOIN`) are equal, so the `PRE-AGGREGATION` becomes a subset of the `REVERSE-JOIN`. This is a good example of how a single auxiliary view can feed multiple different views to reduce the overall cost. The column set $(\{\}_A)$ in the `PRE-AGGREGATION` serves to directly determine the values of `SUM` and `MIN`. Further, the cross-product of column sets $\{\}_A$ and $\{\}_B$ serves to maintain the `JOIN` at the right side of the figure.

The grey arrows in Figure 5 depict the paths along which the client operations are propagated to update the view composition. Instead of re-computing the complete views, just the affected parts are modified. To further reduce overhead, VMS can execute updates on single view column values; the rest of the row remains unaltered.

Example: Now, we show a put operation that triggers a number of updates along the view composition (see Figure 5, the dark grey boxes): (1) The put operation changes the value of row k_3, column c_2 in base table A from 50 to 10. (2) The change is propagated and represented in Delta A as $50 \rightarrow 10$. (3) The corresponding column in the `PRE-AGGREGATION` view (i.e., row key x_2, column $\{c_1\}_A$) is updated. (4) This causes the sum and minimum of row key x_2 in the `SUM` and `MIN` to be re-evaluated. (5) Further, the corresponding entries in `JOIN` are updated to the new value 10. As depicted, VMS always accesses a single column value (to update the view tables); the access is executed with knowledge of row and column key. Since a KV-store guarantees fast row and column key access, – actually, this is what they are built for, – the update propagates rapidly through the composition. Besides the example in Figure 5, more complex view compositions can be created by combining different types of view tables.

5. EVALUATION

[5]One may argue that groups of such an aggregation tend to grow very large (if the total number of aggregation keys is small). However, due to the column-oriented storage format of many KV-store, the values are stored consecutively and indexed by column key – the access still remains fast.

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 cluster comprised of 40 nodes (running Ubuntu 14.04). Out of these, 11 were dedicated to the Apache Hadoop (v1.2.1) installation, one as name node (HDFS master) and 10 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 nodes to be able to scale without interfering with the core system (i.e., the 12 HBase nodes.) Finally, another 8 nodes were reserved for HBase clients to generate the update load on base tables.

Prior to each experiment, we created an empty base table and defined a set of view tables. View definitions and underlying 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. This choice allows HBase to balance regions with high granularity and ensures a uniform distribution of keys among available region servers.

For `COUNT`, `SUM`, `MIN`, and `MAX` views, we created 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 can control 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.

The workload we generate 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 simulates a "hot data" scenario, where only few base table records are updated very frequently.

With our experiments, we primarily evaluate the performance of the system with regards to throughput and view maintenance latency.
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 levels (i.e., selection of 10, 50 and 90 percent of base records), while varying the number of VMs that process the update load. Figure 6 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 6, the performance is monotonically decreasing for a selectivity of 10%. This means, VMs propagate updates as fast as they are applied

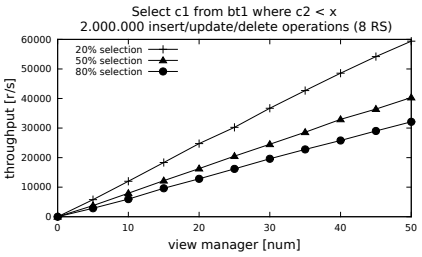

Figure 6: Selection performance

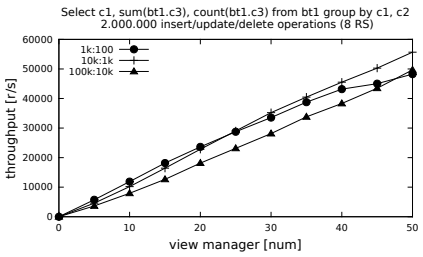

Figure 7: Aggregation performance

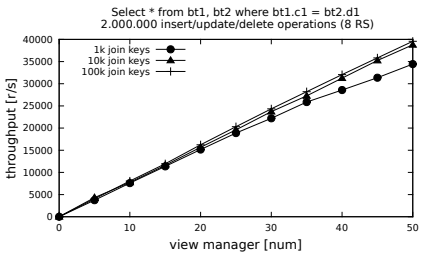

Figure 8: Join performance

to HBase by clients. Increasing the number of VMs only slows down the system as more components are running concurrently.

Figure 7 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 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 8 shows the results for the `JOIN` view under the same conditions as above. Compared to the other view types and not surprising, `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.

Cost 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 parallelise 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 (certainly noting that in practice, this would not be an option; thus, these two approaches are merely serving as a baseline, here.) Our primary objective is to obtain some feel for how VMS fares relative to other potential approaches.

The results are given in Figure **??**, where we measure the latency a client perceives (i.e., the time until results are available) 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 come at a cost. We assess this cost as the performance impact on base table creation time (here, defined as the time to insert 1M tuples into the base table.) Figure **??** shows the base table creation time with and without concurrent view maintenance. We track creation time as the number of clients increases. Figure **??** 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%.

System scalability – Figure 6 and Figure 7 show 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 **??**, 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. Client 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 6 and Figure 7 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

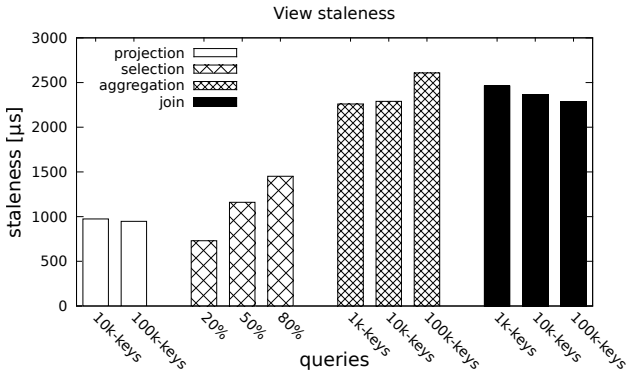

Figure 9: Selection performance

performance, also for VMs, due to the additional I/O capacity.

Figure **??** 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 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 our view states always remain 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.

Impact of data distribution – Figure **??** 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 a small number 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.

Impact of VM crash – Figure **??** 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 transaction log 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 remains 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.

6. RELATED WORK

Research on view maintenance started in the 80s [12,13,17–19]. Blakeley et al. [17] developed an algorithm that reduces the number of base table queries during updates of join views. Zhuge et al. [12] developed the ECA algorithm, which prevents update anomalies. Colby et al. [19] 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 [12, 18, 20]. 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.

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 [9,21–23] and in academia [7,8, 10,11,16,24,25]. Percolator [21] is a system specifically designed to incrementally update a Web search index as new content is found. Naiad [22] is a system for incremental processing of source streams over potentially iterative computations expressed as a dataflow. Both systems are designed for large-scale involving thousands of nodes, but are not addressing the incremental materialization of the kind of views considered in this work.

The Apache Phoenix project [9] develops a relational database layer over HBase, also supporting the definition of views. Little is revealed about how views are implemented, except in as much as limiting view definitions to selection views and materializing views as part of the physical tables they are derived from. 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." [9] The work presented in this paper constitutes a foundation upon which future Phoenix designs could draw.

Agrawal et al. [7] realize incremental, deferred view maintenance in Yahoo!'s PNUTS KV-store [4] in order to raise the level of abstraction of the store's API to expressing equijoin, selection and group-by-aggregation. The approach is based on multiple mechanisms to support this limited set of views. The crux are local view tables (LVTs) that partially materialize views in a synchronous fashion as part of the base table update path on the same storage unit where the view-defining base table resides. At view query time, LVTs are queried to compute aggregates or joins. A remote view table (RVT), potentially residing across the network, stores the mapping between views and constituting LVTs on different nodes. RVTs alone are sufficient to materialize selection views. Our approach significantly differs from this design and uses a single, asynchronous update propagation

mechanisms as basis for a wide range of view types that go beyond the design by Agrawal et al. [7].

In similar spirit, Silberstein et al. [8] designed mechanisms to materialize selection views as underlying abstraction to derive the N most recent events in support of, what the authors refer to as, follow applications, i.e., Twitter etc., in PNUTS. The work only considers selection views over windows of time, a query semantic even more restrictive than selection per se, thus, not applicable to the wide view semantics we aim at realizing.

Interesting materialization approaches are presented in [10,11]. Pequod [11] serve as front-end application cache that materializes application computations. Pequod supports a write-through policy to pass updates on to the back-end store, while serving reads from the cached data. SLIK [10] focuses on materializing indexes in KV-stores.

7. CONCLUSIONS

In this paper, we developed a scalable view maintenance system (VMS), fully integrated with a distributed KV-store. We demonstrated the efficient, incremental, and deferred materialization of selection, index, aggregation, and join views based on VMS and realizes as part of HBase. Our approach is capable of consistently maintain multiple views that may depend on each other. In the spirit today's KV-stores, our view maintenance architecture is designed to be incrementally scalable, thus, accommodating the addition of 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 at large scale. We also address fault tolerance and recovery to react to failing view managers. Our experimental evaluation quantified the benefits and cost of the approach and showed that it scales linearly in view update load and number of view managers running. There are many avenues for future work, such as exploring optimizations for the maintenance of multiple, overlapping view expressions and exploring automatic means for reacting to view maintenance load variations.

8. 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] P. Agrawal et al. Asynchronous View Maintenance for VLSD Databases. SIGMOD, 2009.

[8] Adam Silberstein et al. Feeding frenzy: Selectively materializing users' event feeds. SIGMOD '10. ACM.

[9] Apache Phoenix. `https://phoenix.apache.org/`.

[10] Ankita Kejriwal et al. Slik: Scalable low latency indexes for a key value store. In Memory Database Management Workshop, VLDB, pages 439–455. ACM, 2015.

[11] Bryan Kate et al. Easy freshness with pequod cache joins. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 415–428. USENIX Association, 2014.

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

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

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

[15] Y. Zhuge et al. The Strobe Algorithms for Multi-source Warehouse Consistency. DIS, 1996.

[16] H.-A. Jacobsen, Patric Lee, and Ramana Yerneni. View Maintenance in Web Data Platforms. *Technical Report, University of Toronto*, 2009.

[17] J. Blakeley et al. Efficiently Updating Materialized Views. *SIGMOD Rec.*, 1986.

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

[19] L. Colby et al. Algorithms for Deferred View Maintenance. *SIGMOD Rec.*, 1996.

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

[21] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[22] Derek Murray et al. Naiad: A timely dataflow system. SOSP '13, pages 439–455. ACM, 2013.

[23] FoundationDB. `https://en.wikipedia.org/wiki/FoundationDB`.

[24] Changjiu Jin et al. Materialized views for eventually consistent record stores. ICDE Workshops '10, pages 250–257. IEEE Computer Society,, 2013.

[25] Tilmann Rabl and H.-A. Jacobsen. Materialized views in cassandra. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, CASCON '14, pages 351–354. IBM Corp., 2014.