Dynamic Scalable View Maintenance in KV-stores

Jan Adler
Your Institution

Martin Jergler
Second Institution

Arno Jacobsen
Third Institution

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.

1 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.

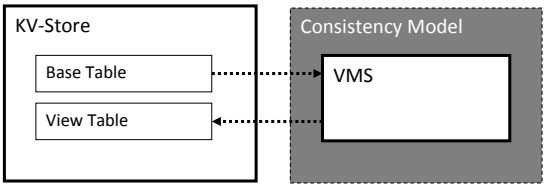

Figure 1: System overview

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.

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 2). As is common in the literature on view maintenance, we use a consistency model (Section **??**) 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 3). 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 5).

2 KV-store Model

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 distill a set of features our VMS requires from a KV-store. As HBase is modeled after Bigtable and Cassandra combines features of Dynamo and Bigtable, we restrict our discussion below to HBase, Cassandra, and PNUTS.

2.1 KV-store Design Overview

Some KV-stores explicitly designate a master node, i.e., HBase or Bigtable, while others operate without explicit master, i.e., 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 (see Figure 2). 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

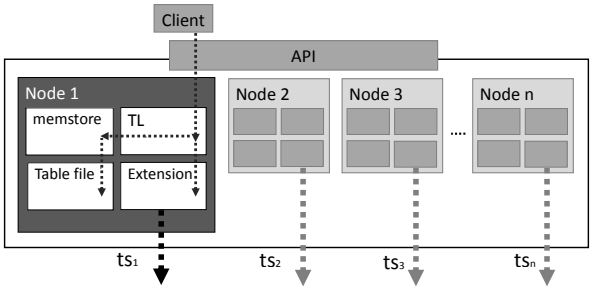

Figure 2: KV-model

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. With regard to key range management, a KV-store supports the set of events specified in Table 1.

2.2 API and 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.

Every node maintains a *transaction log* (TL), referred to as 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.

[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.

2.3 View Maintenance Integration

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," the 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 listed in Table 1 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.

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

Table 1: Administrative events

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 transaction log.

Method 1 may lead to inconsistent view states, as base data may 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, not only by the KV-store, but also by VMS. (iv) The TL contains client operations, not table rows, which is fine for incremental and deferred view maintenance.

[2]Our experiments confirmed that the penalty of reading from the file system are far smaller than intercepting events.

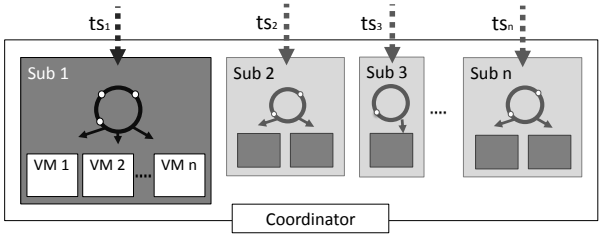

Figure 3: View Maintenance System (VMS)

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 our design. Finally, we discuss fault tolerance mechanisms provided by our design.

3.1 Design Overview

Figure 3 gives an overview of VMS, which is comprised of a *coordinator* and an arbitrary number of *sub-systems*. The coordinator manages system load and recovery, whereas the sub-systems – more specifically the *view managers* (VMs) in a sub-systems – update views. The input of VMS is a set of operation streams (ts_1,ts_2..ts_n); each emitted by a KV-store node (cf. Figure 2). Each VMS sub-system manages one stream of operations. A sub-system distributes the incoming operation stream to VMs. The number of VMs per sub-system is configurable. VMs can be dynamically *assigned* to or *removed* from sub-systems. The coordinator can also re-assign VMs from one sub-system to another.

The VM is designed to be light-weight and 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 to 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 sub-system distributes the arriving base table update stream to its VMs via consistent hashing by maintaining a hash-ring (cf. Figure 4), where active VMs are recorded. Row keys of arriving updates are hashed into the ring and associated in clock-wise direction with active VMs. In this way, a sub-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, while simultaneously guaranteeing the ordered propagation of base table updates to view tables, setting the basis for view table consistency.

Every VM maintains its own transaction log, referred to as *VM-log*. Before processing updates, a VM writes them to the VM-log. Just like the transaction log, the VM-log is kept available by the underlying file system and its 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 `SELECTION` and `COUNT` view V_1 and V_2, both derived from A). Then, VM runs the update program, and submits view table updates (to V_1 and V_2) 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 `SUM` view e.g., the VM reads the current sum from the view before applying the delta of the base table operation. These view queries are always get operations (i.e. single row access) and can be evaluated quickly. To access and update a view table, a VM acts as a client to the KV-store, using its standard client API. Given a base table operation, the VM retrieves (and caches) the view definitions of the views derived from that particular base table. Then, the VM runs the update program and submits view table updates via the client API. Depending on the view type maintained, the VM first queries the view table as part of the update logic. For example, in a `SUM` view, the VM reads the current sum 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 are evaluated efficiently by the underlying store.

To allow for parallel updates of multiple VMs on the same view record, we use a test-and-set mechanism (as suggested in [23]). [3] When updating a table record, a VM sees (tests) whether a record has been concurrently modified between a read and an update.

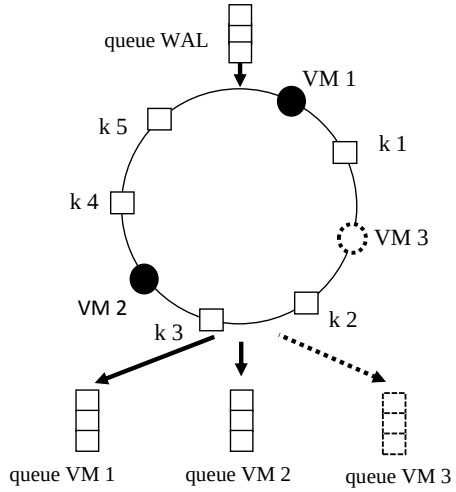

Figure 4: Sub-system setup

Example: Let two VMs VM_1 and VM_2 retrieve the value of the same view record $(x1,\{(col,a)\})$. Let VM_1 add the delta b such that $(x1,\{(col,a+b)\})$ and VM_2 add the delta c such that $(x1,\{(col,a+c)\})$. Now let both VMs write back their values to the view. As a result one of the delta values (either b or c) will be overwritten and not reflected in the view. Using a test-and-set method prevents the scenario. Assuming VM_2 writes second; when trying to put the new value, the test-and-set method tests the old value a. It fails because the old record value changed concurrently to $a+b$. Thus, VM_2 fetches the updated value again and re-computes $(x1,\{(col,a+b+c)\})$. This time the test-and-set succeeds and the record is written.

3.3 Consistency Considerations

Incremental maintenance of views and heavy parallelisation of view update computation increases the probability of data inconsistency significantly. As Jacobsen et al. [23] suggest, we use a consistency model to measure and enforce consistency. The model describes a base table as a sequence of base states (achieved by a number of client updates) and a view table as a number of corresponding view states. It defines the following consistency levels, building on top of each other: convergence, guarantees a correct final view state; weak consistency, guarantees correct intermediate view states; strong consistency, guarantees correct and ordered intermediate view states; complete consistency, guarantees every base state is represented by a view state.

VMS guarantees views converge and views are strongly consistent. Consistent hashing ensures that record timelines are respected in update propagation for view maintenance. Test-And-Set methods guarantees that parallel view updates don't cause inconsistency.

However, these guarantees only hold under the assumption of a "static" VMS configuration: A fixed number of nodes and VMs. Given a "dynamic" context, where VMS assigns and

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

withdraws VMs or allows KV-store to add and remove nodes or move key-ranges, further exacerbates view consistency. The following example shows why:

Example: Consider a sub-system set-up as shown in Figure 4. Two VMs VM_1 and VM_2 are already assigned to the sub-system: 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 VMs. Further assume the sub-system selects VM_2 as the next responsible VM for t_1. Then, t_1 is put in the queue of VM_2. Now assume, VM_3 is assigned to the sub-system 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 the 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 updates 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. This has to be prevented in order to preserve convergence of views.

3.4 Dynamic behaviour

We now refine the behaviour of VMS to allow for dynamic state changes in sub-systems. First, we discuss effects due to VMS actions; second, we discuss effects due to KV-store events (Table 1).

For reasons of recovery, load balancing and scaling, VMS performs the following actions: *add*, makes the VM resource available to the VMS; *remove*, takes away a VM resource; *assign*, assigns a VM to a sub-system; *withdraw*, withdraws a VM from a sub-system; and *re-assign*, re-assigns a VM from one sub-system to another. Adding a VM to VMS (or removing it) does not affect view maintenance; both actions leave the VM idle, waiting for subsequent commands. The re-assign action is a combination of withdraw and assign. Now, we explain how assign and withdraw actions are performed safely (i.e., without hurting consistency).

To process assign and withdraw, we use so called *markers*. Markers identify whether a VM has completed the processing of all operations (that were sent before) and, thus, allow the safe assignment and removal of VMs. Markers are sent just like client operations to the VM: they are inserted into the queue of the VM and become a part of the operation stream. When the VM reads an operation and recognizes a marker, it replies with an acknowledgement to the sub-system.

To realize the marker mechanism, we adapted parts of the sub-system and VM implementation. The logic on the sub-system 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 sub-system to assign (or withdraw) VMs to (or from) itself: *createQueue*(vm) creates a new queue for a particular VM and *deleteQueue*(vm) removes an existing queue for a particular VM at the sub-system. The queue for a VM can only be deleted, if it is empty and no operation is queued. *queue*$(vm,operation/marker)$ inserts, either a base table operation, or a marker into the queue of a particular VM. The methods *activateQueue*(vm) and *deactivateQueue*(vm) start or stop the sending thread that keeps transferring the operations in the queue for a particular VM. A VM can be added to the hash ring by *insertHash*(vm), or removed from the hash ring by *removeHash*(vm).

Assign view manager – When a VM is assigned, it is added to the hash-ring of the sub-system. Unless the hash-ring is empty, the new VM is assigned a key range that is, at the same time, withdrawn from another VM. Figure 4 shows how a sub-system does view maintenance. Recall, that the sub-system selects the responsible VM by applying the 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 sub-system, it must wait, until the first operation has terminated.

Algorithm 1 Safe assignment procedure at sub-system

```
procedure assignVm(vm_a, VM_sub)
    createQueue(vm_a)                    ▷ create queue
    insertHash(vm_a)                     ▷ add VM to hashring
    for all vm ∈ VM_sub do
        queue(vm, m_a)                   ▷ queue markers
    end for
    for all vm ∈ VM_sub do               ▷ wait for acks
        receiveMessage(vm)
    end for
    activateQueue(vm_a)                  ▷ activate queue
end procedure
```

The procedure *assignVm* takes two parameters: vm_a, the VM that should be assigned to the sub-system, and VM_{sub}, a set of VMs that are already assigned to the sub-system. 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 sub-system has received acknowledgements from all VMs, it is guaranteed that no operation in the key range of the newly added VM vm_a is still pending. Referring back to the above example: 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 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 may be violated. At the moment where a VM is withdrawn, i.e., removed from the hash-ring, its queue might still contain operations. If another VM that acquires the key range is fast enough, it might processes operations before the withdrawn VM has finished. Again, the timeline of base records is changed. In order to prevent inconsistencies, we designed Algorithm 2 analogous to Algorithm 1. It takes two parameters: vm_w, the VM that should be withdrawn from the sub-system, and VM_{sub}, the set of VMs that is assigned to the sub-system.

Algorithm 2 Safe withdraw procedure at sub-system

```
procedure withdrawVm(vm_w, VM_sub)
    for all vm ∈ VM_sub ∧ vm ≠ vm_w do
        deactivateQueue(vm)                ▷ deactivate queues
    end for
    removeHash(vm_w)                       ▷ redirect operations
    queueMarker(vm_w, m_w)
    for all vm ∈ VM_sub do
        receiveMessage(vm)                 ▷ wait for acks
    end for
    removeQueue(vm_w)
    for all vm ∈ VM do                     ▷ activate queues again
        activateQueue(vm)
    end for
end procedure
```

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

3.5 Fault Tolerance

Failure detection and recovery play a critical role in VMS. In this section, we analyse the behaviour 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 sub-system. The sub-system 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 VM instance. Upon start-up, it tells the new VM to replay the VM-log of the crashed VMṪhe new VM contacts ZooKeeper and retrieves the last processed sequence number of the crashed VM (cf. Section 3.2). 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.)

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 directly flushed to disk.

The transaction log of a crashed node is still available (due to replication in the underlying file system, HDFS, for example.) Thus, the sub-system that is streaming the operations from the crashed node's TL continues reading to the end of file. As the KV-store starts moving key ranges to different nodes, the VMS reacts as described in Section **??** – consistency is guaranteed in the process. Now, that the stream (of the crashed node's TL) runs dry, the VMS re-assigns all VMs to a different sub-system.

Based on the above reasoning, we conclude that VMS is able to prevent loss and duplication of operations during crashes.

4 View Maintenance Concept

In this section, we develop techniques for maintaining the following view types in VMS: `SELECTION`, `PROJECTION`, `INDEX`, aggregation (i.e. `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`) and join (i.e. `INNER`, `LEFT`, `RIGHT`, `FULL`). Internally, VMS provides a number of auxiliary views, which we refer to as the `DELTA`, `PRE-AGGREGATION` and `REVERSE-JOIN` view. Finally, we explain how these view types compose to form higher level view constructs (e.g., composing an aggregation, a selection, and a join view.)

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, which are shared within and between view definitions. Thus, their use amortizes as more complex views are managed by VMS. Auxiliaries also speed up view maintenance significantly, as we show in Section 5. The update programs that compute the different view types make up most of the VM logic. In what follows, we describe each auxiliary view type, including the problem it addresses and describe how it is maintained given base table updates.

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

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 [23]. 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*. The row key of the base table serves as the row key of the view table and a single base record uniquely maps to a single view table record.

A PROJECTION view selects a set of columns from a base table. Similar to the SELECTION view, the VM uses the row key of the base table as row key for the view table. The computation of the PROJECTION view changes analogously when derived from a DELTA view. To save storage and computation resources, we could combine DELTA, PROJECTION and SELECTION into 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.

Count and Sum – 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 – 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 [23]. 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. The INDEX view is a special case of the PRE-AGGREGATION view. If the indexed key is the same as the aggregation key, we can combine both view types. Further, the PRE-AGGREGATION can be combined with a REVERSE-JOIN view (see description above). Thus, we receive three different view types at the cost of one.

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.

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, respec-

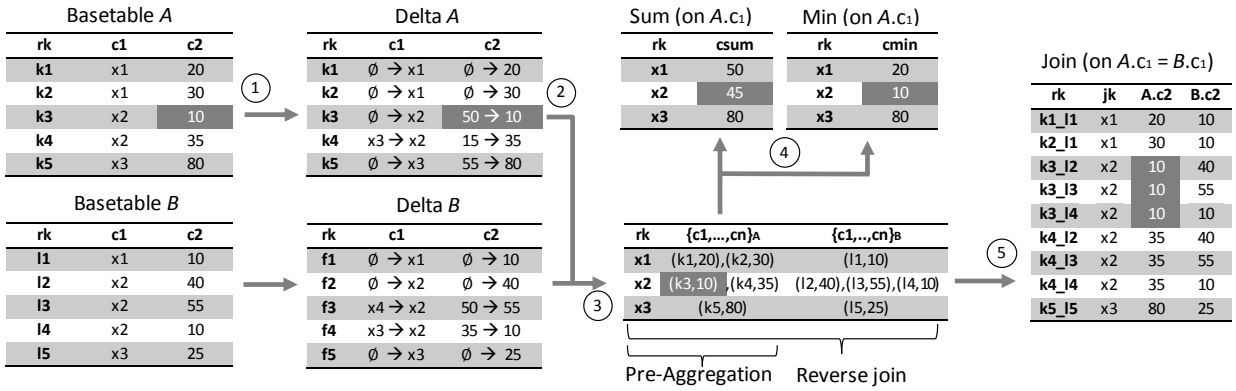

Figure 5: View table example

tively. These views reorder and group the base table records according to a secondary key (found in columns $A.c_1$ and $B.c_1$); one may argue that the groups of such an aggregation key 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, as our experiments demonstrate. In the setup 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

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.

5.1 Experimental setup

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

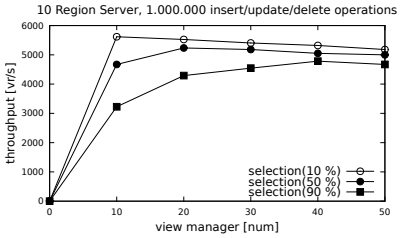

Figure 6: Selection performance

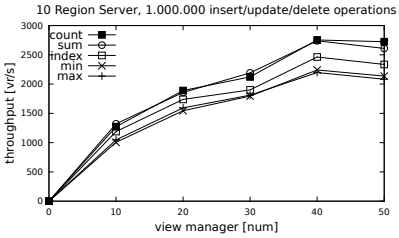

Figure 7: Aggregation performance

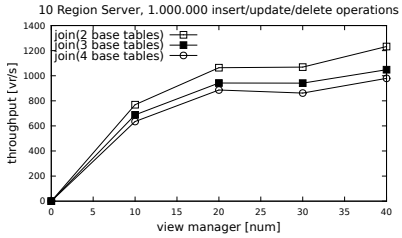

Figure 8: Join performance

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.

5.2 Experiments

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 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 9, 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,

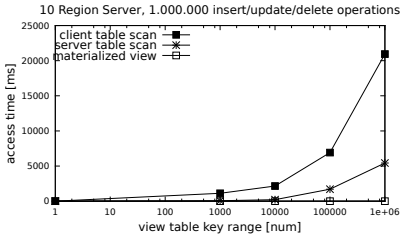

Figure 9: Benefit of view maintenance

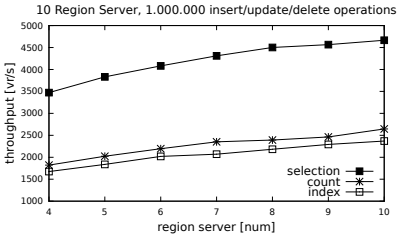

Figure 10: Scale region servers

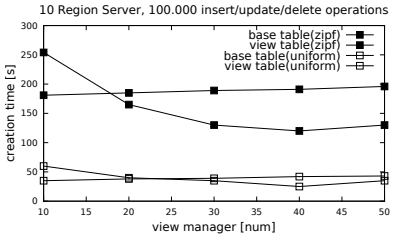

Figure 11: Zipf distribution

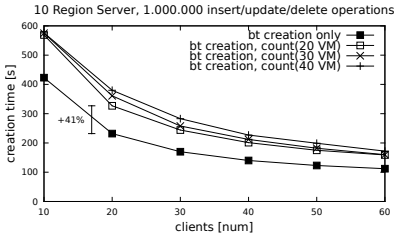

Figure 12: Cost of view maintenance

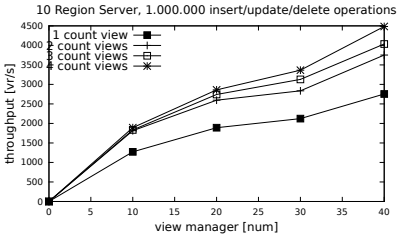

Figure 13: Scale views

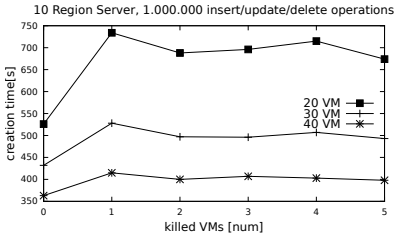

Figure 14: Fault tolerance - Impact of crash

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 12 shows the base table creation time with and without concurrent view maintenance. We track creation time as the number of clients increases. Figure 12 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 10, 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 performance, also for VMs, due to the additional I/O capacity.

Figure 13 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 11 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 14 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 1980s [11, 12, 14–16]. Blakeley et al. [14] developed an algorithm that reduces the number of base table queries during updates of join views. Zhuge et al. [11] developed the ECA algorithm, which prevents update anomalies. Colby et al. [16] 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 [11, 15, 17]. 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 data warehouses, view maintenance has also been studied extensively. Early approaches aimed at integrating incremental, deferred view maintenance based on distributed data sources [12, 18, 19]. Zhuge et al. [19] introduced Strobe, 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. [18] defined the Sweep algorithm, which like ECA and Strobe, compensates via error terms for interfering updates. In these approaches, a data warehouse is considered a central component, where the flow of base table updates is joined and a global timestamp is set. Thus, operations can be serialized in order to obtain an order. In KV-stores update propagation is distributed and no global order exists, thus, the above approaches do not apply.

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 [10, 20–22] and in academia [23–29].

Percolator [20] is a system specifically designed to incrementally update a Web search index as new content is found. Naiad [21] 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 [10] 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." [10] The work presented in this paper constitutes a foundation upon which future Phoenix designs could draw.

Agrawal et al. [24] 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. [24].

In similar spirit, Silberstein et al. [25] 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 [28, 29]. Pequod [28] 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 [29] 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.

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

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

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

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

[18] D. Agrawal et al. Efficient View Maintenance at Data Warehouses. *SIGMOD Rec.*, 1997.

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

[20] 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.

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

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

[23] Ramana Yerneni Hans-Arno Jacobsen, Patric Lee. View Maintenance in Web Data Platforms. *Technical Report, University of Toronto*, 2009.

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

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

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

[27] Tilmann Rabl and Hans-Arno 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.

[28] 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.

[29] 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.