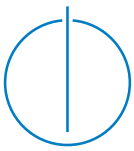
Jan Adler

Materialized views in distributed key-value stores

Technische
Universität
München



Technische Universität München

Fakultät für Informatik

Materialized views in distributed key-value stores

Jan Adler

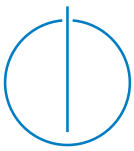
Vollständiger Abdruck der von der Fakultät für Informatik der Technische Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. Jan Křetínský
Prüfer der Dissertation:	1. Prof. Dr. Hans-Arno Jacobsen
	2. Prof. Dr. Georg Groh

Die Dissertation wurde am 25.05.2020 bei der Technische Universität München eingereicht und durch die Fakultät für Informatik angenommen, am 01.07.2020



Technische Universität München

Department of Informatics

Materialized views in distributed key-value stores

Jan Adler

Complete copy of the dissertation approved by the Department of Informatics of the Technical University of Munich in partial fulfillment of the requirements for the degree of

Doktors der Naturwissenschaften (Dr. rer. nat.)

Chair:	Prof. Dr. Jan Křetínský
Dissertation examiners:	1. Prof. Dr. Hans-Arno Jacobsen
	2. Prof. Dr. Georg Groh

The dissertation was submitted to the Technical University of Munich on 25.05.2020 and accepted by the degree-awarding institution of Department of Informatics on 01.07.2020

Abstract

Distributed key-value stores have become the solution of choice for warehousing large volumes of data. However, their architecture is not suitable for real-time analytics. To achieve the required velocity, materialized views can be used to provide summarized data for faster access. The main challenge, then, is the incremental, consistent maintenance of views at large scale. Thus, we introduce our View Maintenance System (VMS) to maintain SQL queries in a data-intensive real-time scenario. VMS can be scaled independently and provides strong guarantees for consistency, even under high update loads. We evaluate our full-fledged implementation of VMS on top of Apache's HBase using TPC-H workloads and queries. Exploiting parallel maintenance, VMS manages thousands of views simultaneously, handles up to $300M$ base updates per second and provides permanent access to fresh view data in under $5ms$.

Zusammenfassung

Verteilte key-value stores sind ein Typ moderner Datenbanken um große Mengen an Daten zu speichern und zu verarbeiten. Trotzdem erlaubt ihre Architektur keine analytischen Abfragen in Echtzeit. Eine Lösung, um die nötige Verarbeitungsgeschwindigkeit zu erreichen und schnellen Zugriff auf berechnete Daten zu erlauben, stellen materialisierte Views dar. Die Herausforderung ist dann, das inkrementelle und konsistente Aktualisieren der Views. Aus diesem Grund präsentieren wir unser View Maintenance System (VMS), das datenintensiven SQL Abfragen in Echtzeit aktualisiert. VMS skaliert unabhängig vom Basissystem und garantiert Konsistenz der berrechneten Ergebnisse auch unter hoher Last. Wir evaluieren unsere vollständige Implementierung von VMS als Erweiterung von Apache's HBase mit TPC-H Datensätzen und Abfragen. Unter Ausnutzung von maximaler Nebenläufikeit verwaltet VMS tausende Views gleichzeitig, berechnet bis zu $300M$ Basistabellen-Updates pro Sekunde und garantiert permanenten Zugriff auf aktuelle Viewdaten in unter $5ms$.

Acknowledgments

This dissertation and the included research was written at the Department of Informatics at the Technical University of Munich under the supervision of Prof. Dr. Hans-Arno Jacobsen. I thank Prof. Jacobsen for deep insights into distributed systems, writing feedback and support, encouragement after bad reviews and having the opportunity to go on great retreats.

I would also like to thank Prof. Dr. Groh for agreeing to be the second examiner and, in general, for being a great technical advisor. Further, I want to thank Prof. Dr. Jan Křetínský for accepting to chair the commitee.

Special thanks go to …

- my colleagues: Martin Jergler for tutoring and writing support, Christoph Doblander for business plans and companionship, Kaiwen Zhang for great ideas and support, Victor del Razo, Amir Hesam Shahvarani, Mohammedreza Najafi, Matthias Kahl for climbing talks and other good stuff, Thomas Kriechbaumer for letting me screw up the cluster, Anwar UI Haq, Elias Stehle for insane table soccer matches, Daniel Jorde for ranting with me about the review system, Pezhman Nasirifard, Ruben Mayer and Alexander Isenko.
- the many students I had the pleasure to supervise for their bachelor or master thesis.
- my mother Renate Adler, my father Walter Schmitt and his wife Petra Schoch-Schmitt
- my sister Anna Adler, her husband Jochen Adler and Elena and Robin.
- my wife Angie and my two sons Louis and Raphael for being the most important persons in my life.

Contents

Chapter 1

Introduction

In this chapter, we motivate the topic of this thesis. Then, we state the research problems and explain the challenges that have to be mastered in order solve them. Finally, we provide our contributions and present the organization of the thesis.

1.1 Motivation

The properties of major Internet players are backed by what has become known as *key-value stores* (KVSs). KVSs serve millions of client requests and handle the production of terabytes of data on a daily basis [1]. Popular examples include Google's Bigtable [2], Amazon's Dynamo [3], Yahoo's PNUTS [4], Apache HBase [5], and Cassandra [6].

As opposed to earlier generation KVSs, for instance, BerkeleyDB [7], whose primary intention was to provide a main-memory database to persist application configurations, the KVSs, we consider in this thesis, are highly distributed systems designed for large-scale data warehousing.

Such KVSs are highly available and provide advanced features like load balancing, fault-tolerance, and incremental scalability [2, 3, 4, 5, 6]. For example, KVSs scale-out horizon-

tally by partitioning data and request load across a configurable number of nodes. To achieve these properties at scale, KVSs sacrifice a relational data model and an expressive query language and only offer a simple API, typically comprised of only *get*, *put*, and *delete* operations on single records (scans can be considered as repeated *gets*).

In a Big Data infrastructure, processing of the data, warehoused in KVSs is done externally. The raw data is loaded into a large-scale distributed computing framework, such as Hadoop or Spark. This approach works to the extent of an off-line batch analysis, but it does not provide the velocity required for real-time analytics of incrementally changing data [8]. To speed up the processing, there is a need to store aggregated data for faster access by the processing engine.

To this end, point solutions appeared raising the level of abstraction of a KVS by either partially or fully materializing the desired application-level queries as views through the underlying store [9, 10, 11]. In this context, secondary indices have been added to KVSs [9, 12], caching of application queries has been introduced [13], and massive processing of selection views (e.g., for following news feeds) has been enabled [10]. However, these solutions lack the ability to manage views for a wide variety of SQL query operators for KVSs, which limit their applicability.

1.2 Problem statement

To address the aforementioned limitations in KVS, and provide results of SQL query operators over large distributed data sets in real-time, there is a number of problems to solve which are described in the following.

First, the simple API of KVS has to be extended with a set of *additional API* functions; this includes a range of common SQL query operators such as projection, selection, aggregation, joins but also a set of more specific SQL functions (e.g., exists or case/when). Second, a mechanism has to be developed and integrated with KVS to provide the results of the additional API functions to clients. Thereby, the focus is not only on delivering the results in reasonable time but also on updating the existing results efficiently, such that fresh view data is always available to clients.

The mechanism we choose in this thesis, to achieve our goals, are *materialized views* that are updated using *incremental view maintenance* (see Chapter 2). While materialized views and incremental view maintenance are well researched topics in centralized environments (see Chapter 3), there are many challenges transitioning them to a distributed context.

In a centralized context, the base table is located on a single machine. The base table can be loaded in a single run and results of analytical queries can be materialized into a view, located on that same machine. All updates that occur on a base table can be directly applied to the materialized view; sequential processing guarantees that updates are always delivered in correct order.

However, in today's large-scale KVS, base tables, holding large amounts of data, are partitioned and distributed over a whole cluster of nodes. Loading a base table requires a coordinated collective action; analytical queries are computed at many nodes in parallel and may require multiple rounds of redistribution. Further, streams of base table updates occur simultaneously and after being routed to different nodes are applied to the materialized view in parallel. Thus, in the first step of the thesis, we design and develop a distributed *view engine* that is capable of performing the additional API functions and uses KVS as view materialization layer. The main challenge then, is the incremental maintenance of views at large scale; the view engine must keep the update ordering – or more generally, sustain consistency – while distributing and applying thousand of updates in parallel.

The amounts of data stored in base tables grow constantly, so does the demand for analytical capabilities [1]. Most of the current research on (distributed) view maintenance is focused on a single base and view table setup [14, 15, 16]. Thus, in a second step, we identify the need to materialize and maintain ten thousands of views in a KVS in parallel. Usual measures of horizontal scaling (i.e., add more resources to the system) do not suffice to handle the sheer amount of updates that arise in the system due to the write amplification (number of updates × number of views). For that reason, we must develop smarter ways to provide more efficient maintenance techniques. Thereby, the main challenge is the identification and exploitation of synergies between the maintenance processes in what we call *multi-view* optimization.

Many modern frameworks and database systems support methods for combined off- and online analysis [17, 18, 19, 20, 21]. Seamless integration of existing data (by batch analysis) and update streams (by online analysis) is the method of choice to further reduce latency and guarantee immediate freshness of results. This also seems to be a perfect fit for the concept of materialized views in KVS. Thus, in a third step, we discuss *hybrid maintenance* strategies in the KVS context. The challenge here, is the definition of basic batch and incremental strategies with regard to KVS architecture and the consistent realization of their hybrid derivatives.

1.3 Contributions

We make the following three main contributions:

1. We introduce the VMS architecture to rapidly materialize and maintain views at scale. Along with VMS, we propose a novel consistency concept allowing for strongly consistent views in a highly parallelized maintenance environment (see Chapter 4).
2. We propose a multi-view concept to optimize the materialization and maintenance of ten thousands of views in parallel (see Chapter 5).
3. We develop basic and advanced incremental and batching strategies in KVS and introduce our concept of consistent hybrid maintenance for maintaining views at scale (see Chapter 6).

1.4 Organisation

We provide an explanation of backgrounds in Chapter 2. We discuss the related work of our three main contributions in Chapter 3.

We discuss our View Maintenance System (VMS) in Chapter 4: we provide an abstract KVS model and introduce the VMS architecture in Section 4.1, we introduce our consistency model and theorem in Section 4.2, we explain how VMS achieves consistency in Section 4.3, we present the supported view types of VMS in Section 4.4, we evaluate VMS in Section 4.5.

We discuss our concept of multi-view optimization in Chapter 5: we introduce the multi-view concept for each view type in Section 5.1, we present a cost model for optimization in Section 5.2, we evaluate our multi-view concept in Section 5.3.

We discuss our concept of consistent hybrid view maintenance in Chapter 6: we introduce basic incremental strategies in Section 6.1, we introduce basic batching strategies in Section 6.2, we introduce hybrid strategies in Section 6.3, we evaluate our strategies in Section 6.4.

We provide a conclusion and our outlook on future work in Chapter 7.

Chapter 2

Background

In this chapter, we provide high-level explanations of the most important aspects of the thesis. The concrete basics needed (for example, KVS or consistency basics) are explained in the respective sections along the thesis.

2.1 Large-scale distributed systems

Before the Big-Data era, many of the transactional and analytical tasks were performed by monolithic systems hosted on single machines. But nowadays requirements, especially on the amounts of (unstructured) data processed, have made it necessary to distribute these tasks to a cluster of machines. This gave rise to a wide range of new distributed system types. Distributed file systems, distributed lock services, distributed key-value stores, just to name a few. These systems, many of them either originating at the big global Internet players (e.g, Google, Facebook, Amazon) or being open sourced by the Apache Foundation (e.g., Hadoop, Zookeeper, HBase), are equipped with scalable processing capabilities and autonomous administration and load balancing features.

Exactly like these systems, the view engine we develop, is an inherently distributed framework. This means, the design of its architecture centers around the following

questions: (1) parallelization: how can each step of the computation be shared evenly by all machines to achieve maximal performance of the total setup, (2) consistency: how can the distributed logic in the system route and compute millions of updates simultaneously and the final results come out correctly, (3) communication: how can the machines be synchronized with a minimum of communication overhead to provide global results, (4) availability: how can results be kept available for clients at all times, and (5) fault-tolerance: what happens when a machine in the cluster crashes and computation has to be resumed by another one.

2.2 Materialized views

Views are perspectives on a database. A view provides data of a database to a client and simultaneously prevents client access to the original database tables. When discussing views, the database tables are always referred to as the *base tables*, whereas the views are always referred to as the *view tables*. View tables can reproduce the data of a database, they can provide a specific selection of data or they can compute new data out of the base table contents. Basically any analytical operation that can be derived from base table data can also be represented in a view table.

A special type of views are *materialized views*. Materialized usually refers to the fact that the contents of the view are stored in form of a separate table, in the database system. Nevertheless, materialized views can be also stored as a file or a data structure in-memory. Opposed to that a virtual view would be loaded from the base table in the moment it is requested. While the on-demand idea behind virtual views is certainly desirable, our context favors high availability of results and as such views are always materialized.

To provide fast results, our system uses two different types of materialized views. One type of views are the *final views* that are actually stored into KVS and accessible by the clients. These view tables are materialized in the original sense because they provide results as database tables. The other type of views are the *intermediate views* which are stored in-memory and which are only used for internal purposes. Definition wise intermediate views are also materialized views, as their data is materialized in-memory.

Arguably, those intermediate views could be also identified as virtual, as their contents are volatile and (depending on the strategy) loaded on-demand. However, when talking about materialized views in the context of this thesis, we primarily refer to the final views that are exposed to the clients.

2.3 Incremental view maintenance

In general, the topic of materialized views is tightly coupled with the topic of incremental view maintenance. In the moment a view gets materialized, it represents the result of an analytical query that is based on the current state of the base table. As soon as updates over the base table are performed, the computed results in the materialized view are out-of-date. As complete reevaluation of the results is a costly task, incremental view maintenance is performed to update only those parts of the view that are related to the updated parts in the database.

While incremental view maintenance is a very efficient method to update a view, it usually complicates the task of providing correct results. The set of updates which has been applied to the base table must be consistently applied to the view to guarantee correctness. Updating records in a base table multiple times leads to multiple different versions of a base table which should not be confused during view maintenance. Ultimately, missing only a single update, already renders the overall result in a view invalid. Thus, materialized views must be carefully maintained, otherwise they are of no use.

2.4 Base and view table definition

To describe tables in the thesis, we define relation sets (as done for relational algebra). Although KVSs support a more flexible (NoSQL) data model, we model base and view tables as sets of attributes and records, as it is done for established SQL databases. This model facilitates understanding and provides a clear conceptual definition. In general we model: (a) $R = (A_1, .., A_n)$, as a table consisting of a set of attributes where the primary key

(i.e., row-key in KVS) of a table is denoted as attribute $\bar{A}$; (b) $r = R(a_1, .., a_n)$, a record with attribute values $a_i \in A_i$ which can be accessed by $r.A_i$; (c) $put(R(a_1, .., a_n))$, a put operation to table R; (d) $del(R(a))$, a delete operation and (e) $g = get(R(a))$, a get operation. We refer to put and $delete$ as update operations in the following. Still, a put can be either an insert into the base table or an update (i.e., modification) of an existing record.

In this thesis, when talking about analytical queries, we mean SQL expressions; syntax as defined by the SQL-92 standard. While our system does not implement the full scope of this comprehensive standard, the complete feature set as defined by the TPC-H benchmark is supported (which largely extends SQL beyond the standard SPJA types). The analytical query constitutes the *view definition*. It is provided on creation of the view and cannot be changed afterwards. It determines the results that are provided by the view and also defines the update process that is required in order to refresh the view.

In general, we describe a set of base tables as $R_{set} = \{R_1, .., R_n\}$ (let each be structured by attributes as defined above) and a view table as V. Then, the relation between both is expressed as $V = View(R_{set})$. Function $View$ is specified by the elements of relational algebra (i.e., $\sigma, \gamma, \bowtie$) which have been obtained from the view definition. It describes the necessary operations to transform the base tables into the view table and is used by the view engine to construct a maintenance plan.

Example 2.4.1*: A small example illustrates the notation. Let a base table be defined as* $R = (\bar{K}, X, Y)$ *with primary attribute* $\bar{K}$ *and secondary attributes* X *and* Y*. Then, let an analytical query be provided as:*

$$SELECT \;\; sum(Y) \;\; FROM \;\; R \;\; WHERE \;\; Y > 5 \;\; GROUP \;\; BY \;\; X$$

The corresponding view table is defined using schema $V = (\bar{X}, sum(Y))$*; the view function translates to* $V = \gamma_{X,sum(y)}(\sigma_{Y<5}(R))$*. Updating base table R, simultaneously requires an update of* V*. The state of the view depends on the state of the base tables.*

2.5 Generalized multi-sets

To illustrate the different view types (starting with Chapter 4.4), we use a bag algebra [22, 23, 24, 25, 26], commonly adopted for SQL operations and for the modeling of incremental aspects in systems [14, 16, 27]. We describe tables R and update sets (ΔR) in form of generalized multi-sets (also known as bags) with positive and negative multiplicities. Thereby, inserts have multiplicity plus one (i.e., $u = (r, +1)$), deletes have multiplicity minus one (i.e., $u = (r, -1)$) and updates are realized as a combination of both (i.e., $u = \{(r', -1)(r, +1)\}$). We capture the change in base tables using the additive union of these multi-sets (e.g., $R \uplus \Delta R$). Further, we describe the process of classical incremental view maintenance as follows:

$$\begin{aligned} View(R \uplus \Delta R) &= V \uplus View(\Delta R) \\ View(R_{set}[R = R \uplus \Delta R]) &= V \uplus View(R_{set}[R = \Delta R]) \end{aligned} \tag{2.5.1}$$

Given that a base table is updated, the system does not recompute the view using the entire base table. Instead it keeps the results materialized in view $V = View(R)$. When updates arrive it computes the delta as $View(\Delta R)$ and merges it with V. In case that there are many base tables, the system computes the delta of each base table separately in relation to all other base tables (i.e., $View(R_{set}[R = \Delta R])$).

Chapter 3

Related work

In this chapter, we review the related work starting from the general and more basic approaches of view maintenance going to the most recent and more specific approaches that can be found in today's large-scale distributed KVSs. We discuss the related work for our three contributions, separately, reviewing each in its own section.

3.1 View Maintenance System

Research on incremental view maintenance started in the 80s [27, 28, 29, 30, 31]. Blakeley et al. [28] developed an algorithm that reduces the number of base table queries during updates of join views. Colby et al. [27] introduced deferred view maintenance based on differential tables that keeps around a precomputed delta of the view table. 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 KVS we consider in this thesis.

Zhuge et al. [30] considered view maintenance in multi-source data warehouse scenarios. An update event causes the warehouse to query affected source systems to calculate

the view relation for the updated record. As base data changes during these queries, compensation queries are needed. Much attention has been given to preventing update anomalies when applying incremental view maintenance [29, 32] (cf., ECA Algorithm). VMS completely avoids the problem by not falling back to base table access (i.e., by not using base table scans) at all.

Still today, incremental view maintenance is a relevant research topic. Formerly applied to standalone warehouses, view maintenance is nowadays used in large-scale distributed (streaming) infrastructures. In [14, 16] approaches for in-memory and batch-wise (incremental) view maintenance are introduced to parallelize and speed-up view maintenance vastly. However, there is no notion of consistency in these works, materialization is done as an exclusive in-memory approach. VMS differentiates itself from these works by providing strong consistency and maintaining materialized views in form of persistent standard KVS tables.

An approach for view maintenance in KVSs together with different consistency models has been presented in [33]. We leverage these consistency models in the thesis, apply them to attain consistency for different view types and greatly extend the scope of views materialized. However in [33], only single view types are materialized (i.e., selection view, aggregation view). Combined or nested queries with multi-table joins are not possible. Further, only weak consistency can be achieved for some of the view types (kfk-joins, min/max views).

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

Cui et al. [34] introduced the concept of auxiliary views as an optimization for the dual purpose of view maintenance and data lineage tracing. However, auxiliary views are the materialization of an intermediate result that is stored to compute the lineage of a view record. We defined the concept of Pre-Processing Views. In contrast to the

auxiliary view [34], the Pre-Processing View represents a preliminary step, which serves to facilitate and speed up the consistent processing of subsequent views. Our Reverse-join, for example, is not just a materialization of an intermediate result. It fulfills several purposes, including the consistent maintenance of a join keys timeline.

In recent years, there has been a rising interest in developing support for the materialization of views in a KVS, both in open source projects and products [8, 11, 35, 36] and in academia [9, 10, 12, 13, 33, 37]. Percolator [8] is a system specifically designed to incrementally update a Web search index as new content is found. Naiad [35] is a system for incremental processing of source streams over potentially iterative computations expressed as a data flow. Both systems are designed for large-scale environments involving thousands of nodes, but are not addressing the incremental materialization of the kind of views considered in this thesis.

The Apache Phoenix project [11] develops a relational database layer over HBase, also supporting the definition of views. Few implementation details about the views are revealed, except for the fact that updatable view definitions are limited to selection views, and view results are generated by periodically scanning the base tables. Also, a long list of view limitations is revealed by Phoenix [11]. *For example, "A VIEW may be defined over only a single table through a simple SELECT * query. You may not create a VIEW over multiple, joined tables nor over aggregations." [11]*

3.2 Multi-view processing

In the context of databases, *multi-query* optimization and *view selection* have been widely discussed in the literature [38, 39, 40, 41, 42]. However, in traditional query processing, incremental updates of base tables and view tables are usually not an issue. Systems work with pure table workloads, there is no need to capture and manage the state of records (e.g., of a selection view, see Section 5.1.1).

View selection, a common sub problem in multi-query optimization, consists of finding the right subset of view candidates to be materialized [43, 44]. Our approach is heavily centered around incremental processing (capturing the state of base records).

In this scenario, the introduced Pre-aggregation and Reverse-join views are materialized as distributed intermediate tables (in-memory) which speed up computation. Thus, the question is not which views should be materialized but how intermediate views can be combined to serve multiple maintenance processes.

The same applies to many modern algorithms for sharing results that are based on the MapReduce paradigm [45, 46, 47, 48]. While providing a similar data model as our solution (records are processed in a key-value format), most MapReduce approaches adopt a strictly batch-oriented processing style. There are some approaches fostering an incremental processing style but they are not optimized for multi-view optimization.

Recent work in the field of view maintenance concentrates on incremental maintenance strategies [14, 15, 16] and embraces in-memory acceleration. However, most approaches foster an incremental processing style, which is feasible only when incremental maintenance is executed starting from an empty base table (i.e., from zero). Likewise, such approaches concentrate on specific aspects (optimization) of incremental maintenance, evaluation is exclusively done in a single view setup (on top of a single query).

In [49] optimization with regard to a multi-view setup can be found. However, the paper relates to a publish-/subscribe context only, in which views are defined as subscriptions evaluating simple selection predicates over XML-tagged data (no nested SQL queries, including, e.g., multi-table joins). While the experimental evaluation focuses mainly on rewrite performance of the view definitions ($100k$ views), the actual workload to be maintained is comparably small (200 documents, $10k$ views).

3.3 Consistent hybrid view maintenance

Hybrid processing modes provide high processing throughput and low latency for SQL-like constructs. As such the hybrid processing approach can be found in many different fields of research. We provide a quick overview of hybrid processing and, finally, we explain the (more specific) hybrid approaches that are known to the field of view maintenance

A number of MapReduce online extensions are available [18, 19, 50]. However, in the named approaches hybrid is created by either combining MapReduce with a streaming system or adding streaming support to MapReduce itself [19, 50]. As these approaches add streaming capabilities to the MapReduce processing style, they can also be used for hybrid view maintenance. However, all named examples are coupled to the MapReduce paradigm. While it is possible to load KVS data to MapReduce (and also write it back), it is still an external solution and not comparable to the in-line view processing provided by us. As such MapReduce solution cannot make use of KVS internals such as filters, observers or write buffers.

There are some interesting approaches in which developers capture both, batch and online processing in an advanced programming paradigm (cf., Map Reduce 2.0) that can be used by developers to define high-level SQL-like constructs. Summing Bird [17] is one such a domain-specific language that compiles into a mixed architecture consisting of MapReduce jobs and an Apache Storm topology. While this language allows for a combined operation mode, the system uses two different databases and the results are combined offline. Our approach combines base table updates already at record level, merging them into one stream of complete and most recent view updates.

Hybrid transactional and analytical processing (HTAP) concepts [51, 52, 53, 54] are benefit-wise relatively similar to materialized views. However, as tightly coupled architectures, they lack the generality and independence of a view concept, which can be applied to essentially any system or database.

There are also some well-known frameworks to capture batching and online capabilities. Apache Spark [55] provides batch processing as well as online processing in form of Spark streaming. However, Spark is using a data lineage model at the granularity of data sets (not at record level as our approach). Apache Flink [21] possesses a combined and integrated runtime to perform batch and stream processing. The Flink model also allows for materialization of intermediate (and final) results and provides certain consistency guarantees (exactly once). Again, both frameworks represent, from a KVS perspective, external systems. While they are highly optimized in their environments, data must be loaded from KVS and KVS internals cannot be used. Additionally, both frameworks work over streaming windows, whereas our system works over the full stream.

The work in the field of view maintenance mostly concentrates on incremental maintenance strategies [14, 15, 56]. Most approaches foster a pure incremental processing style, which is only feasible when the history of update operations is complete. The most recent approaches to incremental view maintenance introduce batch optimization for incremental processing [16, 57]. While the results show improved performance for incremental maintenance, this approach is, likewise, not feasible for combined off and online processing. It batches and computes the entire incremental update load into separate working sets of configurable size, processing style is still incremental. In contrast to that our approach is much more comprehensive. It provides a full-fledged analysis of batching, incremental and hybrid strategies in view maintenance.

Chapter 4

View Maintenance System

In this chapter, we propose the *View Maintenance System* (VMS). As opposed to existing point solutions, our design abstracts from a specific KVS architecture and aims to support a broad spectrum of systems. VMS is based only on a few key features that KVSs need to support. This concise set of base features facilitates the integration of view maintenance across different and heterogeneous KVSs. We describe these features in detail in Section 4.1.1.

Furthermore, we focus on maintaining consistency for materialized views, which is a challenge when dealing with SQL operators and base data coming from a non-relational database. VMS provides mechanisms for *basic materialization* (i.e., the computation of views over existing base data) and consistent *incremental maintenance* of views (i.e., the propagation of base data updates to deriving views), enabling querying for real-time analytics applications in KVSs.

VMS consumes streams of client updates: in case of basic materialization, streams consist of *base table records*; in case of incremental maintenance, streams consist of *base table updates*. As a result, VMS produces updates to view data records (see Figure 4.0.1). Views are, therefore, materialized and maintained within KVS; they are kept as standard tables and all properties such as concurrent access, scalability, availability, fault-tolerance offered by the KVS, apply to them as well.

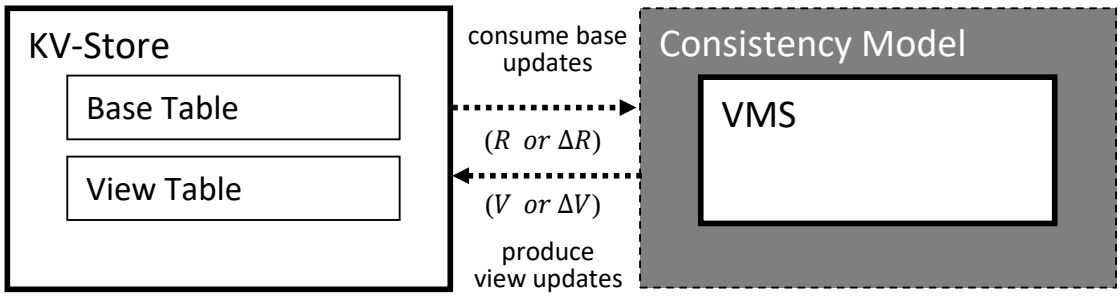


Figure 4.0.1: System overview

To the best of our knowledge, our design, VMS is among the first systems to provide SQL-based materialization and consistent, incremental maintenance of views in KVSs. The closest existing solution, Apache Phoenix, does not consistently and incrementally maintain SQL query results, but rather generates non-materialized results (for a single client) by executing base table scans [11]. We argue that VMS provides both: basic materialization, obtained through full table scans and consistent, incremental updates. In addition, VMS is more scalable, offers faster computation and better read latencies on the views, as shown in our baseline comparison results (see Section 4.5). Thus, in this chapter, we make the following contributions:

1. We provide a detailed consistency analysis using a specific model for view maintenance. We prove how strong consistency can be achieved in a highly parallelizable maintenance system and capture our findings in a theorem.
2. We propose a novel concept called timeline buffering which serves to avoid transitive dependencies in record timelines.
3. We identify the challenging problem of handling multi-row updates for aggregation/join views, and propose a novel split-state mechanism to efficiently maintain atomicity without global locking.
4. We introduce Pre-Processing Views to modularize and speed up the computation of consistent SQL views.
5. We fully validate VMS by extending HBase. We use a TPC-H benchmark (scale factor $100x$, $500x$) to show how VMS materializes and incrementally maintains SQL expressions over views.

4.1 System overview

In this section, we discuss KVS internals that serve us in the remainder of the thesis. We provide a general model which represents existing distributed KVSs such as [2, 3, 4, 5, 6] (see Section 4.1.1). Our objective is to distill a set of features, our VMS requires from a KVS. Furthermore, we present the design of our view maintenance system VMS and describe its components (see Section 4.1.2 and 4.1.3). Additionally, we provide a design rationale.

4.1.1 KV-Store model

The upper part of Figure 4.1.1 shows a general model of a KVS (the lower part of the figure, i.e., VMS, is discussed in Section 4.1.2). Some KVS designs, explicitly designate a master node, e.g., HBase [5] or Bigtable [2], while others operate without explicit master, e.g., Dynamo [3], Cassandra [6], where a leader is elected to perform management tasks, or PNUTS [4], where mastership varies on a per-record basis. In all cases, a KVS *node* (KN) represents the unit of scalability: KNs persist the data stored in the system. The number of KNs can vary to accommodate load change. In contrast to a SQL-based DBMS, a KN manages only part of the overall data (and load).

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

A file system builds the persistence layer of a KN in a KVS. For example, HBase stores files in the Hadoop distributed file system (HDFS). In the file system, all KVS relevant data is persisted and replicated.

A *table* in a KVS 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 on table creation and affect the way the KVS organizes table files.

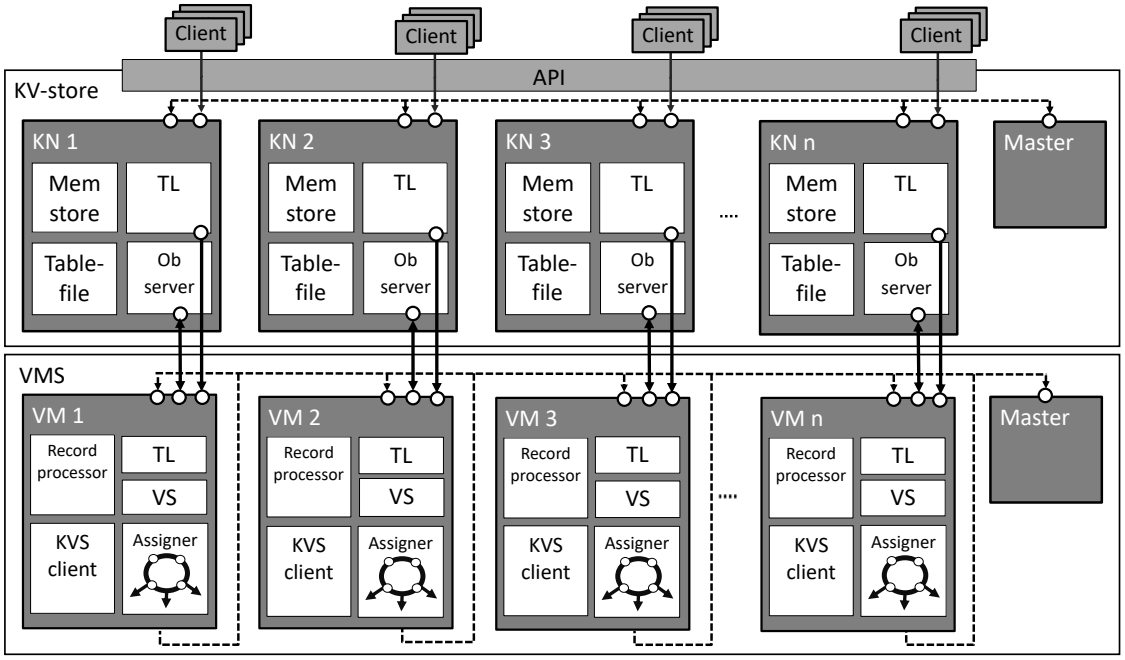


Figure 4.1.1: KV-Store and VMS

Key ranges split a table into multiple *partitions* that can be distributed over multiple KNs. 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*. In general, a KVS can split and move key ranges between KNs to balance system load or to achieve a uniform distribution of data.

Read/write path – In general, the KVS API supports three client-side update operations: *put*, which inserts a record, *get*, which retrieves a record, and *delete*, which removes a record. In the read/write path, when reading or updating a table record, requests pass through as few KNs as possible to reduce access latency. At the end of each request, the client ends up at one particular KN that is serving the key range the client wants to access.

Every KN maintains a *transaction log* (TL), referred to as write-ahead log (WAL) in HBase and commit log in Cassandra. When a client update operation is received, it is first written into the TL of the KN (see Figure 4.1.1). From then on, the update operation is durably persisted. Subsequently, the update operation is inserted into a *memstore*. Memstores are volatile, providing low latency access; they organize records into a tree-like structure.

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.

Modern KVSs (e.g., HBase and Cassandra) provide different types of *observers* that can be registered with KNs and intercept local events (e.g., data manipulation, administrative actions). For example, HBase provides the so-called co-processors. An instance of a co-processor is deployed at each region server and provides a set of predefined methods that can be overwritten in order to execute desired functionality (similar to stored procedures).

KVSs rely on a set of ACID properties. For example, all single-row writes are atomic with isolation (e.g., HBase, Cassandra, Bigtable, PNUTS). This guarantees single-row reads to always return an entire row which is consistent with the write history. KVSs generally do not support transactional writes across rows of a table or even across tables.

4.1.2 VMS architecture

The lower half of Figure 4.1.1 gives an overview of VMS. The input to VMS is a set of streams, each generated by KVS clients and emitted by KNs. A stream either consist of base records (i.e., R) or of base updates (i.e., ΔR).

VMS is comprised of a *master* and *n view managers* (VMs). While the master is responsible for administrative actions only (e.g., load the view definitions), the VMs perform the actual maintenance tasks. Each VM reads one of the streams emitted by the KNs. To (batch) load base tables records, a VM connects to the observer that runs at its assigned KN to locally scan base table partitions. To (incrementally) load base table updates, the VM receives an update stream either via KN observer or via asynchronous read from a TL of the assigned KN.

At the VM, the streams from KN are, then, fed into the *record processor* component, where base updates are transformed into view updates. The transformation is done based on a set of view definitions which are provided in form of SQL statements. Every VM learns about the view definitions at the beginning of the maintenance process.

The *virtual store* (VS) component closely interacts with the record processor. It stores partitions of intermediate view tables in-memory to enable fast local access. In this way, the VS enables fast incremental maintenance and local batchwise computation of base record or update streams. For example, fast local aggregations or fast local determination of join records are done with the help of VS.

Exactly like a KN, every VM is equipped with a transaction log (TL). The TL of a VM can be activated or deactivated, depending on whether fault-tolerance or performance is favored. When activated all updates that a VM receives are written into TL before they are processed. In case a VM crashes, the TL can be used to replay the updates and rebuild the VS.

Simple view definitions (e.g., selection, projection) do not require a redistribution of updates. When a view definition requires multiple rounds of computation (i.e., including redistribution, e.g. for a join), the VM computes its view update and uses the *assigner* component to select and propagate the updates to other VMs.

After executing the last operation of a view definition, a VM computes and materializes the final outcome into a view table in KVS. Thereby, it uses its *KVS client* component. The KVS client maintains a global connection to KVS to either load view definitions or to materialize view records. In contrast to the observer, KVS client can only be used to execute (global) KVS API functions.

A VM is designed to be lightweight and to scale elastically to accommodate changing request loads. It could be allocated to a dedicated node. However, to reduce communication overhead, especially, for complex view definitions, we deploy one VM at each KN, connecting the VM directly to the local observer. In a scenario where many views are maintained in parallel, it is more beneficial to deploy KVS and VMS nodes separately.

Our design provides a multitude of benefits. Seamless scalability: multiple views may have to be updated as a consequence of a single base table update. As VMS exceeds its service levels, additional VMs can be spawned. Operational flexibility: VMs introduce flexibility to the system architecture. All VMs can be hosted together on the same physical node or on different nodes. Exchangeability: every VM can perform any task, a VM can be quickly substituted in case it crashes.

Design Rationale – When a client updates a base table (e.g., put, delete) in the KVS, all derived view tables become stale. Thus, we design VMS to react to all KVS client updates and change the affected view tables accordingly. We determined a number of KVS-common *extension points* to stream incoming client updates of a KVS for processing by VMS (see Figure 4.1.1).

We identify three designs to stream client updates from the KVS to VMS: (1) Using KVS client to access the store's API and retrieve the current record version, (2) asynchronously monitor the TL from the KN, and (3) intercept updates at the KN (via an observer).

Design 1 can lead to inconsistent view states, as base data may change during update processing. For example, a base data record may change again, just before retrieving the record via the KVS-API, occurring just after the record was previously updated. In this case, a base state is missed (the first record update that triggered the retrieval) and won't be reflected in the view state sequence. In addition, this method incurs significant overhead. For example, each update triggers a read and one or more writes to update derived views. Also this design slows down clients of KVS that are accessing the view data concurrently.

In Design 2, reading the 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. Updates in the TL are durably persisted and can be recovered by VMS. However, reading data from TL can be slow (especially when done over network). Also, due to its size, the TL is purged by KVS as soon as records have been written to disk. Performing incremental maintenance requires the KVS to delay purging until data from TL has been propagated for maintenance.

Design 3 is based on observers, one deployed at each KN; observers offer functions for the interception of read and write requests as well as scanning of local key ranges. Thus, Design 3 can be used for both basic materialization and incremental view maintenance. The latter can be either executed by propagating and applying each update separately, or it can be executed delayed by batching update processing. A downside of Design 3 is that updates, retrieved through an observer, are not durably persisted as in Design 2. As a KN crashes, views have to be materialized from scratch.

In VMS, we compare Design 2 and Design 3, where observers forward individual base table updates, offering identical strong consistency guarantees; the actual view computation is also delegated to VMS.

4.1.3 VM processing

In this section, we provide a more detailed explanation of VM processing. In the further course, we explain how VMs manage in-memory acceleration, asynchronous communication and internal data serialization.

Figure 4.1.2 shows the internal processing path and the communication interface of a VM. On the left top of the VM, the bulk loading endpoint is depicted. It is needed when a VM loads complete base tables or delta sets. The bulk loading endpoint retrieves the workloads using a dedicated thread per base table partition. On the top right, the VM provides two communication endpoints (that are both managed by a thread and buffered by a queue): one to receive updates from other VMs and another to receive commands from the master (e.g., to load a query) of VMS. When connected to a live system, the receiver is also used to accept single base table updates that occur in the monitored database.

The record processor is the main thread of the VM. All update operations that a VM receives (from bulk loading or receiving endpoints) flow through this component. Interacting closely with the virtual store (VS), the record processor, in general, handles an update by cycling through the following four steps:

(1) Fetch the old view record (from VS)

(2) Use the update to compute new view record(s)

(3) Store the new view record (into VS)

(4) Pass the update for further processing

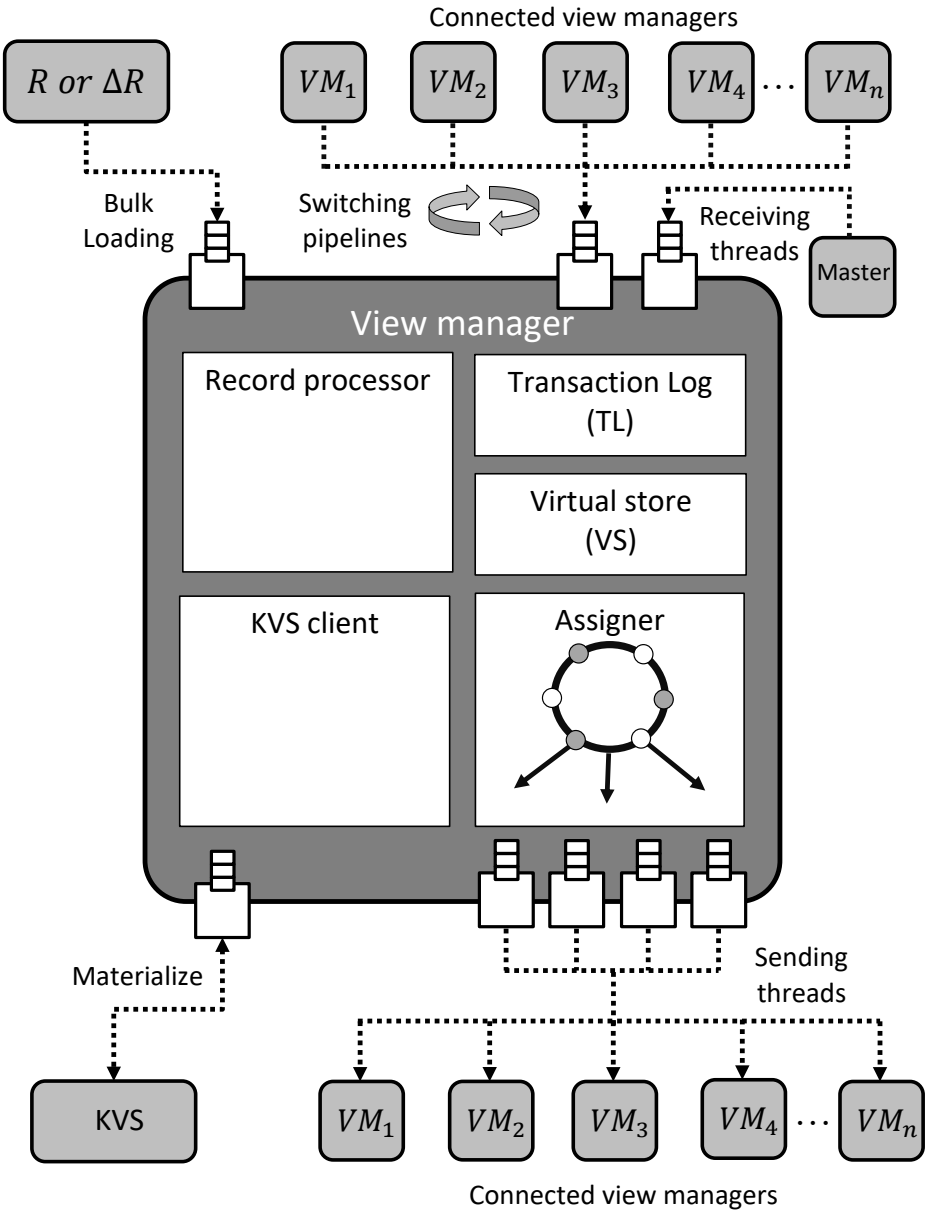


Figure 4.1.2: Internal processing at VM

Receiving an update, the VM fetches the corresponding old view record from VS, deserializes it and applies the update (Step 1 and 2). To store the new view record, the record processor serializes and inserts it into VS (Step 3). Then, depending on the maintenance plan, the record processor passes the updates either to the assigner, for the next operations, or to the KVS client, for final materialization (Step 4). Receiving the next update, the records processor jumps back to Step 1 and repeats the cycle.

The assigner component distributes the ready-to-send updates to the sending threads, which are setup (bottom of Figure 4.1.2) to deliver updates to the remaining VMs of VMS. Record routing in VMS is done via a hashing scheme. The assigner employs a hash ring that organizes all VMs by the hash of their (distinct) name. Every view update that is sent out is hashed by its view row-key and assigned to the next VM in clock-wise direction. This VM then, becomes the destination and the processor of the next operation (in the maintenance plan). The hash ring is synchronized over all VMs such that view row-keys are assigned and sent to the same VMs everywhere.

The KVS client serves two purposes: a) It loads view definitions from the KVS (which have been stored by the master). b) It receives update records from the record processor to be materialized as the final step of a computation (i.e., the last operation in a maintenance plan). Thereby, the client queues and batches update records that are supposed to be stored at the same partition in KVS.

In-memory acceleration – VS is built as collection of hash maps, each representing a local partition of an intermediate view table in a maintenance plan. The row-key of the intermediate view acts as the identifier for the hash maps. The row-key can be a single- or multidimensional construct. For example, when constructing a hash map for join operations, the row-key is 2-dimensional. The values of the hash maps are byte arrays that store the serialized version of a view record.

The API of VS is the same as the API of KVS; that is, it provides the same simple set of methods (i.e., *put*, *delete*, *get*, *scan*) to manipulate and retrieve data. These functions can be called using the table name and specifying the corresponding row-key, exactly as is done for KVS. In this way, the usage is transparent for the VM as there are no differences between storing intermediate results and materializing the final record: a flag decides whether the storage of a view record is quick and volatile or slow and persistent.

Although the usage of VS and KVS is equivalent API-wise, the selection has two implications for the data being stored: (1) data in VS are stored only locally, whereas data in KVS are globally accessible from every node; and (2) data in VS are volatile, whereas data in KVS are fault tolerant.

To address Implication 1, we ensure that view updates are always applied to the node's VS where former updates to the same view key have been processed. All updates are hashed by their next view key to-be-processed. By means of this hash, the updates are assigned and sent to a VM. Thus, all updates with the same view key are sent to (and processed at) the same VM. When a view key changes, responsibility is handed from one VM to the next.

To address Implication 2 and prevent loss of data in volatile VS, the VM optionally writes a transaction log (either to local disc or to a DFS). All arriving updates are then, inserted into the append-only log before being processed. When one VM crashes, a new VM can take control of the transaction log and replay all contained entries to rebuilt the content of the VS. During recovery the system has to interrupt until the new VM has completely built up the VS. Then, processing is resumed.

Establishing fast n:m-communication – VMS distributes incremental updates evenly and allows for high degrees of parallelism, so the distribution performance is dependent mainly on the ability of VMs to communicate.

Thus, establishing fast n:m-communication is a crucial requirement for the system as a large number of updates must be received by and sent to (hundreds of) VMs simultaneously. Especially when executing incremental maintenance, single updates can be sent out to a large number of different VMs in arbitrary order. A setup, for example, in which a new connection is initialized on every request is not feasible in this context. The preferred choice is the creation of $n \times (n-1)$ pipelines; that is, each VM establishes a pipeline to all other $n-1$ VMs. For maximal performance, these pipelines are unidirectional such that there are two pipelines per VM pair.

The next measure that we apply to accelerate communication is parallel update sending. Instead of sending updates sequentially, a VM entertains a defined number of sending threads (each equipped with its own queue, see Figure 4.1.2). The number of sending

threads is independent of the number of VMs and can be varied to achieve the best trade-off.

What has been described as the context for sending messages, also applies for receiving messages. Scaling up the system to hundreds of VMs and views, and at the same time using synchronous communication or providing one reception thread per pipeline leads to massive overhead (as the number of threads is $n \times (n - 1)$). Instead, a VM relies on asynchronous communication, employing a defined number of reception threads (in the figure, a single thread is shown), where each thread handles a list of pipelines together. Thereby, a VM cycles through the list of pipelines and receives updates.

While keeping the received bytes in a buffer, a VM is able to give preference to faster connections and manage slower connections at a later point. This design allows transferring update parts or large update streams; communication is executed via asynchronous sockets (Java NIO API).

Serialization of data – In general, serialization is used at three different points in VMS: (1) when loading existing data or retrieving notifications from a monitored system; (2) when storing/fetching view records from VS; (3) when sending view records to another VM. Because VMS performs these three steps continuously for every update that passes through the system, we identified serialization as one of the actions that are critical to overall performance.

We found that existing serialization schemes are not sufficiently fast or flexible; thus, we propose our own serialization scheme (shown in Figure 4.1.3). In the figure, a *put* update of table R_1 with row-key k is loaded and serialized. Two hash maps, namely, the *table map* and the *column map*, are used to perform rapid serialization; the serialized byte stream of the record is shown in the center of the figure.

The byte stream includes a four-byte identifier that reserves two bytes to encode the base (or view) table and another two bytes to store the status of the operation. Out of the first status byte, two bits are used to encode whether the byte stream is a record, a put or a delete operation. Another four bits are used to encode the number of composite elements in the row-key. The second status byte encodes the number of column-value pairs contained in the stream.

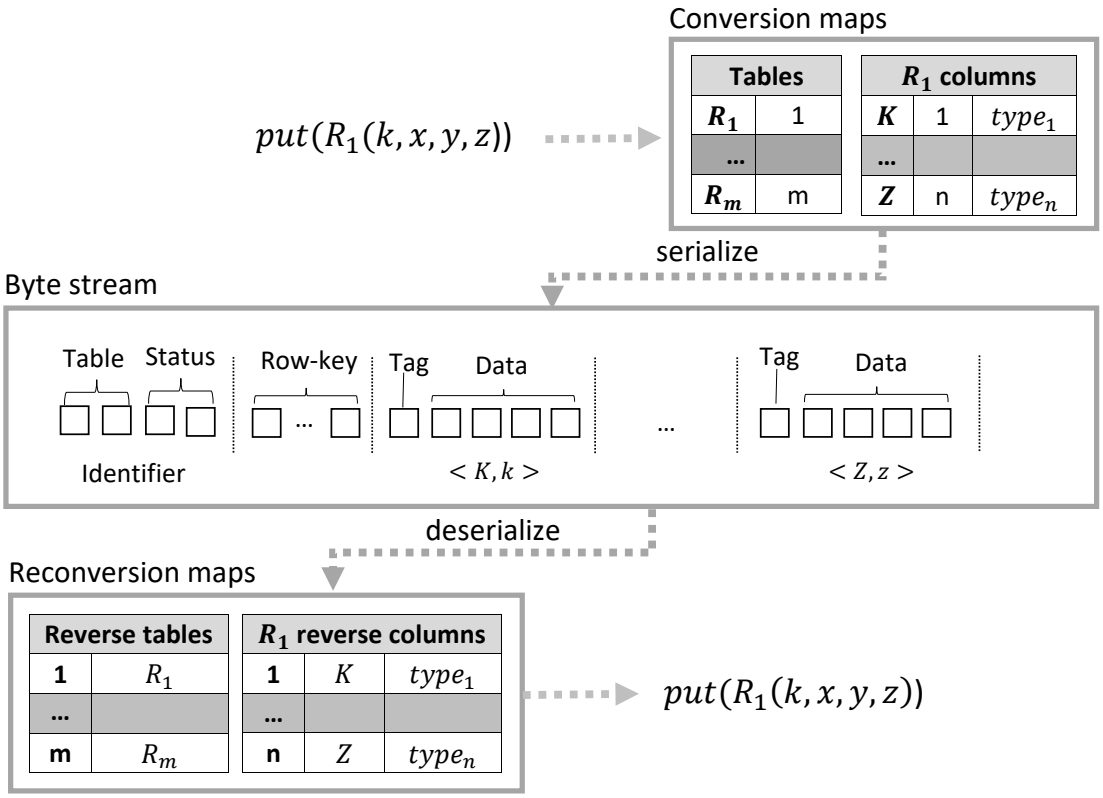


Figure 4.1.3: Serializing VMS records

Consecutively, in the byte stream, the row-key and the map of a record's column-value pairs are appended. The row-key is a composite element composed out of one or multiple attribute values. Thus, instead of storing the row-key itself, we store the indices of the attributes that are given along with the update, so the row-key can be rebuilt from the set of delivered values.

After the row-key, the column-value pairs of the record are encoded via column map that translates the column name into a one-byte column tag. Then, depending on the column's data type, the value follows. Using the tag bytes, we can dynamically transmit different sets of attributes without having to include null values.

4.2 View consistency

In this section, we continue to define the notation used throughout this thesis and give an explanation of view data consistency models, derived from prior work. We then, propose a theorem which identifies three properties required of VMS in order to support strong consistency.

As updates $\Delta\mathcal{R} = \langle u_1, .., u_n \rangle$ (where $\langle\,\rangle$ defines a sequence) are performed on the base tables, their state changes; we depict the state of the base tables as $\mathcal{R}$ and describe a sequence of consecutive states via indices $\mathcal{R}_0.., \mathcal{R}_i, .., \mathcal{R}_f$, where $\mathcal{R}_0$ is the initial state, $\mathcal{R}_i$ is an intermediate state and $\mathcal{R}_f$ is the final state. In a KVS, every put or delete update causes a single record to be modified and affects the state of a base table. A complete update set takes a base table from initial to final state.

$$\mathcal{R}_0 + \Delta\mathcal{R} = \mathcal{R}_f \tag{4.2.1}$$

Two states can be compared by the $\leq$ operator. $\mathcal{R}_i \leq \mathcal{R}_j$ means that the versions of all records in $\mathcal{R}_j$ are equal or newer than the versions of records in $\mathcal{R}_i$ (i.e., $(\forall r \in \mathcal{R}_j) \longrightarrow (r \leq r' \in \mathcal{R}_i)$). If two states can not be compared, which may happen due to the concurrent execution of operations on different row-keys, their relationship is expressed by the $\|$ operator.

We define the incremental *view update* for a view V as follows. Given as input an update u on a record in the base table, a view update reads the current state $\mathcal{V}_i$ of the view table, processes the effect of u according to the semantics of the view, and generates the view state $\mathcal{V}_{i+1}$ for the view table. Note that each view update can therefore consist of several reads and writes or none at all, depending on the operation processed and the current state of the view. For instance, a view update for a projection view does not require any read on the view table since each operation completely determines the value to write. In contrast, a view update for a selection view does not produce any write, if the operation does not satisfy the selection condition.

4.2.1 Consistency model

A *view data consistency model* validates the correctness of a view table. Further, the model evaluates a view table's ability to follow a sequence of base table states and produces a corresponding sequence of valid view table states. The model as well as the different levels of consistency that we establish in the thesis have been widely accepted in view research [30, 31, 33, 58, 59]. Depending on view types, view maintenance strategies, and view update programs, either none, or some, or all of the levels are attainable.

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

Convergence: A view table converges, if after the base tables have gone through states $\mathcal{R}_0, .., \mathcal{R}_f$, and the view table has been updated accordingly, the last view state $\mathcal{V}_f$ is computed correctly. This means it corresponds to the evaluation of the view expression over the final base state $\mathcal{V}_f = View(\mathcal{R}_f)$. 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 every intermediate view state $\mathcal{V}_j$ can be derived from a valid base table state $\mathcal{R}_0 \leq \mathcal{R}_i \leq \mathcal{R}_f$ as $\mathcal{V}_j = View(\mathcal{R}_i)$. Weak consistency ensures that no incorrect intermediate states are provided.

Strong consistency: Weak consistency is achieved and the following condition is true. All pairs of view states $\mathcal{V}_{j1}$ and $\mathcal{V}_{j2}$ that are in a relation $\mathcal{V}_{j1} \leq \mathcal{V}_{j2}$ are derived from base states $\mathcal{R}_{i1}$ and $\mathcal{R}_{i2}$ that are also in a relation $\mathcal{R}_{i1} \leq \mathcal{R}_{i2}$. Strong consistency ensures that successive reads on a view never provide stale data.

Complete consistency: Strong consistency is achieved and every base state $\mathcal{R}_i$ of a valid base state sequence is reflected in a view state $\mathcal{V}_j$. Complete consistency ensures that every change in the base tables is reflected in the view table as well.

Example 4.2.1*: Consider a base table* $R = (\bar{K}, X, Y)$ *and a sum view* $V = \gamma_{X,sum(Y)}(R)$*. This view groups records by their value of* X *and sums the values of* Y *for each group. The initial state of base table is* $\mathcal{R}_0 = \{(k_1, x, 15)\}$ *and the corresponding state of the view table is* $\mathcal{V}_0 = \{(x, 15)\}$*. Now, the following client updates are applied to the base table:*

$$(1)\ \ u_1 = put(R(k_1, x, 15 \longrightarrow 20))$$
$$(2)\ \ u_2 = put(R(k_2, x, 10))$$
$$(3)\ \ u_3 = del(R(k_1, x, 20))$$

KVSs generally provide a consistent per-record ordering, but not for updates across records. Thus, updates (1) *and* (3) *generate a timeline for record with row-key* k_1*, update* (2) *for record with row-key* k_2*. Propagating and maintaining the three updates (in VMS) can be possibly done in six different sequences (i.e., all permutations,* (1)(2)(3), (1)(3)(2)*, etc.) which generates eight different view states.*

Figure 4.2.1 shows all sequences and view states. Thereby, the valid view states (which respect record timelines) are drawn in black and the invalid view states (which do not respect record timelines) are drawn in red.

To achieve convergence, VMS has to compute the final view state as: $\mathcal{V}_f = \{(x, 10)\}$*, which corresponds to the final base table state* $\mathcal{R}_f = \{(k_2, x, 10)\}$*. To achieve weak consistency, any intermediate view state generated must be valid (e.g.,* $\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_4, \mathcal{V}_f$*). Executing base updates in the wrong order can violate weak consistency. For example,* $(3), (1), (2)$ *would generate the correct end result (and allow for convergence) while intermediate view state* $\mathcal{V}_3 = \{(x, -5)\}$ *would be invalid. Likewise, executing updates not atomically can lead to invalid intermediate states.*

To achieve strong consistency view states must be correctly ordered (e.g., $\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_4, \mathcal{V}_f$*). Redistributing the results of* V *for downstream maintenance operations (and not respecting the record timeline of* V*) can violate strong consistency. For example, propagating* $\mathcal{V}_1$ *and* $\mathcal{V}_4$ *to different VMs could lead to global order* $\mathcal{V}_0, \mathcal{V}_4, \mathcal{V}_1, \mathcal{V}_f$*. To achieve complete consistency, all base states must be reflected (e.g.,* $\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_4, \mathcal{V}_f$*). Leaving out intermediate states as it is done for batching of updates (e.g.,* $\mathcal{V}_0, \mathcal{V}_4, \mathcal{V}_f$*) only allows for strong consistency.*

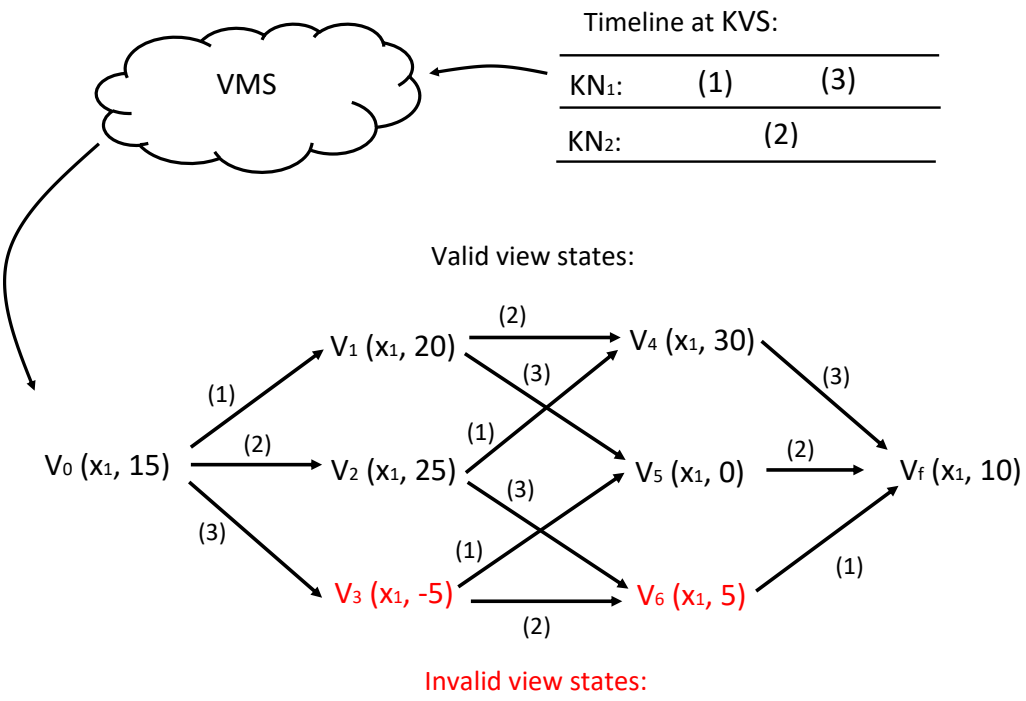


Figure 4.2.1: Possible view states of a given update sequence

4.2.2 Theorem for strong consistency

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

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

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

We now provide a brief explanation of Theorem 1. If we employ Property 1 of the theorem, we ensure that all updates are delivered and applied exactly once. However, Property 1 alone does not guarantee convergence of the view. When using parallel execution, partial updates or multiple updates (to the same view record) might be applied and affect the view correctness.

Property 2 avoids wrong execution in case of partial updates and concurrent view access. However, if Property 1 and Property 2 of the theorem are ensured, weak consistency is still not guaranteed as asynchronous processing and redistribution could lead to reordering of updates.

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

4.3 View maintenance concept

In this section, we first describe in more detail the maintenance concept of VMS, particularly focusing on computation and propagation of updates given a view definition. Then, we show how our design achieves a high degree of concurrency while providing strong consistency via the three properties of Theorem 1 found in Section 4.2. Finally, we discuss how VMS optimizes view maintenance by using different methods of batching.

4.3.1 Distributed model

View maintenance as required by the view definition is brought out in multiple steps. Given a general view definition $V = View(R_{set})$ in which function $View$ is specified by relational algebra (see Chapter 2.4). The function can also be described using a directed acyclic graph (DAG) which in the following is referred to as a *maintenace plan*. The maintenance plan is used by VMS to plan and optimize the maintenance process and by each VM to determine the next *maintenance operation* (e.g., selection, aggregation, join). The maintenance plan is formalized as $M = (R_{set}, O, V, E)$ with base tables R_{set} as the start vertices; maintenance operations O as intermediate vertices; view table V as end vertex; and the connecting edges E.

A *maintenance path* describes the sequence of maintenance operations that start at base table R and end at the final materialized view V in a maintenance plan (see Figure 4.3.1). To execute incremental view maintenance, VMS takes updates of R, processes them along the maintenance path, and finally materializes the updates into V. For each operation in the path, VMS materializes an intermediate view. Intermediate views keep intermediate results ready during the maintenance process. They are stored in-memory (i.e., in VS) to speed up processing of forthcoming incremental updates. The maintenance path is formalized as $M_{R \to V} = \langle o_1, .., o_n \rangle$ (with $o \in O$), for each operation, there is an intermediate view defined as $\{(o_1, I_1), .., (o_n, I_n)\}$.

To minimize communication, VMS processes operations that work over the same partitioning (i.e., that require the same distribution of updates) together at the same VMs. For

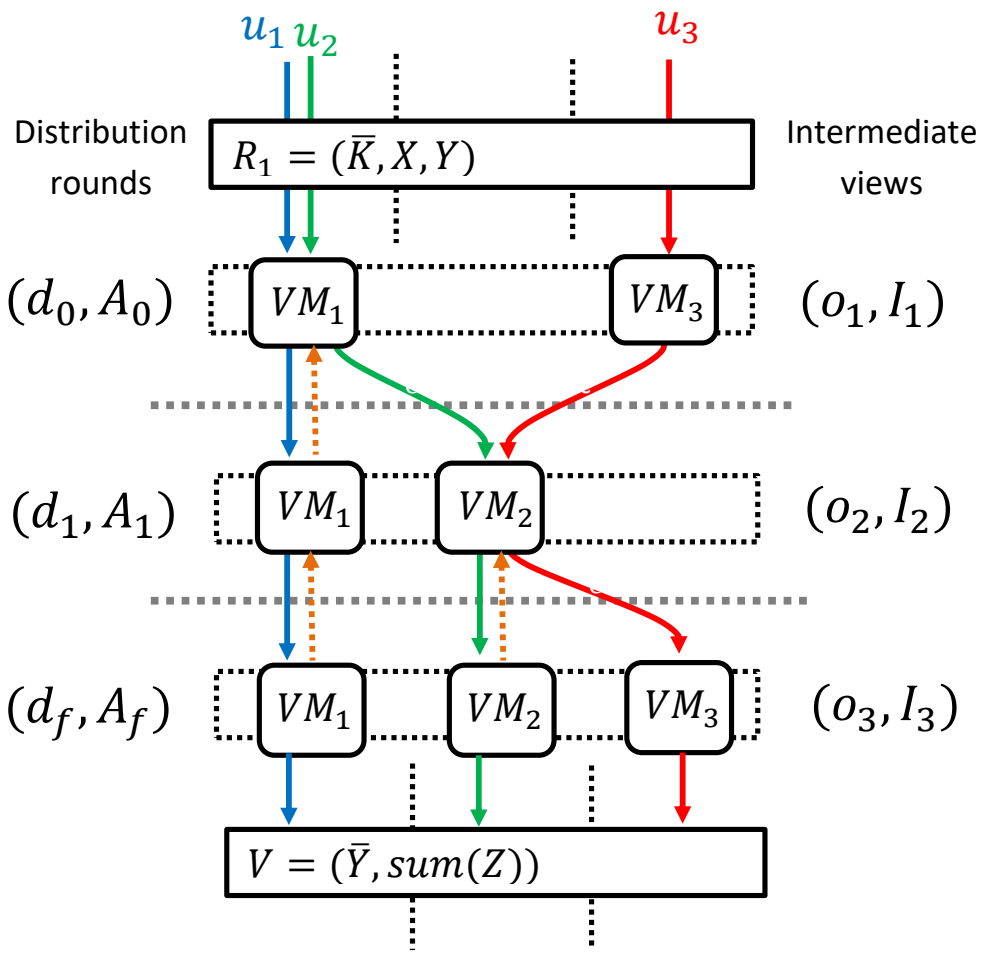


Figure 4.3.1: Processing a maintenance path

that reason, it combines multiple operations into a *distribution round* if possible. While distribution rounds aggregate sequences of maintenance operations, update processing in VMS is not necessarily done in a round-based fashion. VMS supports incremental maintenance of many (single) updates, as well as batch-wise computation of update sets. The maintenance path is then, represented as a sequence of distribution rounds $D_{R \to V} = \langle d_0, d_1, .., d_f \rangle$ (with $d \subseteq M_{R \to V}$). Distribution rounds are formalized along with their *distribution keys* as $\{(d_0, A_0), (d_1, A_1), .., (d_f, A_f)\}$; thereby, the distribution key A_i is the attribute according to which updates are distributed, and also the attribute according to which all intermediate views are partitioned (not necessarily indexed) in the corresponding distribution round.

Example 4.3.1*: Consider base tables $R_1 = (\bar{K}, X, Y)$ and $R_2 = (\bar{L}, X, Z)$. Now, let a view be defined as $V = \gamma_{Y,sum(Z)}(\sigma_P(R_1) \bowtie R_2)$. The maintenance path $M_{R_1 \to V}$, as shown in Figure 4.3.1, is executed in three distribution rounds d_0,d_1 and d_f. Updates are distributed according to base table row-key $A_0 = K$ in the beginning, then get selected, reordered and joined according to (intermediate) distribution key $A_1 = X$ and then aggregated by distribution key $A_f = Y$. Three updates u_1,u_2 and u_3 originate at R_1 and are passed from one VM to the next. Three operations $o_1 = \sigma$, $o_2 = \bowtie$ and $o_3 = \gamma$ are used to compute intermediate views $I_1 = \sigma_P(R_1)$, $I_2 = I_1 \bowtie R_2$ and $I_3 = \gamma_{Y,sum(Z)}(I_2)$. After passing the distribution rounds, the updates are finally applied to V.*

Updates in a distribution round are processed by multiple VMs in parallel (see Figure 4.3.1). Despite of the parallelism of execution, in each distribution round, there is a global processing order. We define the processing order over all VMs in a distribution round d_i as *execution sequence* of updates as:

$$\begin{aligned} \Delta\mathcal{R} &= \langle u_1, .., u_n \rangle \\ d_i(\Delta\mathcal{R}) &\longrightarrow \mathcal{D}_i \end{aligned} \tag{4.3.1}$$

The execution sequence in round d_i is denoted as $\mathcal{D}_i$ and represents a recombination of the original update sequence $d_i(\Delta\mathcal{R})$ on the base table. Thereby, updates get reordered, dropped or they can be duplicated. However, in the following, we are particularly interested in reordering as it threatens the view consistency. The execution sequence $\mathcal{D}_f$,

in the final distribution round, denotes the materialization order into the view table.

$$u_1 <_{D_i} u_2 \quad u_1 >_{D_i} u_2 \quad u_1 ||_{D_i} u_2 \tag{4.3.2}$$

Two updates in an execution sequence $u_1, u_2 \in \mathcal{D}_i$ can either one follow the other or they can be executed simultaneously. We are using notation $>_{D_i}$, $<_{D_i}$ here to indicate that two updates are following each other in execution sequence of $\mathcal{D}_i$ or notation $||_{D_i}$ to indicate that two updates are processed simultaneously in $\mathcal{D}_i$. In subsequent execution sequences (i.e.,$\mathcal{D}_{i+1}$, $\mathcal{D}_{i+2}$), where updates are processed by different VMs, this relation may change.

From one distribution round to the next, a VM redistributes its updates to the set of remaining VMs which manage the updates of the next round. Redistribution of updates is performed using consistent hashing [60]. The assigner (see Section 4.1.2) hashes one of the update's attributes and associates the hash value in clockwise direction to one of the VMs. The VMs are arranged on a hash-ring using the hash values of their VM-IDs. In this manner, a VM assigns updates uniformly across the remaining VMs. The *assign* function (executed by the assigner) can be formalized as follows:

$$asg(u.A) \longrightarrow \text{VM} \tag{4.3.3}$$

An update u is assigned using one of its attributes $u.A$ as parameter; the outcome of the function is one of the remaining VMs. In distribution round (d_i, A_i), a VM evaluates the assign function on next round's distribution key (i.e., $asg(u.A_{i+1}) \longrightarrow \text{VM}$) to determine the receiving VM. The hash-ring is synchronized across all VMs via a distributed lock service. Thus, the same VM is picked to handle the same update everywhere in the system. This is important when regions (of base tables) move from one KN to another.

In the remainder of the section, VMS is designed to provide strong consistency. We argue that convergence on its own is insufficient due to the online nature of our system. Since VMS is designed to incrementally maintain views, the targeted applications must be able to read correct intermediate states. Further, weak consistency is inadequate since a client can perform successive reads on the same view. If the view states are not correctly ordered, the client may enter an inconsistent state (e.g., having to roll-back on a previous state). On the other hand, complete consistency is too costly and can only be detected if the clients repeatedly read from the base tables.

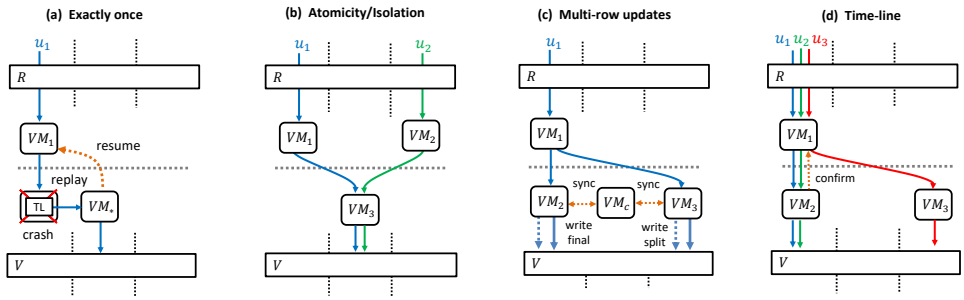


Figure 4.3.2: Achieving consistency implementing Theorem 1

Now, we discuss how the defined properties of Theorem 1 can be achieved in VMS. Figure 4.3.2 shows examples for each property of the theorem: Property 1: (a) Exactly once, Property 2: (b) Atomicity/Isolation, (c) Multi-row updates and Property 3: (d) record timeline. In the following we discuss adherence to each property separately and refer to the examples.

4.3.2 Property 1: exactly once

Property 1 describes the requirement that view updates are applied exactly once to the view table. This is a critical requirement, as views can be non-idempotent. There are two possible scenarios that violate Property 1: (1) an update is received and applied more than once, (2) an update is lost due to a crash (see Figure 4.3.2 (a)). In either case, the view is incorrect (i.e., does not converge). We now describe both scenarios separately.

Scenario 1 – When sending streams of updates through the distribution rounds and connecting different VMs, the system establishes reliability between them. Each VM keeps track of the updates it received and the updates it has sent to other VMs.

$$L = (\text{VMs}, \mathbb{N}) \tag{4.3.4}$$

In the moment an update is processed, the VM tags it with a *local update ID* (see Equation 4.3.4) which is set via attribute $u.L$ of the update. A local update ID is a combination of the VM-ID and a local sequence number. When set in round d_i the local update ID can be used in round d_{i+1} to identify whether an update has been already processed.

To track the maintenance state, each VM uses two map variables L_{rec} and L_{sen} and captures the last local update IDs processed with regard to all other VMs. Variable $L_{rec}[i]$ captures the last local update ID, received from VM_i; $L_{sen}[i]$ stores the last local update ID which has been sent to VM_i. If a VM_i resends an update u a second time, using the same sequence number as before (or a lower one), the receiving VM checks against condition ($u.L \leq L_{rec}[i]$) ; and drops the update if the condition evaluates to true.

Scenario 2 – Under crash failure, when updates are lost, we distinguish between Design 2 and Design 3 (see Section 4.1.2). When realizing Design 2, lost updates can be recovered from the TL of the KN. Realizing Design 3, under the assumption that purging of TL is not managed explicitly by VMS, we cannot replay updates that have been already streamed from a KN. In this case, the only viable solution is deletion of views and a full scan of all base table partitions to recompute the given view definitions.

In Figure 4.3.2 (a) the crash of a VM is depicted. In this case, the system starts a new instance using the same VM-ID as the crashed VM (* in the example). As soon as the new VM instance comes up, it executes a recovery procedure as shown in Algorithm 4.1. For the new VM the VS, and the local variables are initialized blankly. To restore the state of its L_{sen} variables, the VM sends a request message (m_{rec}) to each VM_i (Algorithm 4.1, Line 1-2). It requests the last update ID that VM_i received from the crashed VM (i.e., $L_{rec}[*]$) before the crash; by means of the result, the VM restores its L_{sen} variable.

Algorithm 4.1: Recovery at a new VM

```
1  for (VM_i ∈ {VM_1, ..VM_n}) do
2      L_sen[i] ← VM_i.send(m_sen);                    // restore L_sen
3  for u ∈ TL do                                       // replay updates
4      u.L = (VM_j, n);                                // read local ID
5      L_rec[j] ← u.L;                                 // restore L_rec
6      process(VS, u);                                 // reprocess update
7      VM_a ← asg(u.A);                                // reassign to VM_a
8      if (u.L > L_sen[j]) then
9          VM_a.send(u);                               // resend update
10 for (VM_i ∈ {VM_1, ..VM_n}) do
11     VM_i.send(m_res = (L_rec[i]));                  // resume sending
```

Then, the new VM takes ownership of the abandoned TL and replays all updates, building up its VS from the ground (Algorithm 4.1, Line 3-9). During the replay, VM_* reprocesses and reassigns the updates and tests whether the results should be resent.

After completing the replay, the new VM sends a resume-sending message (m_{res} = ($L_{rec}[i]$)) to each VM (Algorithm 4.1, Line 10-11) such that all VMs know they can resume sending. However, resuming the sending process is not trivial. The crashed VM could have also had updates waiting in its queue; which have not been written into its TL before. For that reason, the new VM puts the last local ID of the update that has been received (and stored into TL) by VM_i into the message m_{res}. Then, each VM_i compares the variable against its own sending variable (i.e., $L_{rec}[i] < L_{sen}[*]$). If the value is equal, VM_i resumes sending, if it is smaller, VM_i resends the missing updates, starting with the next update after $L_{rec}[i]$.

4.3.3 Property 2: atomicity and isolation

According to Property 2, every view update has to be executed atomically and in isolation. First of all, we rely on the semantics that are provided by the KVS to achieve this. We assume (as it is for example the case for HBase and Cassandra) that at record level (i.e., for a single put operation), the execution follows standard ACID guarantees. Thus, we conclude that a single VM can update a view record atomically and in isolation.

However, during maintenance, updates are distributed to many VMs and processed in multiple stages. When two base updates (referring to the same view record) are sent to different VMs, both VMs can update the view record simultaneously (see Figure 4.3.2 (b)). This possibly leads to a violation of Property 2.

$$\begin{aligned}(\exists u_1, u_2 \in \mathcal{D}_i)(u_1.A_{i+1} = u_2.A_{i+1}) \\ \longrightarrow asg(u_1.A_i) \neq asg(u_2.A_i)\end{aligned} \tag{4.3.5}$$

If there are two updates in round (d_i, A_i) which are executed in sequence $\mathcal{D}_i$ and which are targeting the same distribution key in $\mathcal{D}_{i+1}$ (i.e, $u.A_{i+1}$) and which have been assigned to two different VMs, then the VMs can possibly update the same view records simultaneously. VMS prevents the case shown in Equation 4.3.5 by design. Distributing updates according to the hash of the next distribution key completely eliminates the need for further synchronization mechanisms. Given a set of updates processed in distribution round d_i that are consecutively redistributed by distribution key A_{i+1}. Then, the unique-

ness of the assign function ensures that updates, targeting the same distribution keys $u.A_{i+1} = a$, are always forwarded to the same VM, no matter from where they originated. The according equation can be formulated as follows:

$$(\forall u \in \mathcal{D}_i)(u.A_{i+1} = a) \longrightarrow asg(a) = \mathsf{VM} \tag{4.3.6}$$

The VM itself operates sequentially and prevents updates to the same view record from interfering with each other. Thus, we conclude, the execution is isolated. It is also atomic, as the put of a view record is executed according to ACID, and the get does not modify data. However, our definitions are only based on simple view updates (that modify single view records). In complex maintenance processes there might be also *multi-row updates* in a view table. These types of updates are particularly challenging with regard to Property 2 as we explain in the following.

Multi-row updates – During maintenance of aggregation or join operations, a base table update of a single base record can touch many view records (e.g., by changing the aggregation key or by joining multiple records). For the view table, all updated records must be altered at the same time (see Figure 4.3.2 (c)).

$$\begin{aligned}(\forall u = (u_{.1}, .., u_{.n}) \in \mathcal{D}_i)\\ \longrightarrow (\forall \mathcal{D}_k)(k > i)(u_{.1}||_{D_k}..||_{D_k} u_{.n})\end{aligned} \tag{4.3.7}$$

To execute maintenance, VMS can simply split the multi-row update into a set of *partial updates* (i.e., $u = (u_{.1}, .., u_{.n})$) and let them be processed by VMs, just like normal updates. However, to not violate Property 2, the effects of all partial updates must become visible in a view table simultaneously. Given that a maintenance operation produces a multi-row update in round d_i. Then, the partial updates have to be executed in parallel for all rounds to come ($k > i$). However, asserting that intermediate views are not exposed to the clients, we can relax the constraint as follows.

$$\begin{aligned}(\forall u = (u_{.1}, .., u_{.n}) \in \mathcal{D}_i)\\ \longrightarrow (u_{.1}||_{D_f}..||_{D_f} u_{.n})\end{aligned} \tag{4.3.8}$$

Still this constraint is difficult to achieve since we assume that partial updates can be assigned to different VMs in the next distribution round. Also, using a KVS to store view

tables means that view keys of the partial updates can be found in regions of different KNs. Thus, updating them atomically (see Property 2), requires a cross-table transaction. For efficiency reasons, KVSs do not provide this kind of functionality (e.g., HBase offers within-region transactions, only.) VMS demands additional mechanisms to maintain atomicity when handling these kinds of multi-row updates.

$$G = (\text{KNs}, \mathbb{N}) \tag{4.3.9}$$

Global locking is expensive and error-prone to use, thus, is best avoided in large-scale distributed systems. We designed VMS in an asynchronous and lock-free manner, which is also reflected in our solution for handling timelines and multi-row updates. Let all updates be tagged by a *global update ID G* (which can be accessed via attribute $u.G$). A global update ID is provided by the KN an update originates from. In contrast to the local ID, a global ID identifies an update throughout the whole maintenance process. All partial updates belonging to the same multi-row update can be identified by the same global ID as the following equations shows:

$$\begin{aligned}(\forall u \text{ with } u = (u_{.1}, .., u_{.n}) \in \mathcal{D}_i) \\ \longrightarrow u_{.1}.G = .. = u_{.n}.G = g \in G\end{aligned} \tag{4.3.10}$$

In order to support correct execution for multi-row updates, we propose a split-state mechanism where VMS can write additional meta-data with the view update to safely change the state of a view record in KVS. In general, the multi-row update is processed in three steps (see Figure 4.3.3): (1) Synchronization, (2) writing the split-state and (3) resolving the split-state. We now explain each of the steps separately.

Synchronization – When receiving a partial update, a VM buffers it into a hash map (using $g = u.G$ as key), where it is kept along with the global ID until the complete multi-row update is ready to be executed. Notably, the update is not applied such that the targeted view record remains open for modifications through regular updates.

In a next step, each VM evaluates the assign function, using g as parameter (i.e., $\text{VM}_c \leftarrow asg(g)$). As all partial updates contain the same global ID, each VM resolves the same view manager VM_c which will be the *coordinator* for the multi-row transaction. By using our hashing-scheme to determine the coordinator, we achieve two goals, simultaneously:

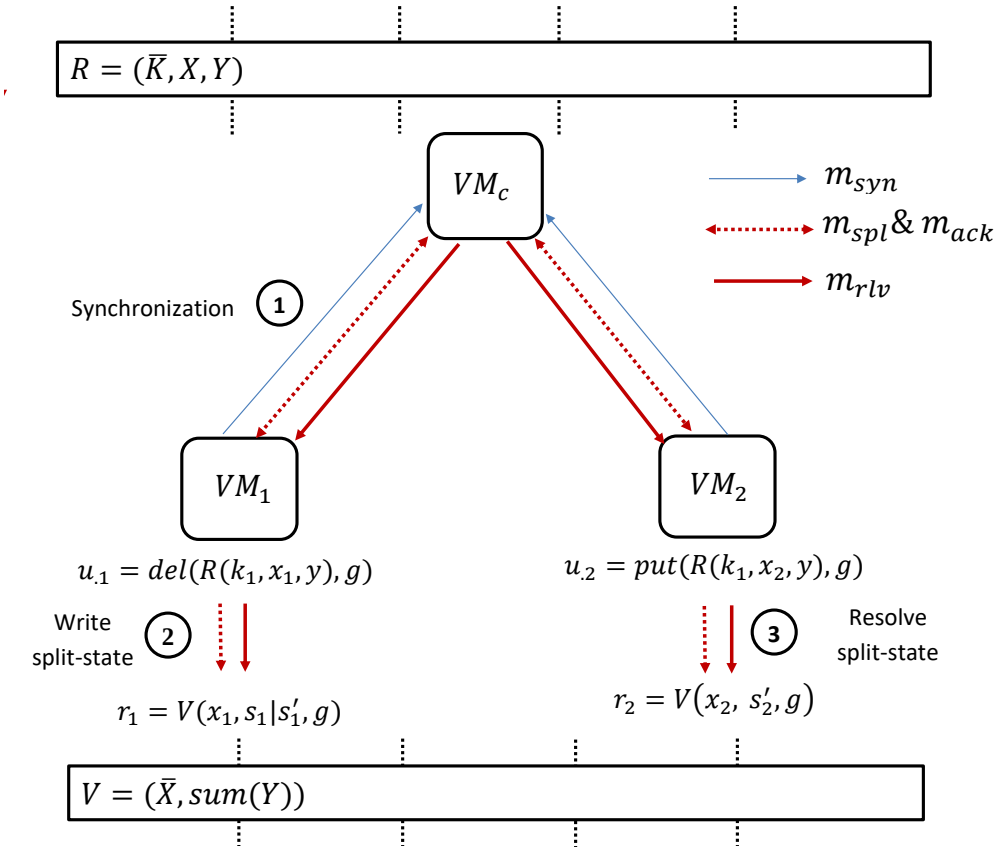


Figure 4.3.3: Execution of a multi-row update

first, we allow the coordinator to be determined rapidly at each VM, (simply using the global ID); second, we guarantee a uniform distribution of coordination tasks. As such, coordination is decentralized and no single VM is overloaded with the task.

Finally, each VM sends a *synchronization request* message, formalized as $m_{syn} = (g)$ to VM_c. The synchronization request is a notification for the coordinator that the VM is ready to process the multi-row transaction.

Writing the split-state – The coordinator waits until all partial updates of the multi-row update have been requested. Then, it sends a write split-state message $m_{spl} = (g)$ containing similar information as the sync message to each VM, advising them to write out their split-states. In doing so, the VMs include the global update ID in the meta data of the written record. This is required in order to identify different multi-row updates, potentially occurring simultaneously. Finally, the VMs acknowledge the writing of the split-state with m_{ack} to the coordinator.

Resolving the split-state – When the coordinator has received all acknowledgments, it starts the third phase and informs all VMs to resolve their split-states using the resolve command m_{rlv}. When writing out the final state (of the transaction) the VMs, again, include the g as identifier.

Example 4.3.2*: Figure 4.3.3 provides an overview of how a multi-row update u consisting of two partial updates is carried out. Consider a base table* $R = (\bar{K}, X, Y)$ *and a sum view* $V = \gamma_{X,sum(Y)}(R)$*. Further consider an update* $u = put(R(k_1, x_1 \longrightarrow x_2, y))$ *on R which is split into two partial updates* $u_{.1} = del(R(k_1, x_1, y), g)$ *and* $u_{.2} = put(R(k_1, x_2, y), g)$ *both being identified by global ID g. First partial update is sent to* VM_1*, second is sent to* VM_2*. Both VMs select the same coordinator* VM_c *and send* m_{syn}*. Then* VM_c *sends* m_{spl} *and advises both VMs to write out their split-states (i.e,* $V(x_1, s_1|s_1', g)$ *and* $V(x_2, s_2|s_2', g)$*). After receiving the acknowledgments,* VM_c *sends* m_{rlv} *and both VMs resolve their split-states to* $V(x_1, s_1', g)$ *and* $V(x_2, s_2', g)$*.*

Client reads – When a client reads view records and finds them in a normal state, it simply fetches the records. In case, the clients finds view records that are tagged with a update ID, it groups the view records according to their update ID. Existing split-states are then, handled on client-side as the following formalization depicts:

$$V_g = (\forall r \in V)(r.G = g) \tag{4.3.11}$$

$$\begin{aligned}(\forall r_i \in V_g)(r_i.A = (a_i|a_i')) \\ \longrightarrow r_i.A = \begin{cases} a_i \ if (\forall r_j \in V_g | r_j.A = (a_j|a_j')) \\ a_i' \ if (\exists r_j \in V_g | r_j.A = a_j) \end{cases}\end{aligned} \tag{4.3.12}$$

We define V_g as all records of the view that possess the same global update ID g. If there are split-states $(a_i|a_i')$ among the attributes of a view record in a group V_g, the client checks the entire group V_g. If there are only split-states, the client takes the old value a_i. There might be more split-states not written yet, such that the new value, together with the non-existing split-states would produce an inconsistent view state. However, if there is a single resolved state among the view records, the client decides to take the new value a_i'. The resolved state indicates that all split-states must have been written and that the new values can be read safely within V_g.

4.3.4 Property 3: record timeline

Property 3 means that sequences of updates on the same base record are not reordered when processed by VMS. Thereby, we assume the correct order of updates, as determined by each KN, is provided as input to the attached VM. While reordering of updates within VMS may be allowed (e.g., due to parallel execution), reordering with regard to the timeline of a base record (or a view record) has to be respected. The record timeline constraint can be formulated as follows:

$$\begin{aligned}(\forall u_1, u_2 \in \mathcal{D}_i)(u_1 <_{D_i} u_2)(u_1.A_i = u_2.A_i) \\ \longrightarrow (\forall \mathcal{D}_k(k > i))(u_1 <_{D_k} u_2)\end{aligned} \tag{4.3.13}$$

When two updates, applied in round (d_i, A_i), are using the same distribution key (i.e., $u_1.A_i = u_2.A_i$) follow each other in a timeline (i.e., $<_{D_i}$,), their order has to be respected during all following sequences $((\forall \boldsymbol{D}_k)(k > i))$.

Since processing at VMs is sequential, a reordering of a timeline can only occur during redistribution. Redistributing the updates according to the hash of the next distribution key A_{i+1} creates a new record timeline which has the following implications: As long as A_{i+1} does not change the assignment, updates are always forwarded to the same VM. If there are two updates, modifying the same record and yielding different assignment results in the next distribution round (e.g., $(u_1.A_i = u_2.A_i)(asg(u_1.A_{i+1}) \neq asg(u_2.A_{i+1}))$ both updates are sent to different VMs. This implies that there is a possible timeline violation that may break consistency as incorrect intermediate view states could occur.

Also there is the problem of *transitive dependencies*. When complex queries (e.g., multiple consecutive joins or aggregations) are processed, the maintenance plan involves multiple stages of redistribution (see Figure 4.3.1) during which updates are reassigned and partitioned based on different view records. Each of the view keys creates its own new timeline. While each timeline could be synchronized separately to solve the problem, dependencies of prior timelines get inherited to the current one which we call a transitive dependency. These transitive dependencies complicate the task of providing consistency severely because the number of dependencies in the system increases exponentially.

Example 4.3.3*: Continuing Example 4.3.1 (see Figure 4.3.1): Given that the base table is in a state* $R_1 = \{(k_1, x_1, y_1)\}$*. Now the following updates occur: A delete* $(u_1 = del(R_1(k_1, x_1, y_1)))$ *followed by a put of* $u_2 = put(R_1(k_1, x_2, y_2))$*, then* $u_3 = put(R_1(k_2, x_2, y_3))$*. After the first round, updates* u_1 *and* u_2 *are redistributed according to key X;* u_1 *is passed to* VM_1 *(hash of* x_1*) whereas* u_2 *is passed to* VM_2 *(hash of* x_2*). In a unfavorable constellation,* u_2 *could be applied before* u_1*, which brakes consistency as* u_1 *(arriving later) deletes the record belonging to* k_1*. Then, the record is reflected in the base table but not in the view table. Further, a dependency between* u_3 *and* u_2 *which is introduced by the timeline of X can be observed. Given that* u_2 *is processed at* VM_2 *before* u_3*. Because* u_2 *is dependent on* u_1*,* u_3 *is not only dependent on* u_2 *but also* u_1 *which is a transitive dependency.*

To solve the problem and synchronize modifications, we introduce the concept of *timeline buffering* to achieve strong consistency. The idea behind timeline buffering is that always

only *one* version of a specific (base or view) record is allowed to travel the system at a time which is called the *active update*. This rule is enforced over all distribution rounds, such that transitive dependencies cannot arise.

Let $\mathcal{D}_i(t)$ be the sequence of updates in a distribution round that have been processed up until point t. Given a sequence of updates in distribution round (d_i, A_i) and a distribution key $a \in A_i$. Out of the updates that are not materialized yet ($u \notin \mathcal{D}_f(t)$), there is always only n active updates with the same key a for all subsequent distribution rounds $\mathcal{D}_k(t)$. What we demand can be expressed as follows:

$$\begin{aligned} &(\forall a \in A_i) \\ &(S = \{u | (\forall \mathcal{D}_k(t)(k > i))(u.A_i = a)(u \notin \mathcal{D}_f(t))\}) \\ &\longrightarrow |S| = n \end{aligned} \tag{4.3.14}$$

In our approach, we restrict the system to $n = 1$ updates. However, we can loosen the constraint to allow $n \geq 1$ versions, simultaneously. But then, additional synchronization measures have to be taken in order to sustain consistency. To implement timeline buffering, each VM is equipped with its own *timeline buffer* (TB). Before each update is processed at a VM it is tested against the TB. The TB remembers the row-keys of all active updates in the system right now. When, during the time an update is actively processed, subsequent updates on the same row-key arrive, they are held and queued into TB. The next subsequent versions are only released when the materialization of the last active update has been confirmed. To make timeline buffering even more efficient, subsequent updates that are queued can be merged together to reflect the respective latest version of the timeline.

Algorithm 4.2: Checking updates against a TB

```
1 for u ∈ VM do
2     a ← u.A_i;
3     if (H_act[a] ≠ ∅) then                    // check for active updates
4         H_act[a] ← u;                         // set active update
5         return u;
6     else
7         H_buf[a] ← u;                         // buffer update
8         return ∅;
```

The TB employs two hash maps: H_{act} which contains the row-key (of the current distribution round) and the global update ID of the active updates and H_{buf} which stores the

buffered updates (see Algorithm 4.2). The row-keys and sequence number of incoming updates are, first, put into H_{act} (Algorithm 4.2, Line 4). If, in the meantime more updates over the same row-key arrive, the are buffered into H_{buf} (Algorithm 4.2, Line 7). If the buffer is already full the buffered update is overwritten by the newer one. In this way, the system is protected against skewed workload distributions, where many updates of the same key are maintained.

If, now and then, the update IDs of the materialized updates arrive, the TB is cleaned up (see Algorithm 4.3). The VM first removes the active update (Algorithm 4.3, Line 3). Then, if there is a buffered update, the VM sets it active and feeds it into the processing cycle (Algorithm 4.3, Line 5-8). During the process of cleaning TB, view maintenance has to be halted to prevent inconsistencies. However, cleaning up is a local process, executed in defined intervals. As such impact on overall latency is low.

Algorithm 4.3: Clean up a TB

```
1 for a ∈ H_act do
2     if (u ∈ D_f(t)) then                    // check if materialized
3         H_act[a] ← ∅                        // remove active update
4         if H_buf[a] ≠ ∅ then
5             u ← H_buf[a];
6             H_act[a] ← u;                   // activate buffered update
7             process(VS, u)                  // process buffered update
8             H_buf[a] ← ∅                    // reset buffer
```

Revisiting Example 2: when a TB is employed at VM_1, the view manager realizes that u_1 and u_2 belong to the same timeline (modifying k_1). Thus, VM_1 will not send out u_2, before the materialization of u_1 has been confirmed. Likewise, VM_2 will consult its TB and hold back u_3 until the materialization of u_2 has been confirmed. The transitive dependency between u_1 and u_3 is not existing (as there is no active dependency between u_1 and u_2).

While still allowing parallelism of maintenance, timeline buffering merely restricts the number of updates processed over a single record. This reduces the amount of updates sent through VMS dramatically. Likewise, it prevents the creation of transitive dependencies, and thus, allows VMS to achieve strong consistency at manageable cost. Moreover, with timeline buffering, it is certain that always one of the records versions is reflected in the view. The downsides are comparably small: synchronization of intermediate record states is slower or in some cases may be skipped due to the merging process in TB.

4.3.5 Batching

From a practical perspective, when computing millions of updates, the system may accumulate a large number of record version as well as multi-row updates and may be overloaded with synchronization tasks to achieve consistency. As we are only materializing (and exposing) the last view computation to clients, the system relies heavily on batching to locally and globally aggregate sets of updates to improve performance. Knowing that during a batch a fixed number of updates is loaded and processed, we can perform the following optimizations.

Record timeline – What a timeline buffer does while buffering updates, can be done to the complete update set before processing the updates. When the VM receives a batch of update operations from a KN, it condenses the timelines to only reflect a single update per base record. The same technique is applied when recursively processing the maintenance plan in batches. For each aggregation or join processing step, the VM condenses the timeline of the results, releasing only one *put* or *delete* update per view record. Again, this reduces the amounts of updates processed significantly.

Multi-row updates – Logically, it is not important for the system to synchronize the results of every multi-row update. For writing out the result after an incremental batch, the batch itself can be considered as one large multi-row transaction, in which every VM transitions from a view state that was valid before the batch operation (i.e., V) to a view state that is valid after the batch operation (i.e., V'). As transaction identifier serves the *batch ID*. The batch ID is contained in a token that is sent after the last update of a batch. During the whole batch, VMs manage a single split state. Thereby, the VMs keep the old value unchanged and apply all modifications to the new value. Then, when the VMs receive the token, they use the hashing scheme to compute the coordinator (i.e., $asg(batchID)$). The coordinator advises all VMs to write out their split-states of the complete batch. When the VMs have acknowledged the operation, the coordinator sends the resolve command, which concludes the update of the complete view state.

4.4 Supported view types

VMS supports a large spectrum of SQL expressions in defining views (e.g., SPJA, exists, case/when and nested query constructs). Here, we mainly develop techniques supporting (strongly) consistent view maintenance. We define a set of Pre-Processing View types that serve to derive standard views without resorting to indirection. To model view maintenance with Pre-Processing View types, we use generalized multi-sets (see Chapter 2.5).

Pre-Processing Views are materialized internally, in a sense that they are neither specified by clients nor exposed to them. Pre-Processing Views are kept as intermediate view tables in VS. Their purpose is two-fold. For some query types, Pre-Processing Views are essential in order to maintain strong consistency. For others (e.g., when used for pre-aggregation), Pre-Processing Views are used to trade off storage overhead in favor of read latency [61], as discussed in Section 4.5. Also, many application-level views can be derived from the same Pre-Processing View, amortizing its cost. Note that VMS handles Pre-Processing Views in the same manner as other views, which includes handling failures and lost updates (see Section 4.3.2). We now describe the three key Pre-Processing View types, we leverage in VMS.

Delta – A Delta view is a Pre-Processing View that tracks base table changes between successive updates. In KVS, client updates only contain the update information but they do not characterize the base record state before or after the operation. For example, a delete update only contains the row-key to be deleted and not the actual row values. For view maintenance, this information is vital; hence, we capture it in a Delta view. The Delta view records base table entry changes, tracking the states between row updates, i.e., the delta between two successive updates for a given key. Given a base table $R \in R_{set}$ in round (d_0, A_0). Then, we define the delta view as intermediate view I_{del} which is partitioned and indexed by base key A_0.

$$I_{del} = I_{del} \uplus \Delta R \tag{4.4.1}$$

We keep the records of R materialized in I_{del}. When a delete is propagated, the system uses I_{del} to find out the row values of the deleted record. To optimize and reduce the storage overhead of a delta view, it can be combined with a selection operator.

Pre-aggregation view – The Pre-aggregation view is a Pre-Processing View that prepares for aggregation by sorting and grouping base table rows. It serves two purposes: (1) pre-aggregating aggregation functions locally, such their results can be combined later (2) storing dis-aggregated records to re-compute a minimum or maximum after deletion. To materialize these aggregates without Pre-aggregation, VMS would have to send updates for every base table refresh. In addition, for min- and max-views, the deletion of the minimum (maximum) in the view would require an expensive base table scan to determine the new minimum (maximum), introducing consistency issues that result from the sought after values changing while a scan is in progress.

Sum/Count 1: Let a sum operation be defined over a universal view function as follows: $\gamma_{X,sum(A)}(View(R_{set}))$ with X, the grouping attribute and $sum(A)$ the grouping function (on some regular attribute A). Let the aggregation, in the interest of generality, be computed on top of a universal view function $View(R_{set})$. Further, let a maintenance plan be defined by the view function with *connecting view* I, belonging to distribution round (d_i, A_i). We call I the connecting view because, as the last intermediate view of $View(R_{set})$, it connects the view function with the sum operator. I can be a base table or an intermediate view, i.e., $I \in R_{set} \vee I \in \{I_1, ..I_n\}$.

To exploit locality, VMS builds and maintains the local pre-aggregate I_{Pre} in the same round d_i. While I_{Pre} is already grouped by X, the table is still partitioned by A_i. To build the global aggregate, a new distribution round (d_{i+1}, A_{i+1}) is added with attribute $A_{i+1} = X$. Then, the global aggregate is stored into a view I_{Agg}. Delta equations for both view are written as follows:

$$\begin{aligned} I_{Pre} &= I_{Pre} \uplus \gamma_{X,sum(A)}(\Delta I) \\ I_{Agg} &= I_{Agg} \uplus \Delta I_{Pre} \end{aligned} \tag{4.4.2}$$

Min/Max 1: Let a minimum (or maximum) operator over a universal view function be defined as $\gamma_{X,min(A)}(View(R_{set}))$. Then, the schema of the local pre-aggregation I_{Pre} can be defined flexibly such that the pre-aggregate in I_{Pre} can be updated with the current minimum and additionally, with the dis-aggregated value itself $\pi_{X,A,n}(\Delta I)$. Thereby X represents the grouping key, A the input value and $n \in \mathbb{N}$ the explicit multiplicity in case that there are many input values with the same value. The flexibility of the KVS schema lets us store both record types at the same time.

This way, when a minimum or maximum is deleted, the next one can be determined quickly by computing $get(I_{Rev}(r.X, A?))$ and evaluating the minimal value of A. This get operation uses fast row-key access to return all dis-aggregated values with $x = r.X$. Thereby, $A?$ is a wild card that returns only the rows of I_{Pre} where A is set (and not $min(A)$). The delta equations of the view construction are performed as follows:

$$\begin{aligned} I_{Pre} &= I_{Pre} \uplus (\gamma_{X,min(A)}(\Delta I) \cup \pi_{X,A,n}(\Delta I)) \\ I_{Agg} &= \gamma_{X,min(A)}(I_{Agg} \uplus \sigma_{A=\emptyset}(\Delta I_{Pre})) \end{aligned} \tag{4.4.3}$$

Again a local pre-aggregation I_{Pre} is used, which stores the dis-aggregated and the aggregate values locally (in VS). It only occupies the storage that is needed to store the dis-aggregation of the local partition. Then, later only the local aggregate is sent out, again to be combined globally in view I_{Agg} according to distribution key $A_{i+1} = X$.

Example 4.4.1*: Let a base table $R = (\bar{K}, X, Y)$ and a view $V = \gamma_{X,min(Y)}(R)$ be defined as before. Given three (local) updates that occur at base table R: $u_1 = put(R(k_1, x, 5))$, $u_2 = put(R(k_2, x, 3))$ and $u_3 = del(R(k_2, x, 3))$ which are propagated to VMS for view maintenance. After the first two updates, I_{Pre} materializes as: I_{Pre}=$\{(x, A{=}5, 1), (x, A{=}3, 1), (x, Min{=}3)\}$. Update u_3 deletes the local minimum and the system queries $min(get(I_{Rev}(r.X, A?)))$. The query delivers $\{(x, A = 3, 1), (x, A = 5, 1)\}$ and reevaluates the minimum as $(x, Min = 5)$.*

Reverse-join view – A Reverse-join view is an intermediate materialized view that supports the efficient and correct computation 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. A matching join row can only be determined quickly if the join-attribute is the row-key of the queried base table, or if an index is defined on the join-attribute for the table. Otherwise, a scan of the entire table is required, which requires a disproportional amount of time, slowing down view maintenance significantly and impacting consistency. However, the use of an index (like, for example, a hash table referencing one of the base tables) reveals the following drawbacks: (1) An index has to be updated to remain consistent with the base data. This adds latency and holds the danger of stale references. (2) While a VM follows a reference of an index, underlying base tables may change, thus, destroying (strong) consistency for derived views. To address these issues, we introduce the Reverse-join view.

Let a join operation be defined over two universal functions $View_A(R_{set}) \bowtie View_B(R_{set})$, where I_A, I_B are the connecting views of both functions and (d_i, A_i), (d_j, B_j) are the respective distribution rounds. Let the two connecting views be defined as $I_A = (\bar{A}_i, X, A'_1, .., A'_m)$ (with $A'_1, .., A'_m$ just being regular attributes) and $I_B = (\bar{B}_j, X, B'_1, .., B'_n)$ (with $B'_1, .., B'_n$ just being regular attributes) which are being joined over attribute X. Again, we use the flexible KVS schema to define a view which stores records from both join tables using $(X, A_i \cup B_j)$ as row-key to rapidly retrieve matching join rows.

When operations are propagated from both sides, they are redistributed according to X. For that reason a new distribution round is added with distribution attribute $A_{i+1} = B_{j+1} = X$. Using the join attribute I_{Rev} can be accessed from either side of the relation. Given that either side is updated with ΔI_A or ΔI_B, the changes are reflected in I_{Rev} as follows:

$$\begin{aligned} I_{Rev} &= I_{Rev} \uplus \pi_{X,A_i,A'_1,..,A'_m}(\Delta I_A) \\ I_{Rev} &= I_{Rev} \uplus \pi_{X,B_j,B'_1,..,B'_n}(\Delta I_B) \end{aligned} \tag{4.4.4}$$

This technique enables inner, left-, right-, and full-join as well as semi-join to directly derive from I_{Rev} without the need for base table scans (or additional indices), as we show below. Further, we can parametrize subsequent joins. Given ΔI_A (ΔI_B can be defined analogous but is omitted for conciseness), we build the join records as follows:

$$\begin{aligned} (I_A \bowtie I_B) \ : \ (\forall r \in \Delta I_A)(S = get(I_{Rev}(r.X, B_j?))) \ : \\ I_{Join} = I_{Join} \uplus (\{r\} \times S) \end{aligned} \tag{4.4.5}$$

$$\begin{aligned} (I_A \ltimes I_B) \ : \ (\forall r \in \Delta I_A)(S = get(I_{Rev}(r.X, B_j?))) \ : \\ I_{Join} = \begin{cases} I_{Join} \uplus \{r\} & if (S \neq \emptyset) \\ I_{Join} & else \end{cases} \end{aligned} \tag{4.4.6}$$

$$\begin{aligned} (I_A \rtimes I_B) \ : \ (\forall r \in \Delta I_A)(S = get(I_{Rev}(r.X, B_j?))) \ : \\ I_{Join} = \begin{cases} I_{Join} \uplus (\{r\} \times S) & if (S \neq \emptyset) \\ I_{Join} \uplus \{r\} & else \end{cases} \end{aligned} \tag{4.4.7}$$

To obtain the join result, the VM uses a get operation $S = get(I_{Rev}(r.X, B_j?))$ to retrieve all stored records with same join key $r.X$ from join table I_B. Thereby, the VM uses wild card $B_j?$ to only load the rows that originate at I_B. Because X is the first row-key and $A_i \cup B_j$

is the second row-key in I_{Rev}, access to fetch S is very fast. Then, the VM computes the cross-product of the update record r and S to determine the join records. A little example illustrates the case:

Example 4.4.2*: Let two base tables* $R_1 = (\bar{K}, X, Y)$ *and* $R_2 = (\bar{L}, X, Z)$ *and a view table be defined as* $V = R_1 \bowtie R_2$*. Also, let three updates be applied to the base tables as:* $u_1 = put(R_1(k_1, x, 5))$*,* $u_2 = put(R_1(k_2, x, 10))$ *and* $u_3 = put(R_2(l_1, x, 3))$*. Applying the first two updates, VMS builds the Reverse-join as* I_{Rev}=$\{(x, K{=}k_1, Y{=}5), (x, K{=}k_2, Y{=}10)\}$*. Now, for Update 3, the system queries* $get(I_{Rev}(x, K?))$ *once, and uses the result to build the natural join* $I_{Join} = \{(k_1, l_1, x, 5, 3), (k_2, l_1, x, 10, 3)\}$.

VMS, in general, executes two main strategies (and their combination) to load and compute a distributed join. Strategy 1, fully partitioned loading; and Strategy 2, partially partitioned loading. Both strategies are standard concepts in query processing.

Strategy 1 – We load a partition of either base table at every VM. Subsequently, we execute the strategy in three rounds. During Round 1, VMS redistributes the keys of both join tables according to the join key and sends them out to VMs. This step involves substantial n-to-m communication, as it triggers a VM to send many updates to different VMs. During Round 2, the VMs, in parallel, store the received records into their Reverse-join view. Ultimately, in Round 3, the join results are obtained.

Strategy 2 – When one of the base tables is sufficiently small, we load it completely at every node together with a partition of the distributed join table. We need only two rounds to compute the join: In Round 1, the intermediate join view is built directly, locally. In Round 2, the VM can build the final join and materialize the records if desired.

4.5 Evaluation

In this section, we report on the results of our extensive experimental evaluation. We fully implemented VMS in Java and integrated it with Apache HBase.

Experimental setup – Experiments are performed on a virtualized environment (OpenStack) of a cluster comprised of 42 physical machines (each equipped with 2x Intel Xeon (8 cores), 128 GB RAM, 600 GB SSD, 1 TB HDD, inter-connected with a 10 GBit/s network). The setup employs up to 222 virtual machines (running Ubuntu 14.04). For, HBase, a set of virtual machines is dedicated to the Apache Hadoop (v2.2.0) installation: one as name node (HDFS master) and a group of data nodes (HDFS). Co-located with HDFS, HBase (v0.98.24) is installed on those same virtual machines with one master and a group of region servers, respectively. The data node/region server machines are properly sized (4 cores, 20 GB RAM, SSDs for fast read/write performance).

TPC-H – The database benchmark contains a record set of typical business data, containing many columns of various data types (addresses, text fields, float values). Our objective is to use VMS to incrementally maintain views over TPC-H data, for fast access to summarized data used in online analytics. For evaluation we use scale factors $100x$ (~$100GB$) and $500x$ (~$500GB$). Using the TPC-H workload lets us evaluate the approach under realistic data loads being used in common applications. TPC-H is designed as an OLAP benchmark: it generates large table sets and also provides update workloads to *orders* and *lineitem* tables. However, generating dynamic OLTP workloads with different distributions is not possible. For that reason, we extended TPC-H to generate large transactional workloads on KVS including *put* and *delete* operations. During our evaluation, we use the full range of TPC-H query templates (except 5 and 21) and denote them with Q_1-Q_{22}. These queries capture a wide range of SQL expression, including specific expressions (e.g., case/when, exists, complex joins) and manifold nested queries. For each query, we load a view definition that can be materialized and maintained by our system.

Bulk loading performance – In this section, we evaluate the sole performance of VMS when materializing or maintaining single views (defined by a single TPC-H query). Therefore, we allocate most of the cluster resources to VMS and deploy 400 VMs on 200

dedicated nodes. HDFS and HBase are deployed only in a small setup, used for view materialization only, on 22 nodes (1 master, 1 ZooKeeper, 20 region servers). We push VMS to its limits, generating workloads with a scale factor of 500 (creating a $500GB$ database with ~4.3 billion records).

For the evaluation, we use two different workloads: A and B. Workload A consists of table records and simulates a scenario where views are materialized from existing base table sets (i.e., R_1, ..R_n). Workload B consists of base table updates (i.e., ΔR_1, ..ΔR_n) and simulates a scenario where updates are streamed from a database. However, both workloads are generated and stored in form of part-files at each VM. To execute them, a VM must simply load the files and read the workload.

The experimental procedure is the following: we configure VMS using one of the 22 TPC-H queries and let VMS load the view definition to all VMs. Then, either workload A or workload B is bulk loaded (either from disc or directly from memory). We let VMS perform the complete maintenance process and measure its execution time. The execution time is the time taken to load the workloads and to fully materialize the results into a view table of KVS.

Figure 4.5.1 shows the execution times of the tested TPC-H queries. Figure 4.5.2 shows the maintenance throughput of the same TPC-H query set. The throughput figures are normalized with the number of base table updates being processed. Thus, throughput is displayed as the overall and average number of base table records (btr) or base table udpates (btu) that VMS can convert into view updates in a second, expressed as btr/s or btu/s.

The best performances are achieved by VMS for pure aggregation queries (i.e., Q_1, Q_6), queries with small base tables (i.e., Q_2, Q_{16}) and queries with high selectivities (i.e., Q_{12}, Q_{14}). VMS always uses pre-aggregation to aggregate keys locally at each VM, and later combines the results. The higher the aggregation ratio is (aggregation functions without *group bys* have the highest ratio), the lower the communication overhead and the materialization cost of KVS. When loaded from memory, query Q_6 achieves the best performance (due to its high selectivity) aggregating ~$320GB$ of data in 13 seconds and achieving an average throughput of ~$235M\ btu/s$ (maximum throughput is ~$348M\ btu/s$).

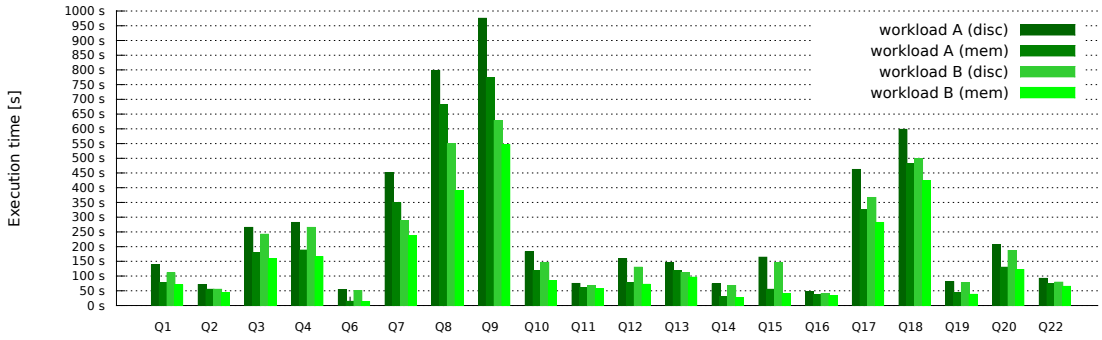


Figure 4.5.1: Bulk loading (execution time)

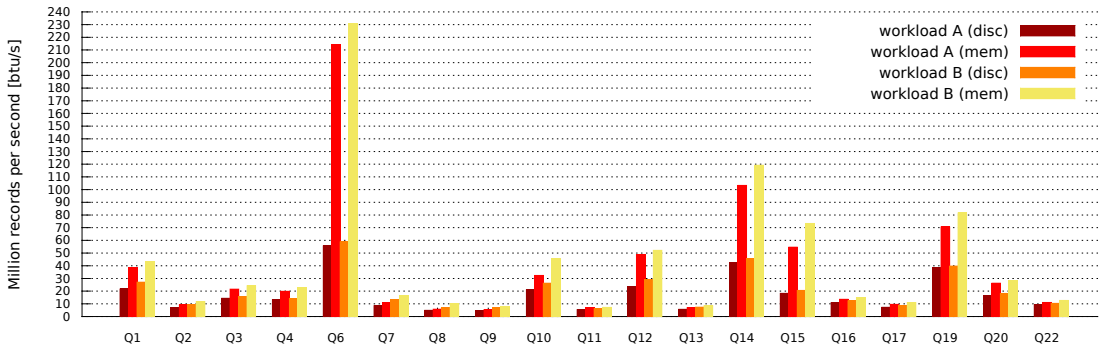


Figure 4.5.2: Bulk loading (throughput)

The next-best results are achieved by query Q_{14}, representing a two-table join ($lineitem$, ~$3B$ records; $part$, ~$100M$ records) followed by an aggregation. When loading from memory, the execution time is 26 seconds and the average throughput is $119M\ btu/s$ (maximal throughput is $259M\ btu/s$). Query Q_{12} (joining $lineitem$, ~$3B$ records; $orders$, ~$750M$ records) can be processed in 72 seconds at $52M$ btu/s. VMS is suitable for joining two large tables: the key-range of both join tables can be partitioned and the resulting Reverse-join (see Section 5.1.3) can be distributed over the VMs.

The longest execution times are observed for manifold join views involving large base tables (i.e., Q_7, Q_8, Q_9). Here, VMS is constrained by the number of updates that can be transported over the network. To accelerate the computation, VMS loads the smaller tables (i.e., $region$, $nation$) completely at every VM (see partially partitioned loading, Section 5.1.3), whereas the larger tables (i.e., $lineitem$, $orders$, $customer$) are split such that a VM loads only a partition (see fully partitioned loading, Section 5.1.3). In this way, some of the joins can be computed locally such that Q_7 executes in 236 seconds when loading from memory. Moreover, an average throughput of $16.2M\ btu/s$ can be maintained (max throughput is $70M\ btu/s$) and the total number of records is ~$3.8B$.

The reductions (in execution time) we obtain by loading records from memory rather from disc are highly depending on the query template. Queries with large base tables and operations that can be applied locally (i.e., selection, pre-aggregation, local join) profit the most. Q_6 and Q_{15} yield a reduction of 74% and 66%, Q_1 yields 44% and Q_2 Q_7 only yield ~23 (due to their high share of communication overhead). In general, it can be said that the improvement for memory loading is quite significant and should be particularly considered when loading queries repeatedly from the same source.

The differences between workload A and workload B are significant for some queries but negligible for other queries. Queries Q_1, Q_3 and Q_4 show a small difference between workload A and workload B (10%-20%), and queries Q_7, Q_8 and Q_9 show a larger difference (32%-43%). For long-running queries (i.e., the latter queries) workload A presents a challenge, as it is a pure insert workload. After a while, inserting records fills the memory, causing the access time to decline. By contrast, workload B provides a well-balanced memory condition, as 35% of the records are deleted over time.

HBase performance – The second set of experiments evaluates VMS in combination with HBase. Thereby, HBase stores base and view table data. VMS either performs basic materialization by loading base tables from HBase or incremental maintenance by monitoring HBase co-processor or TL. In this setup, HBase and VMS share cluster resources. We use 200 nodes each equipped with a HDFS data node, a HBase region server, a co-processor and a VM. Four more nodes are used as master and ZooKeeper nodes.

Basic materialization – In the first step, we load data from existing base tables in HBase and materialize the results into a view table. Prior to each experiment, we create a set of base tables (chosen from TPC-H). These base tables are, then, filled with data using a scale factor of 100.0 (~$100GB$). To fill the tables, we use 200 clients, each generating and loading a part of the overall workload to HBase. We load and execute one of the TPC-H queries via the VMS console and measure its execution time without, first, pre-computing any Pre-Processing Views. The evaluation time always includes the time to materialize the view into an HBase table.

Figure 4.5.3 and Figure 4.5.4 show the results of execution time and throughput. Maximal performance of VMS is achieved for queries with high selectivity and strong aggregations. For example, queries Q_1 and Q_6 are materialized in 42 and 21 seconds from HBase. Average throughputs ranging from $14M$ up to $28M$ btr/s are achieved (maximum is $122M$ btr/s). Also, joining few (large) tables as in Q_3, Q_{10}, Q_{12}, Q_{14}, and Q_{19} can be done with an average throughput of $12.5M$-$31M$ btr/s.

Queries without join operations or with a high selectivity such as Q_1, Q_6, Q_{10}, Q_{12}, Q_{14} are bound by loading and processing speed. To speed up scanning, we process selections (over base table records) already at HBase. Thereby we use HBase filters and hand the selection predicates to the scanner such that non-matching records are directly dropped and not forwarded to VMS. Also, we configure the HBase scanner to fetch only columns that are needed by the query.

Again, the largest operations are found in queries Q_7, Q_8 and Q_9. Here, VMS joins 5, 7 and 6 tables. The number of records involved are $767M$, $778M$ and $852M$. While processing these large joins, VMS is able to maintain a stable average performance of $4M$-$7M$ base records per second. Due to VMS's stream-oriented processing style, join views over very

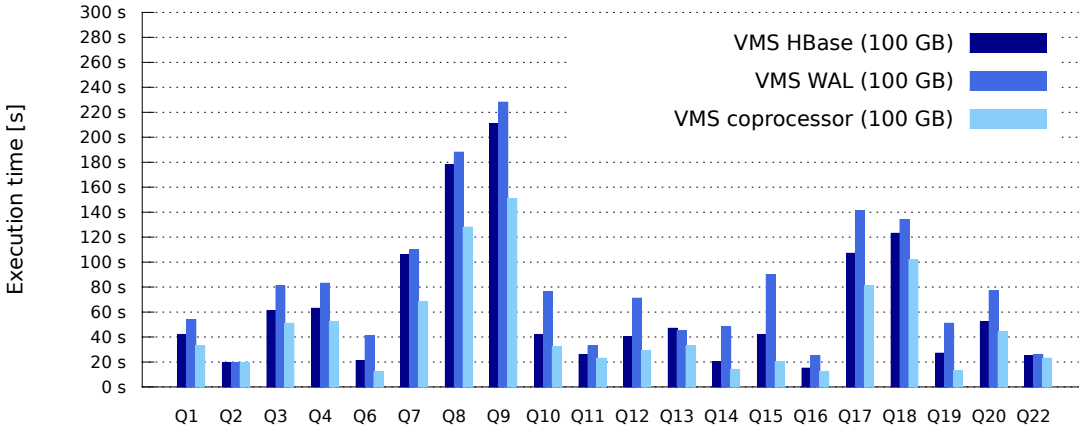


Figure 4.5.3: HBase performance (execution time)

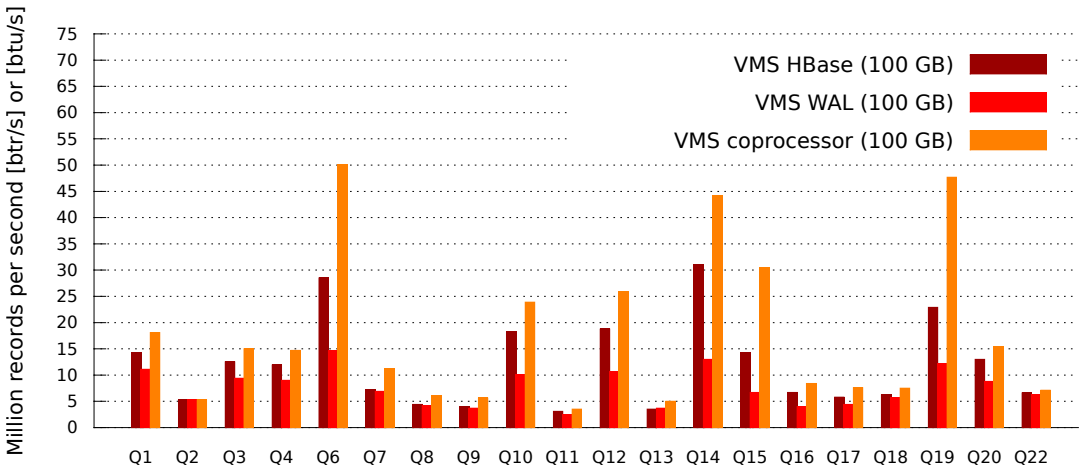


Figure 4.5.4: HBase performance (throughput)

large base tables are supported. The size is only limited by the size of the materialized Pre-Processing Views, if for performance reasons, they are kept in-memory.

While loading data from HBase has some performance drawbacks (in comparison to bulk loading), it is favorable in many use cases considering that the base data is kept in a KVS. All mechanisms to ensure availability, fault-tolerance and preserve the consistent state over a record's lifetime are managed by HBase.

Incremental maintenance – This set of experiments evaluates VMS in incrementally maintaining a set of materialized views, given base table data changes. We evaluate the two primary designs introduced in Section 4.1.2 (Design 2 and Design 3). In Design 2, each VM connects to HDFS and reads the WAL of a (single) region server. Thus, it reads operations in sequence without using multi-threading. In Design 3, we deploy a co-processor at each region server to intercept events that occur due to client base table updates. The events notify the local VM that is deployed along with the co-processor in the same JVM.

Incremental maintenance is inherently faster than basic materialization because there is no need to load complete table sets. Additionally, the number of updates, after which the view table shall be refreshed, can be configured, leading to the intended latency (see view freshness). However, to make a fair performance comparison, we evaluate the execution time using the same number of incremental updates as base records generated for basic materialization.

We create a set of base tables (chosen from TPC-H). Then, we prepare VMS for incremental view maintenance by loading one of the TPC-H queries as views. Finally, we start 200 update clients that apply a mixed update stream to HBase (using scale factor 100 to generate $100GB$ of put and $delete$ updates). Then, we let VMS asynchronously processes the updates.

Figure 4.5.3 and Figure 4.5.4 show the results of execution time and throughput. Co-processor loading achieves the best execution times of $12s$ – $14s$ for Q_6, Q_{14} and Q_{19}. Thereby average throughput ranges from 44-50 btu/s (maximum is $145M$). In contrast to basic materialization, co-processor loading does not require HBase filters; queries with high selectivity are computed rapidly from memory. Also for long running joins Q_7-Q_9,

co-processor loading excels with $6M$-$11M$ btu/s. This might also be related to the high selectivities. Observing 3-table joins Q_3 and Q_4, the advantage diminishes ($51s$ and $52s$ compared to $61s$ and $63s$ for basic materialization).

In general, we measure reading from the WAL as the slowest alternative. The more updates VMS has to load, the more bottleneck becomes the network. For queries Q_2, Q_{11} and Q_{16}, which join many small tables and do not include the large $orders$ and $lineitem$ tables, almost no disadvantage can be observed. Execution times are 19 – $33s$ (compared to 19 – $26s$ for co-processor loading). However, for queries Q_7, Q_8, Q_9 and Q_{17}, Q_{18} which include $orders$ and $lineitem$ and have low selectivities, execution times raise to 110–$228s$ (compared to 68 – $151s$ for co-processor loading). The highest difference can then be observed for large table with high selectivity, for example, Q_6, Q_{12}, Q_{14} and Q_{19} with 41 – $71s$ (compared to 12 – $29s$ with co-processor loading).

Overall, reading the WAL performs slower than co-processor loading with a factor of 1.2-$3x$. VMs have to read the WAL over a network connection. Also filters (for projections and selections) cannot be applied during the reading process such that update operations have to be fully retrieved before they can be dropped (or columns can be projected). However, the advantages of reading the WAL (Design 3) are given through the fault-tolerance capabilities and the non-interference with HBase. Using many small tables with low selectivity also the performance aspect disappears.

Comparing raw processing speeds, generally speaking, we found incremental maintenance (via co-processor) to be executed faster (1.3-$2.1x$ depending on query) than basic materialization (via scan). Even though, basic materialization does not rely on Delta views (as no before-after state has to be tracked) and maximal batching can be applied, a scan always takes more time than loading up the queued update operations. While basic materialization can only be used to process complete base tables, it is still an essential part of VMS when materializing views on top of existing base data or when defining new queries after a stream of updates has been processed (and update operations are not available anymore).

Comparison to Apache Phoenix – In Figure 4.5.5, we compare VMS to the performance of Apache Phoenix (v4.14.1). To the best of our knowledge, Phoenix is the only open source framework that supports SQL query processing and view materialization as

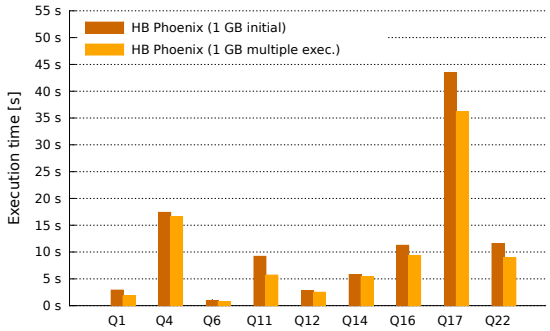


Figure 4.5.5: Apache Phoenix comparison

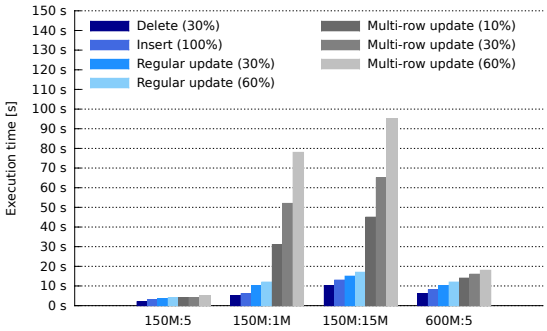


Figure 4.5.6: Multi-row updates

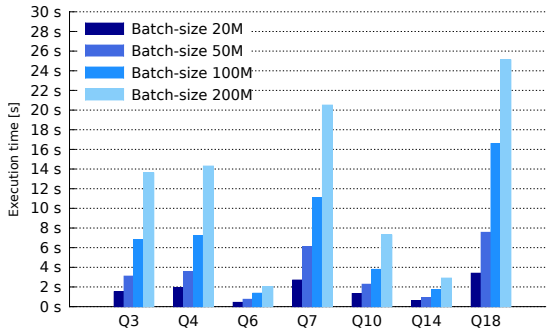


Figure 4.5.7: View freshness

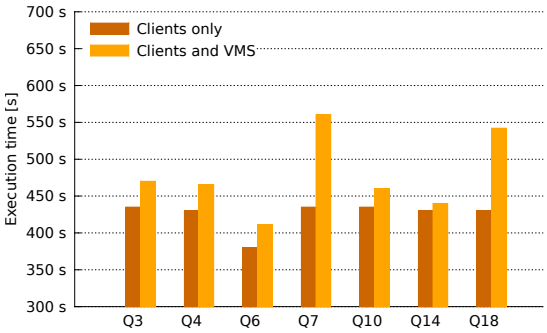


Figure 4.5.8: VMS overhead

a layer on top of HBase. We focus our comparison on VMS basic materialization versus Phoenix as no strong consistency preserving incremental materialization is supported in Phoenix (see Chapter 3).

We use a setup of 200 region servers and deploy a Phoenix instance (that is loaded via co-processors) along with each one of them. We import the TPC-H data, setting the number of salt buckets of each base table to 200 (such that data is evenly distributed among the region servers). We modify the HBase configuration, allowing Phoenix to take $10GB$ of memory (at client and) per server. Then, we start the Phoenix console and issue TPC-H queries against the interface one by one. We measure two execution times, one after HBase startup (initial) and one after HBase warm up (after multiple executions).

While we find pure aggregation performance (i.e., Q_1 and Q_6) up to par with VMS (also at scale factors of 100), join performance of Phoenix does not measure up. For that reason, the figure depicts TPC-H queries at a scale factor of 1. Join queries with high selectivity and low result size (Q_{12}, Q_{14}) terminate relatively quickly (2.8s and 5.8s), some of the larger joins (Q_3, Q_7, Q_{10}) do not complete (despite of using different join implementations).

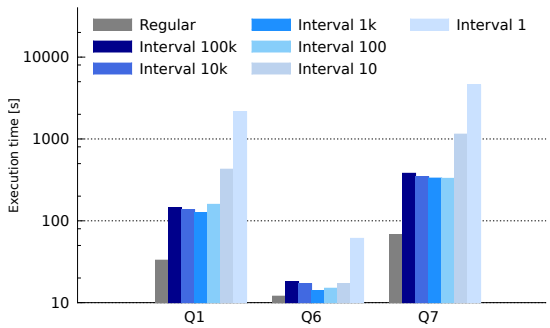


Figure 4.5.9: Fault tolerance

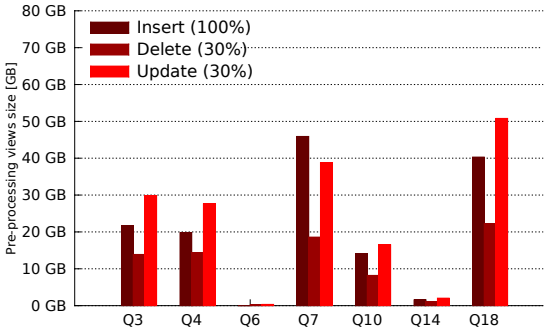


Figure 4.5.10: Pre-processing views

However, for queries Q_{16} and Q_{17}, we conclude that performance at scale factor 1 (11.2s and 43.4s) remains in similar ranges as VMS at scale factor 100 (8s and 58s). While Phoenix uses a last step to assemble and display results at the client, VMS computation remains distributed even for the last materialization step. Views are accessible by hundreds of clients in parallel as HBase tables.

Multi-row updates – Figure 4.5.6 shows the results of applying multi-row updates to the system. To isolate the effect, we run a set of aggregation queries (without selection) using tables *orders* and *lineitem* and different group by columns, establishing different aggregation ratios. Thereby we vary the share of insert, delete and update operations. We use a 30% delete, a 100% insert and two regular update workloads (30% and 60%) where aggregation keys are not changed. Additionally we use multi-row update workloads (10%, 30% and 60%) where aggregation keys are changed for every update.

Results indicate that multi-row updates have a small impact on overall performance when the aggregation is strong. In that case also the meta information of many updates connecting the same aggregation keys can be merged. Aggregating the *status* column of the orders table ($150M : 5$) only causes a neglectable overhead of factor 1.3x. However, when aggregating only moderately using the *clerk* column ($150M : 1M$) or subtle using the *custkey* column ($150M : 15M$) the factor is already at 3.4-5.3x for 10% share and 7.3-13x for 60% share.

Multi-row updates can slow down VMS performance significantly because meta data and split states have to be managed and cleaned up. With increasing result size, the impact worsens. Therefore, it is recommended to use multi-row-updates together with strong

aggregations or in use cases where their share is below the 10% mark.

View freshness – Figure 4.5.7 shows the freshness of results materialized in view tables managed by VMS. We use the incremental setup of Design 2 (i.e., 200 region servers, each with a co-processor and 200 VMs). Via clients, we apply a mixed workload and let the updates be forwarded to the VMs. Depending on the configured batch-size, the VMs process a sub-set of the overall workload and write back results to view tables.

Then, the next incremental batch starts, until the workload is finished. We use batch sizes of $20M$, $50M$, $100M$ and $200M$ updates, and measure the time it takes for the 200 VMs to compute a batch and synchronize.

The batch size is a trade-off between computational effort and freshness of results. VMS can be adjusted according to the use case. When choosing a small batch size (of $20M$ records), the staleness of view tables remains below the one second mark for Q_6 and Q_{14}, below the two second mark for Q_3, Q_4 and Q_{10}, and below the three second mark for Q_7. However, the effort of materializing results is small for query Q_6 (only a single record), whereas the effort for Q_3 and Q_{10} can be up to $1M$ and $2M$ records per incremental batch. In that case a higher batch size, e.g., $200M$ records might be reasonable to operate the system more efficiently.

VMS overhead – In Figure 4.5.8, we evaluate the overhead of VMS when doing view maintenance with HBase while serving a client request load at the same time. We compare the application of a workload to HBase with and without VMS enabled. In the first experiment, we generate a workload for each base table (that is used in the corresponding TPC-H query) and use 200 clients to insert it into HBase. We apply all workloads concurrently and measure the overall execution time. These experiments are labeled "Clients only".

In the second experiment, we use the same setup as before (Design 2 with co-processors), and we prepare the system as follows: we load a TPC-H query using the VMS console, and enable incremental tracking of the TPC-H query. The batch size of the maintenance jobs is set to a fix size of $50M$, such that VMS will collect the same number of updates and process them in one go. These experiments are labeled "Clients and VMS".

The results confirm that VMS runs independently and interferes little with HBase processing. For queries with high selectivity and little communication (Q_6, Q_{14}), the overhead is 7.5% and 3.4%. Also, for queries Q_3, Q_4 and Q_{10}, the overhead remains well in the 3-8% range. Only for the compute and communication intensive queries Q_7 and Q_{13}, the system experiences overheads of up to 22%.

Fault tolerance – In Figure 4.5.9, the impact of running VMS as a fault tolerant system is depicted. Therefore, we compare the basic materialization of queries Q_1,Q_6 and Q_7 using regular execution against execution with transaction logs enabled (at each VM). As explained, transaction logs are standard files which are written sequentially and stored into the underlying distributed files system (HDFS).

The HDFS clients provides a feature to synchronize the files in specified *sync intervals*. We vary this parameter, using values $100k$, $10k$, $1k$, 100,10 and 1. Thereby, the last value 1 provides the best synchronization as every received update at a VM is safely persisted into the log. An interval of $100k$, on the other hand, bears the risk of losing exact the amount of updates which, then, have to be requested from other VMs, again.

During evaluation, we fill the database, load the queries and materialize the results for each sync interval separately. As results differ vastly we present them using a log scale on the y-axis. Depending on the query, writing a transaction logs induces a significant overhead for query materialization: for Q_1 we observe a $3.8x$, for Q_6 a $1.5x$ and for Q_7 a $4.8x$ reduction. Further, we assert that small synchronization intervals (i.e., 10 or 1) increase the execution time, significantly.

Executing Q_1 and Q_7 with sync interval 1, we observe performance penalties of factors 65 and 67. Thus, full synchronization should be avoided. Also for larger sync intervals, a plausibility check reveals the following: for Q_1, fault-tolerant execution yields $126s$, whereas regular execution yields $33s$. We conclude that in a crash scenario a complete reevaluation of views could be executed faster than a fault-tolerant execution.

Pre-Processing Views overhead – In Figure 4.5.10, we investigate the storage overhead of intermediate Pre-Processing Views and determine the space that they occupy (in memory). We let VMS compute the complete workload at once (i.e., without using incremental batches). When the last client update has been applied (just before VMS flushes the

intermediate tables and materializes the results), we stop and determine the size of all Pre-Processing Views involved in the query. We use three different types of workloads: (A) pure insert, (B) 30% deletes, (C) 30% updates.

Depending on the TPC-H query, the size of the used Pre-Processing Views can differ a lot. Again, queries with strong aggregations or high selectivity occupy almost no space, as records are either dropped or heavily pre-aggregated. Q_6 has a maximum of $0.35GB$ (due to Delta view), Q_{14} a maximum of $2.02GB$. For regular join queries, like Q_3, Q_4 and Q_{10}, we observe a difference.

Workload B occupies the least space (8-14GB) because records that are deleted in the base table are also deleted in the Delta and Reverse-join views. Workload A comes second (14-22GB) because the size (of Reverse-join views) is indeed the largest, but Delta views are not needed. Therefore, the largest intermediate sizes (16-30GB) can be found for Workload C. The only exception represents Q_7 where the number of Reverse-join views is the highest (due to the number of join tables).

Chapter 5

Multi-view processing

While materialized views have been extensively researched in the past [34, 62, 63, 64], the focus of view maintenance has shifted from standalone data warehouses to large-scale distributed data storage, including batching and in-memory computation [14, 15, 16, 65]. However, in most approaches, incremental maintenance is a dedicated process that involves computing a large set of update transactions (identified as a delta set) and computing the desired outcome. Therefore, existing approaches are limited to the maintenance of a single large view table derived from a single large (partitioned) base table, where the latter is updated when the former changes.

Materialized views, which are equipped with large-scale processing capabilities, appear to be suited not only to store the results of individual computations (for repeated access) but also for materialization and live tracking of thousands of analytical queries simultaneously. Materialized views can be used in distributed databases to maintain high transactional throughput while providing low-latency access to a large number of result sets. Thus, materialized views provide constant availability, fast access and fault tolerance through the distributed storage system.

In this chapter, we focus on how VMS achieves maximum throughput in a multi-view setup. We exploit techniques from multi-query optimization [38, 39, 40, 41, 42] and propose new methods to perform multi-view optimization in a distributed context. Then,

we explain how VMS uses dedicated intermediate views to combine and share operations of multiple different maintenance plans. Finally, we evaluate how VMS performs efficient materialization and maintenance of thousands of views in parallel. We make the following contributions:

1. We propose a novel *bit vector schema* that enables pre-evaluation of predicates and allows the merging and combined maintenance of many views in parallel (see Section 5.1.1).
2. We propose a method called *decomposed pre-aggregation* to merge and optimize incremental maintenance of multiple distributed aggregation views in parallel (see Section 5.1.2).
3. We explain how VMS uses distributed Reverse-join views to accelerate maintenance of multiple join views built from different types and ranges (see Section 5.1.3).
4. We evaluate VMS in a multi-view setup and show how it scales up to $10k$ materialized views while still maintaining large data sets and providing reasonable execution times of 100 – 500 seconds (see Section 5.3).

5.1 View concepts

VMS supports a broad variety of SQL expressions (e.g., like, exists, case/when) and nested query constructs. However, to define recursive delta rules for multiple queries, we apply the typical SPJA pattern (i.e., selection/projection, aggregation, join). For each operation type, we discuss how VMS performs multi-view optimization and executes the maintenance plan. To describe multi-view optimization with Pre-Processing View types, again, we use generalized multi-sets (see Chapter 2.5).

5.1.1 Selection and projection

When VMS performs basic view materialization (i.e., it selects over R) in the maintenance plan, selection and projection are performed on-the-fly. Thus, VMS simply selects or drops a record depending on the evaluated predicate: there is no view state to be captured. By contrast, in the context of incremental maintenance, ΔR tuples, which represent modifications of the base table, are processed. As such, VMS needs to create an intermediate view I_{Sel} which stores the selected records and awaits incremental updates.

Example 5.1.1*: Given a base table* $R = (\bar{K}, X, Y)$ *and a view defined as* $V = \sigma_{(Y<5)}(R)$*, consider an update* $u_1 = put(R(k, x, 4))$ *matching the predicate that could be followed by an update* $u_2 = put(R(k, x, 7))$ *unmatching the predicate. First, the system adds* $R(k, x, 4)$ *to* I_{Sel}*; then, processing* u_2*, it queries* I_{Sel}*, retrieves* $I_{Sel}(k, x, 4)$ *and deletes* k *from the table. VMS cannot simply drop update* u_2*. It must know about the existence of* k *in order to update intermediate view state of* I_{Sel}*.*

Consider a set of selection views being defined in different maintenance plans over the same base table. Processing those selections in VMS can be easily done by creating n intermediate views I_{Sel} and querying, respectively updating them for each update on the source table. However, querying n views for possibly overlapping record sets is inefficient. Thus, we introduce a combined intermediate view that captures the state of all records selected by the union of predicates. Instead of retrieving the state from n different views (or performing n different intermediate selections), we materialize a single intermediate selection and evaluate the predicates in a combined effort.

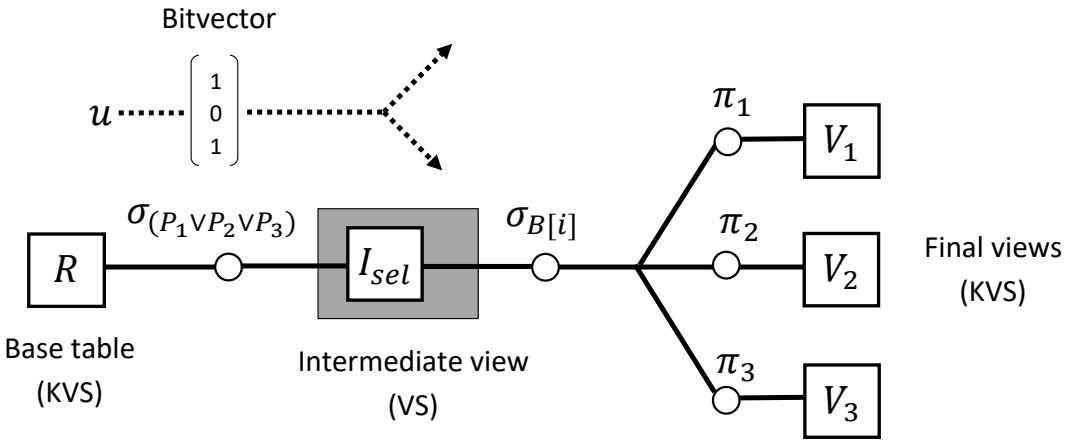


Figure 5.1.1: Selection merge (maintenance plan)

Let a set of n selection views be defined over the same universal view function as $(\forall i \in \{1, .., n\}) : \sigma_{P_i}(View(R_{set}))$. Let the result of $View(R_{set})$ be computed in a maintenance plan with connecting view I. Then, we define a combined selection view as I_{Sel} and write the incremental equations as follows.

$$\begin{gathered} I_{Sel} = I_{Sel} \uplus \sigma_{\bigvee\limits_{P \in P_{list}}}(\Delta I) \\ (\forall i \in \{1, .., n\}) : V_i =: V_i \uplus \sigma_{P_i}(\Delta I_{Sel}) \end{gathered} \tag{5.1.1}$$

Instead of producing n incremental updates and evaluating each P_i separately, we build a list of predicates $P_{list} = \{P_1, .., P_n\}$ and define an operator $\sigma_{\bigvee_{P \in P_{list}}}(R)$. Then, all record states can be maintained in intermediate view I_{Sel}. Final views V_i are only used to materialize results. However, in case that the n view definitions are complex such that each selection operators is followed by subsequent maintenance operators, the results cannot simply be materialized after I_{Sel}. In such a case a mechanism is required to distinguish between the different predicates in the further processing of the maintenance plan. Thus, we introduce the concept of *bit vectors* in the following.

Bit vectors – The intuition of *bit vectors* in view maintenance is the following. When a number of selections (with n different predicates coming from n different view definitions) is applied to a stream of base table updates. Then, what happens is that one base update can transform into a maximum of n updates in the system (when all predicates evaluate to true). Especially defining n complex view definitions, where selections are followed by aggregation or join views, this process represents enormous overhead. Using bit vectors, we can merge all selection operators over the same base table R and treat them as single operator. Thereby, VMS only transmits a single base table update to subsequent maintenance operators and sends the result of the predicate evaluations along with the bit vector.

The bit vector has a length of n bits where each position stores the result of a predicate evaluation over the given base table update ($\mathcal{B}[i] \in \{0, 1, \emptyset\}$). The bit vector is passed along such that subsequent operators in the maintenance plan remember the amount of propagated and dropped updates by looking at the corresponding position in the bit vector. If all positions of a bit vector evaluate to zero (i.e., $(\forall i \in I) : \mathcal{B}[i] = 0$), we also write $\mathcal{B} = 0$ which usually indicates (if the key is nonexistent in the view) that the complete base table update can be dropped.

Example 5.1.2*: Figure 5.1.1 shows an example of merging three different selection queries with definitions* $V_1 = \sigma_{(Y=5)}(R)$*,* $V_2 = \sigma_{(Y<5)}(R)$ *and* $V_3 = \sigma_{(Y<=5)}(R)$*. Instead of maintaining plans separately,* I_{Sel} *evaluates predicates* P_{list}=$\{(Y$=5$), (Y$<5$), (Y$<=5$)\}$ *in a combined effort. Given an update* $u = put(R(k, x, 5))$ *at base table R that is passed to VMS to perform view maintenance, a get operation* $\sigma_{K=k}(I_{sel}) = \emptyset$ *reveals that there is no existing record. The bit vector is set to* $(1, 0, 1)$ *and passed further. When the update are materialized, VMS uses the bit vector to apply u to* V_1 *and* V_3*, whereas view* V_2 *is omitted.*

Splitting bit vectors – As long as subsequent operators (of the n different view definitions) in a maintenance plan can be also merged the updates can be passed along with the bit vector. However, when subsequent operators lead to a separation of maintenance paths, updates have to be split by the information in the bit vector. When splitting the updates (ideally, immediately before their final materialization), the system does not reevaluate the predicates. Instead, $\mathcal{B}[i]$ is used to decide whether to route an update along the i^{th} view path. When the bit at the i^{th} position is set to one, the update is copied and routed further along that path; when the bit is set to zero, the path is omitted. If a

position is not set (i.e., ∅), the system attempts to reevaluate the predicate at the position of the split. For incremental maintenance, the following equation is realized:

Merging bit vectors – When joining records that are passed along with a bit vectors, their bit vectors can likewise be merged. This requires the bit vectors to be computed based on the same list of predicates and, thus, be constructed of the same length. To merge bit vectors we can perform an operation $\mathcal{B}_A \oplus \mathcal{B}_B$ as:

$$\forall(i \in I) \,:\, \mathcal{B}[i] = \begin{cases} \mathcal{B}_A[i] \vee \mathcal{B}_B[i] & \text{if } \mathcal{B}_A[i], \mathcal{B}_B[i] \in \{0,1\} \\ \mathcal{B}_A[i] & \text{if } \mathcal{B}_A[i] \neq \emptyset, \mathcal{B}_B[i] = \emptyset \\ \mathcal{B}_B[i] & \text{if } \mathcal{B}_A[i] = \emptyset, \mathcal{B}_B[i] \neq \emptyset \\ \emptyset & \text{else} \end{cases} \tag{5.1.2}$$

Using bit vectors, we now reformulate Equation 5.1.1. On top of combining the selection views into I_{sel}, we substitute the the selection operator in the second equation with a bit vector. Then, during execution of incremental maintenance on I_{sel}, the bit vector is used to remember the result of the predicate evaluation of the i^{th} selection view.

$$\begin{gathered} I_{sel} = I_{sel} \uplus \sigma_{\bigvee\limits_{P \in P_{list}}}(\Delta I) \\ (\forall i \in \{1,..,n\}) \,:\, V_i =: V_i \uplus \sigma_{\mathcal{B}[i]}(\Delta I_{sel}) \end{gathered} \tag{5.1.3}$$

For selections that are defined over the same base table, we can always rewrite predicates such that they are evaluated together. Even when the result sets of predicates do not intersect, combined evaluation of predicates makes sense as it reduces the number of updates in VMS.

5.1.2 Aggregation

When sets of aggregation views are defined over the same base table, we can exploit the fact that they share similarities. First, we merge multiple Pre-aggregation views that are based on the same set (or subset of) grouping keys such that their aggregated results can be maintained within the same Pre-aggregation using only a single maintenance step. Further, we can substitute functions that can be derived from others.

Let a set of aggregations operations be defined over a universal view function as $(\forall i \in \{1, .., n\}) : \gamma_{X,F_i}(View(R_{set}))$. Let the result of $View(R_{set})$ be computed in a maintenance plan with connecting view I. Let all aggregation functions be stored in a list $F_{list} = \{F_1, ..F_n\}$. Functions that are repetitions or that can be derived from others are eliminated. When redundancy is removed, a list of decomposed functions F_{dec} serves to store the aggregated values. It can be used at a later point to reconstruct all the functions of F_{list}. The incremental equations are built as follows (see Figure 5.1.2).

$$\begin{aligned} I_{Pre} &= I_{Pre} \uplus \gamma_{X,F_{dec}}(\Delta I) \\ I_{Agg} &= I_{Agg} \uplus \Delta I_{Pre} \\ V_i &= V_i \uplus \pi_{X,F_{dec} \rightarrow F_i \in F_{list}}(\Delta I_{Agg}) \end{aligned} \tag{5.1.4}$$

We merge all (local) pre-aggregates, that use the same set of aggregation keys X into one operation and intermediate table I_{Pre}. Additionally, the final aggregate I_{Agg} is built in a single intermediate view using the functions of F_{dec}. In the final step, when maintenance is split again, functions F_i are reconstructed from F_{dec} such that the results can be materialized for each aggregation, separately.

Example 5.1.3*: Let a view be defined as* $\gamma_{X,F_{list}}(R)$*, where a list of aggregation functions is given as* $F_{list} = \{sum(Y) * sum(Z), avg(Y), 5 * count(Y)\}$*. Instead of maintaining the full list, the system computes pre-aggregation using only using decomposed functions* $F_{dec} = \{sum(Y), count(Y), sum(Z)\}$*. For example,* $avg(Y)$ *is built from combination* $sum(Y)/count(Y)$*.*

Decomposing Pre-aggregation – It is a well-known use case in applications that queries with similar structure are executed on different selection ranges. Especially similar aggregation operators are oftentimes used along with different selection predicates. For

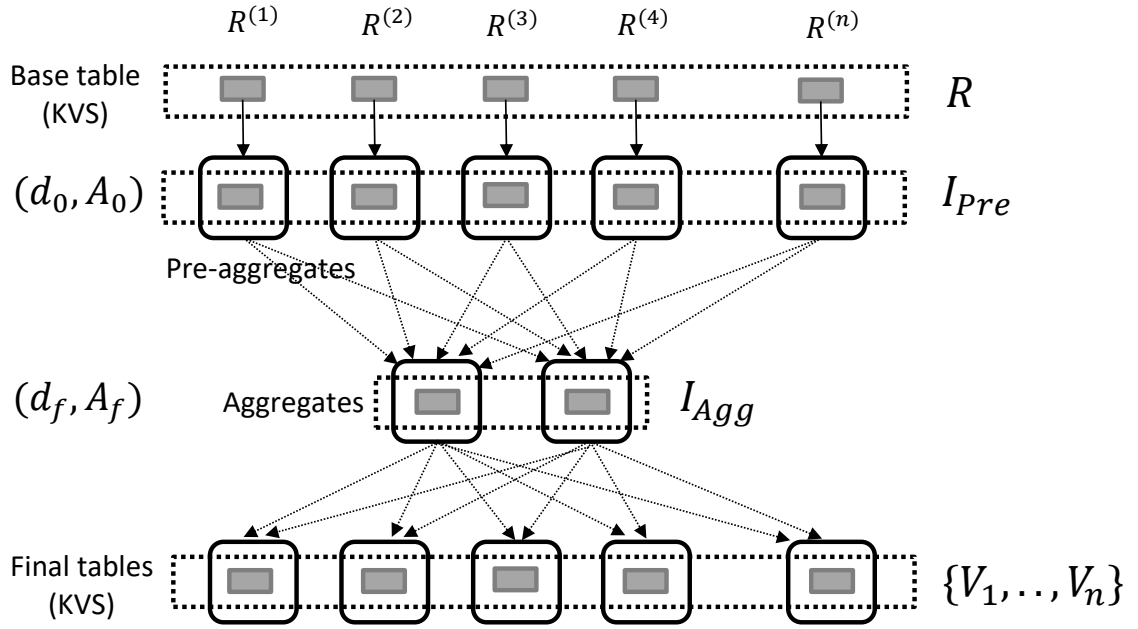


Figure 5.1.2: Execution of multiple aggregation views

VMS, as long as the maintenance plan is not adopted, it will simply maintain n different plans, splitting the maintenance process already at beginning.

Given that the selection predicates of the aggregation queries have already been merged in a previous step (like described in Section 5.1.1) and their updates are propagated along with a bit vector which contains the evaluation results. Then VMS can, likewise, merge all aggregations into a combined operator. This combined operator still computes aggregates of different selection ranges separately. But it does so using a single update. And moreover, when two selections overlap, VMS computes (i.e., incrementally updates) the aggregate of the overlap only once, and later merges it with each aggregate of the remaining selection ranges.

Decomposing Pre-aggregation is performed using three planning steps prior to the actual view maintenance process, which are described in the following: (Step 1) decompose predicates, (Step 2) substitute predicates by tags, and (Step 3) build new aggregation keys.

Step 1 – In the first step, VMS computes a decomposition of the predicates of all aggregation queries. For that reason predicates are considered as sets, covering a specific selection space in the complete value space of attributes. The sets of two predicates selection spaces can be distinct, they can overlap or they can be subsets of each other (e.g., $X < 5$ and $X < 10$). A decomposition describes the minimal number of subsets that cover the whole selection space defined by all predicates.

Let a list of predicates be defined as $P_{list} = \{P_1, .., P_n\}$. Further, let a function $decomp$ (see Algorithm 5.1) compute a decomposition where $decomp(P_{list}) = \{C_1, .., C_m\}$ contains the minimal list of subsets that cover the full range of predicates such that $(C_1 \cap .. \cap C_m) = \emptyset$ and $(C_1 \cup .. \cup C_m) = (P_1 \cup .. \cup P_n)$.

Algorithm 5.1: Decompose predicates $decomp(P_{list})$

```
1  init: P_list = arg;C_set = ∅;
2  for P ∈ P_list do
3      C'_set = C_set;
4      for C ∈ C_set do
5          if (P ∩ C) ≠ ∅ then
6              C'_set = C'_set \ {C};
7              C'_set = C'_set ∪ {(C \ P), (C ∩ P)};
8              P = P \ C;
9      C_set = C'_set ∪ {P};
10 return C_set
```

Now, we use the function $decomp$ to decompose each attribute individually. Given that there are attributes A, B, etc.. and n predicates, we compute $decomp(P_{list}(A))$ such that the decomposition is computed with regard to the share of A in P, i.e., $P_{list}(A) = P_1(A), .., P_n(A))$. We obtain a decomposition of all predicates with regard to A. Each decomposed condition receives a tag stemming from the attribute name and its index in the list (e.g., $decomp(P_{list}(A)) = \{A_1, A_2, ...\}$). This tag identifies the decomposition element and is later used to build the decomposed pre-aggregate. Further, we compute $decomp(P_{list}(B))$ and the decomposition of every contained attribute such that we obtain a list of decomposed conditions that can be used to represent every predicate.

Step 2 – In the second step, the predicates are substituted such that every predicate is described by a number of tags, as defined before. For that reason, we iterate over the predicates in P_{list}, and recursively substitute the conditions of a predicate with the combination of tags that is required to fully cover the predicates selection space. The substituted list is denoted as P_{tag}.

$$(\forall P \in P_{list}) \; : \; P_{tag} = \bigvee_{\{C \in C_{set} | P \cap C \neq \emptyset\}} C \text{ with } C_{set} = decomp(P_{list}) \tag{5.1.5}$$

Step 3 – In the last step, we iterate over the predicates in P_{tag} and use the tags of our decomposed elements to build a new list P_{dec} of (decomposed) keys. These (decomposed) keys can be combined with the defined aggregation key. They subdivide the existing aggregation keys into the decomposed aggregation keys which can be incrementally updated, easily. The following formula is used to build P_{dec}:

$$(\forall P \in P_{tag}) \; : \; P_{dec} = P_{dec} \; \cup \; eval(P) \tag{5.1.6}$$

Algorithm 5.2 is used in the equation. It combines the tags provided in a predicate $P \in P_{tag}$ recursively and returns a list of keys. It treats every conjunction as a concatenation and every disjunction as a numeration of tags.

Algorithm 5.2: Evaluate decomposed keys $eval(P)$

```
1 init: P = arg;
2 if P = P_A ∧ P_B then
3     return eval(P_A) × eval(P_B);
4 if P = P_A ∨ P_B then
5     return {eval(P_A), eval(P_B)};
6 if P = tag then
7     return {P};
```

As a result of Equation 5.1.6, P_{dec} contains the set of (decomposed) predicate keys. Considering that all (similar) aggregation operators are grouped by attribute X, the new aggregation key can be built as follows. The grouping attribute is combined with the predicate keys as (X, P_{dec}) for the pre-aggregation, as well as the final aggregation. As a result, we can use a single pre-aggregation and a single aggregation table to compute the result of many different aggregation views.

Example 5.1.4*: Let a base table be defined as* $R = (\bar{K}, X, Y, Z)$ *. Let a set of three equal aggregation views* V_1, V_2, V_3 *with predicates* P_1*,* P_2 *and* P_3 *be defined as:*

select X*,* $sum(Y)$ *from R..*
.. *where* $P_1 = ((Y < 5)$ *and* $(Z$ *between* 0.1 *and* $0.3))$
.. *where* $P_2 = ((Y < 10)$ *or* $(Z$ *between* 0.2 *and* $0.4))$
.. *where* $P_3 = (Y < 15)$
group by X

In Step 1, we create the selection ranges for the contained attributes Y *and* Z*. We determine the decomposition function as* $decomp(P_{list}(Y))$=$\{(Y<5), (5 \leq Y<10), (10 \leq Y<15)\}$ *for attribute* Y*; then, we determine decomposition* $decomp(P_{list}(Z))$=$\{(0.1 \leq Z<0.2), (0.2 \leq Z \leq 0.3), (0.3<Z \leq 0.4)\}$ *of attribute* Z*. Thus, all overlap has been removed from the predicate ranges. Likewise, the decomposed elements are tagged as* $\{Y_1, Y_2, Y_3\}$ *and* $\{Z_1, Z_2, Z_3\}$*.*

In Step 2, we iterate through the predicates and substitute their conditions with the tags of the decomposed elements which leads to $P_{tag} = \{P_1 = (Y_1 \wedge (Z_1 \vee Z_2)), P_2 = ((Y_1 \vee Y_2) \vee (Z_2 \vee Z_3)), P_3 = (Y_1 \vee Y_2 \vee Y_3)\}$*.* P_3 *is substituted by* Y_1*,* Y_2 *and* Y_3*, as it covers the complete range of the other predicates of* Y*.*

In Step 3, we derive the decomposed predicates from P_{tag} *as* $P_{dec} = \{Y_1, Y_2, Y_3, Z_2, Z_3, (Y_1, Z_1), (Y_1, Z_2)\}$*. When a value falls into the range of the decomposed condition (represented by the key in the list), the corresponding aggregate is updated. For example,* $put(R(k, x, 4, 0.1))$ *updates the aggregates of keys* $\{Y_1, (Y_1, Z_1)\}$*. Notably, an update always updates only a single tag, but this tag can be bound by multiple different conjunctions. To build the final (pre-)aggregation of a view (e.g.,* V_3*), we have to evaluate the tags and use the pre-aggregated values of the keys defined previously (e.g.,* Y_1*,*Y_2 *and* Y_3 *for* P_3*).*

Now, we write the incremental equations of the aggregation views. Let the set of aggregations operations be defined over a universal view function as $(\forall i \in \{1, .., n\}) : \gamma_{X,f(Y)}(\sigma_{P_i}(View(R_{set})))$, with each view definition defining a different selection predicate. Let the result of $View(R_{set})$ be computed in a maintenance plan with connecting view I. Then, optimization can be performed as follows:

$$\begin{aligned} &I_{Pre} = I_{Pre} \uplus \gamma_{(X,P_{dec}),f(Y)}(\Delta I) \\ &I_{Agg} = I_{Agg} \uplus \Delta I_{Pre} \\ &(\forall i \in \{1, .., n\}) : V_i = V_i \uplus \gamma_{(P_{dec},P_i)\rightarrow X,f(Y)}(\Delta I_{Agg}) \end{aligned} \tag{5.1.7}$$

VMS computes both I_{Pre} and I_{Agg} as a combined table using the composite key (X, P_{dec}) as the grouping key. Notably, an update u cannot be directly applied to I_{Pre} as we cannot use attribute X to address the records in I_{Pre} directly. VMS uses the values of the attributes (that are contained in the predicates) to determine the keys ($\in P_{dec}$) of the values to be updated in I_{Pre}. The update of I_{Pre} can, then, be directly applied to update global aggregation values in I_{Agg}. After this step, VMS recomposes the aggregation values in I_{Agg} by reconstructing the grouping key X using an on-the-fly regroup function taking P_{dec} and P_i as parameters.

This technique results in rapid updating of aggregates (which is particularly favorable for incremental updates). Given that predicate ranges heavily overlap (and only disjunctions are used), only a single aggregate has to be updated to update all views. However, the last step (i.e., Equation 5.1.7), which recomposes the aggregation, is the most expensive and should be delayed (using I_{Pre}) as much as possible. Specifically, when a predicate structure leads to many (decomposed) keys in P_{dec}, the system has to iterate over all of them to build the final aggregate.

Decomposing Pre-aggregation views is particularly useful, if the predicates of the involved aggregation queries intersect each other heavily. If not, it is still favorable to derive aggregations from a single Pre-aggregation view to reduce the number of tables.

5.1.3 Multi-join

As described in Chapter 4.4, VMS can incrementally build all kinds of join types using the same Reverse-join view. VMS also exploits this fact when maintaining n view definitions of the same base tables that are using different join types. In addition, like done in the field of query optimization, Reverse-join views can also be modularized and used to share parts of join relations.

Let n join views be defined as follows: $(\forall i \in \{1, .., n\})$: $Join_i(GV_i)$ with $GV_i \subseteq \{(View_1(R_{set}), .., View_m(R_{set}))\}$ being a subset of generalized view functions. Let $\{I_1,..,I_m\}$ be the connecting views of the generalized view functions $\{View_1, .., View_m\}$. Let these intermediate views be always joined over the same attributes. Further let the Function $Join_i$ join the generalized view functions using equi-, semi- or outer-joins in arbitrary order. Then we build a maximum of $(m-1)$ Reverse-join views as $\{I_{Rev(1)}, .., I_{Rev(m-1)}\}$ which serve to update all materialized views $(\forall i \in \{1, .., n\})$: $V_i = Join_i(GV_i)$.

Example 5.1.5*: Figure 5.1.3 shows the maintenance plan of multiple combined join views. Let two base tables be defines as* $R_1 = (\bar{K}, X, Y)$, $R_2 = (\bar{L}, X, Z)$. *Further, two join views are defined as* $V_1 = R_1 \bowtie R_2$ *and* $V_2 = R_1 \ltimes R_2$. *Let three updates be applied to the base tables as:* $u_1 = put(R_1(k_1, x, 5))$, $u_2 = put(R_1(k_2, x, 10))$ *and* $u_3 = put(R_2(l_1, x, 7))$. *By applying the first two updates, VMS will build the Reverse-join view as* $I_{Rev(1)} = \{((x, K = k_1, Y = 5), ((x, K = k_2, Y = 10)\}$. *For Update 3, the system queries* $get(I_{Rev(1)}(x, K?))$ *once and uses the result to incrementally build the natural join* $\{(k_1, l_1, x, 5, 7), (k_2, l_1, x, 10, 7)\}$, *as well as the semi join* $\{(k_1, x, 5), (k_2, x, 10)\}$.

Now, given that a third base table is defined as $R_3 = (\bar{M}, Y)$. *Also, a third join view is defined as* $V_3 = (R_1 \bowtie R_2) \bowtie R_3$ *(see Figure 5.1.3), building on top of the already maintained Reverse-join view* $I_{Rev(1)}$. *Then the system reuses the Reverse-join view that has been built for the first two join views. It adds a second Reverse-join view deriving it from the first one as* $I_{Rev(2)} = I_{Rev(1)} \bowtie R_3$. *In general, when computing join views, VMS uses a cost model to optimize for the best combination of Reverse-join views.*

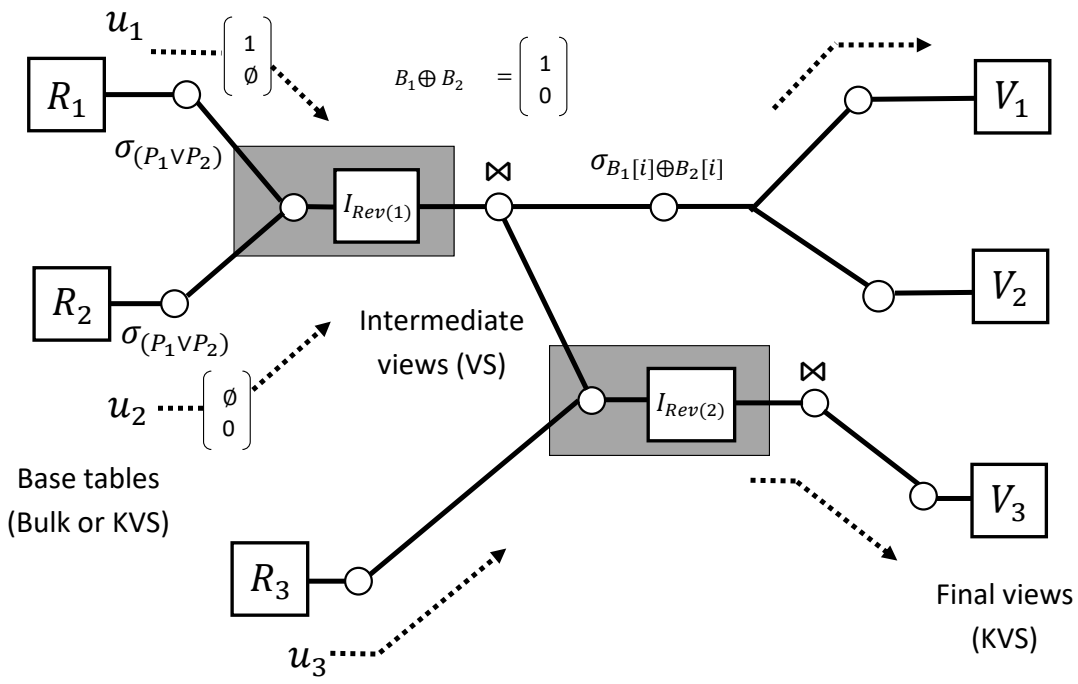


Figure 5.1.3: Join merge (maintenance plan)

Pre-evaluating predicates – As in the aggregation case, joins in applications are oftentimes executed using different selection ranges. In order to not recompute join relations, it is vital to unite the input of the join table into combined intermediate selection views. Building a single join on top of all selection predicates is significantly more efficient than building a separate join for each query. Again, we make use of bit vectors to reduce the number of predicates evaluated and to allow for rapid join construction. We insert a selection operator directly behind the base tables to evaluate predicates instantly and write the results into the bit vector. As the predicate may be defined over attributes from multiple different join tables, the outcome may also evaluate to $\emptyset$. In this case, we write a null value into the bit vector. If the bit vector completely evaluates to zero (i.e., $\mathcal{B} = 0$), we can drop the update at this early point.

Let n join views be defined as follows: $(\forall i \in \{1, .., n\})$: $\sigma_{P_i}(View_A(R_{set}) \bowtie View_B(R_{set}))$, with each view definition defining a different selection predicate. Let I_A and I_B be the connecting intermediate views of both functions. We capture all predicates in a list $P_{list} = \{P_1, .., P_n\}$ and we create two selection views $I_{Sel(1)}$ and $I_{Sel(2)}$ that evaluate all predicates with regard to the updates of I_A and I_B and store the results into bit vectors $\mathcal{B}_A$ and $\mathcal{B}_B$.

$$I_{Sel(1)} = I_{Sel(1)} \uplus \sigma_{\bigvee_{p \in P_{list}}}(\Delta I_A),\ I_{Sel(2)} = I_{Sel(2)} \uplus \sigma_{\bigvee_{p \in P_{list}}}(\Delta I_B) \tag{5.1.8}$$

When building the join using I_{Rev}, VMS simultaneously merges both bit vectors via $\oplus$ operator (see Section 5.1.1). In this way, VMS can determine the predicate status of the join table updates without reevaluating the selection predicates. When computing $\mathcal{B} = \mathcal{B}_A \oplus \mathcal{B}_B$, the result determines whether an update is kept or dropped at position i. In the case that $\mathcal{B}[i] = \emptyset$ (i.e., on both table sides, the predicate could not be evaluated), the system reevaluates the predicate on top of the computed join rows.

$$(\forall i \in \{1, .., n\}) : V_i = \begin{cases} V_i \uplus \sigma_{\mathcal{B}[i]}(I_{Rev}) \ if (\mathcal{B}[i] \neq \emptyset) \\ V_i \uplus \sigma_{P_i}(I_{Rev}) \ else \end{cases} \tag{5.1.9}$$

Example 5.1.6*: In Figure 5.1.3, two base tables R_1 and R_2 are depicted (defined as before). Let the view definitions of V_1 and V_2 be rewritten as:*

$$V_1 = select\ ^*\ from\ R_1 \bowtie R_2\ where\ (R_1.X > 5)\ or\ (R_2.Z = 5)$$
$$V_2 = select\ ^*\ from\ R_1 \bowtie R_2\ where\ (R_1.X = 10)\ and\ (R_2.Z > 15)$$

Following the view definition of V_1 and V_2 both base table are connected to selection views $I_{Sel(1)}$ and $I_{Sel(2)}$ with predicate lists $P_{list(1)} = \{(R_1.X > 5), (R_1.X = 10)\}$ and $P_{list(2)} = \{(R_2.Z = 5), (R_2.Z > 15)\}$ Now, given that there are two updates $u_1 = put(R_1(k, x, 10))$ and $u_2 = put(R_2(l, x, 7))$, the bit vector of u_1 evaluates to $\mathcal{B}_1 = \{1, \emptyset\}$, and that of u_2 evaluates to $\mathcal{B}_2 = \{\emptyset, 0\}$. When combining both vectors during join construction, we build $\mathcal{B}_1 \oplus \mathcal{B}_2 = (1, 0)$. The result indicates that V_1 is updated with the join record, whereas V_2 is not.

5.1.4 Nested constructions

Thus far, we have rewritten queries of similar SPJA structure to capture common aspects and avoid unnecessary overhead. These rules can be applied recursively to rewrite any combination of similar SPJA pattern or nested constructs. Rewriting can be performed from the bottom of a query to the top until the structure diverges. However, the system always strives for merging as many intermediate views as possible such that the maintenance plan is split at the very last opportunity; thereby, only the final materialization step is processed, separately. Let a number of nested analytical queries be defined as:

$$(\forall i \in \{1,..,n\}) : \text{SELECT} \quad * \quad \text{FROM} \quad R \\ \text{WHERE} \quad X > (I_i = \{\text{SELECT} \quad sum(Y) \quad \text{FROM} \quad R \quad \text{WHERE} \quad P_i\})$$

Then, we translate the inner view definition to

$$\begin{aligned} I_{Sel(1)} &= I_{Sel(1)} \uplus \sigma_{P_i}(\Delta R) \\ I_{Pre} &= I_{Pre} \uplus \gamma_{P_{dec},sum(Y)}(\Delta I_{Sel(1)}) \\ I_{Agg} &= I_{Agg} \uplus \Delta I_{Pre} \\ I_i &= \gamma_{(P_{dec},P_i)\rightarrow\{\},sum(Y)}(\Delta I_{Agg}) \end{aligned} \tag{5.1.10}$$

whereas the outer query is derived from connecting view I_i. Modification of the inner query likewise modifies the predicate for the outer query such that the outer query inherits the combined processing. A predicate list $P_{out} = \{(X > I_1),..,(X > I_n)\}$ is constructed to capture the different predicates of the outer query. Note that a sub query within a condition represents a blocking operation. We rewrite the outer query as:

$$\begin{aligned} I_{Sel(2)} &= I_{Sel(2)} \uplus \sigma_{\bigvee\limits_{P \in P_{out}}}(\Delta R) \\ V_i &= V_i \uplus \sigma_{B[i]}(\Delta I_{Sel(2)}) \end{aligned} \tag{5.1.11}$$

We rewrite and split at the last possible point (V_i). As such, we need to keep only a single intermediate table per operation and can amortize its (storage) cost by computing thousands of different view tables on top. Ideally, all intermediate materialization is done in-memory, thereby substantially reducing the overall execution time.

5.2 Cost model

The full cost of the maintenance plan is defined as the sum of the cost of each operation. Let $M = (R_{set}, O, V, E)$ be defined as the maintenance plan of view definition $View(R_{set})$ that consists of operation vertices and edges. Further, let $\Delta = \{\Delta R_1, \Delta R_2, ..\}$ be defined as the update sets of all involved base relations. We sum the total cost as:

$$Cost(M, \Delta) = \sum_{v \in V} Cost(v, \Delta) \tag{5.2.1}$$

The cost of a selection operation (when not performed on-the-fly) is defined over a materialized view in VS for which access results in cost c_{get} and modification c_{up} (with $up \in \{put, delete\}$). The cost model can be further refined to distinguish between put and delete updates.

$$Cost(\sigma_p, \Delta I) = |\Delta I| * c_{get} + |\sigma_p(\Delta I)| * c_{up} \tag{5.2.2}$$

Aggregation views are defined by a pre-aggregation I_{Pre} and a final aggregation I_{Agg}. The cost of redistributing the pre-aggregated records n times (for n VMs) is incurred, and the cost of sending a single update is provided through c_{send}. The stronger the aggregation is, the lower the sending cost.

$$Cost(\gamma_{X,f(Y)}, \Delta I) = (\underbrace{|\Delta I|}_{I_{Pre}} + \underbrace{n * |X|}_{I_{Agg}}) * (c_{get} + c_{up}) + \underbrace{(n * |X| * c_{send})}_{\text{redist}} \tag{5.2.3}$$

Likewise, we determine the cost given by any join pair to compute the cost of any multitable join. To find the best combination of join tables, and the best maintenance plan for multiple join expression we implement existing greedy heuristics (such as dpccp [66]) to reduce the amount of join possiblities. On the basis of these models, the cost of Strategy 1 is determined as:

$$Cost(I_A \bowtie I_B, \Delta I_A, \Delta I_B) = (|\Delta I_A| + |\Delta I_B|) * (\underbrace{c_{send}}_{\text{redist}} + \underbrace{c_{up} + c_{mget}}_{I_{Rev}}) \tag{5.2.4}$$

Strategy 1 is "fully partitioned" and includes a full round of redistribution. We use cost c_{mget} to describe a multi-get to the view (using wild-card $(x, K?)$). While the operation

retrieves multiple join records it cannot be described as $m * c_{get}$ because access is realized over a (single) composite key, loading all join partners from a list (nested hash-map) in one pass. Strategy 2 is "partially partitioned" and runs without redistribution. Because updates to the fully loaded table are being duplicated n times, the cost of Strategy 2 is:

$$Cost(I_A \bowtie I_B, \Delta I_A, \Delta I_B) = (n * |\Delta I_A| + |\Delta I_B|) * (c_{up} + c_{mget}) \tag{5.2.5}$$

The cost of projection is dependent on whether the operation is the last one in the maintenance plan: if it is, no outgoing edges exist, and the cost is determined by KVS updates; else, the view is intermediate, and the cost is determined by VS updates.

$$Cost(\pi, \Delta I) = \begin{cases} |\Delta I| * c_{up}(VS) & \text{if } out(M, \pi) \neq \emptyset \\ |\Delta I| * c_{up}(KVS) & \text{else} \end{cases} \tag{5.2.6}$$

Finally, we determine the benefit (or drawback) of multi-view optimization, where M_{set} defines the set of maintenance plans derived from multiple view definitions and M' defines the new merged maintenance plan that has been obtained by multi-view optimization:

$$\sum_{M \in M_{set}} Cost(\Delta, M) - Cost(\Delta, M') \tag{5.2.7}$$

5.3 Evaluation

In this section, we evaluate the performance of VMS when materializing a large number of views in parallel. For the multi-view evaluation, we use a reduced scale factor of 1 (1 GB, ~8.6M records) because the number of defined views also multiplies the number of updates sent through VMS. Figures 5.3.1 - 5.3.6 materialize views for evaluating TPC-H query templates (i.e., Q_1, Q_3, Q_4, Q_6, Q_{10} and Q_{14}).

For each query template, we use the TPC-H qgen tool to generate a defined number of query instances (i.e., 10, 100, 200, 500, 1k, 5k and 10k). As specified by TPC-H, each query instance varies in terms of the key parameters of the query template (e.g., selection ranges), and the result of every query instance is stored in a separate view table. The corresponding number of view tables are created in KVS before the experiment (for the 10k case, table creation alone takes > 500 seconds). Then, we setup each experiment for three strategies: (1) standard, (2) single batch and (3) double batch.

Standard: Describes the naive strategy of simply building and executing a single maintenance plan for each view.

Single batch: Describes the strategy of merging all maintenance plans into a single plan, and sharing (and distinguishing) all selections, aggregations and joins, until the very last step, where each view is materialized separately (see Section 5.1). This strategy is called single batch because in the three-round process of aggregation (see Figure 5.1.2), only the first round (i.e., the pre-aggregation I_{Pre}) is a batched operation.

Double batch: As for single batch, all maintenance plans are merged into a single plan. The strategy is called double batch because in the process of aggregation (see Figure 5.1.2) Rounds 2 and 3 are both batched. In Round 3, pre-aggregates are collected and stored (using the composite key (X, P_{dec})); only then, is the final result released.

Further, for the single and double batch strategies, VMS merges join operations using the bit vector mechanism described in Section 5.1.3 (see Figure 5.1.3). VMS resolves all selection predicates prior to the join in a combined effort and stores the results in the bit vector. The bit vector is then passed along with the update and merged via $\oplus$ operator

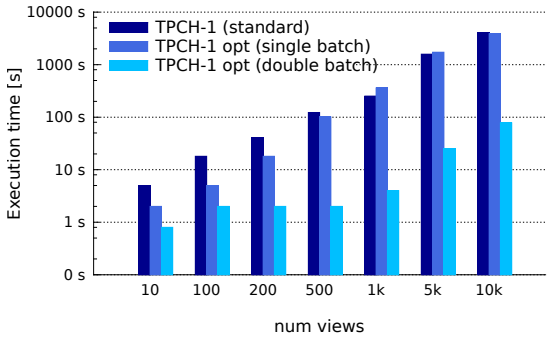


Figure 5.3.1: Q_1 multi-view

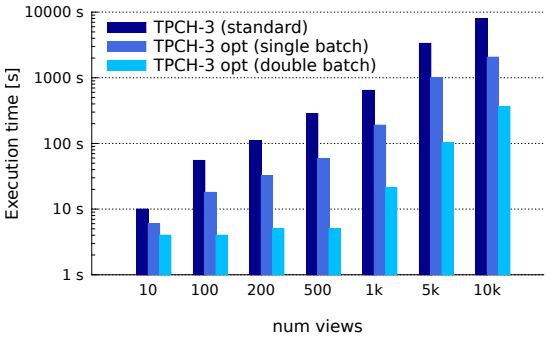


Figure 5.3.2: Q_3 multi-view

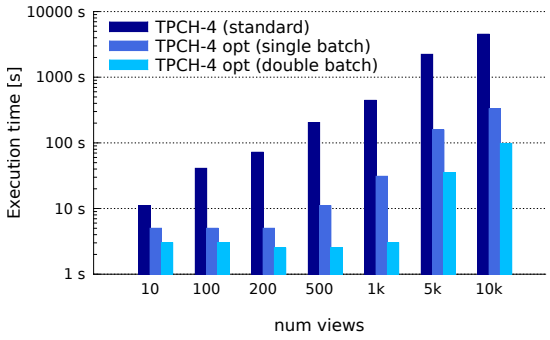


Figure 5.3.3: Q_4 multi-view

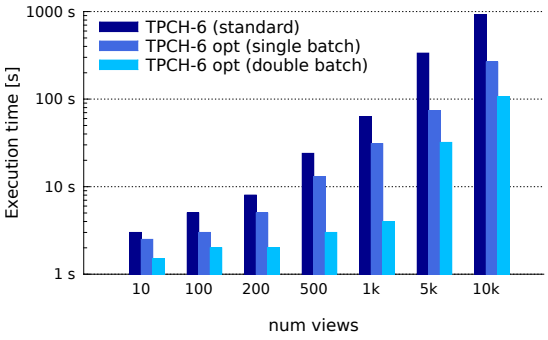


Figure 5.3.4: Q_6 multi-view

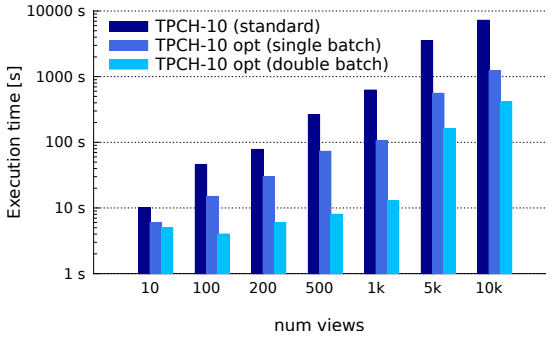


Figure 5.3.5: Q_{10} multi-view

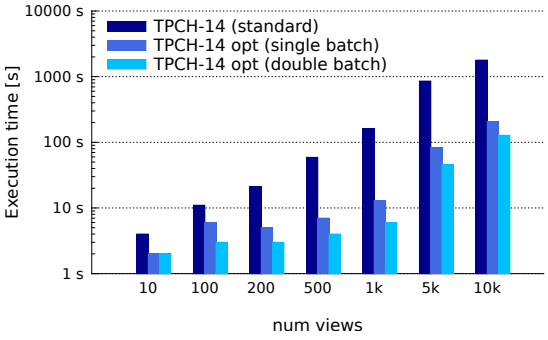


Figure 5.3.6: Q_{14} multi-view

during join processing. In this way, VMS keeps the number of internal updates to one per base table update, but the bit vector can still grow to a large sizes (1250 bytes without and 2083 with null values at $10k$ instances).

Figure 5.3.4 shows the results of the three strategies for query template Q_6. Note that the execution time (on the y-axis) is presented on a log scale since the results differ substantially. The results of computing 10 views are comparable, single batch achieves a $1.2x$ speed up an double batch achieves a $2x$ speed up. The greater the number of views computed in parallel is, the larger the gap becomes. When processing 500 views in parallel, single batch achieves $1.8x$ and double batch already $8x$.

VMS maxes out when materializing $10k$ views in parallel. With the standard strategy, the execution time is $912s$; with single batch optimization execution time is $265s$ and with double batch optimization, it is $107s$. For both queries, VMS computes the $lineitem$ table at scale factor $1x$, i.e., $6M$ updates over $10k$ views. Because of write amplification, the standard strategy (i.e., a non-optimized system) processes the equivalent of $6M \times 10k = 60B$ update operations. The standard strategy only provides somewhat reasonable execution time for Q_6 because of the high selectivity of records (over 95% of records are dropped).

Comparing to query Q_1 which also operates over the $lineitem$ table, we observe the following: standard strategy executes in $4081s$, single batch executes in $3895s$ relatively close to the standard strategy. Then, the execution time drops significantly to $78s$ for the double batch strategy

For queries Q_3 and Q_{10}, the speed up for double batch is also quite significant; both view definitions include aggregations and joins which can be processed as combined operations. Q_{10} is a four-table join with consecutive aggregation. Using the standard strategy execution time is $7210s$, for single batch it is $1217s$ and for double batch, we can materialize the results within 411 seconds. Q_3 is a three-table join with consecutive aggregation. For Q_3, results over $10k$ views can be materialized in $7843s$ for the standard strategy, in 2075 for single batch and in 362 seconds for double batch.

In this case, the speed up of single batch compared to standard computation is $5.6x$ and that of double batch is $17.5x$. Notably, for Q_3 and Q_{10} VMS processes updates of join tables

$lineitem$,$orders$ and $customer$, simultaneously. At scale factor $1x$ this means a total of $7.65M$ base records which are processed over $10k$ views. Again, the standard strategy (i.e., a non-optimized system) processes the equivalent of $7.65M \times 10k = 76.5B$ update operations.

Only the described methods of multi-view optimization in combination with the double batch strategy allow very reasonable execution times of 100 – 500 seconds; only this way the write amplification can be successfully prevented. The maintenance plan for the $10k$ view definitions is split at the very last point and, in addition, all update operations are executed in a single run. All aggregations are evaluated combined; splitting the updates and materializing the final views at the end of the maintenance plan only requires writes; get operations can be avoided, completely.

Chapter 6

Consistent hybrid view maintenance

Increasing volume and velocity have been a driving factor for Big Data analytics. While an ever growing data stock has to be scanned and evaluated (batch processing), at the same time, base data is updated (online/incremental processing), rendering the already computed results obsolete. Following that large-scale processing frameworks mainly employ two different processing styles to compute and track results upon large tables: batching computations are executed with bulk processing frameworks (e.g., MapReduce [67]), incremental updates are often times realized with the help of event or stream processing systems (Apache Storm [68], Apache Flink [21]).

The need to combine the advantages of both processing styles (i.e., high throughput, and low latency) into a single solution gave rise to a number of hybrid processing frameworks and paradigms. Hybrid processing strategies have been discussed as extensions to MapReduce, as combinations of different architectures and frameworks (batch processing and streaming) or domain specific languages [17, 18, 19, 20, 21]. Analogous to the development of Big data infrastructures, the research on (materialized) views has also centered around two main strategies: batch and incremental maintenance.

Batching strategies [65] in view maintenance related to concepts that read and process complete data sets as input to build up (or update) a materialized view in one big run. Batch strategies perform well in analysing large tables, but they lack the ability to track

small changes and provide efficient real-time updates of results. The staleness of a view between one view refresh and the next might not be sufficient for clients.

Incremental strategies [14, 16, 56] relate to concepts that receive a stream of update operations on a data set and incrementally apply updates to a materialized view. Incremental strategies excel at their ability to track modifications in real-time; they are inherently efficient and provide low latency to the client of the view. Incremental strategies are particularly useful for large tables that are updated infrequently. On the downside incremental strategies cannot be used over existing data (as all the operations needed to be replayed) and they are susceptible to non-uniform distributions (e.g., Zipf).

Hybrid strategies in view maintenance try to overcome the aforementioned weaknesses by combining aspects of batching and incremental strategies. But unlike in research on large-scale infrastructures (where hybrid approaches are well represented), the contribution to hybrid strategies in the field of view maintenance has been relatively small. To this point, only solutions exist that discuss the use of (micro-)batching for incremental strategies [16, 57].

However, a comprehensive study, researching the trade-offs of different hybrid approaches, and their coexistence with established strategies is still missing. As can be found, not a single (hybrid) strategy is sufficient to match all given scenarios. Multiple different incremental, batching, hybrid strategies can be used to adapt view maintenance to its context and, ultimately, to balance the triangle of throughput, cost and latency requirements (view staleness).

In this chapter, we make the following contributions:

1. We present a novel integrated concept for distributed KVS to support a variety of different incremental, hybrid and batching view maintenance strategies by only relying on a set of primitives (i.e., operation streams, scans, snapshots).
2. We provide a comprehensive study of different incremental, batching and hybrid view maintenance strategies. We provide a classification by their nature, a definition of their application, as well as different suggestions to their realizations (see Section 6.1, 6.2 and 6.3).

3. We propose a concept for hybrid maintenance in KVS that achieves strong consistency (also for mixed insert/update/delete workloads) on the one side and operates highly parallelized on the other side. Thereby, we propose a novel data structure (called MK tree) to efficiently synchronize records and operation streams at process level (Section 6.3).

4. We conduct an extensive experimental study using an implementation over VMS and HBase. We offer a comparison of all defined incremental, hybrid and maintenance strategies with regard to cost, staleness and performance and provide a recommendation of which strategy to use in which context (see Section 6.4).

In Figure 6.0.1, an overview of all strategies is depicted. On the very top of the figure a sequence of updates, applied to a base table by clients, can be found. The base table receives a (uniformly distributed) stream of update operations. The point at which a client inserts or updates a record into the base table is called *insertion point*. Formally, we describe the insertion sequence of updates into the base table as $\Delta\mathcal{R}$. Given that table is in state $\mathcal{R}_0$, in the beginning, applying $\Delta\mathcal{R}$ takes it to state $\mathcal{R}_f$ (see Chapter 4.2).

$$\begin{aligned} \mathcal{R}_0 =, \Delta\mathcal{R} &= \langle u_1, .., u_n \rangle \\ \mathcal{R}_f =& \mathcal{R}_0 + \Delta\mathcal{R} \end{aligned} \tag{6.0.1}$$

Below the base table in Figure 6.0.1, the different view maintenance strategies are depicted: two incremental, three batching and three hybrid strategies. In the figure, the lines on top of the boxes illustrate the time span from retrieval of a record or update to its execution. Thereby, the start point of the lines marks the *retrieval point*, the moment when a record or update is fetched from KVS. Retrieval, in our architecture is done through KVS observer or TL. To describe the retrieval order for VMS, we formalize an additional *retrieval round* (d_{ret}, K) which can be triggered through a snapshot, a scan or an incremental update stream.

$$\mathcal{D}_{ret} = d_{ret}(\Delta\mathcal{R}) \tag{6.0.2}$$

The end points of the lines mark the beginning of the maintenance process. The stronger the slope of the lines the higher the delay of maintenance – and ultimately the staleness of the view table. We use red lines to indicate that an update is propagated and maintained

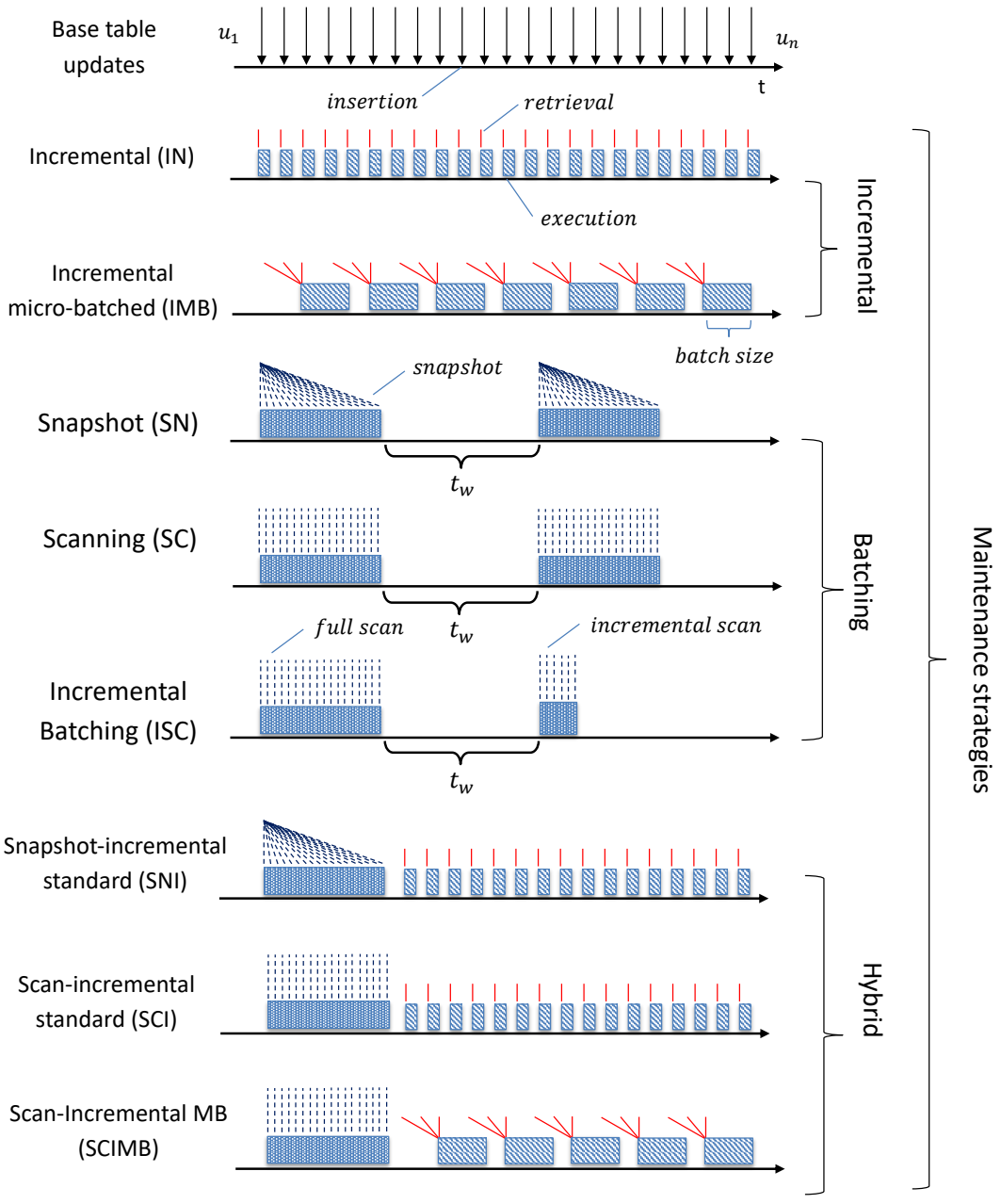


Figure 6.0.1: Strategies overview

by the system. We use blue dashed lines to indicate that a record has been fetched from the database (e.g., through a scan or a snapshot) and is now processed.

Finally, the blue boxes below the lines represent the *execution point*, where updates are applied to the view table. The length of the boxes indicate the length of the actual maintenance process; which is shorter for incremental and longer for batching strategies. The execution order of the first round, is then formalized as $\mathcal{D}_0 = d_0(\Delta\mathcal{R})$ as described in our model before (see Chapter 4.3).

6.1 Incremental strategies

In this section, we describe and formalize the basic incremental strategies that our system leverages to maintain a set of view tables stored in a KVS.

6.1.1 Basic incremental

For pure incremental maintenance (IN), what is being maintained, is a number of base update sub-streams, each on emitted by one of the KNs. Despite of that, the sub-streams produce a global execution sequence (see Equation 6.1.1) which is important for subsequent view maintenance. For incremental maintenance, insertion, retrieval and execution points can be found close together (see Figure 6.0.1), indicating that there is little delay between insertion and execution (given no updates are queued); the update is propagated in the moment the client inserts it. Retrieval and view updates follow an instance later to update the materialized view. Incremental maintenance during the retrieval round is formalized as follows:

$$\mathcal{D}_{inc} = d_{inc}(\Delta\mathcal{R}) \tag{6.1.1}$$

In round (d_{inc}, K), VMS retrieves the updates which are propagated according to sequence $\mathcal{D}_{inc}$. The distribution key K of the round is also the row-key of the base table and the distribution key of the first distribution round $\mathcal{D}_0$.

Since only updates of $\Delta\mathcal{R}$ are propagated during incremental maintenance, the view will not be complete at the end of the process. If records were existing in the base table before, which have not been updated, their state is not reflected in the view.

$$(\exists r \in R) \longrightarrow (r.K \notin \Delta\mathcal{R}.K) \tag{6.1.2}$$

Still, we can assert that retrieval at a VM is sequential and based on a specific partition. Multiple updates to the same row-key (inducing a record timeline) are always forwarded to the same VM. Thus, record timeline requirements are not violated during the retrieval round.

6.1.2 Incremental micro-batched

Incremental micro-batching (IMB) is an optimization of the standard incremental strategy. This means, streams of base updates are consumed and sets of base updates (number is determined by the batch size) are combined and maintained together. There are known approaches [16, 69] to compute incremental maintenance in batches of different sizes. This way, sets of update operations are processed together, the result is only materialized once. Formalization of this strategy can be expressed as follows:

$$\begin{aligned} \Delta\mathcal{R}(i,j) &= \langle u_i, .., u_j \rangle \\ \mathcal{D}_{inc}(x) &= d_{inc}(\Delta\mathcal{R}(([x-1]*b+1),(x*b))) \\ \text{IMB} &= \langle \mathcal{D}_{inc}(1), .., \mathcal{D}_{inc}(n) \rangle \end{aligned} \tag{6.1.3}$$

Input consists of delta stream of updates $\Delta\mathcal{R}$ as before. Let $\Delta\mathcal{R}(i,j) \subseteq \Delta\mathcal{R}$ be the sub-sequence of of updates from update u_i to u_j (with $i < j$). Then, depending on batch size b, the input stream is cut into chunks of sub-sequences processing b updates per sub-sequence. After each of these sub-sequences, results are synchronized with the materialized view table. Micro-batching is independent of time intervals, i.e., VMS fetches the amount of b updates from the queues, if the queue size, in total, is smaller than b, VMS simply fetches the entire all updates.

Micro-batching allows for several optimizations. We make use of two different methods to apply micro-batching to a stream of update operations: (1) combine update operations that are defined over the same row-key (i.e., condense a row-keys timeline); (2) pre-process results at different stages of a maintenance plan (such as local pre-aggregation before a global aggregation step) and merge them later on.

Combine operations – Combining operations is a strategy to protect incremental maintenance against highly skewed key distributions. If many update operations to a single key (range) are issued, a lot of incremental update steps have to be performed by the same VM to update the same row-key over and over again. To overcome the problem, given a set of batched updates, VMS condenses the timeline of row-keys into single updates.

Example 6.1.1*: We consider a base table* $R = (\bar{K}, X, Y)$ *and a sequence of four updates* $u_1 = put(k_1, x_1, 5)$, $u_2 = put(k_2, x_2, 7)$, $u_3 = del(k_1)$ *and* $u_4 = put(k_2, x_2, 15)$*. Given that all operations are processed in a single batch, the system is able to condense the update stream from* $\mathcal{D}_{inc}(1) = \langle u_1, u_2, u_3, u_4 \rangle$ *to* $\mathcal{D}_{inc}(1) = \langle u_3, u_4 \rangle$*. Thereby,* u_3 *overwrites* u_1 *and* u_4 *overwrites* u_2 *because row-keys are equal and only the respective last update is relevant to the view (we disregard partial row-updates, here).*

Pre-process results – A maintenance system can also benefit from micro-batching, if the evaluated view operator is capable of pre-processing steps. This way, during micro-batching we are able to reduce the number of transmitted updates, significantly.

Example 6.1.2*: Consider a base table* $R = (\bar{K}, X, Y)$ *as before and a view table defined as* $\gamma_{X,SUM(Y)}(R)$*. Given that operations* u_1*-*u_4 *(as defined in the example before) are processed at a VM, the pre-aggregate can be already build using updates of* $\mathcal{D}_{inc}(1)$ *as* $I_{pre} = \{(x_2, 15)\}$ *condensing the four operations locally. The pre-aggregate is then, in another distribution round combined to form the global aggregation result.*

Pre-processing results is very beneficial for intermediate aggregation as well as join operators. Especially strong aggregations or joins with low join key cardinality provoke bottlenecks as there is a high number of input updates waiting to be processed on the same view records. Here, pre-processing (as optimization for micro-batching) reduces the intermediate update load significantly, leading to a higher throughput of the system.

In general, micro-batching can be used to adapt the freshness of views by varying the batch size. Smaller batch size means higher freshness but also higher materialization cost. Larger batch size means lower freshness but also lower materialization cost.

Basic incremental strategies are excellent for refreshing view tables on update operations as they keep the staleness of the refreshed views to a minimum. However, they always require a full history of update operations to compute the correct state of a view table. Foremost, this is the case when a base table is empty and the view tables are generated beforehand. In reality, views are oftentimes defined on top of existing data sets, where the history of update operations is either not known or too large to be recomputed. In that context a batch-oriented strategy is required.

6.2 Batching strategies

Naive batching strategies to materialize and update views include the creation of snapshots or the scan of a base table in defined intervals. In the following, we describe strategies using both methods as primitives.

6.2.1 Repeated snapshots

View maintenance via repeated snapshots (SN) may be executed in defined intervals (see Figure 6.0.1, t_w). After each interval, the view table is recreated and the view data is recomputed, entirely. A snapshot of the base table, taken previously, serves as a basis. The data set in the snapshot is, in contrast to the actual base table, not modified during view maintenance. Exactly like the base table, the snapshot is divided by key ranges. Each VM connects to one of the KNs to request a (local) snapshot of its key ranges.

A sloped blue dashed line (see Figure 6.0.1, SN), as it is drawn for the snapshot, indicates that retrieval and execution of base table records deviate. Creating a snapshot means all records are fetched at a single point in time t, still view maintenance takes some time. A snapshot represents a consistent state of a data set, drawn from a base table in KVS at a specific point in time. We assume that, having taken the snapshot at a point t, we are loading the respective latest record versions with regard to t from the base table. For a snapshot, we demand that our data set is immutable, record versions do not change.

$$\begin{aligned} \Delta\mathcal{R}(i) &= \langle u_1, .., u_i \rangle \\ \mathcal{R}_i &= \mathcal{R}_0 + \Delta\mathcal{R}(i) \\ \mathcal{D}_{snap}(t) &= d_{snap}(\mathcal{R}_i) \end{aligned} \tag{6.2.1}$$

Let $\Delta\mathcal{R}(i) \subseteq \Delta\mathcal{R}$ be the sub-sequence of updates that have been already applied to the base table until t. Then, VMS retrieves the records of the base table in state $\mathcal{R}_i$. The records are propagated and received in sequence $\mathcal{D}_{snap}(t)$. Running the snapshot strategy, VMS retrieves a snapshot, and computes the results; then, waits for interval $t_{i+1} = t_i + t_w$,

removes the view tables completely and repeats the cycle. The strategy is formalized as:

$$\text{SN} = \langle \mathcal{D}_{snap}(t_1), .., \mathcal{D}_{snap}(t_n) \rangle \tag{6.2.2}$$

While providing excellent consistency and conserving the state of the table separately (which facilitates the work of the maintenance system), snapshot strategies suffer from the overhead a snapshot produces and defer subsequent maintenance operations (as the snapshot has to be created and stored).

6.2.2 Repeated scans

Exactly like the snapshot strategy, the repeated scans (SC) works within an interval (see Figure 6.0.1, t_w), after which the view is deleted and recomputed. A scan works over the active records stored in a table; its computation can also be distributed and parallelized such that each VM requests a (local) scan from its associated KN to load all partitions stored there.

The straight blue dashed line (see Figure 6.0.1, SN) indicates that (like for incremental maintenance) retrieval and application of a record to the view table are close together. The main difference between a scan and a snapshot is the versions of their records. While a snapshot represents a closed consistent data set, during a scan the base table remains open for modifications by clients. Records that are fetched later in a scan can have updated versions as they might have undergone multiple changes in the meantime.

$$\begin{aligned} \mathcal{R}_{i..j} &= \{r | r \in \mathcal{R}_i \vee .. \vee r \in \mathcal{R}_j\} \\ \mathcal{D}_{scan}(t_s, t_e) &= d_{scan}(\mathcal{R}_{i..j}) \end{aligned} \tag{6.2.3}$$

The version of a scanned record either corresponds to a version updated before the scan or it corresponds to a version updated during the scan (with t_s and t_e being the start and end point of the scan). Let the start and end points of the scan t_s and t_e be associated with base table states $\mathcal{R}_i$ and $\mathcal{R}_j$ (with $i < j$). Then we conclude, the scanned record version can be either found in $\mathcal{R}_i$, $\mathcal{R}_j$ or in all states the base table accepted in between (i.e., computing sequence u_i to u_j). We express this condition as base table state $\mathcal{R}_{i..j}$. Running the scanning strategy, VMS retrieves a snapshot, and computes the results; then, waits

for interval t_w, removes the view tables and repeats the cycle.

$$\text{SC} = \langle \mathcal{D}_{scan}(t_1, t_2), .., \mathcal{D}_{scan}(t_{n-1}, t_n) \rangle \tag{6.2.4}$$

A distributed scan is easy to implement and (in comparison to a snapshot) holds no storage overhead; we are fetching table records directly from the base table. Still, scans are very problematic with regard to consistency. The version of a record that is scanned at an earlier point, can differ a lot from one that is scanned at a later point.

6.2.3 Incremental snapshots

Incremental snapshots (ISN) is a strategy optimizing the repeated snapshot strategy (SN). As such it avoids repeated snapshots of records that have not changed through the last and the current batching interval. ISN almost always provides reduced cost in comparison with repeated snapshots. Figure 6.2.2 shows an ISN strategy. Thereby, a full snapshot is followed by (many) incremental snapshots which update the view table.

While the ISN strategy might resemble an IMB strategy, the implementation and execution differs considerable. Snapshots work over record versions of the base table and not over update streams (like IMB). Further, the time interval of execution is configurable, whereas the IMB strategy purely works over batch sizes. We formalize the retrieval round as:

$$\begin{aligned} \mathcal{D}_{snap}(t) &= d_{snap}(\mathcal{R}_i) \\ \mathcal{R}_j \setminus \mathcal{R}_i &= \{r | r \in \mathcal{R}_j \neq r \in \mathcal{R}_i\} \\ \mathcal{D}_{snap}\Delta(t_1, t_2) &= d_{snap}(\mathcal{R}_j \setminus \mathcal{R}_i) \end{aligned} \tag{6.2.5}$$

A regular snapshot, which is used first in ISN, is defined as before (for SN) and records are retrieved in sequence $\mathcal{D}_{snap}(t)$. However, the following snapshot is an incremental snapshot; it only incorporates the records that have changed in time interval t_1 to t_2. Let t_1 and t_2, again be identified with base table states $\mathcal{R}_i$ and $\mathcal{R}_j$ (with $i < j$). Then, we define operator $\mathcal{R}_j \setminus \mathcal{R}_i$ to load only records of $\mathcal{R}_j$ that are different in $\mathcal{R}_i$. The incremental snapshot is retrieved in sequence $\mathcal{D}_{snap}\Delta(t_1, t_2)$. Concatenating a regular snapshot and an

arbitrary number of incremental snapshots the complete strategy can be formalized as:

$$\text{ISN} = \langle \mathcal{D}_{snap}(t_1), \mathcal{D}_{snap}\Delta(t_1, t_2), .., \mathcal{D}_{snap}\Delta(t_{n-1}, t_n) \tag{6.2.6}$$

To catch incremental updates that occur during the snapshots (and between them), we establish so-called *tracking phases*. During a tracking phase each VM enables its observer component such that it is notified about all keys that are updated in the associated KN. The VM stores the keys of the updated records into a MK tree.

MK tree – The MK tree is a b+-tree that keeps track of maintained keys during incremental maintenance (see Figure 6.2.1). Once an incremental update (put or delete) arrives, it is inserted into the tree. The tree can, then, tell whether a specific row-key has been already maintained or not.

The b+-tree is predestined for incremental batching. The VM can rapidly insert keys (of modified records) into the tree. Additionally, the b+-tree keeps the keys in a sorted order and allows for fast ordered access of all stored keys via the intermediate pointers that are interconnecting the leaf nodes. During a tracking phase the b+-tree gets filled with modified keys. Then, the VM hands it to an observer which performs a fast scan on the local partitions. However, a b+-tree still stores each row-key separately. When maintaining large tables spanning million row-keys and more, the data structure will occupy large portions of the memory. This is why we introduce the MK tree.

In KVS, row-keys are used to uniquely identify records. They are compared and sorted lexicographically (see HBase [5]). While this is a necessary property to build and manage key ranges and balance them over different nodes in the network, we use it to create MK tree.

Given a compare operator and step width (or a key space), a VM is able to identify if two keys are adjacent or not. If yes, it will merge the keys into a combined key range. This way, the MK tree, first increases in size, but later – when thousands of incremental updates are done – shrinks again. Having maintained the complete key range should finally lead to a tree only hosting a single element. If however, the step width (or the key space) of a base table is not known, we cannot determine the neighbors of a key. Then, we fall back and use the b+-tree like described above.

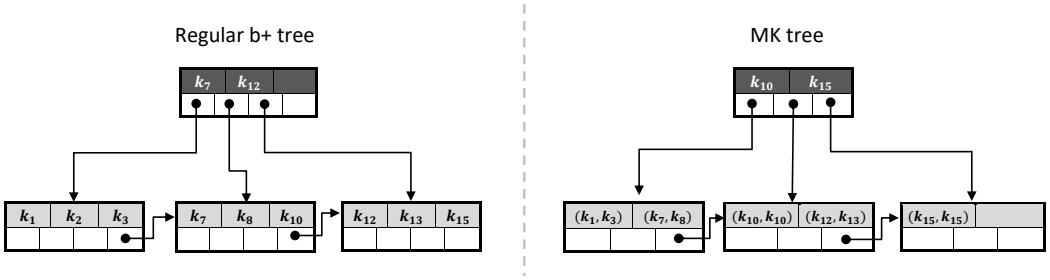


Figure 6.2.1: MK tree (d=2)

Figure 6.2.1 depicts an example of an MK tree. On the left side, a b+-tree is shown that has been built by a sequence of row-keys (emitted during the tracking phase). On the right side, the same row-keys are shown using the MK tree. We define the MK tree using a recursive structure:

$$
\begin{aligned}
MK_{tree} &= (\mathcal{N}, \mathcal{P}) \\
\mathcal{N} &= \{N_1, .., N_m\} \; with \; N_i = \begin{cases} (k) & if (h(N_i) < h_{max}) \\ (k_s, k_e) & else \end{cases} \\
\mathcal{P} &= \{MK_{tree(1)}, .., MK_{tree(n)}\}
\end{aligned}
\tag{6.2.7}
$$

An element within the tree structure consists of a list of search row-keys, an element in the leaf nodes consists of a list of key-ranges.

As explained, we use MK tree to store the record keys of base table updates. To fill the MK tree with row-keys of updated records, we establish so-called *tracking phases* (see Figure 6.2.2). As shown in the figure, tracking and maintenance phases are not completely synchronized with each other. This is due to the fact that snapshots in a KVS are not globally synchronized and we cannot determine the exact execution point. Thus, to not miss any updates, we initiate the tracking phase (for the respective next snapshot), shortly before we take and compute the (actual) snapshot.

Each tracking phase builds up a MK tree, which is then used during the next tracking phase to perform an incremental snapshot. The first snapshot is a full snapshot and, thus, does not need a tracking phase. Hence, the MK tree of the first tracking phase serves to provide the updated keys to the second (incremental) snapshot. The MK tree of the second tracking phase serves to provide the keys for the third (incremental) snapshot.

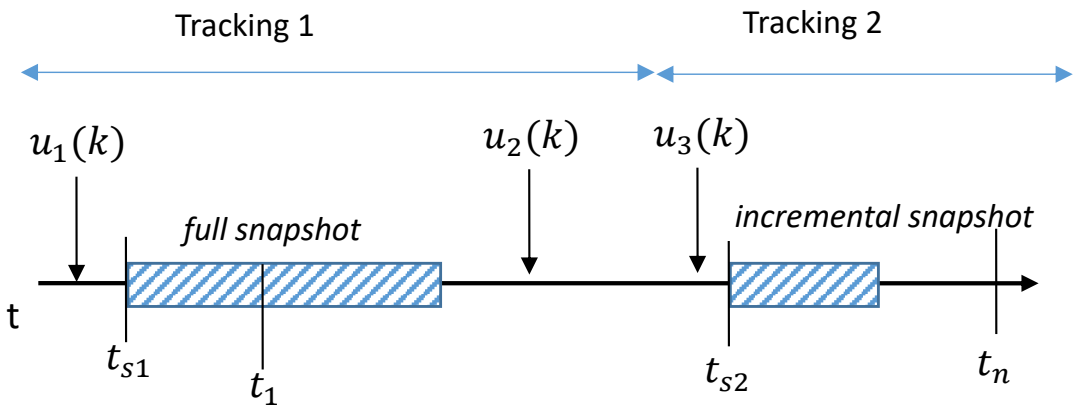


Figure 6.2.2: Incremental snapshots with tracking phases

Example 6.2.1*: In Figure 6.2.2, an update u_1 over key k that is applied in the first tracking phase before the snapshot. Given that the full snapshot already incorporates u_1; but u_1 is also inserted into MK tree and will during the second snapshot be applied, again. Consistency wise, this is not a problem. Using snapshots only, u_1 cannot overwrite, for example, an update u_2 on the same key. What is being used during the second (incremental) snapshot is not u_1, but only its key k to retrieve the most recent (snapshot) version of the record.*

6.3 Hybrid strategies

In this section, we describe how hybrid strategies can be used to combine batch and incremental maintenance. We start by given a rationale of our design decisions and go on explaining transitions of the hybrid processing types.

6.3.1 Rationale

When combining a batch and an incremental based strategy, we always mix a set of records (representing a certain state) with a stream of updates, each provoking a state change on one of the records. As a result, we obtain multiple different record versions of the same record: one version due to the batch job, and, depending on the update distribution, an arbitrary number of versions due to incremental updates. An incremental update operation may be older, newer or equal to the record version that has been obtained during the batch job.

However, depending on our realization, the point at which both strategies are united to present a consistent result, differs. We identified the following realizations as the most relevant ones.

(1) Compute batching and incremental strategy separately and merge the results

(2) Compute batching and incremental strategy combined, check consistency on execution of view update

(3) Compute batching and incremental strategy combined, check consistency on retrieval of base updates

Realization 1 as it is done in [17, 20] is a solution to let a bulk processing and a streaming framework collaboratively compute results with high throughput and low latency. Consistency has to be achieved by providing an exact cut between the batch records and the update stream. While this solution provides a clean separation of concerns, it is not

integrated; it requires multiple architectures/processes to be setup. Also, the additional merging step at the end has a negative effect on the availability of results. Merging cannot be done locally. If both result sets are large (aggregation with high cardinality), there is a high cost of transferring the result sets and performing the merge.

Realization 2 combines batch records and incremental operations already during processing. Thereby, consistency is checked in the moment a view update is executed upon the view table. In this context signatures can be used [33] to establish consistency. The timestamps of records and updates merge into the view record and update its signature. Then, the signature is used to identify and drop out-of-date updates. As a downside of this realization, an update has to be send to the maintenance process and is dropped at the very last point (i.e., at execution time). Also signatures can grow very large as every update operation has to be reflected (e.g., in heavy aggregated records).

Realization 3 sorts out obsolete versions already at retrieval time and produces a stream of up-to-date records and updates that provide a full and coherent view angle of the actual base data. It handles consistency already at maintenance process (i.e. VM) level. Data structures are used to compare and drop duplicates as early as possible which avoids unnecessary compensation or recomputation.

In the following, we build upon Realization 3 to avoid unnecessary overhead and provide an integrated solution. The naive data structure to provide consistency is a simple hash map, storing all keys and their timestamps (i.e. versions) of the maintained records. However, as a hash map grows with number keys inserted and is likely to exceed memory of a VM at some point, in the following, we develop more efficient ways to check for consistency.

6.3.2 Hybrid transitions

We define hybrid strategies, in general, as a combination of a basic batch and basic incremental strategy (see Figure 6.0.1). A hybrid strategy defines consistent transitions between the batch and the incremental execution. Figure 6.3.1 shows the transitions of three hybrid strategies (SNI, SCI and SCIMB). We distinguish between two types:

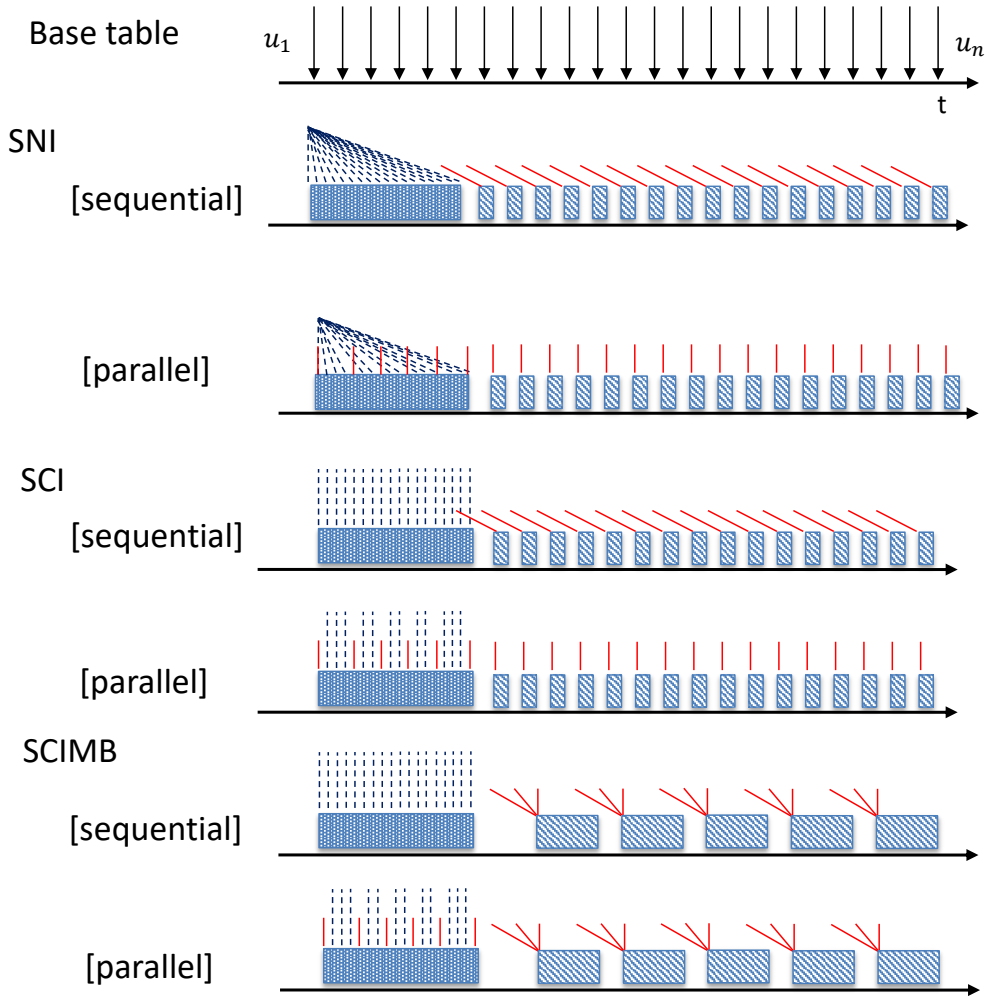


Figure 6.3.1: Hybrid strategy transitions

(1) *Sequential:* Existing base table data is completely processed. While incremental maintenance starts afterwards, client modifications that are made during batch processing are caught and buffered. Still, incremental maintenance is delayed and staleness increases.

(2) *Parallel:* Batch processing is applied and at the same time incremental updates are incorporated; this is desirable when large base tables are processed that delay the processing of incremental updates for a longer period.

SNI – The first mixed hybrid strategy, we are discussing, is a snapshot that is combined with incremental maintenance (SNI). We start with sequential transition.

Sequential: The system takes a snapshot at a specific point in time. It will completely compute and materialize the result of the snapshot. Only then, incremental view maintenance starts. While we compute results of a snapshot, base tables remain available for modifications. Clients still access the database and insert their records. However, the VM queues up update operations, the processing of incremental maintenance is delayed, as it must be halted until the result of the snapshot is fully evaluated. We formalize the distribution round of the strategy as follows:

$$\mathsf{SNI}_{seq} = \langle \mathcal{D}_{snap}(t), \mathcal{D}_{inc} \rangle \tag{6.3.1}$$

Figure 6.3.2 shows how computation of the snapshot (blue phase) is followed by incremental maintenance (red phase). All incremental updates that are collected during the tracking phase are released during the red phase such that incremental maintenance can be executed.

Example 6.3.1*: In Figure 6.3.2, let during tracking phase, two updates (i.e., u_1 and u_2) occur over the same base table key k. Let t_s be the point, where the snapshot is taken. The snapshot includes the most recent versions (with regard to t_s) of all records that have been inserted or updated in the table. Thus, in the figure, the version of snapshot record k at point t_1 corresponds to the update u_1 made before. As u_1 is already incorporated in the snapshot record, and is an incremental update (cf. Example 6.2.1) it should not be maintained.*

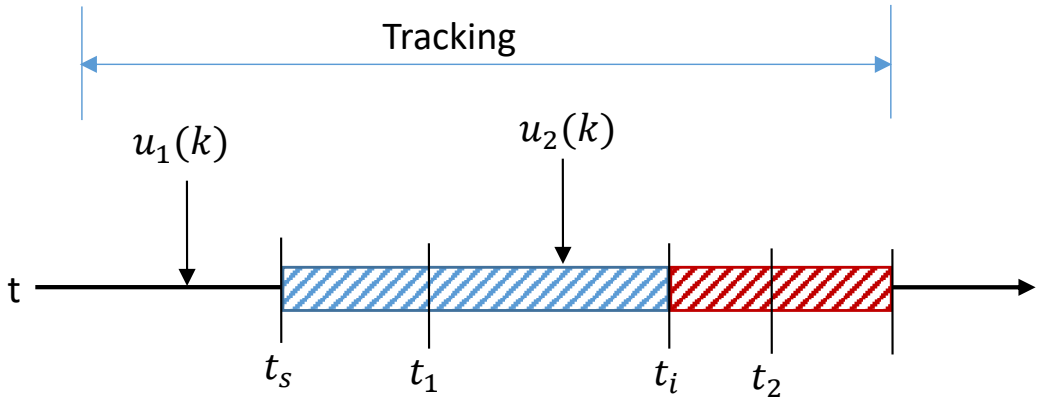


Figure 6.3.2: SNI sequential

Determining the cut-off point – In order to avoid conflicts between snapshot record versions and incremental updates, the system needs to find a cut-off point between the two basic strategies. Also the system has to assure that no base table update is omitted. Some KVS provide the capability of online snapshots that are locally consistent. While the snapshot represents a consistent data set (each taken at one of the KNs), a global point in time can neither be fixed, nor is it defined after the snapshot. As a result, we can also not define a cut-off point, globally.

Thus, we determine a local instance of t_s at each VM. During the blue phase, each VM computes its partition of the snapshot. Thereby, it evaluates the record with the highest timestamp, i.e. $t_s = max(\{t(r)|r \in \mathcal{D}_{snap}\})$. We use function $t(r)$ to determine the timestamp (i.e., the insertion point into the base table) of a record or an update. Timestamp t_s, then, becomes the orientation timestamp; every snapshot record has the most recent version with regard to that timestamp. When incremental maintenance starts, all updates that are older than t_s are dropped. This selection process is formalized in the following equation.

$$(\forall u \in \mathcal{D}_{inc})(t(u) < t_s) \longrightarrow \mathcal{D}_{inc} = \mathcal{D}_{inc} \setminus \{u\} \tag{6.3.2}$$

The sequential transition benefits from the efficiency and simplicity of its implementation. As a downside, incremental view maintenance gets delayed – which counter-acts its original purpose to provide real-time analytics. During computation of the snapshot

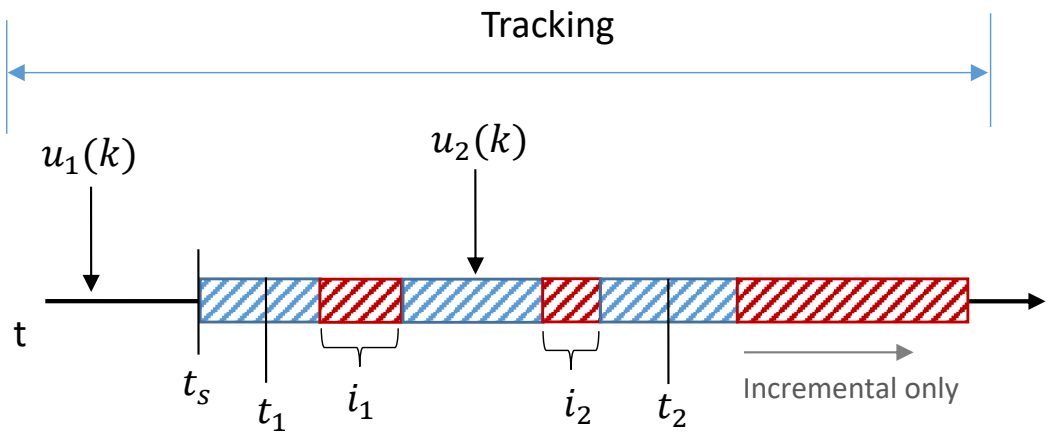


Figure 6.3.3: SNI parallel

the maintenance system experiences a down-time, during which the views remain stale. After that the systems has to catch up with the insertion rate of clients.

Parallel: When using parallel transition, we are trying to catch updates in real-time. We are taking the snapshot, loading its records and at the same time, we are letting the incremental updates stream in. Records and updates are combined, the results are computed and materialized before (pure) incremental maintenance starts. Figure 6.3.3 shows an example of snapshot maintenance and incremental phases alternating each other. As can be observed, the retrieval of incremental operations is more in-time, but it overlaps with the computation of the snapshot. We formalize the problem as follows:

$$\begin{aligned} \mathcal{R}_i \cup \mathcal{R}\Delta &= \{e | e \in \mathcal{R}_i \vee e \in \mathcal{R}\Delta\} \\ \mathcal{D}_{sni}(t) &= d_{sni}(\mathcal{R}_i \cup \mathcal{R}\Delta) \\ \text{SNI}_{par} &= \mathcal{D}_{sni}(t_1) \end{aligned} \tag{6.3.3}$$

Record versions and incremental updates are loaded from KVS simultaneously. Thus, the input stream $\mathcal{R}_i \cup \mathcal{R}\Delta$ transforms into one single large execution sequence $\mathcal{D}_{sni}(t)$. As illustrated in Figure 6.3.3, the parallel transition is a concatenation of smaller transitions between snapshot and incremental processing (and vice versa). As such, in the following, we treat processing of updates and records differently. Let $\mathcal{D}_{sni}(t)$ provide the updates that have been loaded in the maintenance process so far.

Snapshot maintenance starts with the lowest record key and iterates up to the highest. We store this information in form of the maintained key range where k_l marks the lowest and k_h the highest key maintained so far. When the row-key of an incremental update falls into the maintained key range and its timestamp $\leq t_s$, we drop the update.

$$\begin{aligned} k_l &= min(\{r.K | r \in \mathcal{D}_{sni}(t)\}) \\ k_h &= max(\{r.K | r \in \mathcal{D}_{sni}(t)\}) \end{aligned} \tag{6.3.4}$$

When alternating between snapshot and incremental maintenance, it is not possible to evaluate the highest timestamp of the complete snapshot. However, the timestamp can be evaluated as $t_s = max(\{t(r) | r \in \mathcal{D}_{sni}(t)\})$, the highest timestamp that has been maintained so far. We process or drop incremental updates based on the following equation:

$$(\forall u \in \mathcal{D}_{sni}) \longrightarrow \mathcal{D}_{sni} = \begin{cases} \mathcal{D}_{sni} \setminus \{u\} & if((t(u) < t_s) \wedge (k_l \leq u.K \leq k_h)) \\ \mathcal{D}_{sni} & else \end{cases} \tag{6.3.5}$$

This way, we make sure that no update overwrites an already maintained record. Still we need to cover the inverse case, where a record overwrites an already (incrementally) maintained operation. To prevent this from happening, we use the maintained time range. This is possible because update streams are in correct (local) time ordering.

$$\begin{aligned} t_l &= min(\{t(u) | u \in \mathcal{D}_{sni}(t)\}) \\ t_h &= max(\{t(u) | u \in \mathcal{D}_{sni}(t)\}) \end{aligned} \tag{6.3.6}$$

When a records timestamp $t(r)$ can be found within the time range, we know its has been maintained and can be discarded. However, when $t(r)$ is not within the time range a decision can be ambiguous, as the following example demonstrates:

Example 6.3.2*: In Figure 6.3.3, two base updates u_1, u_2 to a row-key k are depicted. Further, the timeline shows two possible points t_1 and t_2 at which the snapshot record, belonging to k, could be maintained; the record r with row-key k, retrieved by the snapshot corresponds to an earlier version (with $t(r) = t(u_1)$), which was not captured in the tracking phase. Let the incremental maintenance of update u_2 be executed during phase starting at i_2. Now, we can make different observations based on the different points, the snapshot record is maintained: t_1, incremental maintenance has been executed, but not over key k. The snapshot record shall be applied; t_2, incremental maintenance has been executed, also over key k (representing*

version u_2). The snapshot record shall be dropped in order to not overwrite the incremental update. In both cases $t(r) < t_l$.

The system has to remember what row keys have already been maintained incrementally. Only then, it can make a decision about applying a snapshot record or not. To solve the problem we make use of MK tree, again. Every snapshot record is tested against MK tree – which is filled with all keys that have been incrementally updated before, i.e., MK tree = $\{u.K | u \in \Delta\mathcal{R}(t)\}$. If the key can be found within the tree, the snapshot record is dropped; if it cannot be found the snapshot record is forwarded.

$$(\forall r \in \mathcal{D}_{sni}) \longrightarrow \mathcal{D}_{sni} = \begin{cases} \mathcal{D}_{sni} \setminus \{r\} & if \left(t_1 \leq t(r) \leq t_h\right) \ \vee \left[(t(r) \leq t_l) \wedge (r.K \in MK_{tree})\right] \\ \mathcal{D}_{sni} & else \end{cases} \tag{6.3.7}$$

SCI – The next combination, we are discussing is a scan plus consecutive incremental maintenance (SCI). Again, we start with sequential transition and, then, discuss the parallel case.

Like stated before, a scan does not provide a consistent set of records. While the snapshot retrieves all records with regard to a given point in time, the scan also incooperates changes made during the scanning process. However, it does not incooperate all changes as it misses those updates on records that have been already scanned. Since scans can take a very long time, incremental view maintenance should be enabled to not miss any updates. We denote the time span of a scan as (t_s, t_e). Accordingly, the scanned records can take either the latest version before t_s or any version within the time span.

Sequential – Sequential transition means that the system first executes the scan, and once it is taken, incremental view maintenance starts. As the system always allows for client updates, incremental updates are still retrieved during the scan; they are queued until the scan (and its computation) has finished. We formalize the problem as follows:

$$\text{SCI}_{seq} = \langle \mathcal{D}_{scan}(t_s, t_e), \mathcal{D}_{inc} \rangle \tag{6.3.8}$$

Example 6.3.3*: Figure 6.3.4 shows a scan starting at cut-off time t_s and completing at point t_e. In the figure, three incremental operations u_1, u_2 and u_3 on top of row key k occur; The*

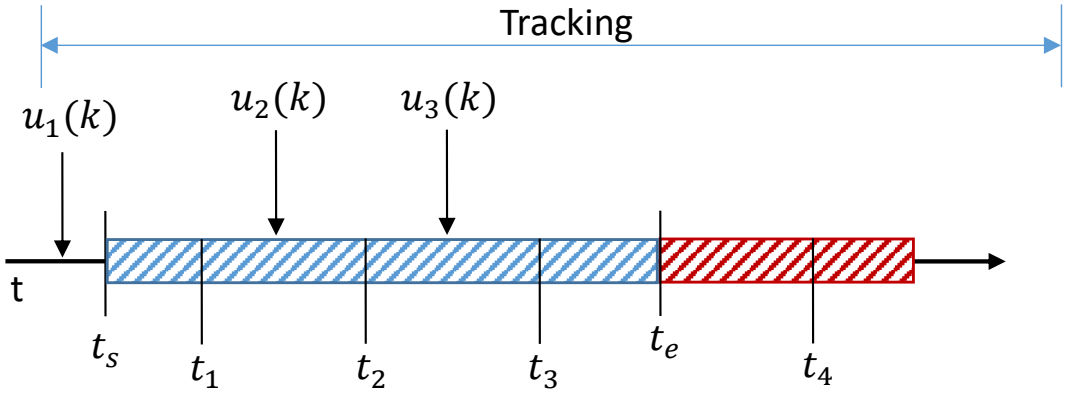


Figure 6.3.4: SCI sequential

timeline shows the following: three points t_1, t_2 and t_3 at which the scanned record, belonging to k, could be maintained; Scanning at point t_1 results in record version $t(u_1)$, which means incremental updates u_2 and u_3 can perfectly maintained at point t_4. However, scanning record k at point t_2 or t_3 may result in a scanned record version $t(u_)2$ or $t(u_3)$; if we proceed and compute the incremental update u_1 at point t_4, it will overwrite the already scanned version. As already mentioned, overwriting newer with older records is not acceptable.

Again, we need a data structure to identify and drop updates at process level. A scanned record can possibly represent any update version during scan time. Thus, we need the keys of the scanned records along with their timestamp (i.e., the version). We consider a hash-map as simple data structure to test against; during the scan we build up the hash-map, storing key/timestamp pairs. Then, during incremental maintenance, we can load a row key's timestamp and decide whether to drop the incremental operation or not.

The naive solution, inserting the row-keys of all scanned records into a hash-map along with their timestamps ($H_{key}[k] \longrightarrow t\}$) yields a significant memory overhead and can lead to crash of a VM. To reduce the size of the hashmap, significantly, we make use of the MK tree, again. During the tracking phase, we insert the keys of all update operations into the tree. In contrast to before (see SNI parallel), we use MK tree to store the tracked keys (and not the incrementally maintained keys), i.e., MK tree = $\{u.K | u \in \Delta\mathcal{R}_{tracked}\}$.

Thereby, we track a key *before* it is inserted into the base table, such that we can be certain it is contained in the tree before it becomes visible for a scan.

Then, while scanning, we test whether the key of a scanned record can be found within the tree. If yes, we know that there have been modifications during the scan. Therefore, we store a key/timestamp pair of the key into the hashmap. The hashmap entry serves during incremental maintenance phases to identify and drop duplicate versions.

$$(\forall r \in \mathcal{D}_{scan}) \longrightarrow \begin{cases} H_{key}[r.K] = t(r) & if\,(r.K \in MK_{tree}) \\ \emptyset & else \end{cases} \tag{6.3.9}$$

$$(\forall u \in \mathcal{D}_{scan}) : \begin{cases} \mathcal{D}_{scan} \setminus \{u\} & if\,(t(u) \leq H_{key}[u.K]) \\ \mathcal{D}_{scan} & else \end{cases} \tag{6.3.10}$$

Using the MK tree, we make sure that only those scanned keys are stored into the hashmap, which have undergone changes. Also, we point out that there is an alternative. Instead of using the hashmap, we could also drop scanned records of keys that we find in the MK tree. We know that the key is maintained during incremental phase and, thus, we can skip the scanned version. As explained before, losing intermediate view states does not hurt our consistency criterion.

Parallel – When using parallel strategies, along with a scan, we assume that scan records and base updates stream in at the same time and can be processed in alternating order. We formalize the problem as follows:

$$\begin{aligned} \mathcal{R}_{i..j} \cup \mathcal{R}\Delta &= \{e | e \in \mathcal{R}_{i..j} \vee e \in \mathcal{R}\Delta\} \\ \mathcal{D}_{sci}(t_s, t_e) &= d_{sci}(\mathcal{R}_{i..j} \cup \mathcal{R}\Delta) \\ \mathsf{SCI}_{par} &= \mathcal{D}_{sci}(t_1, t_2) \end{aligned} \tag{6.3.11}$$

Exactly as for SNI parallel, we have to ensure consistency during a sequence of smaller batch and incremental maintenance phases. Again, we use a hash map along with an instance of MK tree to provide consistency for incremental operations. The hash map is build from the already scanned records as Equation 6.3.12 demonstrates. The selection of operations can, then, be done equivalent to Equation 6.3.10. To prevent records from overwriting newer versions, a tree is not needed as the following example demonstrates.

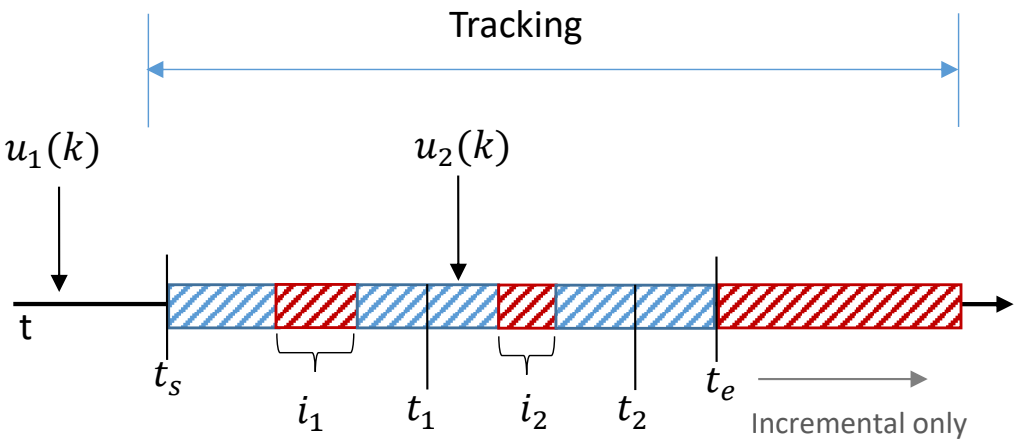


Figure 6.3.5: SCI parallel

Example 6.3.4: *In Figure 6.3.5, we see a similar setup as in Figure 6.3.3: two points t_1 and t_2 where a scanned record r could possibly be maintained; above (see SNI parallel), we found that the critical decision is between those two points as the system is not able to tell (by time range) whether r has been incrementally maintained or not. Now, processing a scan, we can be certain that, if the record would have been incrementally maintained, the scanned version would also incorporate the change.*

There are only two possible cases: the scanned version lies within the maintained time range (defined as (t_l, t_h)) and needs to be dropped; the scanned version lies outside the time range and needs to be maintained. In essence, to make a decision, we only need to consider the maintained time range. The formalization of the selection process can be found in Equation 6.3.13.

$$(\forall r \in \mathcal{D}_{sci}) : \begin{cases} H_{key}[r.K] = t(r) & if\,(r.K \in MK_{tree}) \\ \emptyset & else \end{cases} \tag{6.3.12}$$

$$(\forall r \in \mathcal{D}_{sci}) : \begin{cases} \mathcal{D}_{sci} \setminus \{r\} & if\,(t_1 \leq H_{key}[r.K] \leq t_h) \\ \mathcal{D}_{sci} & else \end{cases} \tag{6.3.13}$$

SCIMB – The maintenance of SCIMB is very similar to that of SCI and can be also executed as a sequential or a parallel strategy. The only difference is the execution of the incremental part, starting after materialization of the scan results at point t_e. Here, SCIMB micro-batches incremental updates and materializes them in one run.

6.3.3 View states

One of our requirement is to keep views dynamically, i.e., they should be added and removed during maintenance. A view that is created at the beginning of the update process has a different state than a view that is created later. To keep track of that we attach a state variable to each view. Views that are created at the same time can share an instance of the variable. When running a SNI_{par} strategy, we create a tuple $\mathcal{S} = (V_{set}, \ (t_l, t_h), \ (k_l, k_h), \ MK_{tree})$. V_{set} is the set of views, maintained time range, key range and the required data structure MK_{tree}. When creating a set of new views, we create an instance of variable $\mathcal{S}$ for all of them. The variable stores the maintenance state and helps to perform correct maintenance. If another view is created at a later point in time, a new variable $\mathcal{S}$ is instantiated.

6.4 Evaluation

In this section, we report the results of an extensive experimental evaluation of our approach. We fully implemented our standard and hybrid view maintenance algorithms on top of VMS in Java and integrated it with Apache HBase.

Static-workload scenario – For the base-line experiments, we evaluate VMS maintenance performance using a *static workload*. This means, we create the TPC-H base tables in HBase (i.e., *lineitem*, *orders*, *customer*, *part*, *partsupp*, *supplier*, *nation*, *region*). Each base table is pre-partitioned (by key-range) over the 200 region server. Then, we start 200 clients which use the dbgen tool to fill base tables with a scale factor of $100x$ resulting in a maximum of $600M$ records being inserted into the $lineitem$ table. For each run, we configure one of the TPC-H queries in VMS. Then, we either choose SC or SN as

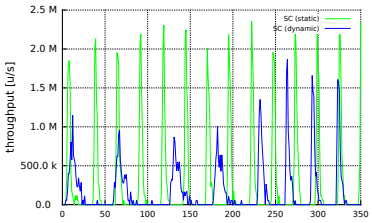


Figure 6.4.1: SC (Q_6)

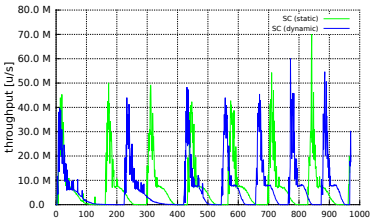


Figure 6.4.2: SC (Q_7)

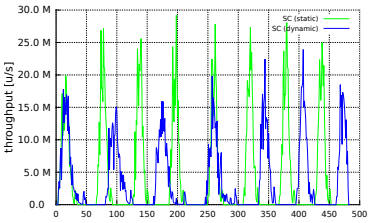


Figure 6.4.3: SC (Q_{10})

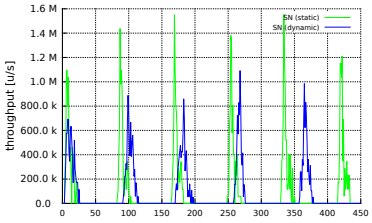


Figure 6.4.4: SN (Q_6)

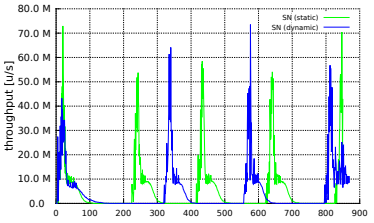


Figure 6.4.5: SN (Q_7)

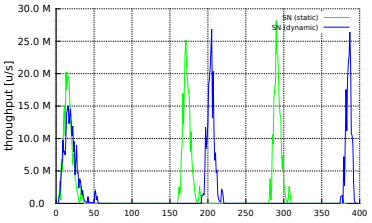


Figure 6.4.6: SN (Q_{10})

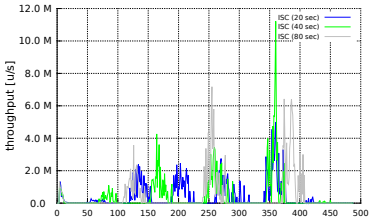


Figure 6.4.7: ISC (Q_6)

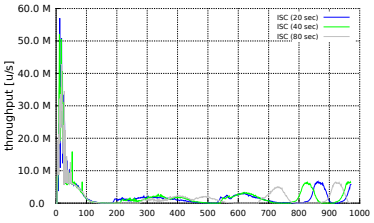


Figure 6.4.8: ISC (Q_7)

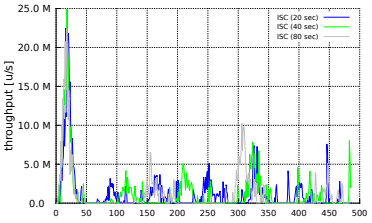


Figure 6.4.9: ISC (Q_{10})

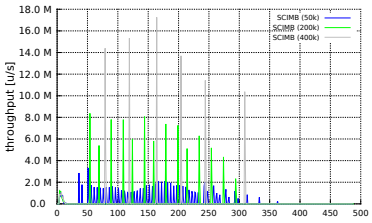


Figure 6.4.10: SCIMB (Q_6)

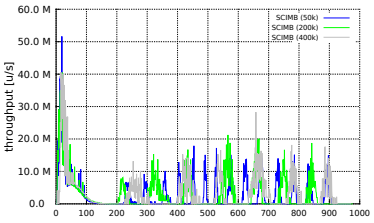


Figure 6.4.11: SCIMB (Q_7)

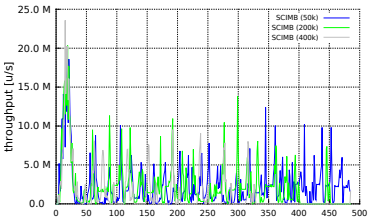


Figure 6.4.12: SCIMB (Q_{10})

maintenance strategy because those are the only strategies applicable in a static context (ISC and SCIMB are only needed when base data is updated). We let VMS execute the maintenance strategies during a defined time interval and measure its throughput in real-time. The throughput is determined at each VM and, then, aggregated at the master node. The base-line experiments are depicted as blue graphs in Figure 6.4.1-6.4.3 and Figure 6.4.4-6.4.6.

Dynamic-workload scenario – For the full comparison, we evaluate VMS maintenance performance using a *dynamic workload*. This means, we create a setup as described in the base-line experiments, first. We create the TPC-H entities as base tables in HBase and start 200 clients to fill them (again, with scale factor $100x$).

Then, we start VMS, configure one of the TPC-H queries and choose a the maintenance strategy: SC, SN, ISC or SCIMB. For strategies SC and SN $t_w = 0$ can be set to zero such that one scan directly succeeds the next one. A larger t_w can be used when latency bounds are lower than the actual scan time. For strategy ISC, we test configurations $t_w \in \{20s, 40s, 80s\}$ because the number of updates that can be accumulated is depending on t_w. A low t_w means many executions with few update whereas a high t_w means few executions with many updates. Likewise for SCIMB, we use different configurations of batch sizes with $b \in \{50k, 200k, 400k\}$. Again, using smaller b lets the system do many executions with few updates whereas larger b lets the system do few executions with many updates.

In a next step, we start the 200 clients, again, and let them insert an update workload of $100M$ records into the base tables consisting of 50% update and 50% delete operations. While the clients are inserting their updates, we execute one of the maintenance strategies and let VMS catch up to the current state of the base tables.

SC – Figures 6.4.1-6.4.3 show the maintenance plot of the SC strategy for Q_6,Q_7 and Q_{10}. Albeit t_w has been set to zero, the SC strategy (also SN) experiences a break of 5 – $30s$ between the cycles. This is related to the fact that after a cycle, VMS has to delete all view tables, reload the maintenance plan and recreate the view tables.

Evaluating Q_6, we measure 13 maintenance cycles using the static and 8 maintenance

cycles using the dynamic workload. For Q_7, VMS achieves 3 cycles using the static and 2 cycles using the dynamic workload (in a comparable time frame of 350s). Query Q_{10} is at 6 cycles for the static and 5 cycles for the dynamic case.

Despite of having the highest number of maintenance cycles, Q_6 achieves the lowest throughput in VMS (2M updates per second). When running scans on base tables, each VM requests a local scan from a region server of HBase. In doing so, the VM already applies selection criteria given along with the query. If there is a high selectivity, like for query Q_6, the throughput of VMS is low as most of the records are already filtered at region server side. When measuring the throughput of base table updates per second, the combined system (HBase plus VMS) is at 200M per second.

Q_7 shows a throughput of up to 60M updates per second. The maintenance curve is steep, at first, but then it slowly declines as Q_7 involves multiple joins which require consecutive redistribution of updates. The largest delays in the dynamic setup of Q_7 can be observed during the first 400 seconds where congestion due to client request is the highest. In the moment the client requests decline, the latency immediately returns to the normal level (which is measured for the static workload).

SN – Figures 6.4.4-6.4.6 show the maintenance plots of the SN strategy. The SN strategy achieves 6 and 5 (static and dynamic) maintenance cycles for Q_6, 1 and 2 cycles for Q_7 and 2 and 3 cycles for Q_{10}. For all three queries, we measure longer breaks in between the maintenance cycles (~60s for Q_6, ~120s for Q_7 and for Q_{10}). The breaks are related to query reloading, view recreation (~25%) and snapshot creation (~75%). Thereby, the impact of snapshot creation on overall execution time is dependent on the query. For Q_6 the impact is the highest (up to 2.8x) because the query also executes the fastest. For long running, multi-table joins, like Q_7 the impact of snapshot creation is not as significant (only 30%).

ISC – Figures 6.4.7-6.4.9 show the results of the ISC strategy. The amount of maintenance cycles for ISC are determined by the interval $t_w \in \{20s, 40s, 80s\}$ that we use during evaluation. For Q_6, we achieve 4,3 and 2 maintenance cycles for the corresponding parameters in t_w. However, the number of cycles for Q_{10} is identical.

After the initial scan, we observe multiple small/incremental scans. The breaks between

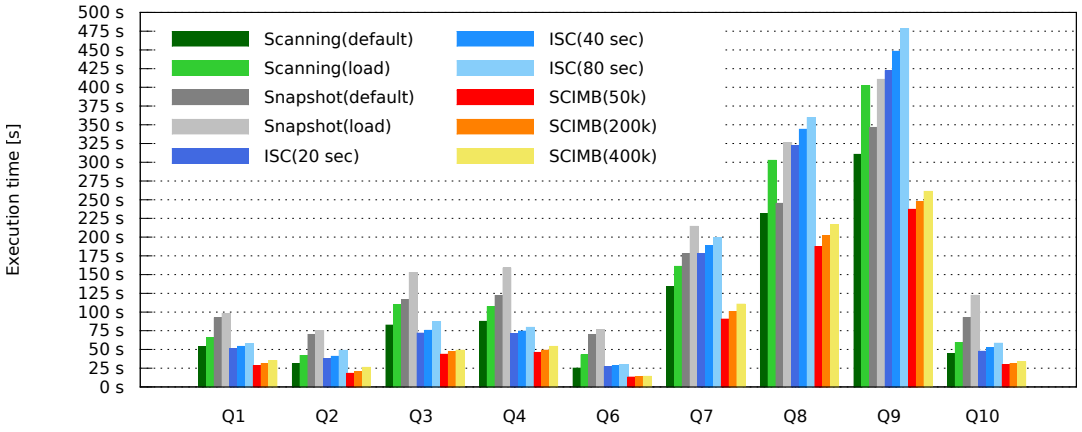


Figure 6.4.13: Maintenance strategies overall results (1)

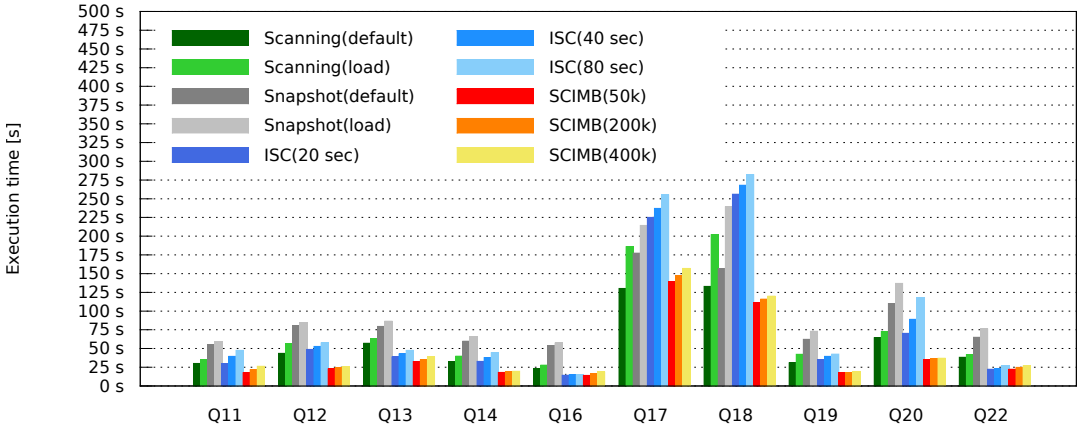


Figure 6.4.14: Maintenance strategies overall results (2)

the scans are completely determined by t_w as there is no time spent to reload the plan and recreate the view tables. For Q_6 the latency of incremental scans converges towards $21s$ at $t_w = 20s$. For Q_7, we observe a significant slow down (up to $320s$) of incremental scans during the peak time which later recovers (to match $92s$).

SCIMB – Figures 6.4.10-6.4.12 depict the results of the SCIMB strategy. The amount of maintenance cycles is determined by the batch size $b \in \{50k, 200k, 400k\}$. For Q_6, VMS achieves 52, 15 and 7 maintenance cycles, for Q_{10} VMS achieves 26, 17 and 9 cycles.

After the initial batch job has been executed, we observe very crisp incremental maintenance intervals. Since a recreation of views is not required the latency converges towards $1s$ for Q_6, $21s$ for Q_7 and $12s$ for Q_{10}. Configuring the batch size differently, also different throughputs are achieved: for Q_6, it is $2M$ updates per second at $50k$, $8M$ at $200k$ and $16M$ at $400k$ batch size. In general, the maintenance performance of SCIMB is very constant and predictable. During peak times, maintenance executes similarly fluent as during idle times.

Overall comparison – Figures 6.4.13 and 6.4.14 show the aggregated results evaluating TPC-H queries Q_1-Q_{22}. The results of the strategies are presented showing the average execution time of maintenance, which defines the latency of the maintained view tables. For a fair comparison, the latency of hybrid strategies is defined as the combination of the batch and the incremental maintenance part. However, in reality the latency of a hybrid strategy converges towards the incremental processing. For the ISC and SCIMB strategies, we exclude the waiting times from the execution time, as the system is simply waiting for more updates.

In general, SC provides a much better maintenance performance as SN (only ~46-62% of SC). While throughputs of both strategies are very comparable, the delays of SN are mainly caused by snapshot creation. The creation time of snapshots can last from $20s$ up to $100s$ if many large base tables are involved (e.g., Q_7). While snapshots are handled using references in HBase, their creation also occupies additional storage space (example).

We observe that on average 28% latency is added when the SC strategy is performed using a dynamic workload (in comparison to the static setup). For SN, on average only 18% latency is added using a dynamic workload. We conclude that the SN strategy is

much more robust during peak times of client requests. After the snapshot has been created, maintenance is performed separately and does not interfere with the actual base table processing.

Another strength of the SN strategy (that cannot be measured in terms of performance) lays in the quality of results. SN views are calculated with regard to a single point of time. The results of SC, while computed correctly, represent a time span of the base table. Thus, results obtained with the SN strategy are more exact.

If compared to the SC strategy, ISC has a slight advantage (5 – 10% faster), especially comparing the dynamic setup (of SC). Considering the fact that incremental scans involve less records, their execution times are not proportional. In contrast to normal scans, VMS cannot advise HBase to apply filters at server-side. Thus, MK tree has to be traversed and records have to be retrieved from HBase subsequently. Especially for long running queries (i.e., Q_7,Q_8,Q_9,Q_{17},Q_{18}), we observe a degradation of performance.

However, ISN can be configured to use smaller maintenance intervals (e.g., < 5s), which SC cannot. Also efficiency-wise ISC is far ahead, as the SC strategy requires the repeated scan of base records ($13x600M$ for Q_6), whereas ISC only needs the initial scan plus updates ($700M$ for Q_6). The ISN strategy (which is based on snapshots) is a very viable alternative to ISC, as it mitigates the impact on maintenance during peak times.

SCIMB provides by far the best and most predictable latencies for view tables. Like ISC, SCIMB does not require recreation of views after the initial batch has been carried out. The average incremental throughput of SCIMB is much higher as for the ISC strategy, as updates are queued up into memory and upon execution are released and directly processed by VMs.

The drawbacks of SCIMB can be its efficiency and its memory consumption. Especially for highly skewed distributions or small tables with many updates, ISC performs better. Using ISC, all updates to the same base record will be executed by a single get, as they are represented by a single row-key in MK tree. By contrast SCIMB, queues up all update operations into memory, which can overload the VM. Using the combine operations method (see Section 6.1.2) the effect. However, the update operations still have to be buffered into memory before processing.

SNI and SCI are not depicted as they are conceptually the same as SCIMB (only inferior in performance). Computing very large data sets, micro-batching is a requirement to provide reasonable execution times. Also SNIMB is not depicted (similar performance), still it is a convenient option as the delay for creating a snapshot is only incurred once.

Chapter 7

Conclusions

In this chapter, we provide a summary of our findings and give an outlook in which areas we identify potential for future work.

7.1 Summary

In Chapter 4, we presented VMS, a scalable view maintenance system which operates together with a distributed KVS. VMS provides efficient, incremental, strongly consistent, and asynchronous view materialization and maintenance. Pre-Processing Views facilitate and speed up maintenance and avoid expensive table scans. Building on the Pre-Processing Views, VMS can consistently compose and maintain SQL queries. As a proof of concept, VMS was implemented on top of HBase. Our experimental evaluation with TPC-H benchmark showed that VMS can handle large amounts of update streams and provide reasonable execution times at the same time. We demonstrated the system's ability to perform real-time processing, and quantified the benefits and drawbacks of the approach.

In Chapter 5, we showed that the maintenance of many views can be scaled massively (up to $10k$ views, materialized in < 100 seconds) by applying traditional optimization

techniques like pre-aggregation of aggregates or using fully partitioned or partially partitioned join loading, depending on the table sizes. Moreover, we developed new optimization techniques such as pre-evaluating predicates (via bit vectors), decomposed pre-aggregation or combination via Reverse-join to merge the maintenance plans of views and use synergies of executed operators (e.g., overlapping selection predicates). Especially for views, sharing a query template with different parameters, materialization performance can be improved significantly and write amplification can be prevented successfully.

In Chapter 6, we presented a concept for hybrid view maintenance in KVSs. We developed formal techniques and data structures to achieve consistency when maintaining KVS primitives (scans, snapshots) or streams of KVS update operations (co-processor). Further, We conducted an extensive study of distributed maintenance strategies including batch, incremental and hybrid types. We showed how hybrid maintenance strategies can materialize views from existing data sets and simultaneously perform incremental maintenance to provide fresh results already from the get-go. Finally, we implemented our approach in VMS and provided a comprehensive evaluation on top of HBase using a variety of different maintenance strategies. Thereby, we illustrated the strength of hybrid maintenance strategies to efficiently combine off- and online analysis, to provide low latency and fresh view data and to outperform their basic counterparts.

7.2 Future work

Protecting against VM overload: To have the maintenance work evenly distributed over VMs and to achieve the best results possible, we used mainly uniform distributions as a workload. Skewed distributions, like for example a Zipf distribution in a base table can be already handled at KVS level using techniques like salting. However, if this is not the case, single VMs would get overloaded fast. VMS can prevent VM overload relatively good at planning stage. Low cardinalities of aggregation or join keys (and the resulting big update loads), can be already countered by using batched Pre-aggregation or Reverse-join operators in the maintenance plan. However, when VMs get overloaded with updates, there is no dynamic mechanism to split the work load and redistribute it to another set

of VMs that might be running idle at the same time.

Adapting to partition reassignment: As explained before evaluation has been performed with uniformly distributed work loads and more or less static partition assignment. In a real world setup the addition or removal of resources of the KVS or peaks in the client loads can lead to dynamic reassignment of partitions by KVS. In such a case a partition is relocated from one KN to another KN. While this does not impact the maintenance plan and its distribution rounds, the responsibility of the base records is also handed from one VM to another VM. If there is no clear cut, i.e., both VMs are processing the same partition simultaneously there is a possible timeline violation.

Predicting load scenarios: When computing large-scale batch or incremental maintenance plans, VMS provides an excellent overall performance and has shown very reasonable execution times for combinations of various SQL operators. However, knowing some of the maintenance parameters in advance, can lead to much better results, as the maintenance plan can be adapted beforehand. For example, when planning n-table joins, the order of joining the tables can be oriented on the table size, for batching strategies, and on the update frequencies for the incremental strategies. Even very approximate values of these parameters can lead to a much improved maintenance performance, especially for some of the corner cases.

Trading off consistency guarantees: Like described in Chapter 4.3.4, we use a very strict constraint and allow only $n = 1$ versions of a record keys timeline to achieve strong consistency. Like suggested in the chapter, this constraint could be relaxed such that $n > 1$ versions of the same record key are possible. However, in that case, additional measures have to be invented such that strong consistency is guaranteed. One possible approach would be the synchronization of the versions at the end of the maintenance process, using a coordinator like done for the multi-row updates.

List of Acronyms

VM View Manager

VMS View Maintenance System

KVS Key-Value Store

KN Key-Value Store Node

TL Transaction Log

WAL Write-Ahead Log

VS Virtual Store

TB Timeline Buffer

IN Incremental

IMB Incremental micro-batched

SC Scanning Strategy

SN Snapshot Strategy

SCI Scanning + Incremental Strategy

SNI Snapshot + Incremental Strategy

ISC Incremental Scanning Strategy

ISN Incremental Snapshot Strategy

SCIMB Scanning + incremental (micro-batched) Strategy

SNIMB Snapshot + incremental (micro-batched) Strategy

TPC-H Transaction Processing Performance Council (Benchmark H)

HDFS Hadoop Distributed Files System

SQL Structured Query Language

SPJA Selection Projection Join and Aggregation

List of Figures

List of Algorithms

Bibliography

[1] J. Parikh. *Data Infrastructure at Web Scale*. `http://www.vldb.org/2013/keynotes.html`. 2013 (accessed February 3, 2020).

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data." In: *ACM Trans. Comput. Syst.* 26.2 (June 2008). ISSN: 0734-2071.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-Value Store." In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, pp. 205–220. ISBN: 9781595935915.

[4] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s hosted data serving platform." In: *PVLDB* 1 (Aug. 2008), pp. 1277–1288.

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

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

[7] M. A. Olson, K. Bostic, and M. Seltzer. "Berkeley DB." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '99. Monterey, California: USENIX Association, 1999, p. 43.

[8] D. Peng and F. Dabek. "Large-Scale Incremental Processing Using Distributed Transactions and Notifications." In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 251–264.

[9] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. "Asynchronous View Maintenance for VLSD Databases." In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, Rhode Island, USA: Association for Computing Machinery, 2009, pp. 179–192. ISBN: 9781605585512.

[10] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. "Feeding Frenzy: Selectively Materializing Users' Event Feeds." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 831–842. ISBN: 9781450300322.

[11] S. Akhtar and R. Magham. *Pro Apache Phoenix: An SQL Driver for HBase*. 1st. USA: Apress, 2016. ISBN: 1484223691.

[12] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. "SLIK: Scalable Low-Latency Indexes for a Key-Value Store." In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '16. Denver, CO, USA: USENIX Association, 2016, pp. 57–70. ISBN: 9781931971300.

[13] B. Kate, E. Kohler, M. S. Kester, N. Narula, Y. Mao, and R. Morris. "Easy Freshness with Pequod Cache Joins." In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 415–428. ISBN: 9781931971096.

[14] Y. Katsis, K. W. Ong, Y. Papakonstantinou, and K. K. Zhao. "Utilizing IDs to Accelerate Incremental View Maintenance." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1985–2000. ISBN: 978-1-4503-2758-9.

[15] W. Zhao, F. Rusu, B. Dong, K. Wu, and P. Nugent. "Incremental View Maintenance over Array Data." In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, pp. 139–154. ISBN: 978-1-4503-4197-4.

[16] M. Nikolic, M. Dashti, and C. Koch. "How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates." In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pp. 511–526. ISBN: 978-1-4503-3531-7.

[17] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. "Summingbird: A Framework for Integrating Batch and Online MapReduce Computations." In: *Proc. VLDB Endow.* 7.13 (Aug. 2014), pp. 1441–1451. ISSN: 2150-8097.

[18] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu. "DEDUCE: At the Intersection of MapReduce and Stream Processing." In: *Proceedings of the 13th International Conference on Extending Database Technology*. EDBT '10. Lausanne, Switzerland: ACM, 2010, pp. 657–662. ISBN: 978-1-60558-945-9.

[19] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. "Muppet: MapReduce-style Processing of Fast Data." In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), pp. 1814–1825. ISSN: 2150-8097.

[20] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. "Naiad: A Timely Dataflow System." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 439–455. ISBN: 978-1-4503-2388-8.

[21] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. *Apache Flink: Stream and Batch Processing in a Single Engine*. Tech. rep. TU Delft, 2015.

[22] J. Albert. "Algebraic Properties of Bag Data Types." In: *Proceedings of the 17th International Conference on Very Large Data Bases*. VLDB '91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 211–219. ISBN: 1558601503.

[23] J. Albert. "Algebraic Properties of Bag Data Types." In: *Proceedings of the 17th International Conference on Very Large Data Bases*. VLDB '91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 211–219. ISBN: 1558601503.

[24] L. Libkin and L. Wong. "Query Languages for Bags and Aggregate Functions." In: *J. Comput. Syst. Sci.* 55.2 (Oct. 1997), pp. 241–272. ISSN: 0022-0000.

[25] S. Grumbach, L. Libkin, T. Milo, and L. Wong. "Query Languages for Bags: Expressive Power and Complexity." In: *SIGACT News* 27.2 (July 1996), pp. 30–44. ISSN: 0163-5700.

[26] L. Libkin and L. Wong. "Some Properties of Query Languages for Bags." In: (Oct. 1995).

[27] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. "Algorithms for Deferred View Maintenance." In: *SIGMOD Rec.* 25.2 (June 1996), pp. 469–480. ISSN: 0163-5808.

[28] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. "Efficiently Updating Materialized Views." In: *SIGMOD Rec.* 15.2 (June 1986), pp. 61–71. ISSN: 0163-5808.

[29] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. "Maintaining Views Incrementally." In: *SIGMOD Rec.* 22.2 (June 1993), pp. 157–166. ISSN: 0163-5808.

[30] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. "View Maintenance in a Warehousing Environment." In: *SIGMOD Rec.* 24.2 (May 1995), pp. 316–327. ISSN: 0163-5808.

[31] H. Wang, M. Orlowska, and W. Liang. "Efficient Refreshment of Materialized Views with Multiple Sources." In: *Proceedings of the Eighth International Conference on Information and Knowledge Management*. CIKM '99. Kansas City, Missouri, USA: Association for Computing Machinery, 1999, pp. 375–382. ISBN: 1581131461.

[32] K. Salem, K. Beyer, R. Cochrane, and B. Lindsay. "How To Roll a Join: Asynchronous Incremental View Maintenance." In: vol. 29. June 2000, pp. 129–140.

[33] H.-A. Jacobsen, P. Lee, and R. Yerneni. "View Maintenance in Web Data Platforms." In: *Technical Report, University of Toronto* (2009).

[34] Y. Cui, J. Widom, and J. L. Wiener. "Tracing the Lineage of View Data in a Warehousing Environment." In: vol. 25. 2. New York, NY, USA: Association for Computing Machinery, June 2000, pp. 179–227.

[35] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. "Naiad: A Timely Dataflow System." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 439–455. ISBN: 9781450323888.

[36] C. Chrysafis, B. Collins, S. Dugas, J. Dunkelberger, M. Ehsan, S. Gray, A. Grieser, O. Herrnstadt, K. Lev-Ari, T. Lin, M. McMahon, N. Schiefer, and A. Shraer. "FoundationDB Record Layer: A Multi-Tenant Structured Datastore." In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1787–1802. ISBN: 9781450356435.

[37] C. Jin, R. Liu, and K. Salem. "Materialized views for eventually consistent record stores." In: *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*. 2013, pp. 250–257.

[38] A. Kementsietsidis, F. Neven, D. Van de Craen, and S. Vansummeren. "Scalable Multi-query Optimization for Exploratory Queries over Federated Scientific DBs." In: *Proc. VLDB Endow.* 1.1 (Aug. 2008), pp. 16–27. ISSN: 2150-8097.

[39] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. "Pipelining in Multi-query Optimization." In: *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '01. Santa Barbara, California, USA: ACM, 2001, pp. 59–70. ISBN: 1-58113-361-8.

[40] A. Assefa and F. Getahun. "Multi-query Optimization for Semantic News Feed Query." In: *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*. MEDES '12. Addis Ababa, Ethiopia: ACM, 2012, pp. 150–157. ISBN: 978-1-4503-1755-9.

[41] G. Graefe and W. J. McKenna. "The Volcano Optimizer Generator: Extensibility and Efficient Search." In: *Proceedings of the Ninth International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1993, pp. 209–218. ISBN: 0-8186-3570-3.

[42] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. "Efficient and Extensible Algorithms for Multi Query Optimization." In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. SIGMOD '00. Dallas, Texas, USA: ACM, 2000, pp. 249–260. ISBN: 1-58113-217-4.

[43] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. "Materialized View Selection and Maintenance Using Multi-query Optimization." In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD '01. Santa Barbara, California, USA: ACM, 2001, pp. 307–318. ISBN: 1-58113-332-4.

[44] D. Theodoratos and W. Xu. "Constructing Search Spaces for Materialized View Selection." In: *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP*. DOLAP '04. Washington, DC, USA: ACM, 2004, pp. 112–121. ISBN: 1-58113-977-2.

[45] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. "MRShare: Sharing Across Multiple Queries in MapReduce." In: *Proc. VLDB Endow.* 3.1-2 (Sept. 2010), pp. 494–505. ISSN: 2150-8097.

[46] S. Vemuri, M. Varshney, K. Puttaswamy, and R. Liu. "Execution Primitives for Scalable Joins and Aggregations in Map Reduce." In: *Proc. VLDB Endow.* 7.13 (Aug. 2014), pp. 1462–1473. ISSN: 2150-8097.

[47] G. Wang and C.-Y. Chan. "Multi-query Optimization in MapReduce Framework." In: *Proc. VLDB Endow.* 7.3 (Nov. 2013), pp. 145–156. ISSN: 2150-8097.

[48] F. N. Afrati and J. D. Ullman. "Optimizing Joins in a Map-reduce Environment." In: *Proceedings of the 13th International Conference on Extending Database Technology*. EDBT '10. Lausanne, Switzerland: ACM, 2010, pp. 99–110. ISBN: 978-1-60558-945.

[49] K. Karanasos, A. Katsifodimos, and I. Manolescu. "Delta: Scalable Data Dissemination Under Capacity Constraints." In: *Proc. VLDB Endow.* 7.4 (Dec. 2013), pp. 217–228. ISSN: 2150-8097.

[50] J.-H. Böse, A. Andrzejak, and M. Högqvist. "Beyond Online Aggregation: Parallel and Incremental Data Mining with Online Map-Reduce." In: *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*. MDAC '10. Raleigh, North Carolina, USA: ACM, 2010, 3:1–3:6. ISBN: 978-1-60558-991-6.

[51] F. Coelho, J. Paulo, R. Vilaça, J. Pereira, and R. Oliveira. "HTAPBench: Hybrid Transactional and Analytical Processing Benchmark." In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. Italy: ACM, 2017, pp. 293–304. ISBN: 978-1-4503-4404-3.

[52] J. Arulraj, A. Pavlo, and P. Menon. "Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads." In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pp. 583–598. ISBN: 978-1-4503-3531-7.

[53] F. Özcan, Y. Tian, and P. Tözün. "Hybrid Transactional/Analytical Processing: A Survey." In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, pp. 1771–1775. ISBN: 978-1-4503-4197-4.

[54] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. "BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications." In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, pp. 37–50. ISBN: 978-1-4503-4197-4.

[55] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster Computing with Working Sets." In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, pp. 10–10.

[56] K. Y. Lee, J. H. Son, and M. H. Kim. "Efficient Incremental View Maintenance in Data Warehouses." In: *Proceedings of the Tenth International Conference on Information and Knowledge Management*. CIKM '01. Atlanta, Georgia, USA: Association for Computing Machinery, 2001, pp. 349–356. ISBN: 1581134363.

[57] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. "Stateful Bulk Processing for Incremental Analytics." In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 51–62. ISBN: 978-1-4503-0036-0.

[58] X. Zhang, L. Ding, and E. A. Rundensteiner. "Parallel Multisource View Maintenance." In: *The VLDB Journal* 13.1 (Jan. 2004), pp. 22–48. ISSN: 1066-8888.

[59] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. "The Strobe Algorithms for Multi-Source Warehouse Consistency." In: *Proceedings of the Fourth International Conference on on Parallel and Distributed Information Systems*. DIS '96. Miami Beach, Florida, USA: IEEE Computer Society, 1996, pp. 146–157. ISBN: 081867475X.

[60] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. "Web Caching with Consistent Hashing." In: vol. 31. 11–16. USA: Elsevier North-Holland, Inc., May 1999, pp. 1203–1213.

[61] M. Athanassoulis, M. Kester, L. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. "Designing Access Methods: The RUM Conjecture." In: *International Conference on Extending Database Technology (EDBT)*. Bordeaux, France, 2016.

[62] Y. Cui and J. Widom. "Lineage Tracing for General Data Warehouse Transformations." In: *The VLDB Journal* 12.1 (May 2003), pp. 41–58. ISSN: 1066-8888.

[63] G. Moro and C. Sartori. "Incremental Maintenance of Multi-source Views." In: *Proceedings of the 12th Australasian Database Conference*. ADC '01. Gold Coast, Queensland, Australia: IEEE Computer Society, 2001, pp. 13–20. ISBN: 0-7695-0966-5.

[64] L. Ding, X. Zhang, and E. A. Rundensteiner. "The MRE Wrapper Approach: Enabling Incremental View Maintenance of Data Warehouses Defined on Multi-relation Information Sources." In: *Proceedings of the 2Nd ACM International Workshop on Data Warehousing and OLAP*. DOLAP '99. Kansas City, Missouri, USA: ACM, 1999, pp. 30–35. ISBN: 1-58113-220-4.

[65] B. Liu, E. A. Rundensteiner, and D. Finkel. "Restructuring Batch View Maintenance Efficiently." In: *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*. CIKM '04. Washington, D.C., USA: ACM, 2004, pp. 228–229. ISBN: 1-58113-874-1.

[66] G. Moerkotte and T. Neumann. "Dynamic Programming Strikes Back." In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 539–552. ISBN: 978-1-60558-102.

[67] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. "Map-reduce-merge: Simplified Relational Data Processing on Large Clusters." In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China: ACM, 2007, pp. 1029–1040. ISBN: 978-1-59593-686-8.

[68] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. "StormAtTwitter." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 147–156. ISBN: 978-1-4503-2376-5.

[69] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. "Nova: Continuous Pig/Hadoop Workflows." In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. Athens, Greece: ACM, 2011, pp. 1081–1090. ISBN: 978-1-4503-0661-4.

Appendices

Appendix A

Proof of consistency

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

P1: View updates are applied *exactly once*.

P2: View updates are processed *atomically* and *isolated*.

P3: Record *timeline* is preserved.

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

A.1 Notation

First, we define the following notation for keys, operations on keys, and the ordering of operations. Let k_x denote a key of a base table A, where $x \in X = \{1, \dots, m\}$, and X is the table's key range. X can be qualified with the name of the table when multiple tables are involved (e.g., X_A for table A, X_C for table C). Further, let an operation on key k_x be

defined as $t[k_x]$. A totally ordered sequence of operations S on a single table of length N is denoted by $\langle t_1[k_{x_1}], t_2[k_{x_2}], t_3[k_{x_3}], \dots, t_N[k_{x_n}]\rangle$, where $\forall i \in \{1, \dots, n\}, x_i \in X$. In other words, this sequence contains operations over an arbitrary key in the base table, where it is possible that a key is updated several times or not at all.

The index i in $t^{(i)}[k_x]$ is used to express a sequence of operations on a single row-key (i.e., the record timeline). For example, a sequence of operations on row-key k_x is denoted as $\langle t^{(1)}[k_x], t^{(2)}[k_x], \dots, t^{(\omega)}[k_x]\rangle$.

The last operation on a particular row-key is always denoted with ω. Note that each operation in the timeline also exists in the overall sequence for the table containing the record: $\forall i \in \{1, \dots, \omega\}, \exists j \in \{1, \dots, n\}, t^{(i)}[k_x] = t_j[k_x]$.

A sequence of operations $S = \langle t_1[k_{x_1}], \dots, t_N[k_{x_n}]\rangle$ over table A produces a sequence of base table states $\langle B_0, \dots, B(t_N[k_{x_n}])\rangle$. B_0 is the initial state of A, and $B(t_n)$ is the state of the table after applying some transaction t_n. We also call the final state of the table B_f, where $B_f = B(t_N[k_{x_n}])$. $B_f(k_x)$ for $x \in X$ denotes the final base table state for a specific key k_x.

Lemma 1*: For any two operation sequences on* k_x*:*
$S_1 = \langle t^{(1)}[k_x], t^{(2)}[k_x], \dots, t^{(\omega)}[k_x]\rangle$ *and*
$S' = \langle t'^{(1)}[k_x], t'^{(2)}[k_x], \dots, t'^{(\omega)}[k_x]\rangle$*. Let* $B_f(k_x), B'_f(k_x)$ *be the final state of* k_x *after applying* S *or* S'*, respectively. Then,* $t^{(\omega)}[k_x] = t'^{(\omega)}[k_x] \iff B_f(k_x) = B'_f(k_x)$.

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

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

A.2 Convergence

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

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

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

One-to-one mapping – selection, projection views define a one-to-one mapping between base and view table. We first prove the following lemma:

Lemma 2: Given a sequence of operations applied for a view (using selection, projection, only) on key k_x: $S = \langle t^{(1)}[k_x], \dots, t^{(\omega)}[k_x] \rangle$. Let $V_f(k_x)$ be the final state of the view on k_x after applying S. Then, $View(t^{(\omega)}[k_x]) = V_f(k_x)$.

Proof. Since selection and projection are both idempotent stateless operations, the final state of the view maintained on k_x is equivalent to applying the view operator on the final operation in the sequence. □

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

We use a similar proof for the Delta operator:

Lemma 3: Given a sequence of operations applied for a view (using DELTA, only) on key k_x: $S = \langle t^{(1)}[k_x], \dots, t^{(\omega-1)}[k_x], t^{(\omega)}[k_x] \rangle$. Let $V_f(k_x)$ be the final state of the view on k_x after applying S. Then, $View(\langle t^{(\omega-1)}[k_x], t^{(\omega)}[k_x] \rangle) = V_f(k_x)$.

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

Suppose that $V_f \neq View(B_f)$ for Delta. Then, $\exists x \in X$, $View(B_f(k_x)) \neq V_f(k_x)$. According to Lemma 3, $t^{(\omega)}[k_x] \neq s^{(\omega)}[k_x] \vee t^{(\omega-1)}[k_x] \neq s^{(\omega-1)}[k_x]$, where $s^{(\omega-1)}[k_x]$, $s^{(\omega)}[k_x]$ are the last two operations processed by the view for k_x. According to properties P1 and P3, the last two operations processed by the view and the base table must be identical and in the same order. Therefore,
$t^{(\omega)}[k_x] = s^{(\omega)}[k_x] \wedge t^{(\omega-1)}[k_x] = s^{(\omega-1)}[k_x]$,
$View(B(\langle t^{(\omega-1)}[k_x], t^{(\omega)}[k_x]\rangle)) = V(\langle s^{(\omega-1)}[k_x], s^{(\omega)}[k_x]\rangle)$,
which contradicts $View(B_f(k_x)) \neq V_f(k_x)$.

Many-to-one mapping – Pre-aggregation, aggregation and index views define a many-to-one mapping between base and view table. The row-key of the view table is the aggregation key. Multiple row-keys in the base table can relate to a particular aggregation key. However, a base table row has always only one aggregation key. A correct view record with aggregation key x is defined as the combination of multiple base records $k_{x_1}..k_{x_j}$, related to the particular key. We prove the following lemma:

Lemma 4: Given a sequence of operations applied for a view (using Pre-aggregation and aggregation) on table A. Let V_f be the final state of the view after applying the sequence. Let S be an arbitrary sequence containing only the last operation on each key $k_x, x \in X$: $S = \langle t^{(\omega)}[k_1], \ldots, t^{(\omega)}[k_m]\rangle$. Then, $View(S) = V_f(k_x)$.

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

Suppose that $V_f \neq View(B_f)$ for Pre-aggregation, aggregation or index. According to Lemma 4, $\exists x \in X$, $t^{(\omega)}[k_x] \neq s^{(\omega)}[k_x]$, where $s^{(\omega)}[k_x]$ is the last operation processed by the view for k_x and $t^{(\omega)}[k_x]$ is the last operation processed by the base table for k_x. According

to properties P1 and P3, the last operation processed by the view and the base table for key k_x must be identical. Therefore, $t^{(\omega)}[k_x] = s^{(\omega)}[k_x]$, $View(B(\langle t^{(\omega)}[k_1], \dots, t^{(\omega)}[k_m]\rangle)) = V(\langle s^{(\omega)}[k_1], \dots, s^{(\omega)}[k_m]\rangle)$, which contradicts $View(B_f) \neq V_f$.

Many-to-many mapping: Reverse-join and join views define a many-to-many mapping between base and view table. The row-key of the view table is a composite key of both join tables' row-keys. Multiple records of both base tables form a set of multiple view records in the view table. Since the joining of tables takes place in the Reverse-join view, we prove convergence only for this view type. A Reverse-join view has a structure that is similar to an aggregation view. The row-key of the Reverse-join view is the join key of both tables. All base table records are grouped according to this join key. But in contrast to an aggregation view, the Reverse-join view combines two base tables to create one view table. A correct view record with join key x is defined as a combination of operations on keys $k_1..k_n$ from join table A and operations on keys $l_1..l_p$ from join table B. This property provides the basis for the following lemma:

Lemma 5: Given a sequence S of operations applied for a view (using Reverse-join and join) on tables A and C. Let V_f be the final state of the view after applying S. Let S be an arbitrary sequence containing only the last operation on each key $k_x, x \in X_A = 1, \dots, m$ in A and each key $y_x, x \in X_B = 1, \dots, m'$ in C: $S = \langle t^{(\omega)}[k_1], \dots, t^{(\omega)}[k_m], t^{(\omega)}[y_1], \dots, t^{(\omega)}[y'_m]\rangle$. Then, $View(S) = V_f$.

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

For Reverse-join and join, convergence is achieved if $V_f = View(B_f, B'_f)$ where B_f, B'_f are the final view states for tables A, C involved in the join, respectively. Suppose that $V_f \neq View(B_f, B'_f)$. According to Lemma 5, $\exists x \in X_A \cup X_C, t^{(\omega)}[k_x] \neq s^{(\omega)}[k_x]$, where $s^{(\omega)}[k_x]$ is the last operation processed by the view for k_x and $t^{(\omega)}[k_x]$ is the last operation processed by a base table for k_x. According to properties P1 and P3, the last operation processed by the view and the base table containing key k_x must be identical.

Therefore, $t^{(\omega)}[k_x] = s^{(\omega)}[k_x]$, which contradicts $View(B(B_f, B'_f)) \neq V_f$.

A.3 Weak consistency

Weak consistency has been defined as follows: Weak consistency is given if the view converges and all intermediate view states are valid, meaning there exists a valid sequence of operations from which they can be derived from the state of one or more base tables ($V_j = View(B_i, \dots, B_k)$). As we already proved convergence, we need to show that all the intermediary view states are correct.

For view expressions which depend on one key in table A (excluding DELTA), suppose that for some intermediate view state V_j, $\forall i \in 1, \dots, N, V_j \neq View(B_i)$. Let $T = \langle t_1[k_{x_1}], t_2[k_{x_2}], t_3[k_{x_3}], \dots, t_N[k_{x_n}] \rangle$ be the sequence of operations applied on A. Let s_j be the operation that produced V_j: $V(s_j) = V_j$. By Lemmas 2, $\forall i \in 1, \dots, N, s_j \neq t_i$. In other words, the view processed an operation which was never processed by the originating base table. However, property P1 and P2 ensure that each operation corresponds to a base table operation and is fully processed. By contradiction, $V_j = View(B_i)$.

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

For view expressions which depend on multiple keys in table A, suppose that for some intermediate view state V_j, $\forall i \in 1, \dots, N, V_j \neq View(B_i)$. Let $T = \langle t_1[k_{x_1}], t_2[k_{x_2}], t_3[k_{x_3}], \dots, t_N[k_{x_n}] \rangle$ be the sequence of operations applied on A. According to Lemma 4, ;et S be an arbitrary sequence containing only one operation on each key $k_x, x \in X_A = 1, \dots, m$ such that $V(S) = V_j$. According to the definition of intermediate view states, $\exists k_x, x \in X_A, t(k_x) \in S, t(k_x) \notin T$. In other words, there exist at least one operation on some key in A which was processed by the view, but not by the base table. However, property P1 and P2 ensure that each operation corresponds to a base table operation and is fully processed. By contradiction, $V_j = View(B_i)$.

For Reverse-join and join on tables A and C, the argument is similar to selection and projection. According to Lemma 5, every state V_j is created on a pair of operations of A and C. Using property P1, it is guaranteed that both operations have been previously processed in the originating tables.

A.4 Strong consistency

Strong consistency has been defined as follows: Weak consistency is achieved and the following conditions hold true. All pairs of view states V_i and V_j that are in a relation $V_i \leq V_j$ are derived from base states B_i and B_j that are also in a relation $B_i \leq B_j$. Since weak consistency is already proven, we only need to prove the statement $V_i \leq V_j \Longrightarrow B_i \leq B_j$. If this statement is false, then only two of the following cases can occur: Either $V_i \leq V_j \Longrightarrow B_i \parallel B_j$ or $V_i \leq V_j \Longrightarrow B_i \geq B_j$. Both cases can only be constructed by breaking the record timeline. To be precise: At least one record has to exists, whose timeline is broken. Formally, we demand $(\exists t_l \in B_i)(\forall t_k \in B_j) : (r(t_l) = r(t_k)) \wedge (l > k)$. Because P3 prevents the breaking of timelines, we conclude that both cases are not possible. Thus, we have proven strong consistency by contradiction.