Dynamic Scalable View Maintenance

Jan Adler[*]
Technische Universität
München
1932 Wallamaloo Lane
Wallamaloo, New Zealand
j.adler@intum.de

ABSTRACT

Todays internet platforms like twitter and facebook serve millions of user and process large amounts of data. The inherent distributed architecture of those systems supports performance, scalibility and fault tolerance. However, the lack of consistency quarantees and the distributed nature of the systems make it difficult to analyse the collected data. Especially in the case of view maintenance the technical requirements have changed and the techniques once applied to stand-alone data warehouses is today not powerful anymore. We propose a system that allows scalable and dynamic calculation of views even in the scenario mentioned previously. Taking away the burden of view maintenance from the storage servers we share the maintenance work among a variable set of view manager components. We integrate our solution into an existing distributed data store and show how concerns of consistency, update distribution and failure recovery can be handled.

1. INTRODUCTION

Views are just another term for analyzing data in a database system. In the background of large scale internet-platforms views are essential to make information available. The most recent data in those systems is usually kept in fast accessible and small databases. In constrast, data which remains in the system longer than a day, is considered old. Nevertheless the old data, which represents the significantly largest amount of the data volume, is kept in huge distributed storages. It is selected, aggregated and joined in order to keep it available to the user. Moreover the old data is evaluated to gain information about the systems or users behaviour.

To accelerate access to aggregated data materialized views have always been a vital option. Materialized views enable multiple users to retrieve information rapidly on the one side, but introduce the problem of view maintenance on the other side. In the field of research on view maintenance two kinds of maintenance strategies have become apparent. Immediate and deffered view maintenance. Since we are dealing with heavy loaded servers it is in our interest to deffer the view calculation from the storage servers to another network node. This process is called propagation.

[*]Dr. Trovato insisted his name be first.

The approach we choose integrates the concept of incremental, deffered view maintenance into an exisiting distributed storage system. A distributed storage offers performance, scalability and recovery mechanisms in the presence of failure. We want to inherit those desirable properties to our view maintanence system. Like for a distributed key value store our system should be able to handle every size of workload. There should be a component of scalibility that can process view updates and that can be spawned in arbitrary numbers. Likewise the view maintenance system should have the ability to shift the workloads between the components to achieve an efficient usage of resources.

Simultanously we want to take care of the negative effects, that can appear when calculating views of distributed data sets incrementally. The lack of global synchronization and transactional guarantees in distributed database systems can easily lead to inconsistent data. Since corectness of a calculated view is a crucial aspect we want to assure it all cost. However we don't have to provide complete consistency. To identify the needed consistency level we will use a consistency model and project it onto our systems architecture.

When operating large scale internet platforms we can state that failure is not the exception but the norm. Especially in the case of horizontal partitioning, commodity hardware is likely to fail. Therefore we want to equip our view maintenance system with recovery mechanims. The system should be able to detect crashed components and recover from lost view updates.

2. RELATED WORK

There has been a lot of research in the field of view maintenance with regard to materialized views. This research has layed the foundations to caclulate materialized views efficiently [1–5]. However, some of researched concepts are not valid anymor but others can be easily adapted to fit the requirements of large scale distributed data storages.

Materialized views can be refreshed by recalculating the complete view or by updating those records of the view which are affected by base table updates. This very efficient way of recomputation is called incremental view maintenance. Incremental view maintenance can save a lot of computing time but in return is a big challange towards consistency. Lots of efforts have been made to prevent update anomalies when using incremental view maintenance [2,3,6].

There have been used versioning and timestamp based approaches to determine the delta of a view [6, 7]. We cannot rely on those mechanisms because time is not synchronized and transactions are not serializable in our distributed system. However, we can use sort of a local versioning.

For calculating materialized views two strategies of view maintenance have proven to be relevant: Immediate and deffered view maintenance. When immediate view maintenance is applied the update of the viewtable is performed as part of the basetable update. For deffered view maintenance on the contrary, the update of the view table is shifted to a later point in time. Deffered view maintenance decouples the base table from the view tables. This is extra valuable if multiple viewtables are derrived from one base table or the base table system should not know anything about the views referenced.

Especially in the context of datawarehouses incremental, deffered view maintenance has been researched extensively. to separate data sources from the warehouse [3–5, 8, 9]

There is also some more recent research work to immluminate the challanges of view maintenance in the context of large scale internet environments. [7, 10–12].

The research in [10] solves concsistency problems in distributed data sets and also covers the questions of view synchronization. consistency is established by using logical clocks rather than timestamps. The transactions are serialized with the help of transaction ids. Even this is promising, there is a lot of information that has to be kept at the data sources, producing additional overhead. Moreover

The approach of

In [12]vthe theoretical foundations of our work can be found. Here the term view manager as an independent component for updating views is defined for the first time. We also use the algorithms that are suggested in this paper to ensure consistency in our system.

The advantages of consistent hashing has been examined in [13]. Consistent Hashing is a principle which consistently maps resources to servers or caches. When a resource is delivered to the system the hashring determines which network node is responsible for it. In contrast to a standard modulo function the consistent hashring allows adding resources, without reassigning most of the resources to other network nodes.

[1–19].

3. ARCHITECTURE

3.1 System Overview

In figure 1 you can see the overview of the complete system. On top of the architecture resides the client issuing updates to the basetable. Since we are aiming at using a distritubed key value store, the clients can only issue put and delete operations.

The basetable stores the base data which is used to calculate the view tables. Since all data in the basetable is depending on a primary key, the basetable can be split into key ranges. A key range is called a region. Regions are used to distribute the base table over many region servers and to minimize the access time to parts of the table.

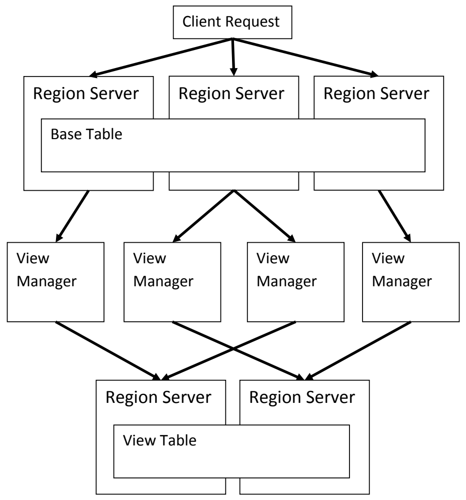

Figure 1: system overview

The region servers are part of a distribute storage system. They act as a container for the regions and they can be deployed in any number. A region server can hold multiple key regions at a time. Key regions can be moved between region servers to balance the load of the system or to achieve a smooth distribution. We will not have to implement the region server architecture by ourself because there several existing system. Nevertheless we have to extend the region server functionality to integrate it into our scalable view maintenance system.

We are using incremental, deffered view maintenance. Therefore the updates on the basetable are not processed at the region servers. Instead they are propagated to the view managers. The view managers remove the burden of updating the view from the region servers. Once the view manager has received a basetable update it retrieves the viewtable record. Then it calculates the new view record and updates the viewtable.

A view manager is the atomic unit of scalability. The relationsship between region servers and view managers is a one-to-many relationsship. Multiple view managers can be attached to a region server to parallelize the update processing of the region server. A view manager is only responsible for one region server. A view manager is always able to perform any update operation on any viewtable. Since we are building the system with regard to scalability view managers should be exchangable at any time.

The distributed nature of our storage system doesn't provide us a powerful query language. Nevertheless we are able to create viewtables of the following types: count, sum, min, max, join, selection. In our system numerous view tables can be calculated out of a base table at a time. Like the basetable, the viewtables can also be split in multiple regions and distributed over the region servers. There is nothing said about where viewtables are located. Key regions of base and viewtables can exist on the same region server at any network node.

3.2 Hbase

For storing our base table updates we want to use a well known and established distributed storage system. Hbase is the open source version of google's Big Table. The architecture of Hbase acts representativley for most types of distributed storage systems. It implements the already introduced region server concept. The architecture aims particularly at performance, scalibility and fault tolerance. Therefore we use the systems architecture as the fundament of our work.

As most of the distributed key value stores Hbase is very limited in terms of calculating views. Obviously calculating a view by scanning over the whole base table is not a good option. Another possibility that can be used in Hbase are update hooks. Here a piece of program code is deployed on every region server. When issuing a put operation the hook is called before the insert and updates the appropriate view table. This is a more efficient solution since we refresh the view locally and recordwise. But on the downside the update is not deffered and the region server is still responsible for calculating the view table. Especially in a scenario where hundereds of views have to be updated this will overload the region servers.

Hbase is equiped with a master component. The master component is not part of the data path. It is just responsible for controling the region servers. By performing assign and remove commands it relocates regions and ensures a uniformly distributed load. We want to use a similar mechanism to manage our view managers. The view managers in our system should be attached to the region servers dynamically with regard to the actual base table update situation. Moreover there should be a master, that takes care of the systems running condition. In case of a view manager crashing the master should take measures to qurantee an errorless recovery.

Hbase uses the coordination service zookeeper [16] to store configuration information and to detect failure of single components. In addition zookeeper is the first contact point for all clients of the the storage system. Because zookeeper has also a scalable architecture the client requests can be routed one node of a zookeeper ensemble.

3.3 Master component

To control our view maintenance process and to balance the view managers we use a master component that is similar to the master of hbase. It continously monitors the systems state and is able to control the components remotely. The master component consists of four subcomponents: the event processor, the load balancer, the recovery manager and the component controller.

The event processor observes the events occuring in the system. If any system component is added, removed or crashes the event processor is informed. Then it creates an event and passes it either to the load balancer or the recovery manager. All typical procedures like adding and removing components are forwarded to the load balancer, while the crash event is handed over to the recovery manager.

The load balancer tries to balance the systems load by assigning view managers to the region servers. The load balancer is always called when a component is added, removed or has been crashed. Likewise it is able to perform a temporary load balancing and shift the hardly loaded or idle view managers from one region server to another. To calculate the load plan, the load balancer needs to be informed about the utilization of the components. Therefore the number of updates that are currently pipelined is aggregated in a treewise fashion from the view manager to the region server to master node. The load balancer holds a reference to compnent controller to execute its load plan.

The recovery manager is informed in case one of the components crashes. When a view manager crashes the recovery manager performs an action to find out the last processed updates. It then informs the region server to replay and reassign the updates that might have been lost during the crash of the view manager.

The component controller represents the masters interface to control the entire view maintenance system. It contacts the compenents and uses a message protocol to transmit commands. The component controler is used to realize the caclulated load balancing actions. It also receives confirm messages from the components to verify that the given instructions. This is vital to keep a consistent reference of the systems state.

3.4 region server extension

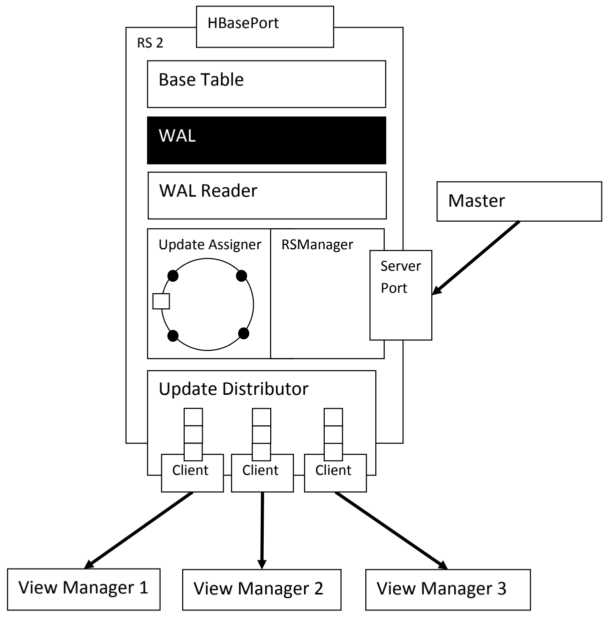

Figure 2: region server extension

The region server of our view maintenance system extends the architecture of a region server of hbase. The extended region server is then able to, not only persist the update operations of the clients, but also distribute them among any number of registered view managers.

The basic region server implementation of hbase consists of a write-ahead- log and multiple regions.

Then the client transmits the update to the responsible region server. When client updates arrive at the region server they are put to the Write Ahead Log where they are immediatly Since memory data is volatile the region server first needs to write the data to a write-ahead-log. memstore of the region server to keep the recent data in memory and guarantee fast access. The updates are not flushed to disk before the memstore exceeds its capacity. In the moment the update is written to the log it can be considered as persisted safely. The write-ahead-log is then written to an underlying distributed file system and replicated multiple times.

The components, which are added to the region servers for view maintenance purposes, can be seen in figure 2: the wal-reader, the update assigner, the update distributor. These components are processing the base table updates sequentially in the mentioned order.

The wal-reader compenent can just connect to the file system and start reading the updates. By doing so we can directly propagate the updates without interferring the region server at all. The updates are listed in the write-ahead-log in strict fifo order and they are tagged with a for the region server unique, sequential number. Later, this helps us to globally identify the updates.

The WAL-Reader acts like a pointer to the write-ahead-log of the hbase region server. It constantly polls the log-folder of the region server. If it has found new update entries it points to the line in the log-file and starts reading the updates sequentially. If problems occur during update propagation the WAL-Reader is able to jump back to a previous line of the write-ahead-log and to read the updates several times. This procedure is called replaying the write-ahead-log. After the WAL-Reader has extracted the updates it forwards them to the update assigner.

The update assigner is responsible for assigning the updates to the view managers. It does so by using a consistent hashring. Every view manager, which registers at the region server, is put on the hashring by calculating a hash-value from its systemID. If a view manager leaves it is removed from the hashring. To assign a key to a view manager again a hash-value is calculated out of the The update is tagged appropriatly.

The update distributor creates a separate queue for every view manager. Depending on the assigment of the update is put to the corresponding queue. From there the updates are sent to the view managers in fifo order.

On the one side the rs-controller is conducting all internal components, on the other side it is the interface through which other components can talk to the region server. The master for example uses the rs-controller to remotely controll the functions of the region server, for example.

3.5 view manager

The view manager is our component of scalibility and is able to process view updates at record level.

The view manager consists of the following subcomponents: the pre-processor, the update processor and the vm-controller.

The pre-processor resolves the view tables that are mapped to a base table update. This procedure can change the number of processed updates dramatically, especially if a lot of views are assinged to one base table. For that reason the pre-processor measures the current load of the view manager. The information about the load state is then send to the region servers periodically. Because the

The update-processor performs the actual work of computing the view deltas. It can retrieve view records from the view tables by using the aggregation key. The update-processor then queries the base table. This is an optional step since most of the updates are local. For example all

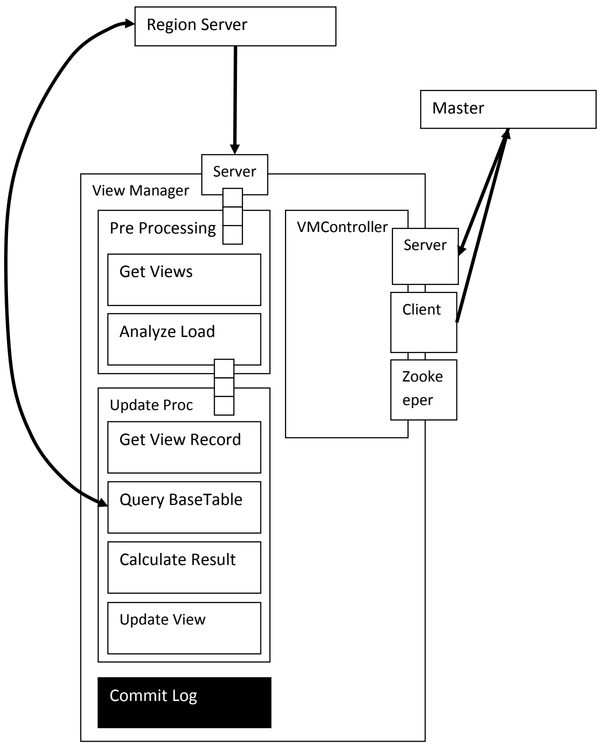

Figure 3: view manager

update operations on a sum view are local because only the difference of the client update has to be added or subtracted from the view table record. A global operation would be the deletion of a current minium in a in view. Then the whole base table had to be queried to find a new minimum.

As the most important step the update-processor calculates the new value of the view record and updates the view. Because the view can also be a distributed dataset the update-processor may contact multiple region servers.

Additionally the update-processor keeps writing to a commit log. There it puts the base table updates indicating that they have already been processed. The commit log is like the write-ahead-log written into the distributed file system laying beneath the database.

The vmcontroller acts just like the rscontroller as an internal manager and an external interface.

3.6 system operations

To regulate the view maintenance system we have to define the possible system operations. The system operations should leave our vm system in a consistent state and therefore we demand that they are running according to the ACID principles. System operations can be either triggered by system events or by the master to transform the systems state. The master keeps a model of the system stored in the systems configuration. This model contains a list of view manager, region servers and a map of assignments, showing which view manager has been assigned to which region server.

3.6.1 add view manager

Adding describes the process of adding a new view manager resource to the view maintenance system. When a view manager is added to the system primarily it contacts the zookeeper ensemble. To register at the view maintenance system it creates a session node in the view manager branch of zookeeper. The session nodes are called ephemeral nodes in zookeeper. The master component keeps an observer registered to the view manager branch and is informed as soon as a new node is created. When the master notices the creation of a new view manager it updates the system configuration and immediately tries to use the view manager when calculating the load balacing plan.

3.6.2 *remove view manager*

Removing describes the process of taking away a view manager resource from the view maintenance system. A remove operation can either by initiated by the master or by the viewmanager. In both cases the view manager proceeds by executing a withdraw operation to deregister itself from the region server.

3.6.3 *assign view manager*

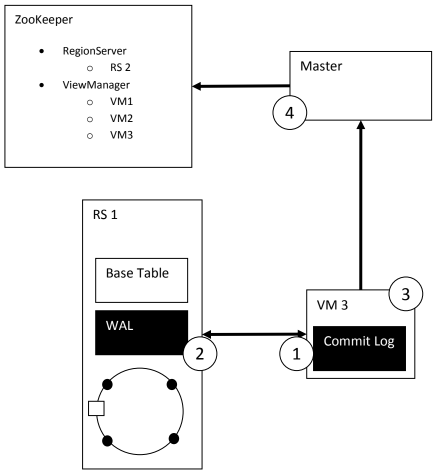

Figure 4: assign view manager

Assigning a view manager describes the procedure of registering it to a region server. We don't want to overburden the master with communication tasks. Especially when scaling to several hundreds of nodes this aspect is critical. Hence the master only initiates the process by sending an assignment command, including the targeted region server as a parameter, to the view manager. The view manager then handles the assignment by himself. He sends a request message to the region server. The region server then initializes the new view manager by putting it onto its hashring. It creates a new message queue and then starts to deliver base table updates to the view manager.

3.6.4 *withdraw view manager*

As we have seen, no matter how an operation is performed. I it is always commited by the master updating the systems configuration accordingly. This is supporting atomicity, because if a system operation somehow fails then the system configuration is not updated. Only a completely successful operation is recorded by the master.Isolation of system operations is quaranteed by the masters component controller. The component controller executes the commands, which are performed as part of the system operations, one after the other. There is a property specifying a fixed number of retries in case a component is not answering. A view manager changes its state from "running" to "assigning" when receiving an assign command. In case a second command is sent, while the view manager is executing the first command, then the former is beeing denied by the view manager. However, the system operations are not perfectly consistent. A system operation, like the assignment of a view manager, may be executed successfully and then the final message to inform the master component gets lost. The system state will not be reflected in the systems configuration then. But regarding the case we can also conclude that the systems operability will not be hurt. The region server will forward updates to the view manager, which will process the updates typically. As soon as the master performs the next load balancing it will notice an idle view manager and reassign it reasonably. Ideally the systems configuration is persisted safely by storing it to the zookeeper, guaranteeing durability of the system operations.

4. VIEW CONSISTENCY

4.1 consistency model

In order to measure and change our level of consistency we need to define a consistency model. Since there has already been research on consistency in view maintenance systems, e.g. in [8,12], we adopt a consistency model and apply it to our architecture.

In our consistency model B represents our base table, whereas V represents our view table. Both tables change as we are starting to apply updates to them. Therefore we use the notion B_i and V_j to specify the state of the respective table. Because we are interested in view consistency at record level we write $B_i(r)$ and $V_i(r)$ to indicate a state of a single record of the table.

4.2 non idempotent updates

In our distributed view maintenance system updates are sent over the underlying network several times. Since we can not always rely on perfect transmisson message have a chance of beeing duplicated. Also in our process of recovering from view manager crashes we are forced to replay the write ahead log of the region server. In both cases we end up with an exactly similar update sent multiple times to a view manager. Usually this hasn't any impact on selection and join views because the record is inserted once. If the same update comes again the old view record is just substituted, the view remains in the same state. What happens in the case of a sum view is shown in the following example. We define a base table $R(\underline{K}, X, Y)$ and view table $D(\underline{X}, S)$. The view is defined as

```
SELECT SUM(Y) FROM R GROUP BY X
```

The initial table states are $B_0\{R(k1, x1, 50)\}$ and $V_0\{D(x1, 50)\}$. There is one insert $i_1(R(k2, x1, 100))$. Because of transmission problems or recovery actions insert i_1 gets duplicated and is transferred to view manager VM_1 twice. VM_1 also updates the view record twice leading to a final view table state $V_f\{(x1, 250)\}$ where it should be $V_f\{(x1, 150)\}$.

To solve the problem of idempotent view updates we have to use signatures of view records. Signatures are a suggestion in [12]. The signature reflects the state of a view records. Every time a base table update is applied to the view record the signature is recalulated by appending the id of the base table update. If the recalculation of the signature produces the same result, the base table update has already been applied. The id of the base table update should uniquely identify the update across the whole system.

In our hbase architecture we can easily derive a global id for the base table updates. A region server orders its base table updates by using a sequential number. When an update is written to the write ahead log the sequence number is increased by one. The sequence number is only a local identifier but if we put it together with the id of the region server we receive a global id.

4.3 wrong update order

If we want to establish consistency in a view maintenance system it is vital to stick to the order in which the base table updates are inserted at the region server. In a distributed storage system like hbase, there is nothing like a global time. It is impossible to serialize the base table transactions across multiple region servers. However, for most classes of views it is enough to preserve the timeline of a single record. In the moment this requirement is violated then even convergence cannot be quaranteed. For example if we have a selection view with a base table $R(\underline{K}, Y)$ and view table $S(\underline{K}, Y)$. The view is defined as

```
SELECT K,Y FROM R WHERE Y < 300
```

There are two updates $u_1(R(k1, 100))$ and $u_2(R(k1, 200))$. view manager VM_1 propagates update u_1 and VM_2 propagates u_2 delivering it faster than VM_1. This means the updates are inserted into the base table in right order but into the view table in wrong order. The final table states are $B_f\{(k1, 200)\}$ and $V_f\{(k1, 100)\}$.

Another example, showing the importance of record timeliness is the following: A join view with two base tables $R_1(\underline{K}, X)$, $R_2(\underline{K}, Y)$ and view table $S(\underline{K}, X, Y)$.

```
SELECT K,X,Y FROM R1,R2 WHERE R1.K = R2.K
```

The initial table states are $B_0\{R_2(k1, 200)\}$ and $V_0\{\}$. There are two updates $i_1(R_1(k1, x1, 100))$ and $d_2(R_1(k1, x1, 100))$. View manager VM_1 propagates update u_1 and VM_2 propagates d_2 delivering it again faster than VM_1. This means the deletion d_1 of the row is delivered before the insertion i_1, leading to divergent final states $B_f\{\}$ and $V_f\{(k1, x1100)\}$.

The examples show the importance of record timeliness. The records of a single region server are always written to the write-ahead-log in fifo order. This means we receive the updates in the right order. If we are able to route the updates made to a single base table row always to the same region server, then we can quarantee record timeliness. In the following we will call this mechanism *flow control*. To establish flow control the consistent hashring used in the update assigner component of the region server suits just fine. The hash value, which is calculated out of the key of the base table update, remains the same if the keys remains the same. Therefore multiple updates on a base table record get always assigned to the same view manager.
This is an easy procedure as long as we are bound to a static context.In a dynamic context view managers can be assinged to or withdrawn from the region server while the stream of updates is continueing. Then there is a short period of time, in which the order of updates can be changed. For example, if an update u_1 is taken from the write ahead log. In the next step the update assigner calculates a hash value $h(k_1)$ from key $k1$. The hash value of the view manager $h(VM_1)$, who is hit first in the hashring, is determined responsible. Another view manager joins with a hash value $h(VM_2)$ lying between $h(k_1)$ and $h(VM_1)$. The next update u_2 on the same key $k1$ is propagated. The view manager determined responsible is now VM_2. Taking into account that VM_1 has already queued a lot of updates and VM_2 is just processing its first update, we can assume that u_2 is likely to pass u_1. Again the record timeline has been broken. To overcome this situation we establish a control mechanism. If a new view manager gets registers at the region server, then the region server put it on the hashring like in normal operation mode. But the assinged updates of the new view manager are not delivered. Instead the message queue of the new view manager is put on hold and a marker message m_1 is inserted into the stream of all remaining view managers. The view managers are processing their update streams and after a while they entcounter the marker message m_1. They react to the message by sending an acknowledgement to the region server. After the region server has collected all marker messages m_1 from the view managers it releases the previously queued updates. Now it can be sure, that no update of k_1 is still in progress at any of the view managers.

4.4 concurrent commit

Our view maintenance architecture aims at a highly parallelizing updates. As we already discovered we can use flow control to sequentially process updates of a single base table record. But views aggregate the base table data and therefore updates of multiple base table records are merged together. These updates can of course be processed in parallel. A single view record can be updated by two different view managers concurrently. This can violate the convergence of a view table as the following example shows. Imagine a base table $R(\underline{K}, X, Y)$ and view table $D(\underline{X}, S)$. The view is defined as

```
SELECT SUM(Y) FROM R GROUP BY X
```

The initial table states are $B_0\{R(k1, x1, 50)\}$ and $V_0\{D(x1, 50)\}$. There are two inserts $i_1(R(k2, x1, 100))$ and $i_2(R(k2, x1, 200))$. view manager VM_1 propagates update i_1 and VM_2 propagates simultaneously i_2. Since the keys of both insert transactions are different this is not violating our record timeline. Both view managers retrieve the view record $D(x1, 50)$. Then they add the delta of their insert operation and write back the view record. Since both view updates are calculate on the same view record value one of the updates gets lost, in our case its i_2.

4.5 update interference

Update interference is a long existing issue in the reasearch of view maintenance. As mentioned above, when calculating updates incrementally, there are local and global transaction. Examples for global transactions is the delete operation of a min/max-view or the insert operation of a join view.

In case of the min/max-view we need to compute a new global minum/maximum. In case of the join view we have to globally query the base table to determine the matching rows of the join. However, when querying a large set of base table updates it can happen that base table records are updated meanwhile. Then the updates are included twice. They are in the result of the predecessing query and they trigger an update by themselves. For the min/max-view this leads to a violation of strong consistency as following example shows. For the join view results are equal as this example shows:

5. LOAD DISTRIBUTION

5.1 update distribution

Especially in the case of load balancing the operations should be executed isolated. A new system state should only be calculated by the load balancer if the system has establish the last demanded state.

5.2 global load balancing

5.3 local load balancing

6. FAILURE DETECTION

There are to cases we want to distinguish

7. IMPLEMENTATION

8. EVALUATION

9. CONCLUSIONS

This paragraph will end the body of this sample document. Remember that you might still have Acknowledgments or Appendices; brief samples of these follow. There is still the Bibliography to deal with; and we will make a disclaimer about that here: with the exception of the reference to the LaTeX book, the citations in this paper are to articles which have nothing to do with the present subject and are used as examples only.

10. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and

11. ADDITIONAL AUTHORS

Additional authors: John Smith (The Thørväld Group, `jsmith@affiliation.org`), Julius P. Kumquat (The Kumquat Consortium, `jpkumquat@consortium.net`), and Ahmet Sacan (Drexel University, `ahmetdevel@gmail.com`)

12. REFERENCES

[1] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, June 1986.

[2] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *SIGMOD Rec.*, 22(2):157–166, June 1993.

[3] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. *SIGMOD Rec.*, 24(2):316–327, May 1995.

[4] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. *SIGMOD Rec.*, 25(2):469–480, June 1996.

[5] Hui Wang, Maria Orlowska, and Weifa Liang. Efficient refreshment of materialized views with multiple sources. In *Proceedings of the Eighth International Conference on Information and Knowledge Management*, CIKM '99, pages 375–382, New York, NY, USA, 1999. ACM.

[6] Kenneth Salem, Kevin Beyer, Bruce Lindsay, and Roberta Cochrane. How to roll a join: Asynchronous incremental view maintenance. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 129–140, New York, NY, USA, 2000. ACM.

[7] Jingren Zhou, Per-Ake Larson, and Hicham G. Elmongui. Lazy maintenance of materialized views. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 231–242. VLDB Endowment, 2007.

[8] Yue Zhuge, Hector Garcia-Molina, and Janet 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, pages 146–157, Washington, DC, USA, 1996. IEEE Computer Society.

[9] Ki Yong Lee, Jin Hyun Son, and Myoung Ho Kim. Efficient incremental view maintenance in data warehouses. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, CIKM '01, pages 349–356, New York, NY, USA, 2001. ACM.

[10] Songting Chen, Bin Liu, and Elke A. Rundensteiner. Multiversion-based view maintenance over distributed data sources. *ACM Trans. Database Syst.*, 29(4):675–709, December 2004.

[11] Parag Agrawal, Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, and Raghu Ramakrishnan. Asynchronous view maintenance for vlsd databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 179–192, New York, NY, USA, 2009. ACM.

[12] Ramana Yerneni Hans-Arno Jacobsen, Patric Lee. View maintenance in web data platforms. *ACM Trans. Program. Lang. Syst.*, 15(5):795–825, November 2009.

[13] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. In *Proceedings of the Eighth International Conference on World Wide Web*, WWW '99, pages 1203–1213, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[14] Xin Zhang, Lingli Ding, and Elke A. Rundensteiner. Parallel multisource view maintenance. *The VLDB Journal*, 13(1):22–48, January 2004.

[15] Vamshi Krishna Konishetty, K. Arun Kumar, Kaladhar Voruganti, and G. V. Prabhakara Rao. Implementation and evaluation of scalable data structure over hbase. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ICACCI '12, pages 1010–1018, New York, NY, USA, 2012. ACM.

[16] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[18] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

12.1 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references).

APPENDIX

You can use an appendix for optional proofs or details of your evaluation which are not absolutely necessary to the core understanding of your paper.

A. FINAL THOUGHTS ON GOOD LAYOUT

Please use readable font sizes in the figures and graphs. Avoid tempering with the correct border values, and the spacing (and format) of both text and captions of the PVLDB format (e.g. captions are bold).

At the end, please check for an overall pleasant layout, e.g. by ensuring a readable and logical positioning of any floating figures and tables. Please also check for any line overflows, which are only allowed in extraordinary circumstances (such as wide formulas or URLs where a line wrap would be counterintuitive).

Use the `balance` package together with a `\balance` command at the end of your document to ensure that the last page has balanced (i.e. same length) columns.