Masterpraktikum: Internet Scale Distributed Systems

Technische Universität München

View Maintenance in Web Data Platforms

-Technical Report-

Jan Adler

Betreuer: Martin Jergler

Contents

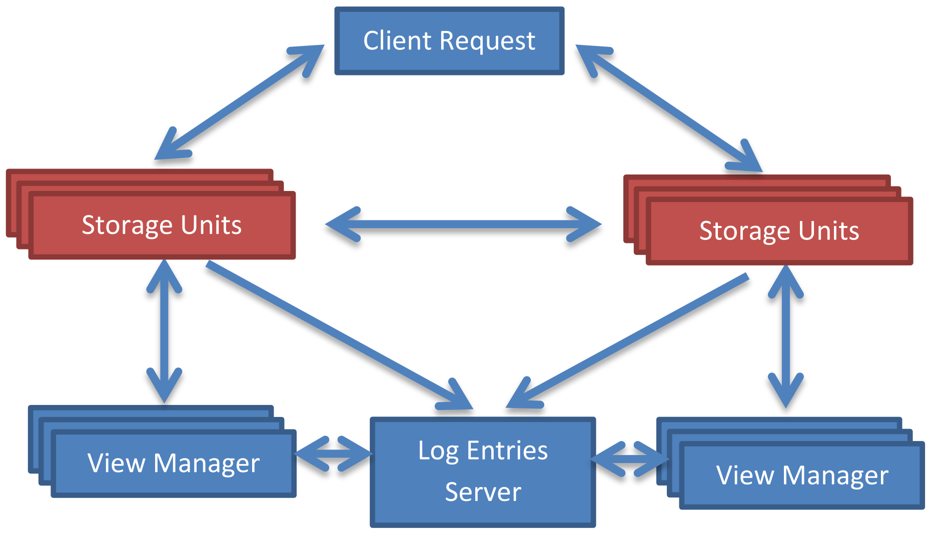
Pictures

Basics

Web Data Platforms

Web Data Platforms are a product of the progress in the IT Industry over the past few years. Today large Scale Internet Services, for example Google Maps, Facebook or Twitter are used by many people over the world. The user numbers have increased strongly, growing from thousands to millions of users. Moreover the exchanged data has become larger, consisting not only of text but also of photo and video messages. For that reason the client-server architectures and relational database system that have been used until now cannot provide sufficient performance to serve the continuously increasing load and store the big amounts of data.

Web data platforms are usually distributed system with multiple servers running in a cluster. Instead of using high-performance servers web data platforms are often realized with the help of commodity hardware. The web data platforms should be easy to maintain and scalable with regard to the amount of running servers and the served users. Therefore one of the main design goals for web data platforms is to have components that communicate locally and handle their work independently. Centrality means always a possible bottleneck exists in the system and whenever a central component fails the whole system will crash. In Picture 1 a typical structure for a web data platform is shown (Jacobsen, Lee, & Yerneni, 2009).



Picture 1: Architecture of a Web Data Platform

Views

Views are a well-known terminology since the development of relational database systems. Since there are a lot of queries on the data in a database system, views can be computed. To watch it this way a view is just used to make a query accessible under a certain name. This makes life easier for a programmer because he doesn't need to repeat queries that are frequently used. Moreover it reduces complexity to have a concisely name for a query which hides the details, for example if the query spans over multiple tables and contains a lot of conditions.

But beyond the practical aspects, a view represents a logical concept. This becomes clear when thinking about the query that is executed by the view. The query on the tables of a database system is nothing more than a table by itself. The query selects aggregates or attaches data but the result set is always a number of data rows which can be stored like a table. The only difference to a data table is that the data of the view is calculated out of existing data in other tables. So the view can be seen as angle of view on the data. In further descriptions the term view table refers to a table representing a view and the term base table refers to all data tables.

Key Value Stores

No-SQL data stores are a new kind of database concept which has gained importance because of the growing user numbers and data amounts. Key value stores have been created with focus on performance and flexibility. They are built to run in distributed environments. The key value store is designed in a distributed fashion. This architecture allows the definition of a table that spans over multiple servers and contains big amounts of data, a scenario where a classical relational database would exceed its limits.

The key value store can be seen like a simple Java map which allows the operations put, get and delete to manipulate the data of the store. The stored value is always referenced via the appropriate primary key (key value). The data itself is rather unstructured. Specification of data types and relations is avoided. The system has to be flexible to split the data into any number of partitions. If a table grows too big then the key range of the table is automatically split into two parts. If the new parts grow again then the procedure is repeated. This guarantees a fast access to the table data and enables the database system to host the different table parts on different network nodes.

The architecture of the key value store consists of one master and multiple region servers. The master server handles the client requests. If a client request comes in to put or get the value of a particular key, the master server routes the request to the region server that manages the appropriate key range. The region servers then answer the request of the client. They can manage one or more key ranges and they can be deployed and removed on the fly.

Moreover key value stores possess suitable features for the development of distributed systems. For example code can be executed directly on the different region servers of the key value store. With the help of this feature results can be calculated locally and then sent to client and added up globally.

Analysis

View Maintenence

Views are an integral part of relational databases. Nevertheless views haven’t been used a lot by programmers for various reasons. In MySQL for example the view is recalculated during every client request. This corresponds to a query executed for every request. In a small application with little users this may be sufficient. However, in the context of web data platforms repeated tables scans are an unfeasible solution.

There have been approaches to use materialized views and store the view tables directly on disk like normal tables. This improves performance significantly since the view is not recalculated on every request. It is only updated if an insert/update/delete-command has been executed on the data of the base table. In the context of web data platforms a materialized view would save a lot of needless recalculations, which is desired in terms of high user numbers and millions of accesses per minute/second. But again there is the downside of recalculation. A complete recalculation of the view table has to be exercised if the data changes in the base table. Thinking of large amounts of data and a high number of write operations to the data in the base table, materialized views will not lead to satisfying results.

To achieve both goals at a time, high access and large amounts of data being written, the update procedure of the view tables has to be kept as small as possible. This is only possible if the view table is updated piecewise. For every base table insert/update/delete-command being processed the corresponding entry in the view table has to be determined and manipulated. This approach guarantees a minimal amount of recalculation. It allows a high number of write operations to the data of the base table, while simultaneously providing fast access to the aggregated data in the view tables.

Aggregation

Aggregation views are all kinds of views where data of the base table is merged depending on a particular key. The data can be merged as sum, count, min, max or average of the base table records. Technically the update procedure remains the same for all cases. By taking the sum for example, the sum over all base records with the same aggregation key is calculated. When using piecewise updates the aggregation key of the base record is looked up in the view table. If the corresponding view record is found then it is increased or reduced by the delta of the aggregation value.

Selection

In selection views base records are just selected according to a condition and then they are put into the view table. As we will see a selection view can select a range of base table values which then leads to the non-idempotent view update problem.

Join

Joins are the most complex case considering view table maintenance. Because at least two tables with two primary keys are involved in a join a lot of special cases have to be considered. To reduce the complexity the join possibilities have to be limited. For a first try we will only consider key-foreign key join view (KF-K)

Consistency

By introducing views immediately the concern of view table maintenance comes up. Every time data is kept redundant, even if it is only calculated data, consistency problems can occur. This may be because the calculated data is wrong updated or it is updated not at all when the underlying numbers of the calculation change. A distributed system like a web data platform can consist of multiple network nodes and many unreliable network connections. In such an environment the described problems get even worse. Packages can be lost or duplicated in the network. Single nodes can fail and cause severe problems with consistency.

Furthermore the CAP theorem states that the three properties consistency, availability and performance in distributed systems have a strong influence on each other. By having two of them at a time the third one is nearly impossible to achieve. As mentioned before, the approach described in this paper uses the possibility of piecewise updates to gain performance and availability. As a result consistency is sacrificed to a certain degree. This becomes clear when thinking of the view table itself. Every time a base table update is triggered the view table is changed exactly once at that datum. If an error happens during this procedure the view table comes out of synchronization with the base table data and will be left in an inconsistent state until it is completely recalculated again. Since consistent data is one of the most important properties of database systems it cannot be ignored. Despite the CAP theorem different measures can be taken to reduce the possibility of inconsistency to an agreeable level.

Consistency is a very vague term and it is difficult to prove, a more precise definition has to be specified. Therefore 3 levels of consistency are introduced (Jacobsen, Lee, & Yerneni, 2009). This helps determining the degree to which a distributed system can provide consistency guarantees. The three levels are weak, strong and complete consistency. The levels of consistency are built on top of each other. This means a system that guarantees complete consistency automatically guarantees weak and strong consistency.

Convergence

Convergence is a requirement for any kind of consistency. It means that the final state of the base table corresponds to the final state of the view table. In other words the updates have been processed and aggregated correctly to the view table. A violation of this property is not acceptable because the view table would be useless representing not a valid state of the base data.

Weak consistency

A system is weak consistent if it converges and every state of the view table during the update process is valid. A state in the view table is valid if it corresponds to a state in the base table.

Strong consistency

A system is strong consistent if it is weak consistent and the all the base table updates have been processed in the correct order.

Complete consistency

A system is completely consistent if it is strong consistent and every state of the base table is reflected in the view table. It is important that the view tables always hold a consistent state. Otherwise it can happen that a client receives inconsistent data. This should be avoided under any circumstances. But there is no need to reflect every base table state in the view tables. Therefore complete consistency is not as important as the other consistency levels.

Violations of Consistency

Concurrent update propagation

A concurrent processing of updates can lead to inconsistency in the view table. For example if two view manger process an update in parallel that references the same view table record. Both of the view managers then read the value of the view table record and add their delta. This results in a wrong view table records because one of the view managers should use the result of the other view manger to add his delta.

Non-idempotent view updates

In a distributed system a single network node can always crash. For example a view manager can crash right after he has propagated an update. In this case the update is propagated again when the view manager has been restarted. Because the view manager doesn't know that he has already propagated the update he processes it again, which leads to an inconsistent view table state.

Out of order update propagation

Due to the fact that the view mangers are processing updates in parallel it can always happen that the order of updates is changed. In case of aggregated sums this doesn't matter because the order of summands can be switched. But when it comes to selection of values in a specific range then the order can decide the final state of the view table, for example $x < 10$ can result in $x=9$ or $x=1$

Design/Implementation

Software Components

In order to simulate view table updates in a distributed environment the different components of a web data platform have to be implemented or at least an existing implementation has to be chosen which meets the claimed requirements.

Client request

For example a client request component has to be realized. The component should trigger all different kinds of updates into the base table, generating random data or using defined test data.

Storage unit

The storage units contain the base tables and in most of the cases the view tables, too. The storage units should be distributed over the network. Only in a distributed environment, where data is spread over multiple network nodes and accessed decentrally, the requirements for consistency and performance can be evaluated properly. Moreover the storage units should offer enough flexibility to add and remove storage units from the cluster. This can be necessary because of growing amounts of data or growing numbers auf users.

The storage unit is a component of the web data platform that doesn't have to be implemented. There are already mature database systems that can be used. Either a non-distributed system can be taken and replicated over multiple servers or a fully distributed system can be integrated in the architecture. Because of their great flexibility and performance key value stores are the instrument of choice.

View manager

This component has to implement the view manager functionality of a web data platform. A view manager propagates all the updates from the base table to the view table. While doing so it takes care of the consistency of the updates. For example an update should be rejected if it is send twice to the view table. How the updates are communicated from the base table to the view manger is left open. However, the architecture of the view managers should be distributed and scalable exactly as the architecture of the storage units. In times of high traffic the number of view managers should be increased to propagate more updates simultaneously.

Log Entry Servers

The Log Entry Server collects all the base table updates and provides them asynchronously to the view managers. The Log Entry server doesn't implement any functionality by itself, it just queues the updates and creates a decoupling in the time dimension. As an asynchronous access can be implemented in different ways a log entry server may not be needed for a web data platform. Anyway, it is important that propagation of updates can be delayed because a view manager can crash and then it should be able to come up and replay the updates that have been made in the meantime.

View Components

To be able to use the base table records for the calculation of aggregations, selections and joins we have to make some assumptions about them. This is done by specify the appropriate Java-Interfaces. The interfaces can then be implemented by the storage instance which is delivering the base table updates. This way the view manager knows how to extract the information from the base table updates.

Aggregation

The aggregation base record always needs an aggregation identifier which is used to merge all the values where the aggregation identifier is equal. The merged values named aggregation values. Because it doesn't matter if we calculate a sum, count, average or min/max-aggregation we only need one interface for all cases.

```
public interface IBaseRecordAggregation extends IBaseRecord{
	public String getAggregationIdentifier();
	public int getAggregationValue();
}
```

Picture 2: Interface IBaseRecordAggregation

Selection

The base record for the selection view has to provide a method which checks if the base record matches a certain criteria. A valid selection condition would be 25, a valid selection operator would be greater than. This is required by the view manager to check if a base record should be inserted in the view table or not.

```
public interface IBaseRecordSelection extends IBaseRecord, IViewRecord{
	public static int EQUAL=0;
	public static int GREATER_THAN=1;
	public static int LOWER_THAN=2;
	public boolean isMatching(int selectionCondition, int selectionOperator);
}
```

Picture 3: Interface IBaseRecordSelection

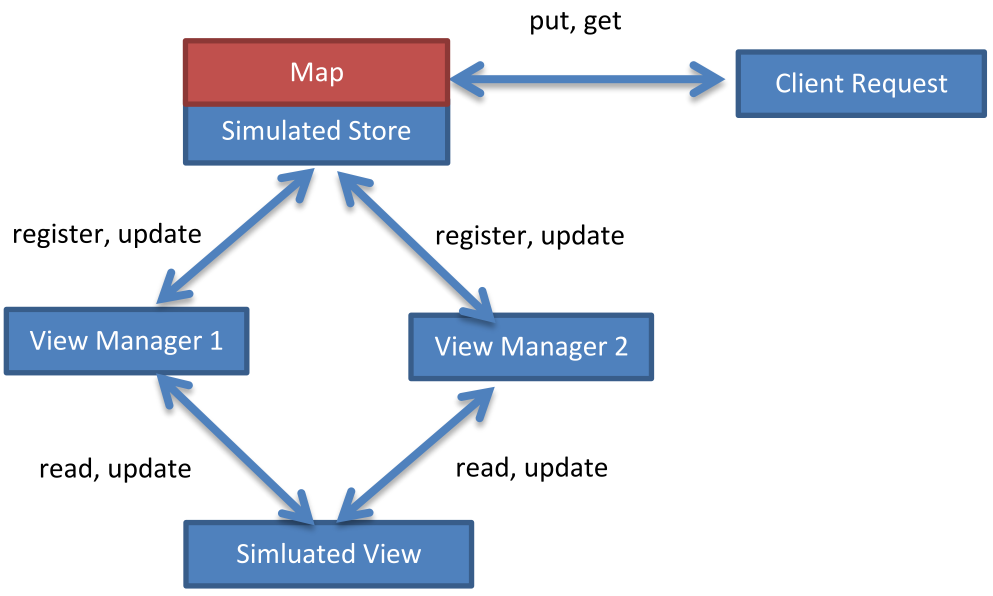
Join

For the base table updates of joins it has always to be considered if the right or the left side of the join relation is updated. As stated before we use a k-fk join view. This means that the left table is the leading table. Its entries have to provide a foreign key that references the primary key of the right table. When processing updates of the right table we just extract the value that is stored there.

```
public interface IBaseRecordJoinLeft extends IBaseRecordJoin{
	public String getForeignKey();
	public void setForeignKey();
}
```

Picture 4: Interface IBaseRecordJoin

Version 1: Distributed processes



Picture 5: Architecture Distributed Processes

In a first approach we implement the system in a distributed fashion with a simulated key value store as you can see in Picture 5. Every entity in the domain runs as a separate process on a separate node. The different nodes of the network are connected over the network. The first version lacks some of the key features stated above, for example the storage unit is not distributed. But it is only a mockup version to implement the logic of the view managers, without having to implement the entire web data platform. The different components are realized in form of java programs that are connected via the Java RMI framework. Each process can run independently from others on one of the network nodes.

On the very top right side there is the client process which triggers the inserts, updates and deletes of the base tables. For testing purposes the client can execute its requests either sequentially or with the use of multiple threads in parallel.

```
public interface IStorage extends Remote{


	public void put(String tableName, String key, IBaseRecord value) throws Exception;

	public IBaseRecord get(String tableName, String key) throws Exception;


	public void register(String name, IViewManager viewManager) throws RemoteException;

	void deregister(String name) throws RemoteException;


	public Update<IBaseRecord> requestUpdate(String viewManagerName) throws
	RemoteException;


	public void showStorage() throws RemoteException;

	public void shutdownStore() throws RemoteException;

}
```

Picture 6: Interface IStorage

The key value store implements the Interface IStore (Picture 6) and uses a simple Java Map<String, BaseRecord> internally. Later this interface should be used to connect a real distributed storage unit to the web data platform. Down below you can see the interface. It contains the put and get-procedures which correspond to the very low level data access of a key value store. Furthermore there are methods for a view manager to register and request the updates from the store. Since this is a simplified version of a web data platform there is no Log Entry Server which collects the updates, but instead the updates are polled from the storage unit directly.

```
public interface IViewManager extends Remote{


      public String getName() throws RemoteException;

      public void register() throws RemoteException;

      public void pollUpdates() throws RemoteException ;


      public void propagateInsertAggregation(String k, IBaseRecordAggregation bra,...);

      public void propagateDeleteAggregation (String k, IBaseRecordAggregation bra,...);

      public void propagateUpdateAggregation (String k, IBaseRecordAggregation oldBra,
      IBaseRecordAggregation newBra...);


      public void propagateInsertSelection(String k, IBaseRecordSelection brp, long eid);

      public void propagateDeleteSelection(String k, IBaseRecordSelection brp, long eid);

      public void propagateUpdateSelection(String k, IBaseRecordSelection oldBrp,
      IBaseRecordSelection newBrp, long eid)throws RemoteException;


      public void propagateInsertJoin(String k, IBaseRecord joinRecord, long eid);

      public void propagateDeleteJoin(String k, IBaseRecord joinRecord, long eid);

      public void propagateUpdateJoin(String k, IBaseRecordJoin oldJoinRecord,
      IBaseRecordJoin newJoinRecord, long eid);


      public boolean read(String key);

      public boolean hasProcessed(Signature signature, long eid);

      public Signature generateSignature(Signature signature, long eid);

}
```

Picture 7: Interface IViewManager

The view manager interface can be found in Picture 7. There are methods to register the view manager to the storage, respectively to the log entry server and ask for updates. This polling for updates has the advantage that an arbitrary number of view managers can be used. If more updates are propagated additional view managers can be instantiated to distribute the load. Moreover the implementation is very simple and appropriate for a first try.

One of the downsides of polling the updates is the stress of the network. If there are multiple view managers working at a time and polling updates in an interval of every millisecond then the internal network traffic of the web data platform will be very high and the update propagation will be slowed down. To solve this problem a mechanism for update distribution could be established in such a way that a view manager is informed by the storage unit if an update is ready to be processed.

Moreover the delivery of updates is assigned to a single, central instance. In a distributed environment updates should be delivered in a distributed way. But this is not a simple question since the view managers then have to know where to register to establish a consistent network with the storage units and the log entry servers. They should do it in a way that the updates of every storage unit are processed and the update load is distributed equally among the view managers.

The most important part of the view manager interface are the propagate Insert/Update/Delete methods. They are implemented according to the theory of (Jacobsen, Lee, & Yerneni, 2009) and there are three times three methods needed to cover the aggregation, selection and join-updates. The Insert/Update/Delete methods of the three different cases could surely be combined together but then a lot of special cases would have to be treated and the overview would get lost. The conceptual strength of this approach lies in the modularity and extensibility of the algorithms.

To avoid the concurrent update propagation problem, those algorithms are equipped with TAS-Methods. Every time the view manager asks the view table for a specific record, that should be updated, it gets back the view table record plus a signature. This signature should be unique and represent the current state of the view table record. In our implementation the signature is a list of timestamps where every timestamp corresponds to an update. Then the update is processed and the view manager appends the timestamp of the current update to the signature by calling the generate Signature method. Finally the view manager sends the new value of the view record together with the modified signature to the view table. The view table then checks with the help of the signature whether the view manager, who propagated the update, had a current version of the view record. If the view record has changed in the meantime, the view table returns the Boolean value false. This way the view manager knows that the update has been rejected because he has used obsolete data. The view manager overcomes this situation by requesting the view record again. Afterwards he recalculates and resents the update. Another simple implementation for a signature would be sequential numbers that increase a count variable plus one every time the value of the view record is changed.

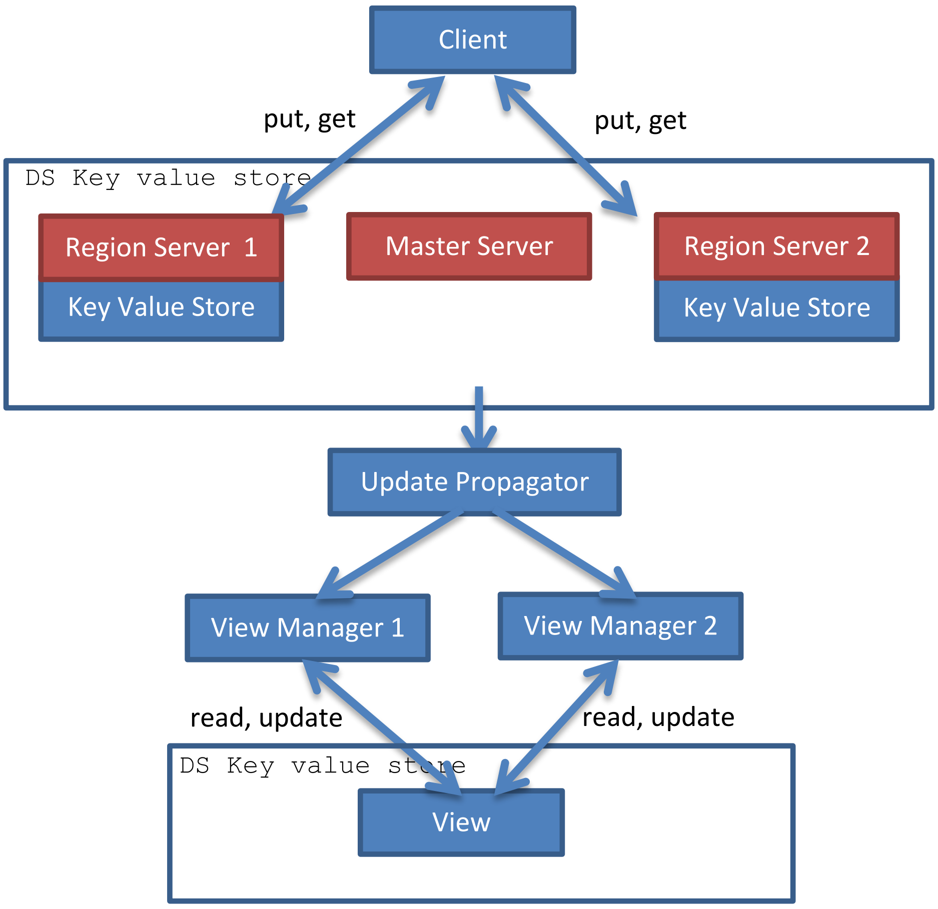
Due to the fact that we implement the signature as a list of timestamps, there is a list for every view table record. The list can be kept inside the view table record. But view table records can also be deleted, for example if a sum turns zero. So it may be a better solution to keep the signatures of the base table updates in a separate table. This way all the update signatures for an aggregation key can be retrieved and checked independently of the view record existing or not.

The view manager also takes counter measures for the non-idempotent view update problem. Like mentioned before, every view table record owns a unique signature. Similar to the TAS-Methods we strive for a simple implementation of the signatures and use timestamps. When a view manager has to deal with duplications he firstly treats them as regular updates. But then he retrieves the signature of the view record and calls the hasProcessed method. This method finds the timestamp of the current update in the signature and marks it as already processed.   Sequential numbers can unfortunately not be used here. They would have to be assigned somewhere before the view manager, so the view manager can compare them to the processed sequential numbers. As a result this would impose a global order of sequential numbers to have a unique signature. Those numbers could only be established by a central component, which is not wanted for the web data platform.

For the last problem, the out of order update propagation, it is a little bit more difficult to find a distributed solution because a global order has to be applied. If multiple view managers process an update for the same base record they probably take different time spans and this can change the order of the updates. A possible solution would be to process all updates of a specific base record with the same view manager, to keep the order of the updates.

Since this is very strict measure, again the timestamps of the signature are used to identify the order of the base table updates. It is very improbable that two base updates for the same view records create the same timestamp on two different servers, but in the end the possibility exists. To solve the problem, we can combine the timestamp with a regionId provided by responsible region server. This composite key should then uniquely identify every base table update. Finally there is an issue that should always be considered when working with timestamps. The use of timestamps assumes distributed servers that are synchronized in time to certain accurateness, otherwise the order of updates are mixed up again.

Version 2: Integrated Key value store

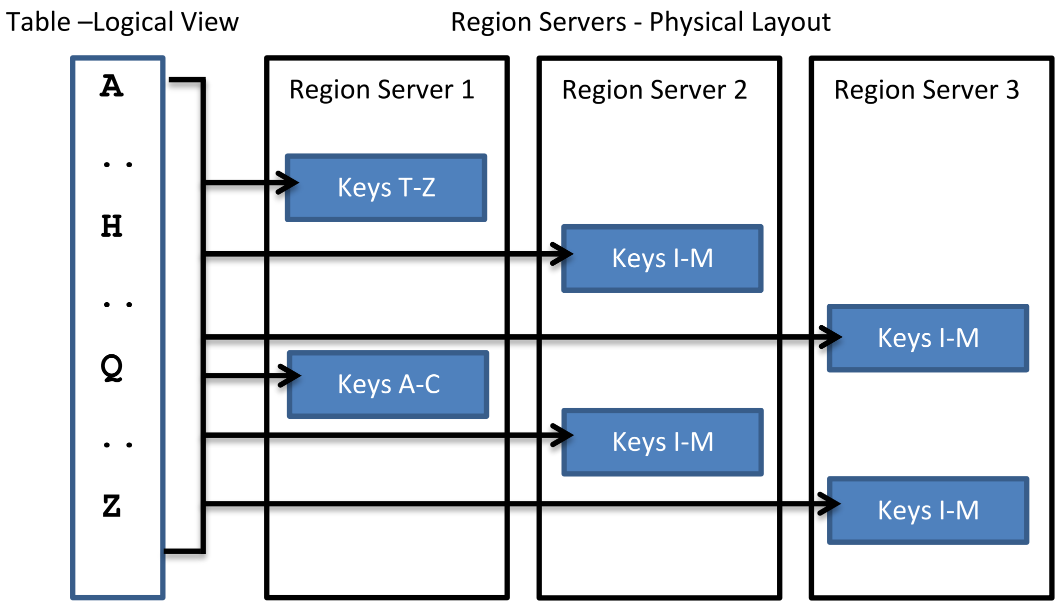


Picture 8: Integrated key value store

In the next step we integrate a distributed database system in form of a key value store into the web data platform as can be seen in Picture 8. The key value store represents the functionality of the storage units and keeps the base table as well as the view table. However, the base table has to be split over multiple tables whereas the view table can be stored in one piece. Like described above the client asks the master server first which region server he has to query. In the second step the client issues his request to the appropriate region server. The region server sends an update to the update propagator. The update propagator is a component similar to the log entry server. It queues up all the updates and the registered view manager can poll the updates at the moment he has free capacities. As in the implementation before as many view managers as needed can be used.

The distributed key value store which is taken for this implementation is Apache HBase. HBase is an open source clone of Google's Big Table and builds on top of HDFS, the Hadoop distributed file system. HDFS has the advantage that the stored data is replicated over the network nodes so that single nodes can crash and the database system is still available.

HBase implements the already outlined master-/region server architecture. As Picture 9 shows the key ranges of the table are dynamically distributed over the region servers (George, 2011). Of course the system tries to assign an equal number of key ranges to every region server. In the default configuration key ranges are managed automatically. A split size can be specified at which the system splits up the key range. If wanted a fixed key range can be defined. We use this setting to test our implementation against multiple key ranges without generating big amounts of data.



Picture 9: Region Servers

Data definition and manipulation in Hbase can be done via the Java client. Hbase stores its data as raw bytes, which means there is not much definition to create a table. First of all an HTableDescriptor with a table name is instantiated. Then an HColumnDescriptor is added. In providing a maximum of flexibility key value stores define column families and not fixed columns. A column family contains an arbitrary number of columns, which can change for each row. In our case only one column family is added and the data definition part is finished.

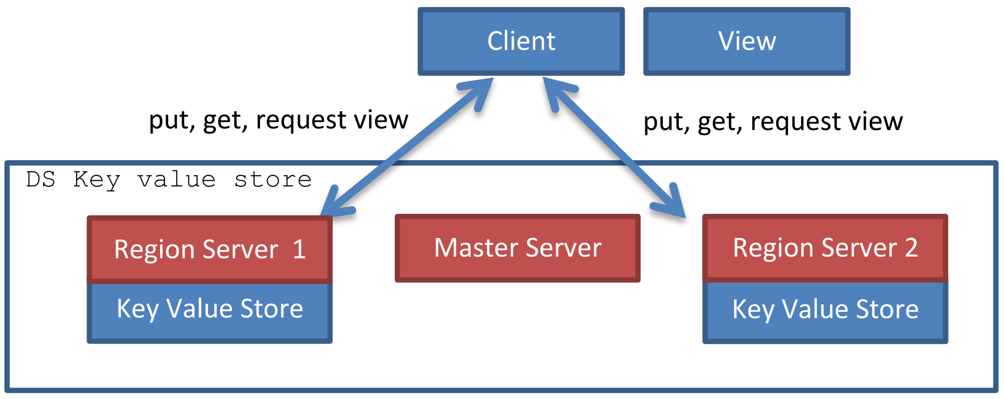
The first goal is to fill the table with aggregation data. Base table records are generated that contain a primary key, an aggregation key and an aggregation value. The values are transformed to bytes and stored in HBase with the help of a Put command. This way the whole base table is constructed. To remain consistent with our existing implementation and to minimize the effort of exchanging our storage unit we implement the IStorage interface a second time and call the implementation HBaseStorage. This way the client doesn't have to be changed at all. It still sends the put and get commands to the known storage interface.

To forward the base table updates to the update propagator an alternative is needed to transfer data from HBase to an external process. Therefore we use a mechanism of HBase called Coprocessors. Coprocessors are literally java programs that extend predefined classes. Those programs can be packaged as jar files and deployed to the HBase Server. The programs can be deployed and run concurrently on all servers of the cluster.

Two of the main classes that can be extended are the MasterObserver and the RegionObserver. The MasterObserver watches all kinds of data definition events and for that reason it is deployed to the master server. The RegionObserver watches all kinds of data manipulation events and is deployed simultaneously to all region servers. For our purpose we extend the RegionObserver class and implement the postPut method. This method is triggered right after a put has been executed by the client. The postPut method takes an object of the Put class and the environment of the region server as input parameters. The Put object gives access to all the information the client has bundled in his request. At this point the update operation is labeled with a timestamp and the regionId of the region server, which later will be part of the signature that is stored together with the view record. Moreover the Put object is transformed to an object implementing the interface IBaseRecordAggregation, such that the view manager can read and relate the data of the given updates.

For the case of a selection the table structure is similar but instead of the IBaseRecordAggregation we transform the put object to an object that implements the IBaseRecordSelection interface. In case of the Join we have to define to a left and right table. A complicating fact here is that at some point in the implementation we have to recognize if we are dealing with the left or the right side table of the join. Accordingly the Put object has to be transformed to an object implementing the IBaseRecordJoinLeft or implementing the IBaseRecordJoinRight interface. In the end the view manager knows exactly whether he is dealing with an aggregation, a selection or a join.

Version 3: batch processing



Picture 10: Architecture batch processing

The last version of our web data platform is constructed for evaluation purposes only (Picture 10). In the beginning of the report the use of views in MySQL was described as simply recalculating the view table on every client request. Since we want to test the performance of our piecewise update concept we need an implementation to test against. The recalculation on every request offers the best solution in terms of consistency but in terms of performance it is the worst solution possible. So if the performance of the piecewise updates is not significantly better than the complete recalculation, the recalculation is always the preferable solution.

The distributed key value store has the advantage of build-in mechanisms to completely recalculate a table. Therefore the client sends a batch process in form of executable code to every region server. This code may calculate the sum over a particular aggregation key. Then the code is executed on every region server and the region servers send back the partial result. Back on the client the results are combined to build a complete consistent view table.

This kind of recalculation can speed up the calculation of big data tables considerably. But nevertheless it is obvious that the solution is not good enough to survive in a web data platform environment. If the view is recalculated for every client request and there are thousands of requests per minute then a distributed key value store would collapse under the load of repeated recalculations.

Evaluation

Test Setup

For the test setup the components are deployed on different network nodes with different ip addresses. To facilitate the network setup all nodes are installed as virtual machines. For this evaluation Ubuntu 12 is installed on 4 virtual machines, running in a VMware player. The installation process is described in detail in the installation log. In a next step Java 7 is setup on all virtual machines. One of the machines is used to host the client process, as well as the update propagator and the view manager processes.

On the other three machines our distributed storage units in form of Apache Hbase are provided. The distributed version of Hbase runs on top of a distributed file system. For that reason similarly configured versions of Hadoop HDFS are installed on all three machines. According to the architecture of HDFS one of the three machines is configured as a name node, the other two are configured as data nodes. The name node is the most important component in a HDFS because it stores all the file paths in a treelike structure. Besides that the metadata, for example file- or block sizes, is managed on the name node. If the name node crashes then the HDFS cannot be used anymore. Therefore the name node should be replicated multiple times in a productive environment (White, 2011). For our test cases this is not essential because the image of the virtual machine can always be reset. The data nodes of the HDFS actually store and deliver the data for the distributed file system and an arbitrary number of data nodes can be used. The name node of the HDFS communicates with the data nodes via SSH connections. To establish SSH connections without password request, certificates are generated on the name node and copied to the data node machines. At the end the HDFS is tested with the help of command line client. If it is working files from the local file system can be stored in the HDFS and retrieved again.

After preparing HDFS Hbase can be setup on the three machines. Again we use one server as master server and two servers as region servers. The HBase installation is copied to all three servers with the same configuration. The configuration includes the addresses of the other HBase servers as well as the HDFS path. SSH connections between the servers are already established. The self-developed java program that extends the RegionObserver class is deployed to all region servers. In the end the installation is tested by running the HBase command line client and creating/deleting a table.

Summary/Perspective

The technical report combined two main parts of implementing view maintenance in a web data platform context. First part was the implementation of the view manager. Interfaces for the view manager and the data it uses have been specified and implemented. To prevent the three types of consistency problems that can occur in distributed view table maintenance, three feasible solutions were developed. The appropriate algorithms guaranteeing consistency have been sketched and realized for the three families of view tables: aggregation, selection and join relations. Then a first version of our web data platform was built around the view managers. The components client request and storage unit were implemented and a first process network was established with the help of Java RMI connections.

The second part was the integration of an existing distributed key value store into the web data platform. Therefore the interfaces for input and output had to be defined. For the ingoing data the Java client API of HBase has been used. For the outgoing data a java program, extending the RegionObserver class, was deployed on the server. This program was triggered by every put operation to forward the updates to an update propagator. There the updates were distributed to the view managers. In the end the view managers retrieved, recalculated and updated the view records corresponding to the received base table update.

In a continuative work especially the architecture concerning the update propagation has to be researched. Ways have to be found so that the part of the update propagation which follows behind the storage units can run distributed and asynchronously. It is desirable that the update propagators can register locally at different storage units and then the view mangers can connect to different update propagators. Because we want to design the system with regard to scalability the numbers of update propagators and view managers should be freely selectable. Since the components don't know which predecessor they should ask for updates, so that the global update propagation is equally distributed, some kind of distributed algorithm has to be applied to determine the communication relation of the components between each other.

Another starting point for extensions is the different view types, especially the join view. Other, more complicated types of joins could be constructed and examined with reference to consistency. As a completion of the subject the view types could be combined together in a composite pattern

Finally there is a rather technical thing to be improved. The setup of network nodes with the help of virtual machines is a very fast procedure but it could be further automated. To vary the numbers, not only for the logical, but also the physical components, a script could be created. The script could install and deploy the virtual machines automatically with the desired components.

Literaturverzeichnis

George, L. (2011). *HBase The Definitive Guide.* Gravenstein: O Reilly Media.

Jacobsen, H.-A., Lee, P., & Yerneni, R. (2009). *View Maintenance in Web Data Platforms.* Toronto: University of Toronto.

White, T. (2011). *Hadoop The Definitive Guide.* Gravenstein: O Reilly MEdia.