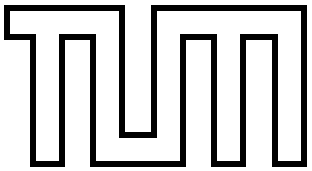


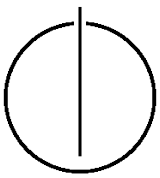
FAKULTÄT FÜR INFORMATIK

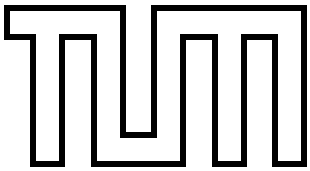
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Materialized views with Apache Spark

Saroj Gautam





FAKULTÄT FÜR INFORMATIK

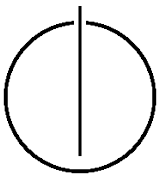
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Materialized views with Apache Spark

Materialisierte views mit Apache Spark

Author:	Saroj Gautam
Supervisor:	Prof. Dr. Hans-Arno Jacobsen
Advisor:	M. Sc. Jan Adler
Date:	August 15, 2016



I assure the single handed composition of this master's thesis, only supported by declared resources.

München, August 15th, 2016 Saroj Gautam

Acknowledgments

I would first like to thank my advisor Jan Adler for providing me the opportunity to work on a interesting topic and supervising me throughout the research.

I thank Prof. Hans-Arno Jacobsen for providing me an opportunity to write my thesis under the Chair of Distributed Systems. I thank my advisor for providing me valuable feedbacks and suggestion from the very beginning till the end.

I would also like to thank my family for giving me the motivation and moral support. I would also like to thank my friends for creating the positive environment by cracking jokes and releasing off the pressure during coffee breaks.

Abstract

In today's world, billions of people exchange information online. Service providers like Facebook, Twitter, Whatsapp store and process tremendous amount of data. Those service providers need distributed scalable storage systems to store and process a big volume of data. Even though data are stored in a distributed storage systems, still the huge size of data presents a bottleneck regarding performance optimization. Scanning tens of millions of rows and few million columns each time are expensive regarding execution time and processing power. $Materialized\ Views$ solve this problem by precomputing expensive queries and storing the result in a physical table or disk. One of the bottleneck for this approach is constantly maintaining consistency between the base table and view table. In this thesis, we propose a $Incremental\ View\ Maintenance$ approach to maintain consistency between base table and view table.

Contents

1 Introduction

1.1 Introduction

The Internet has touched many aspects of human lives. It has revolutionized the way we communicate, interact, shop, etc. Over the years, the internet has become an infinite repository of information. For instance, social networks such as Facebook records 500 TB of data each day[27], including 2.7 billion likes and 300 million photos. As of 2012, Facebook already has 100 petabytes of photos[27]. Google, on the other hand, processes 3.5 billion requests per day [27]. To deal with the scale of the ever growing information, we are in the need of efficient and scalable databases than ever before.

In the early 2000s, where there were fewer data shared on social media, data were stored in a relational database. Relational databases were designed in such a fashion to store a small amount of data and maintain integrity between them[5]. The amount of information we share on social media is expected to grow from 4.4 zettabytes in 2013 to 44 zettabytes in 2020(1 zettabyte is 1 trillion gigabytes)[5]. The scaling in RDBMS depends on adding more powerful CPU's and memory, i.e. only the vertical scaling is possible which is rather expensive. One of the advantages of the big data storage system is that it can be scaled horizontally and is also useful for storing unstructured or semi-structured data.

To address the large-scale data storage and retrieval various database systems have been developed. HBase is one of the most popular open source sortedMap datastore from Apache Software Foundation for large scale database. HBase supports horizontal scalability, i.e. the table can be stored in a distributed filesystem and can be queried efficiently. In particular, HBase breaks a table into multiple pieces and distributes the storage in a distributed storage allowing fast storage and retrieval. However, the demand for efficient processing of queries is highly desired due to ever increasing data churn rate. Hence, for petabytes of data, scanning each part of the table for a single user query is still considered to be expensive regarding the processing time.

In this thesis, we discuss several algorithms that allow efficient maintenance and retrieval of data in the context of HBase database. However, the ideas discussed in this thesis generalizes to any sortedMap datastore. In particular, we will discuss *Materialized Views* approach for databases.

1.2 Motivation

Our motivation in this thesis is to maintain consistency between base tables and view table that are split across multiple nodes in a cluster. In the era of early 90's when Internet access was limited, relational database systems were widely used to store data generated through the Internet. In the early 2000, when more people had an access to Internet, huge amount of data were generated and new research were carried out to store data in a distributed storage systems. In the year 2007, Apache Software Foundation released the first prototype of HBase, an open source sortedMap Datastore that runs on top of Hadoop. HBase data storage system provided a solution for horizontal scalability, i.e. data was stored in a distributed fashion. In the late 2000's, Internet became more accessible and the size of date grew bigger and bigger. Though data was stored in a distributed fashion, there were researchers carrying out research on faster retrieval of data from the big chunk of datasets. Oracle first introduced the concept of Materialized Views that provided a faster querying mechanism over a large dataset. Materialized Views are the database object that stores the result of a query in a table or a disk. Instead querying a table, queries were redirected to the Materialized Views for faster retrieval. However, there was a challenge to maintain consistency between a table and its materialized view. Our aim here is to maintain consistency between base tables and a view table in a HBase database systems. One of the solutions were to re-create materialized view whenever there were changes in base tables. Re-creating view table every time when changes happened in the base table was not a ideal solution. HBase released a feature called coprocessor from version 0.92 onwards that provided a functionality of triggers used in relational database management systems. We used coprocessor functionality to maintain consistency between base tables and a view table incrementally in our implementation.

1.3 Contribution

In Chapter 2, we will briefly discuss the fundamentals of Materialized Views, HBase and Incremental View Maintenance. In Chapter 3, we will present past research on Incremental View Maintenance. In Chapter 4, we discussion about the failure scenarios and prevention. In Chapter 5, we define some prerequisite and propose an algorithm to maintain consistency among base tables and view table. In Chapter 6, we perform different experiments to prove our hypothesis. In Chapter 7, we provide conclusion of our experiments and in Chapter 8, we provide some ideas that can be implemented in the near future.

2 Background

In this chapter we will first discuss about the fundamentals of $Materialized\ Views$ and $View\ Types$. We will further explain about the technologies used widely in today's Distributed Storage Databases.

2.1 Views

In a relational database management system, a $View$ is defied as result set of a query. View can be subset of a table or joins from multiple tables. Views in relational database systems are generally created for frequently accessed queries involving multiple joins to reduce cost of the operation. Views are nothing but a $SELECT$ statements to fetch desired result sets and are given certain name and saved in database. Views can also hide a complexity of a query. In a large dataset, when a computation is required to fetch data from several tables involving complex business logic, all the complex business logic can be moved to a $view$, and then just use $SELECT$ statement to get data from that view thus hiding the complexity of a query. Views also provide a layer of security mechanism to our database table. We can create a view without the columns containing confidential information, and restrict access to the base table. We can then provide access to the view and carry out desired operation using that view.

In relational database systems, Views are widely used. However, there are also certain disadvantages of $Views$. In a scenario where base table is deleted, the view of that table becomes inactive. In $MySQL$ database, for every client request, a view is recalculated. This might not be a problem for small applications containing few hundred rows or columns, but re-calculating views for every client request in large dataset can be a bottleneck for performance optimization. To overcome this bottleneck, a new approach called $MaterializedView$ is used.

2.2 Materialized Views

Materialized view is defined as the database object that stores the result of a query in a table or a disk. Materialized views are widely used for gaining performance advantage, i.e. to speed up query processing time over large datasets. The need for Materialized view addresses the problem of having to query large datasets that often needs joins and aggregations between multiple tables. These kind of queries are very expensive regarding execution time and processing power. Materialized views speed up the query processing

time by pre-computing joins and aggregations before execution and stores these results in a table or disk[10].

2.3 View Maintenance

Once the Materialized views are created, our query is redirected to Materialized View table rather than base table. Whenever there is an update in the base table, the Materialized View table also has to be updated accordingly. One of the solutions would be recomputing the entire Materialized View from the scratch or using the heuristic of inertia[16] approach i.e. incremental maintenance with respect to the base table.

2.4 Incremental Maintenance of Materialized View

"A view V is considered consistent with the database DB if the evaluation of the view specification S over the database yields the view instance (V = S(DB)). Therefore, when the database DB is updated to DB0 , we need to update the view V to V0 = S(DB0) in order to preserve its consistency"[3].

Recomputing Materialized view from scratch every time there is an update on base table is expensive. The other approach is to update the part of Materialized view table with respect to the update in Base Table. Our target is to maintain consistency between Materialized views and base table whenever there is an update on the base table.

2.4.1 Aggregation

In Aggregation view type, the data from the base table is merged on the basis of a particular key. In our implementation, we've implemented basic aggregation functions like sum, count, min and max. All these operations are carried out based on a particular key. So a unique key has sum, count, min and max operations. Whenever an update is triggered to update value for a particular key in the base table, in this case, count remains same and sum, min and max has to be recalculated. If a delete is triggered for a particular key in the base table, each of the aggregation functions has to be recalculated.

2.4.2 Join and Aggregation

In Join and Aggregation case, we have at least two base tables. Joins being one of the complex structure itself, incremental view maintenance implementation involves a lot of complex cases. Here, to reduce complexity, we join two base tables on the basis of key to form a new intermediate table. We group all the values of both base tables based on their keys. This way, for any update or delete trigger, the complexity of scanning whole base table is reduced to a single row. In our intermediate table, each of the base table is merged to a column family, join is applied and then result is stored in the view table.

In the intermediate table, the unique keys from both the base tables act as the row key, both column families from base table are merged in the intermediate table. Now for a particular row key, the values are selected from base table and plotted in the intermediate table. Now join is applied between both column families of a particular row key, and sum of the join is inserted in the view table.

2.4.3 Join and Selection

Join and Selection case is similar to the Join and Aggregation case, the only difference is instead of applying aggregation function, the join is applied for a particular row key and value is selected and inserted into the view table.

2.5 HBase

Before the evolution of HBase, Relational database systems were used particularly for storing and processing of data. Relational database systems have been used widely over a decade and are considered to be successful. In a relational database, multiple tables are used to store different types of data, this segregation of data gives more clear and systematic view of the data[22]. However, one of the biggest drawbacks in Relational database is the difficulty of scaling horizontally. The major disadvantage of relational database design is the performance if the number of tables between which the relationships has to be defined is large, i.e. more operation power needed for computation[22].

HBase is an open source sortedMap Datastore from Apache Software Foundation. HBase is modeled after Google's BigTable framework. It is a Hadoop database that is used for storing and retrieving data with random access. HBase architecture is designed to run on a cluster of computers rather than a single machine [13]. HBase aims to scale horizontally by adding more machines to the cluster. HBase runs on top of HDFS(Hadoop Distribution File System) that provides the functionality alike of Google's Big Table and provides a fault-tolerant way of storing a large volume of semi-structured and unstructured data[11].

HBase is built on top of Hadoop and Zookeeper[13]. Both Hadoop and Zookeeper are open source projects from Apache Software Foundation. Apache Hadoop is an open source framework that facilitates storing and processing large dataset in a distributed fashion. Zookeeper, which was developed under Apache software foundations, as a subproject of Hadoop, is an open source distributed configuration service for large distributes applications. A basic table structure of HBase consists of Row Key, which is similar to the primary key in a relational database table, Column Family and Column Qualifier. The figure below describes a HBase table and it's mapping to the relational database table.

HBase Table in Tabular view

RowKey	Column Family	
	Column Qualifier1	Column Qualifier2
1	A	10
2	B	20
3	C	30

HBase Table mapping to RDBMS Table

Row Key	Data
1	columnfamily:{'columnqualifier1':'A','columnqualifier2':'10'}
2	columnfamily:{'columnqualifier1':'B','columnqualifier2':'20'}
3	columnfamily:{'columnqualifier1':'C','columnqualifier2':'30'}

Row Key: Translates as Primary Key in relational table

Column Family: Group of Column Qualifier having same characteristics are placed together in a Column Family. In the table below, there are two different column families, *columnfamily1* and *columnfamily11*. A Column Family can have more than 1 column qualifiers, in the table below, *columnfamily1* has 2 column qualifiers, *columnqualifier1* and *columnqualifier2*.

Row Key	Data
1	columnfamily1:{'columnqualifier1':'A','columnqualifier2':'10'} columnfamily11:{'columnqualifier1':'A1','columnqualifier2':'11'}

Column Qualifier: Column Qualifier maps to a column in RDBMS terms. A single column can have different values at different timestamps.

Row Key	Data
1	columnfamily:{'columnqualifier':'A'@timestamp=1417524574905, 'columnqualifier':'B'@timestamp=1417524575978}

Data is always stores as byte[ ] in HBase.

Figure 2.1: HBase

2.5.1 HBase Architecture

HBase architecture consists of three major components and three sub components. The major components are Master, Region server and zookeeper. The three sub components are Write-Ahead-Log(WAL), HFile and Memstore[20]. HBase architecture is based on Master-Slave architecture, where the Master is known as HMaster, is the master node and Region Servers are the slave nodes. Whenever the write request is sent, HMaster receives the request and forwards it to the respective Region Server[20].

HMaster

HBase Master is mainly responsible for region assignments within the region servers and DDL operations like creating and deleting tables[23]. Apart from these roles, HMaster is also responsible for assigning the regions and re-assigning of the regions for recovery or load balancing[23]. HMaster also monitors all the instances of Region Servers in the cluster[23] and mainly provides administrative operations.

Region Servers

Region servers are systems within HBase that acts like a data node[20]. When a HMaster receives a write request, it forwards the request to the Region Server. Region server can have multiple regions within it, and it directs the request to the specific region. Region servers are mainly responsible for handling data related operations and communication. Region servers handle the read/write request for all the regions within it. A Region Server runs on data node and it has four sub components as described below[23]

- Write Ahead Log(WAL): Write Ahead Log is basically a log file. Region server adds each request to WAL first before sending that request to the appropriate region. It is mainly used for recovery in case of failure[20]. If the request is not written in the WAL file, there is a possibility of data loss in case of Region Server failure.

- BlockCache: BlockCache is the read cache that is used to store frequently read data[20]. When the cache is full, last read data is removed from the cache.

- MemStore: MemStore is the write cache. All the new data that has not been written to the disk are stored in MemStore. It is mainly responsible for keeping tracks of all the logs for read and write operations to be performed for a specific Region Server[23]. Each column family in a region has one MemStore[20].

- HFile: In HBase, column family is a collection of multiple HFiles. HFiles are used to store rows as keyValue pairs and are immutable and sorted[20].

Zookeeper

Zookeeper an open source project under Apache Software Foundations, is a distributed software system that provides a infrastructure for synchronization across the clusters. It provides coordination between distributed processes across the cluster so that client receives consistent data. The architecture of Zookeeper is based on client-server model. The client acts as a node that make use of the service and server acts as a node that provides the service[15]. Many Zookeeper servers can be collected together, that is known as $Zookeeper\ ensemble$[15]. Each server node of the zookeeper at a given time can handle large number of client connections. It is essential to know if the connection is alive, so the client node sends a ping request to the server it is connected to make sure it is connected and alive[15]. The server, after receiving ping request, sends an acknowledgement to indicate that server is alive. If the client doesnot receive acknowledgement within a given specific time, then the client connects to another zookeeper server within a $Zookeeper\ ensemble$ and the client session is transferred to the new zookeeper server[15].
HBase has a tight integration with Zookeeper. HBase uses Zookeeper as a distributed coordination service to facilitate synchronization between the servers in a cluster. HBase also uses Zookeeper to keep track of state of the servers, which servers are alive and available[23]. Whenever a HBase instance is started, it automatically starts Zookeeper instance, as Zookeeper comes integrated with HBase[20]. Zookeeper is used to keep tracks of the number of regions servers available, and the data hold by each region servers.

The figure below explains the HBase Architecture and its components.

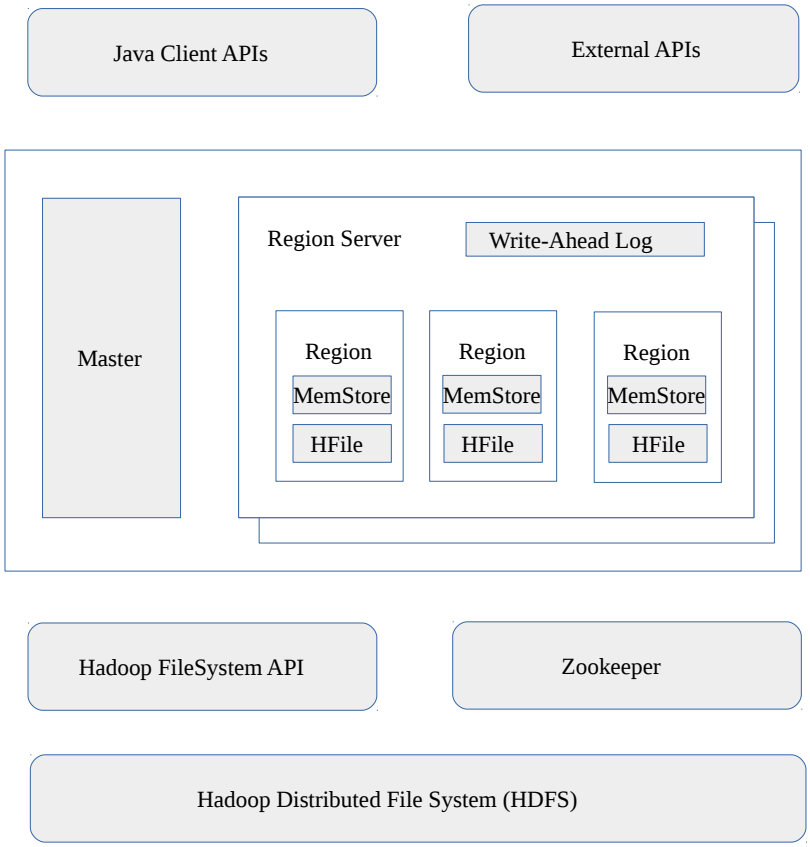


Figure: HBase Architecture

Figure 2.2: HBase Architecture

2.6 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is an open source distributed file system developed for the Hadoop framework. HDFS is designed to store very large data sets and run on a commodity hardware. In HDFS, each files are divided into blocks of fixed size and are stored across multiple machines[12]. HDFS is also based on client-server architecture and each HDFS cluster consists of a single NameNode, also called as Master Node and multiple DataNodes, known as Slave nodes. All the metadata are stored in NameNode wheres the application data resides in DataNodes[8].

2.6.1 NameNode

NameNode is the master node of HDFS file system. NameNode is the centerpiece in the HDFS architecture and is responsible for keeping the directory tree of all the files in a system[29]. Namenode does not store any data of the files, it stores only the metadata like namespace information and block information of HDFS[29], and the data are stored in Data nodes. NameNode maps files into the set of blocks, and maps those blocks to a data nodes and directs data nodes to execute I/O operations[29]. For example, when a client wants to locate a particular file, Namenode intercepts the request from the client and returns response by retrieving all the possible data nodes where the file resides. Since there is only one NameNode in the HDFS cluster, it is subjected to a single point of failure in HDFS Cluster. If a Namenode is down, all the running processes will terminate as a result of which the entire HDFS cluster goes offline[29].

2.6.2 SecondaryNameNode

As the name suggests, it is assumed that SecondaryNameNode is used as a backup node in case of single point of failure. From subsection 2.6.1, we know that NameNode stores meta information like namespace and block information. All these informations are stored in main memory and also in the physical disc for persistence storage[26]. Whenever a NameNode is started, the snapshot of the file system is stored in fsimage file and logs of the changes made after NameNode is started is written in Edit logs. There might be an issue when edit logs become very large and hard to manage. So SecondaryNameNode is used as a checkpoint in the HDFS. It fetches the edit logs from the namenode in regular interval and updates fsimage with edit logs.The recent fsimage is copied back to the NameNode[26]. Since SecondaryNameNode cannot process the metadata to the disc[12], it can not be used as a substitution to the NameNode.

2.6.3 DataNode

DataNodes are the slave nodes in the HDFS file system. There can be one or many data nodes in a HDFS cluster. The data nodes are responsible for storing the files in a HDFS

cluster. When the DataNode is started, it sends information about all the files and blocks stored in that node to the NameNode[24]. DataNode, likewise NameNode, is also expected to fail at some point. But this doesnot let the HDFS cluster to go offline. In such scenario, NameNode will replicate the blocks and files managed by failed DataNode[29].

2.7 Coprocessor

HBase Coprocessor framework provides a library to run user code in the HBase Region Server. The advantage of this framework is that it decreases the communication overhead of transferring the data from HBase region server to the client, thus improving the performance by allowing the real computation to happen in the HBase region server[18]. There are two types of coprocessor, Observer coprocessor which acts more like relational database triggers and Endpoint coprocessor that resembles stored procedures of RDBMS[21]

2.7.1 Observer coprocessor

Observer coprocessor as stated earlier, is more like database triggers that executes our code when certain events occur. In the figure below, we first try to explain a simple life cycle of put() operation as an example[13]. Observer coprocessor resides between the client and the HMaster. Observer coprocessor can be triggered after every get(), put() or delete() command. The CoprocessorHost class is responsible for observer registration and execution[13]. During the life cycle of events, Observer coprocessor allows us to hook triggers in two stages. The first one is before the occurrence of the event and the other is after the completion of the event. For example, if we want to perform some computations before the occurrence of put event, we can use prePut() method to perform our custom computation. Then the life cycle of put event starts and after the life cycle of put event is completed, we can use postPut() method to perform custom computation. In the figure below, we try to explain the lifecycle of observer coprocessor when a put event is fired[13]. There are four types of Observer Interfaces provided as of HBase version 1.1.3[14].

1. RegionObserver: RegionObserver runs on all the Region of a HBase table. RegionObserver provides hook for data manipulation for events like put(), get() add delete() events. All the data manipulations are done with pre-hook and post hook[14] such as pre and post observers. For instance, preGetOp() and postGetOp() provides hook for manipulating get request.

2. RegionServerObserver: Likewise in RegionObserver, RegionServerObserver provides a hook for data manipulation for events like merge, commits and rollback. All the data manipulation are done with pre-hook and post hook such as preMerge() and postMerge().

3. WALObserver: WALObserver interface provides a hook for Write-Ahead-Log(WAL)[14] related operations. This interface provides only preWALWrite() which is triggered

before WALEdit is written to Write-Ahead-Log and postWALWrite() which is triggered after WALEdit is written to a Write-Ahead-Log.

4. MasterObserver: MasterObserver Interface provides a hook for data manipulation for DDL events such as table creation, table deletion or table modification[19]. For instance, if the secondary indexes need to be deleted when primary table is deleted, we can use postDeleteTable(). The MasterObserver runs on the master node.

In the figure below, We can see the life-cycle of a put request with observer coprocessor implemented.

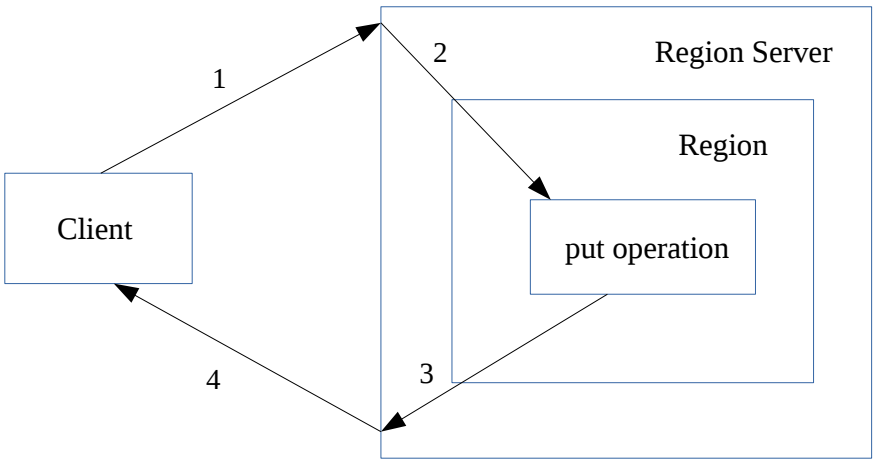


Figure: Lifecycle of a put() request in HBase

1. Client sends a put request to a Hmaster
2. HMaster dispatches the request to the appropriate Region Server and Region
3. The Region receives a put request, performs operation and returns the response back to the Region Server
4. Region Server then returns the response back to the client

Figure 2.3: Life cycle of put request

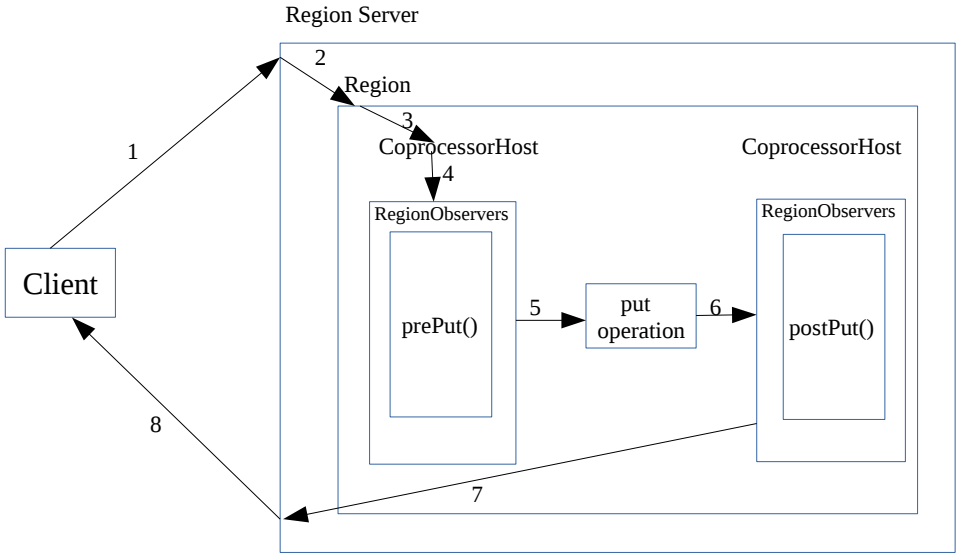


Figure: Lifecycle of put request with Observer coprocessor

1. Client sends a put request to a HMaster and HMaster dispatches request to appropriate Region Server
2. Region Server dispatches the request to the appropriate Region
3. Put request is intercepted by CoprocessorHost and invoices prePut() on each Region server
4. CoprocessorHost dispatches the request to RegionObservers and prePut() method of Observer Coprocessor is triggered.
5. After the completion of prePut(), the request is forwarded to the put operation of the request lifecycle
6. Assuming no interruptions, the request is carried out to postPut() method by CoprocessorHost
7. After postPut() produces the result, the response is forwarded to Region Server
8. Region Server then returns the response back to the client

Figure 2.4: Life cycle of put request with observer coprocessor

2.7.2 Endpoint coprocessor

Endpoint coprocessor is similar to the Stored Procedures in RDBMS. This type of coprocessor is more useful in the scenario where the computation is needed for the whole table and are not provided by observer coprocessor[6]. Invoking the endpoint coprocessor is similar to invoking any other commands in HBase from the client's point of view but the result is based on the code that defines the coprocessor[13]. The figure below explains the Aggregation example[13].

When a request is invoked from a client, an instance of Batch.call() encapsulates the request invocation and the request is forwarded to coprocessorExec() method of HTableInterface. Then the coprocessorExec() handles the request invocation and distributes the request to all the Regions of the RegionServer. Assuming that no interruptions occurs and all the requests are completed, the results is then returned to client and aggregated[13].

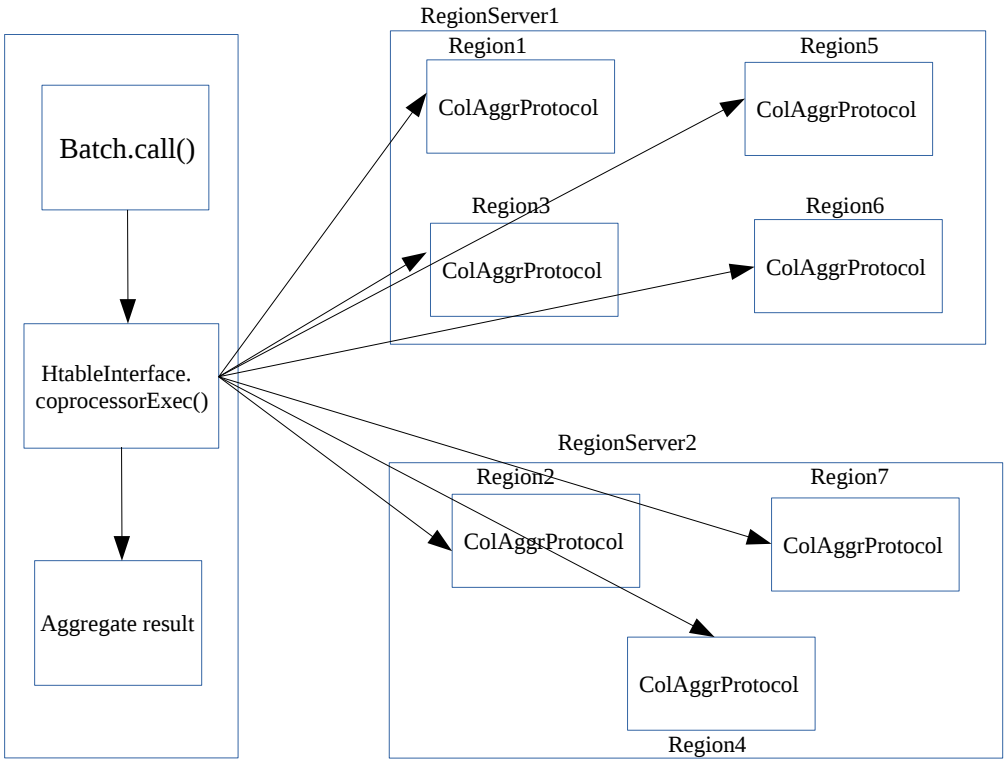


Figure 2.5: EndPoint Coprocessor

3 Related Work

In this chapter, we will discuss the existing research that have been made to maintain consistency between base table and view table.

3.1 Foundations

In the early 90's when relational database systems were popular and widely used, several research were conducted to optimize the query processing time. The idea of using materialized view when-ever possible to evaluate a query for the benefit of improved query processing was proposed more than a decade ago[9]. Several methods have been proposed for incremental view maintenance in the past[7, 17, 4].

In[25], the researchers have investigated the problem of incremental maintenance of a materialized view. The researchers in paper[25] have proposed an auxiliary relations to reduce the cost of view maintenance. They proposed a view that can be represented by an operator tree[28] where leaf nodes represented database relations and non-leaf nodes represented relational algebraic operations. An auxiliary relation was maintained for each node, and the key of auxiliary relations is a foreign key that matched the primary key of each relation, thus maintaining referential integrity between auxiliary relations and base relations[28]. These auxiliary relations are also changed in response to the base relations changes.

In paper [3], the researchers have demonstrated an algorithm for incremental view maintenance based on graph-based data model and query language Lorel developed at Stanford. Their algorithm produces a set of queries that computes the changes to be made to the view table based on the changes to the base table. Researchers proposed a view specification extension to Lorel query language[3] that introduced two objects in the view model: $select - from - where$ and $with$. The $select - from - where$ model specified the primary objects imported to the view and the later one $with$ model specified paths from primary objects to the adjunct objects[3]. Their algorithm generates a set of maintenance statements for a given view and a database object, evaluates the updates on the database to generate new set of view updates and finally installs the updates in the view[3].

In paper [16], the researchers have classified four dimensions along which the view maintenance problems has to be studied.

- Information Dimension: This dimension deals with the amount of information available for view maintenance. Some of the prior information regarding integrity con-

straints and keys, access to materialized views has to be known before developing an algorithm for incremental maintenance.

- Modification Dimension: This dimension deals with problem statements related to modification of a system. Some prior knowledge has to be acquired such as what modifications can be handled by a view maintenance algorithm, how update tuples are handled, are they handled directly or are they modeled as deletion followed by insertion tuples.
- Language Dimension: This dimension addresses problem related with select-project-join query, i.e. does view consists of entire SQL or subset of SQL. It also defines problem statements whether SQL statement can use aggregation function, recursion function or closure.
- Instance Dimension: This dimension addresses problems related to instance of database such as if view maintenance algorithm works for all the instances of the database for just for some particular instances.

In paper [9], researchers found that blind applications using materialized views resulted in much worse results than application not using materialized views. The research found out that using materialized views to optimize query performance depends on the query and statistical properties of the database[9]. The statistical properties of the database are time-varying and also most of the times, queries are generated using tools, $cost - based$ decision has to be taken whether to use or not to use materialized views to answer a given query in the database[9]. There might also be cases where more than one materialized view can be relevant for a given query, so in this case, incorrect alternatives has to be avoided to gain performance advantage. Researchers in paper [9] have proposed an algorithm for optimizing materialized views in three steps. In the first step, the query is translated into canonical unfolded form, i.e. system that supports views. In the second step, they identify possible ways to generate one or more materialized views for a given query. In the third step, they use efficient join enumeration algorithm to predict the cost of each alternative formulations and the path with least cost is selected[9].

4 Failure Detection

Many of the large scale distributed systems are often subjected to Failure. Likewise, HBase is also subjected to failure. However, like other large scale distributed systems, HBase also guarantees availability and reliability in case of node failures. In such a large scale distributed systems where there are hundred's of nodes, manually detecting failure nodes and replacing them with new nodes is nearly impossible. In our design, we mainly focus on master failure and region server failure.

4.1 Master Failure

The primary job of HBase Master is to monitor all the available Region Server Instances in the cluster. In order for HBase to be up and running, HBase Master should always be available. HMaster, with no exception, is subjected to failure at some point of time. If no master is running at some point of time, the entire system goes offline. So to avoid such failure, HBase always has a single running master and multiple backup masters running at the same time. The master and backup masters are managed by Zookeeper. When the running master goes offline, Zookeeper appoints a new master from the available backup masters.

4.2 Region Server Failure

HBase Master monitors all the available Region Servers in a cluster. There can be multiple ways in which the region servers can go offline. When a particular region server fails, all the regions within that region server also go offline. Zookeeper then finds out about the failed region server when it looses heart beat with it. After the failed region server is detected, Zookeeper notifies HMaster about the failed Region Server. HMaster then reassigns all the regions from failed region server to the active region server. Now HMaster has to recover memstore edits from the failed region server. The Write-Ahead-Log(WAL) of the failed region server is split into separate files and stored in new region server's data node by HMaster.

5 Implementation

In this section, we will first discuss the prerequisite of implementation and then the proposed method for our research.

5.1 Prerequisite

Before we begin with our implementation of the coprocessor, there are few steps to load coprocessor into our HBase table. The coprocessor can be loaded to the base tables in two ways: statically and dynamically[1].

5.1.1 Static Loading of coprocessor

We have to define coprocessor properties in a $hbase\text{-}site.xml$ file inside a <property> element followed by <name> and a <value> sub element. The <name> sub element should have one of the followings[2]:

1. hbase.coprocessor.region.classes for RegionObservers and Endpoints coprocessor
2. hbase.coprocessor.wal.classes for WALObservers
3. hbase.coprocessor.master.classes for MasterObservers

The <value> sub-element should contain the full path of the coprocessor implementation class. A typical example for static loading of coprocessor looks as,

<property>
<name>hbase.coprocessor.region.classes</name>
<value>`org.apache.hbase.HBase_coprocessor.HBaseCoprocessor`</value>
</property>

If we have multiple classes, then the path in <value> sub-element should be comma separated. In this setup, the framework will attempt to load all the configured classes, so we have to create a jar with dependencies, for all the classes and place the location of the jar to HBase classpath. For that, we have to export /path/to/jar in $hbase\text{-}env.sh$ file. A typical example for exporting classpath is given below,

export HBASE_CLASSPATH='/path/to/jar'

Now if HBase is restarted without any errors, we have managed to load system coprocessor successfully.

5.1.2 Static Unloading of coprocesssor

1. Delete entry from *hbase-site.xml*
2. Delete entry for *hbase-env.sh*
3. Restart HBase

5.1.3 Dynamic Loading of coprocessor

In this approach, rather than loading coprocessor to all the tables in a Region, the coprocessor is loaded to specific tables of the region. There are two implementations of loading coprocessor dynamically, from HBase shell or using Java API[2].

Using HBase shell

1. disable table
   hbase>disable '<table_name>'
2. load coprocessor using the following command
   alter '<table_name>',
   METHOD =>'<table_att>', 'coprocessor' =>'/file/to/path|
   /source/path/to/impementation/class|1001|'

   A typical example looks like,

   alter 'BaseTableA', METHOD =>'table_att', 'coprocessor' =>'file:///home/sarojgautam/Documents/HBase-coprocessor-0.0.1-SNAPSHOT-jar-with-dependencies.jar|
   org.apache.hbase.HBase_coprocessor.HBaseCoprocessor|1001|'
3. enable table
   See if coprocessor is loaded successfully. We can see it by seeing the table properties.
   hbase>describe '<table_name> should list the coprocessor under TABLE_ATTRIBUTES.

In the above scenario, the coprocessor tries to read class information from table_att property. There are certain arguments separated by pipe (|). The first argument in the value is the file path to the jar file that contains the implementation class. The second argument contains the full classname of the implemented coprocessor. The last argument represents

the execution sequence of registered observers. If this field is left blank, the framework will itself assign a default priority value[2].

Using Java API

Prior to HBase version 0.96, the coprocessors were loaded in a different way. After HBase version 0.96 and newer, HTableDescriptor class provides addCoprocessor() method that helps to load coprocessor in an easier way. A code snippet[1] below will give us a basic insight of how coprocessor is loaded dynamically from Java API in older versions and newer versions of HBase.

Older than 0.96

```
String path = "/path/to/jar"
admin.disableTable(<table_name>)
hTableDescriptor.setValue("COPROCESSOR$1", path + "|"
    + RegionObserverExample.class.getCanonicalName() + "|"
    + Coprocessor.PRIORITY_USER);
admin.enableTable(<table_name>)
```

0.96 or newer

```
String path = "/path/to/jar"
admin.disableTable(<table_name>)
hTableDescriptor.addCoprocessor(<class_name>.class.getCanonicalName(),
            path, Coprocessor.PRIORITY_USER, null);
admin.enableTable(<table_name>)
```

5.1.4 Dynamic Unloading of coprocessor

Dynamic unloading of coprocessor can also be done in two ways, from shell and from Java API.

Using HBase shell

1. disable table hbase>disable '<table_name>'
2. alter table, remove coprocessor hbase>alter '<table_name>', METHOD =>'table_att_unset', NAME=>'coprocessor$1' =>
3. enable table hbase>enable '<table_name>'

Using Java API

Using Java API, in the newer version we can use removeCoprocessor() method provided by HTableDescriptor class and in the older version, we can use setValue() to unload coprocessor.

5.2 Proposed Method

In this section, we will explain about the algorithms we've implemented to maintain incrementally materialized views for

1. Aggregation
2. Join and Aggregation
3. Join and Selection

One of the most important features in our implementation is the introduction of intermediate table. We have introduced an intermediate table in order to restrict scanning of the entire base table for a simple get, put or delete operation. Scanning billions of rows for such operations can be expensive in terms of processing power and CPU usage. When an update/delete operation happens in the base table, updates in view table are calculated on the basis of updates in the intermediate table.

5.2.1 Creation of an Intermediate table

We create an intermediate table in such a way that we have a single row for all the unique keys in the base table. This architecture restricts scanning the entire base table for any update or delete operation. Instead, we only scan particular row containing the key for which an update or delete operation is triggered. In our implement, there are two ways in which we create our intermediate table.

Single Base Table

In this scenario, we have one base table and we have to create an intermediate table from the base table. The $columnFamily$ of the base table becomes $columnFamily$ in the intermediate table. The unique values of column Key of the base table becomes $rowKey$ in the intermediate table and $rowKey$ of the base table becomes $column$ in the intermediate table. Once we create our intermediate table, we put the values from the base table into the intermediate table. Fig(a) in 5.1 shows a graphical transformation of a single base table into an intermediate table. Now for any update or delete operations for a particular Key/Value in the base table, only a single row for that particular Key is updated in an intermediate table. This implementation restricts scanning of the whole base table for a single update/delete operation.

Two Base Tables

In this scenario, we have two base tables and we have to create an intermediate table from the two base tables. The both $columnFamilies$ of base tables are merged together into two $columnFamilies$ in the intermediate table. The unique values of column Key of both base table becomes $rowKey$ in the intermediate table and $rowKey$ of the base table becomes $column$ in the intermediate table. Once we create an intermediate table from two base tables as described, we put values from both the base tables into the intermediate table. Fig(b) in 5.1 shows a graphical transformation of a two base tables into an intermediate table. Now for any update or delete operations for a particular Key/Value in the base tables, only a single row for that particular Key is updated in an intermediate table. This implementation restricts scanning of the whole base table for a single update/delete operation.

5.2.2 Creation of View Table

A view table contains resultset of a query on the base table. In our implementation, view table is created on the basis of an intermediate table. Once we have our intermediate table, we apply certain algorithms on the intermediate table and create view table. Once we have our view table, our task is to maintain consistency between base table and view table. If any operation changes the state of base table, we reflect those changes in intermediate table and view table. The algorithm we've implemented maintains consistency incrementally, not re-computing view table for every changes in the base table. We will discuss our approach of this incremental maintenance in the sections below.

Aggregation Table

RowKey	HColumn	
	Key	Value
x1	A	10
x2	B	20

Aggregation Intermediate Table

RowKey	HColumn	
	x1	x2
A	x1,10	
B		x2,10

Fig(a): Creation of Intermediate table from a single base table

Join and Aggregation Table A

RowKey	HColumnA	
	Key	Value
x1	A	10
x2	B	20

Join and Aggregation Table B

RowKey	HColumnB	
	Key	Value
y1	A	10
y2	C	20

Join and Aggregation Intermediate Table

RowKey	HColumnA		HColumnB	
	x1	x2	y1	y2
A	x1,10		y1,10	
B		x2,20		
C				y2,20

Fig(b) : Creation of Intermediate table from two base tables

Figure 5.1: Creation of an Intermediate Table

5.2.3 Aggregation

For Aggregation view type, we have a single base table with multiple key,value pairs. We create an intermediate table from a single base table as described in subsection 5.2.1. The view table for Aggregation basically stores the result of aggregation functions namely Sum, $Count$, Min and Max. These aggregation functions are carried out based on Key of the base table. In this section we describe how we can maintain consistency incrementally between base table and view table.

Once we have a base table, an intermediate table, a view table and coprocessor successfully loaded on our base table, we are ready to go ahead with our implementation. There are certain scenarios where coprocessor is triggered for an update and delete operations.

1. New row is inserted
2. Existing value of a row is updated
3. Existing key of a row is updated
4. Existing row is deleted

New row is inserted

When a client issues an insert operation to insert a new key,value pair on the base table, we have to insert that key,value pair in our intermediate table and our view table has to be updated accordingly. The view table stores result of aggregation functions and the changes in the view table has to be updated incrementally. Using prePut() and postPut() triggers from observer coprocessor, we perform all the required operations.

As we have already discussed put() request life cycle in 2.7.1, before the key,value is inserted, we catch the request using prePut() method provided by the observer coprocessor. In the prePut() method, we verify the inputs and check if new row is inserted or existing row is updated. After we verify that new row is inserted, we let the request to insert new key,value pair into the base table. After new key,value pair is inserted into the base table, we again catch the request in postPut() method. In postPut() method, we put new key,value pair in the intermediate table. Once the intermediate table has been updated, we need to update all the aggregation functions in the view table. We create a list and put all the key,value pairs for that particular rowKey of the intermediate table. From the list we update our aggregation functions and then update the result on the view table. The figure 5.2 explains the scenario when a new row is inserted. The left side tables are the default tables and right side tables explain the behavior when a new row is inserted. The text displayed in red marks the changes that are happening on base table, intermediate table and the view table.

Base Table

AggrTable

	HColumnA	
	Key	Value
1	A	10
2	A	20
3	B	20
4	B	40
5	C	60

AggrIMTable

	AggrColFam				
	1	2	3	4	5
A	1,10	2,20			
B			3,20	4,40	
C					5,60

AggrViewTable

	AggrColFam			
	Sum	Count	Min	Max
A	30	2	10	20
B	60	2	20	40
C	60	1	60	60

Base Table when new row is inserted

AggrTable

	HColumnA	
	Key	Value
1	A	10
2	A	20
3	B	20
4	B	40
5	C	60
6	D	30

AggrIMTable

	AggrColFam					
	1	2	3	4	5	6
A	1,10	2,20				
B			3,20	4,40		
C					5,60	
D						6,30

AggrViewTable

	AggrColFam			
	Sum	Count	Min	Max
A	30	2	10	20
B	60	2	20	40
C	60	1	60	60
D	30	1	30	30

Figure 5.2: New row insert

Existing value of a row is updated

Whenever an existing value of a key is updated, the base table is updated accordingly. Before the base table is updated, we catch the request via prePut() method of observer coprocessor. In the prePut() method, we get the key,value for which the value is going to be updated and also we verify if the value is being updated or the key is being updated. After we verify that value is updated, then we release the request and the value is updated in the base table. After the insertion, we catch the request via postPut() method of observer coprocessor, and then plot the updated value in our intermediate table for particular row key. Since we already have the row key, we only need to can that particular row, instead of scanning the whole intermediate table. This saves a lot of execution time and processing power. Once we plot updated value in the intermediate table, we then calculate aggregation functions for that particular row key and then update our view table accordingly. Now we need to update aggregation functions in the view table. We create a list and put all the key,value pairs for that particular rowKey of the intermediate table. From the list we update our aggregation functions and then update the result on the view table. The figure 5.3 explains the process in more detail. The updated value is marked in red on the right table, and also from the figure, we can see that we only iterate over a particular row key instead of scanning the whole base table and view table.

Existing key of a row is updated

Whenever there is a trigger to update a Key of the particular rowKey, we first catch the request via prePut() method. In this scenario, we first save key,value pair of the row to be deleted from base table and new key,value pair to be inserted into the base table. Then we release the request and the Key is updated in the base table. In this case, now we have old key,value pair and the new key,value pair.

In the postPut() method, first we find the column to be deleted from the intermediate table. Then we delete that particular column from the intermediate table. After we delete column from an intermediate table, we need to update aggregation functions in the view table. We create a list and put all the key,value pairs for that particular rowKey of the intermediate table. From the list we update our aggregation functions and then update the result on the view table. After the process is complete without any interruptions, the process is similar as of inserting new key,value pair. We put new key,value pair in our intermediate table and now we need to update aggregation functions in the view table. We create a list and put all the key,value pairs for that particular rowKey of the intermediate table. From the list we update our aggregation functions and then update the result on the view table. In the figure 5.4, the old key A is updated to new key B. In the intermediate table, the plotting for old key A is deleted and aggregation functions for old key A are also updated in the view table. After the process is completed, new values for updated key B is plotted in the intermediate table and then the view table for row key B is also updated accordingly.

Base Table

AggrTable

	HColumnA	
	Key	Value
1	A	10
2	A	20
3	B	20
4	B	40
5	C	60

AggrIMTable

	AggrColFam				
	1	2	3	4	5
A	1,10	2,20			
B			3,20	4,40	
C					5,60

AggrViewTable

	AggrColFam			
	Sum	Count	Min	Max
A	30	2	10	20
B	60	2	20	40
C	60	1	60	60

Base Table when value is updated

AggrTable

	HColumnA	
	Key	Value
1	A	10
2	A	50
3	B	20
4	B	40
5	C	60

AggrIMTable

	AggrColFam				
	1	2	3	4	5
A	1,10	2,50			
B			3,20	4,40	
C					5,60

AggrViewTable

	AggrColFam			
	Sum	Count	Min	Max
A	60	2	10	50
B	60	2	20	40
C	60	1	60	60

Figure 5.3: Update value for a existing row key

Base Table

AggrTable

	HColumnA	
	Key	Value
1	A	10
2	A	20
3	B	20
4	B	40
5	C	60

AggrIMTable

	AggrColFam				
	1	2	3	4	5
A	1,10	2,20			
B			3,20	4,40	
C					5,60

AggrViewTable

	AggrColFam			
	Sum	Count	Min	Max
A	30	2	10	20
B	60	2	20	40
C	60	1	60	60

Base Table when Key is updated

AggrTable

	HColumnA	
	Key	Value
1	A	10
2	B	20
3	B	20
4	B	40
5	C	60

AggrIMTable

	AggrColFam				
	1	2	3	4	5
A	1,10				
B		2,20	3,20	4,40	
C					5,60

AggrViewTable

	AggrColFam			
	Sum	Count	Min	Max
A	10	1	10	10
B	80	3	20	40
C	60	1	60	60

Figure 5.4: Update Key for a existing row key

Existing row is deleted

When an existing row is deleted in the base table, we first get key,value pair in prePut() method. In the postPut() method of observer coprocessor, we first delete the entry for that particular key from the intermediate table. But before deleting, we have two scenarios here. If the key to be deleted has more than one values in the intermediate table, then we delete the particular plotting in the intermediate table and update aggregation functions for that key in the view table. If the Key in the intermediate table has only one value, then we delete that row from an intermediate table and then also delete the row from view table.

In the figure 5.5, we have a delete() call for row key 5. The key for row key 5 is C. Now we delete the row key 5 from the base table. After that, we delete the plotting for key C in our intermediate table. Since the key C has only one plotting, we delete entry for row key C from the view table instead of recomputing aggregation functions for row key C.

Base Table

AggrTable

	HColumnA	
	Key	Value
1	A	10
2	A	20
3	B	20
4	B	40
5	C	60

AggrIMTable

	AggrColFam				
	1	2	3	4	5
A	1,10	2,20			
B			3,20	4,40	
C					5,60

AggrViewTable

	AggrColFam			
	Sum	Count	Min	Max
A	30	2	10	20
B	60	2	20	40
C	60	1	60	60

Base Table when row is deleted

AggrTable

	HColumnA	
	Key	Value
1	A	10
2	A	20
3	B	20
4	B	40

AggrIMTable

	AggrColFam				
	1	2	3	4	5
A	1,10	2,20			
B			3,20	4,40	

AggrViewTable

	AggrColFam			
	Sum	Count	Min	Max
A	30	2	10	20
B	60	2	20	40

Figure 5.5: Delete an existing row

5.2.4 Join and Aggregation

For Join and Aggregation, we have two base tables with key,value pairs. We create an intermediate table from two base tables as described in subsection 5.2.1. For Join and Aggregation, we join two base tables on the basis of k-fk join, apply sum aggregation function on the resultset and put result in the view table. In this approach, there are certain scenarios where we have to maintain consistency between base tables and a view table.

1. New row is inserted
2. Existing value of a row is updated
3. Existing key of a row is updated
4. Existing row is deleted

New row is inserted

When a client inserts new row with key,value pair in a base table, we have to update our intermediate table and view table accordingly to maintain consistency between them. Updating intermediate table is not a big problem here, we just need to put new key,value pair for that particular rowKey in a corresponding column family. In the prePut() method, we determine the key,value pair to be inserted into the base table. When the request is triggered in postPut() method, we need to update intermediate table and view table accordingly. Since k-fk join is involved, we need to figure out the new rows generated with respect to the newly inserted key,value pair. The first list contains newly inserted key,value pair for that particular rowKey. The second list contains all the key,value pairs for the same rowKey. Now we get a cartesian product of key,value pairs, value containing the sum of the k-fk joins that are to be inserted into the view table. Now the result is inserted into the veiw table. This way, we are able to maintain consistency whenever a new row is inserted in a base table. The figure 5.6 describes the scenario in more detail. The updated table on the right has new values plotted in red.

BaseTableA

	HColumnA	
	Key	Value
a1	A	10
a2	B	20
a3	C	30
a4	D	40

BaseTableA

	HColumnA	
	Key	Value
a1	A	10
a2	B	20
a3	C	30
a4	D	40
a5	E	30

BaseTableB

	HColumnB	
	Key	Value
b1	A	10
b2	A	20
b3	B	30
b4	B	40
b5	E	50

BaseTableB

	HColumnB	
	Key	Value
b1	A	10
b2	A	20
b3	B	30
b4	B	40
b5	E	50

IMTable

	HColumnA				**HColumnB**				
	a1	**a2**	**a3**	**a4**	**b1**	**b2**	**b3**	**b4**	**b5**
A	a1,10				b1,10	b2,20			
B		a2,20					b3,30	b4,40	
C			a3,30						
D				a4,40					
E									b5,50

IMTable

	HColumnA					HColumnB				
	a1	**a2**	**a3**	**a4**	**a5**	**b1**	**b2**	**b3**	**b4**	**b5**
A	a1,10					b1,10	b2,20			
B		a2,20						b3,30	b4,40	
C			a3,30							
D				a4,40						
E					a5,30					b5,50

ViewTable

	HcolumnV
	Sum
a1,b1	20
a1,b2	30
a2,b3	50
a2,b4	60

ViewTable

	HcolumnV
	Sum
a1,b1	20
a1,b2	30
a2,b3	50
a2,b4	60
a5,b5	80

Figure 5.6: New row is inserted for Join and Aggregation

Existing value of a row is updated

When an existing value of a particular key is updated in a base table, we update the particular value in our intermediate table. In the prePut() method, we determine key,value pair for the row to be updated and in the postPut() method, we update our intermediate table and view table accordingly. The update process in an intermediate table is straightforward i.e. we just update value for that particular row and column. Now in order to update our view table, we need to find all the rows that are affected by an update statement as k-fk join is involved. We create two lists and the first list contains key,value pair of the row which was just updated. The second list contains all the key,value pairs for the same rowKey. Now we get a cartesian product of key,value pairs, and the value containing sum of k-fk joins that are to be updated into the view table. This way we maintain consistency between base tables and view table when an existing row is updated in the base table.

Existing key of a row is updated

In this scenario, we first save key,value pair of the row to be deleted from base table in prePut() method. After the entry has been deleted from the base table, we delete an entry from the intermediate table in the postPut() method. Now we have to find all the rows to be deleted from the view table for that particular key. In this process, we create two list and find out all the rows to be deleted from the view table. The first list contains key,value pair which we saved earlier. The second list contains all the key,value pairs for the same rowKey. Now we get a cartesian product of key,value pairs that are to be deleted from the view table. Then we delete all the affected rows from the view table. Now we have to update old key to a new key in a base table. Once the base table has been updated, we update our intermediate table accordingly. Now we need to find out the rows to be inserted into the view table as it involves k-fk join among the base tables. We do that by creating two lists, first list containing new key,value pair and second list containing all the key,value pairs for that rowKey. Now we get a cartesian product of key,value pairs, and the value containing sum of k-fk joins that are to be inserted into the view table.

Existing row is deleted

When a row is deleted from the base table, we simply delete an entry from the intermediate table. The next task is to find all the rows that are to be deleted from the view table. This is a similar process as described in the sub section above. We first create two lists here, first list contains the key,value pair of the row that was deleted and the second row contains all the key,value pairs for the same row. Now we get a cartesian product of key,value pairs that are to be deleted the view table. We delete all the rows from the view table. This way we can maintain consistency between base table and view table when a row is deleted from the base table.

5.2.5 Join and Selection

The approach implemented in this subsection is similar to the approach described in subsection 5.2.4. We have two base tables with key,value pairs. We create an intermediate table from two base tables as described in subsection 5.2.1. For Join and Selection, we join two base tables on the basis of k-fk join and put result in the view table. In this approach, there are also certain scenarios where we have to maintain consistency between base tables and a view table.

1. New row is inserted
2. Existing value of a row is updated
3. Existing key of a row is updated
4. Existing row is deleted

New row is inserted

When a client inserts new row with key,value pair in a base table, we have to update our intermediate table and view table accordingly to maintain consistency between them. Updating intermediate table is not a big problem here, we just put new key,value pair for that particular rowKey in a corresponding column family. Since k-fk join is involved, we need to figure out the new rows generated with respect to the newly inserted key,value pair. The first list contains newly inserted key,value pair for that particular rowKey. The second list contains all the key,value pairs for the same rowKey. Now we get a cartesian product of key,value pairs that are to be inserted into the view table. Now we are able to maintain consistency whenever a new row is inserted in a base table.

Existing value of a row is updated

When an existing value of a particular key is updated in a base table, we update the particular value in our intermediate table. The update process in an intermediate table is straightforward i.e. we just update value for that particular row and column. Now in order to update our view table, we need to find all the rows that are affected by an update statement. We create two lists and the first list contains key,value pair of the row which was just updated. The second list contains all the key,value pairs for the same rowKey. Now we get a cartesian product of key,value pairs that are to be updated into the view table. This way we maintain consistency between base tables and view table when an existing row is updated in the base table.

Existing key of a row is updated

In this scenario, we first save key,value pair of the row to be deleted from base table. After the entry has been deleted from the base table, we delete an entry from the intermediate

table. Now we have to find all the rows to be deleted from the view table for that particular key. In this process, we create two list and find out all the rows to be deleted from the view table. The first list contains key,value pair which we saved earlier. The second list contains all the key,value pairs for the same rowKey. Now we get a cartesian product of key,value pairs that are to be deleted from the view table. Then we delete all the affected rows from the view table. Now we have to update old key to a new key in a base table. Once the base table has been updated, we update our intermediate table accordingly. Now we need to find out the rows to be inserted into the view table as it involves k-fk join among the base tables. We do that by creating two lists, first list containing new key,value pair and second list containing all the key,value pairs for that rowKey. Now we get a cartesian product of key,value pairs that are to be inserted into the view table.

Existing row is deleted

When a row is deleted from the base table, we simply delete an entry from the intermediate table. The next task is to find all the rows that are to be deleted from the view table. This is a similar process as described in the sub section above. We first create two lists here, first list contains the key,value pair of the row that was deleted and the second row contains all the key,value pairs for the same row. Now we get a cartesian product of key,value pairs that are to be deleted the view table. We delete all the rows from the view table. This way we can maintain consistency between base table and view table when a row is deleted from the base table.

6 Evaluation

In this chapter we perform different kinds of experiments on both Pseudo Distributed mode (single node cluster) and Fully Distributed mode (multi node cluster). We will further discuss about every types of experiments we performed and the dataset we used. We will then present the result of our experiments.

6.1 Experiment Setup (Pseudo Distributed Mode)

In pseudo distributed mode, we have performed four different kinds of experiments for each of the three scenarios. First experiment $View\ Re\text{-}computation\ vs\ Maintenance$ is performed on three different datasets while rest of the experiments are performed on fixed dataset.

6.1.1 Deployment

We performed our experiment on a single node cluster (Ubuntu 16.04 LTS, Intel Core i5-3230M CPU @ 2.60GHz, 3.9GB RAM, 23GB HD). We installed hadoop in pseudo distributed mode. The services like JobTracker, TaskTracker, Namenode and Datanode runs as a separate Java process in a single cluster. We installed hadoop version 2.6.4 and hbase version 1.1.5 for our experiments.

6.1.2 Table Configuration

For $Aggregation$, we first create one empty base table, an intermediate table and a view table. We first insert records in the base table. After that, we read data from base table and write into intermediate table as explained in section 5.2.3. Once write in the intermediate table is completed, we perform different aggregation functions like Sum, Max, Min and $Count$ based on the key of intermediate table and write the result in view table.

For $Join$ & $Aggregation$ and $Join$ & $Selection$, we create two base tables as it involves k-kf joins, an intermediate table and a view table. We first insert records in both the base tables, read data from first base table and insert into first column family of an Intermediate table, and again read data from second base table and insert into second column family of an Intermediate table as explained in sections 5.2.4 and 5.2.5 respectively. After we have our Intermediate table ready, we apply k-kf join and insert data into view table accordingly.

6.1.3 Control Parameter

There are certain control parameters defined for our experiments to determine performance.

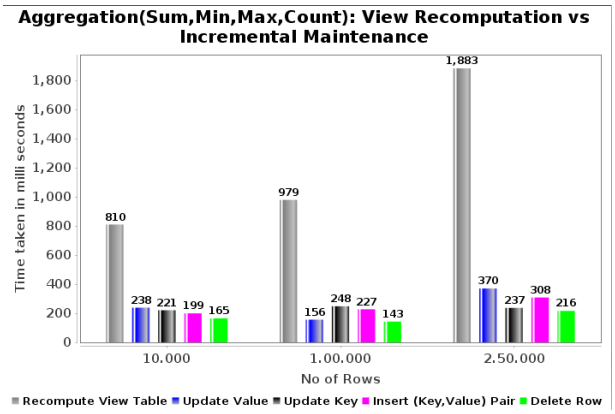
- $noOfRegions$: The number of regions within a Region Server
- $typesOfOperation$: The type of operation performed by the client. In our experiment, we've performed insert, update and delete operation.
- $typesOfViews$: This parameter defines the types of views we have implemented in our experiments such as $Join$, $Selection$, Sum, $Count$, Max, Min.
- $timeInMillis$: This parameter defines the time taken in milliseconds to perform certain operations.

6.2 Experiment 1 (Aggregation)

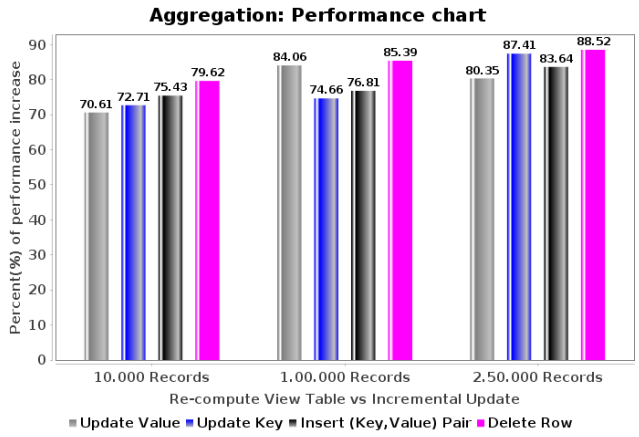
To evaluate the performance of $Aggregation$ view type, we perform three different experiments. In the first experiment, we compare the single region execution time for $View\Re$-$computation$ and $Incremental\ maintenance$ on three different datasets. Next, we perform the same experiments on a dataset with 100,000 records by varying the number of regions on a single region server.

6.2.1 Aggregation: View Re-computation vs Maintenance

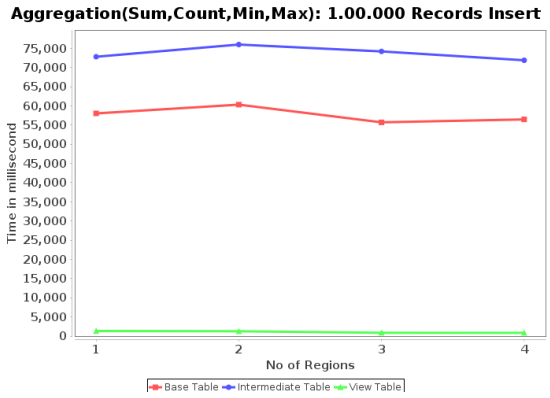
To evaluate performance in View Re-computation vs Maintenance, we first create a base table with 10,000 rows and generate corresponding intermediate and view tables. The view table consists of four different aggregations of the base table data, namely Sum, $Count$, Min and Max. For operation in a base table, view table can be updated to reflect changes in two different ways, (a) re-compute (b) incremental update. In our experiments, we compare the execution time for these two methods. Further, we perform the experiments for 100,000 and 250,000 records. In our experiments, we report the execution time in milliseconds. In the figure 6.1, in x-axis, we show types of client operations performed in base table and in y-axis we show performance optimized in percent.



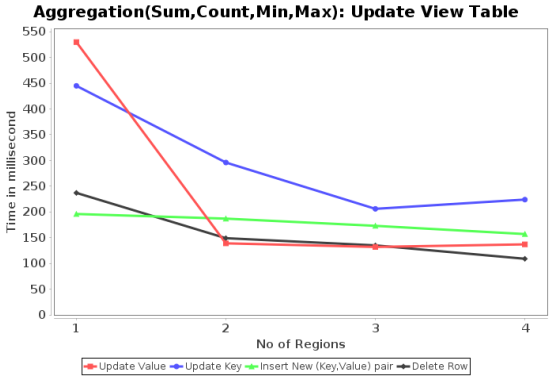
Fig(a): Recomputation vs Incremental Maintenance



Fig(b): Aggregation Performance Chart



Fig(c): Insert records



Fig(d): Update View Table

Figure 6.1: Aggregation Experiment Standalone

In Figure 6.1, $Fig(a)$, we compare the time taken while inserting records using the above methods. In figure 6.1, $Fig(a)$, we observe that incremental update outperforms re-compute method by orders of magnitude. Further, In Fig 6.1, $Fig(b)$, we report the percentage improvement achieved by an incremental update for four different operations over the re-compute baseline. Here we observe that incremental method yields up to 88% improvement over the baseline.

$$\text{performance} = \frac{\text{time taken to compute view table} - \text{incremental update}}{\text{time taken to compute view table}} \times 100 \qquad (6.1)$$

The reason for the significant improvement in performance of incremental update over the baseline is due to read-write and recompute overhead. In the re-compute method, we need to read the whole base table and reconstruct the view table. However, in the incremental method, we do not reconstruct the view table but only update the rows that are affected by the operation on the base table.

6.2.2 Aggregation: Insert Records

In this experiment, we wanted to prove that splitting table across multiple regions improves overall performance. In the figure 6.1, $Fig(c)$, in x-axis we show no. of regions and in y-axis, we show time taken to insert records in millisecond. To prove our hypothesis, we first inserted 1.00.000 records without splitting a table and recorded time taken to insert records. In the second scenario, we split our table in two regions and note time taken to insert 1.00.000 records. We carried out same experiment by splitting our table in three and four regions simultaneously. We noted time taken for each experiments and plotted a graph. From the graph, we see that splitting our table across regions takes less time to insert 1.00.000 records. The reason for this improvement is that the table is split horizontally across all the regions. The $HexStringSplit$ algorithm partitions the table evenly across all the regions, as a result of which the load is distributed equally among all the regions.

6.2.3 Aggregation: Update View Table

In section 6.2.2, we proved that we can improve optimization if we split our tables across multiple regions. In this experiment, we wanted to see how it affects the performance if we split $View\ Table$ across multiple regions. In the figure 6.1, $Fig(c)$, in x-axis we show no. of regions and in y-axis, we show time taken to update view table incrementally in millisecond. We first insert 1.00.000 records in base table and pre-compute view tables accordingly. In the first scenario, we compute view table without pre-splitting view table. In the second scenario, we pre-split view table across two regions. Similarly, we also pre-split view table across three and four regions and compute view tables accordingly. When client issues update operation on base table, we reflect changes in view table incrementally and note time taken for each type of update operation. Then we plot graph with the cap-

tured results and from $Fig(d)$, we proved that splitting table across multiple regions can improve overall performance.

6.3 Experiment 2 (Join and Aggregation)

For $Join$ & $Aggregation$, we create two base tables as it involves k-kf joins, an intermediate table and a view table. We insert 3.000 rows in both the base table, apply k-fk joins and calculate sum for those joins and insert result in the view table. We then run experiments to see how we can improve performance as we split our tables across multiple regions.

6.3.1 Join and Aggregation: Insert records

In this experiment, we insert 3.000 records in each of the base tables and apply k-fk join in the two base tables. First we run this experiment in a single region and see the number of rows generated in the view table after applying k-fk joins. We run this experiment for 2, 3 and 4 regions and split our tables accordingly. In the figure 6.2, Fig(a) shows the graph of no. of records inserted in view tables. We use this statistics for running other sub-experiments.

6.3.2 Join and Aggregation: View Re-computation vs Maintenance

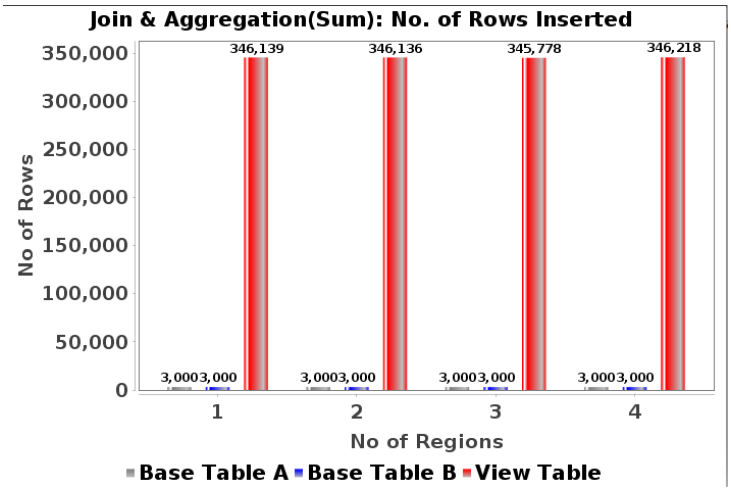
In this experiment, we wanted to show that by incrementally maintaining our view tables, we can increase the overall performance of the system. We ran this experiment on the basis of data gathered in sub-section 6.3.1. As a proof, we first ran this experiment on dataset of 1.53.761 records. From the graph 6.2, $Fig(b)$ shows the time required for re-computing view table vs time required to incrementally update view table. If we need to re-compute view table, in such scenario, we first need to perform read operation on the entire base table and then write operation in view table. Whereas in incremental maintenance, we only update the affected rows in view table. To ensure our hypothesis works, we ran our experiment on bigger dataset of 3.45.658 rows. From the experiment results, we proved that maintaining views incrementally is significantly less expensive than re-computing view table for every client request in base table.

6.3.3 Join and Aggregation: Insert Records

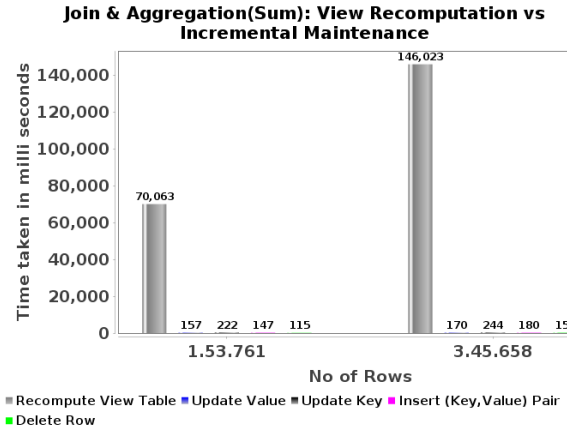
We ran this experiment on the basis of data collected in section 6.3.1. In this experiment, we split our tables into multiple regions, and we want to show that splitting table across multiple regions enhances performance of the system. From the graph 6.2, $Fig(c)$ shows that we were able to optimize performance by splitting tables into multiple regions. In x-axis we show no. of regions and in y-axis we show time taken in milliseconds to insert records in base table, intermediate table and view table.

6.3.4 Join and Aggregation: Update View Table

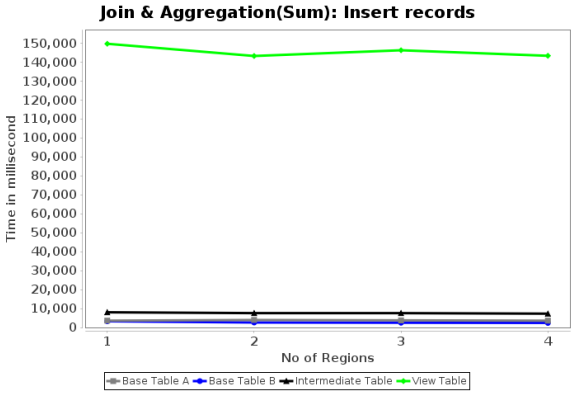
In section 6.3.3, we showed that client operation time can be reduced significantly if we split our tables across multiple regions. In this section,we ran an experiment to see how we are able to optimize performance if we split $View\ Table$ across multiple regions. For that, we split $View\ Table$ across multiple regions and ran this experiment. From the graph 6.2, $Fig(d)$ proved that we also can reduce operation time if we split $View\ Table$ across multiple regions. In x-axis we show no. of regions and in y-axis we show time in milliseconds required to complete client operation. We ran this experiment on different types of client operations and for each operation we were able to decrease time taken to complete required operation.



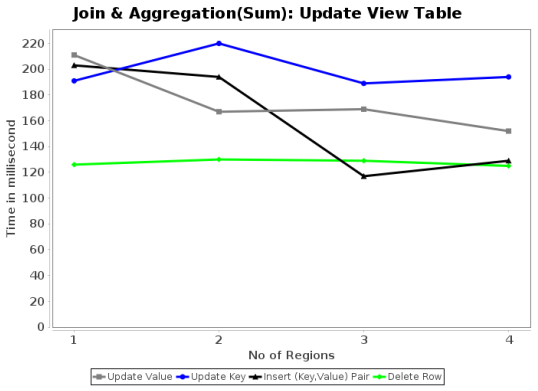
Fig(a): Rows Inserted k-fk join



Fig(b): View Recomputaion vs Maintenance



Fig(c): Insert Records



Fig(d): Update View Table

Figure 6.2: Join and Aggregation Experiment Standalone

6.4 Experiment 3 (Join and Selection)

For $Join$ & $Selection$, we create two base tables as it also involves k-kf joins, an intermediate table and a view table. We insert 3.000 rows in both the base table, apply k-fk joins and select values for those joins and insert result in the view table. We then run experiments to see how we can improve performance as we split our tables across multiple regions.

6.4.1 Join and Selection: Insert records

In this experiment, we insert 3.000 records in each of the base tables and apply k-fk join in the two base tables. First we run this experiment in a single region and see the number of rows generated in the view table after applying k-fk joins. We run this experiment for 2, 3 and 4 regions and split our tables accordingly across those regions. In the figure 6.3, Fig(a) shows the graph of no. of records inserted in view tables. We use this statistics for running other sub-experiments.

6.4.2 Join and Selection: View Re-computation vs Maintenance

In this experiment, we wanted to show that by incrementally maintaining our view tables, we can increase the overall performance of the system. We ran this experiment on the basis of data gathered in sub-section 6.4.1. As a proof, we first ran this experiment on dataset of 1.53.561 records. From the graph 6.3, $Fig(b)$ shows the time required for re-computing view table vs time required to incrementally update view table. If we need to re-compute view table, in such scenario, we first need to perform read operation on the entire base table and then write operation in view table. Whereas in incremental maintenance, we only update the affected rows in view table. To ensure our hypothesis works, we ran our experiment on bigger dataset of 3.45.049 rows. From the experiment results, we proved that maintaining views incrementally is significantly less expensive than re-computing view table for every client request in base table.

6.4.3 Join and Selection: Insert Records

We ran this experiment on the basis of data collected in section 6.4.1. In this experiment, we split our tables into multiple regions, and we want to show that splitting table across multiple regions enhances performance of the system. From the graph 6.3, $Fig(c)$ shows that we were able to optimize performance by splitting tables into multiple regions. In x-axis we show no. of regions and in y-axis we show time taken in milliseconds to insert records in base table, intermediate table and view table.

6.4.4 Join and Selection: Update View Table

In section 6.4.3, we showed that client operation time can be reduced significantly if we split our tables across multiple regions. In this section,we ran an experiment to see how

we are able to optimize performance if we split $View\ Table$ across multiple regions. For that, we split $View\ Table$ across multiple regions and ran this experiment. From the graph 6.3, $Fig(d)$ proved that we also can reduce operation time if we split $View\ Table$ across multiple regions. In x-axis we show no. of regions and in y-axis we show time in milliseconds required to complete client operation. We ran this experiment on different types of client operations and for each operation we were able to decrease time taken to complete required operation.

6.5 Experiment Setup (Fully Distributed Mode)

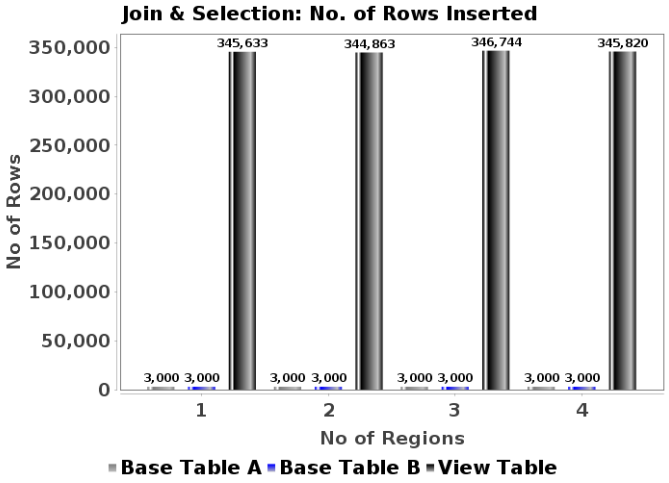
In fully distributed mode, we have performed four different kinds of experiments for each of the three scenarios. First experiment $View\ Re\text{-}computation\ vs\ Maintenance$ is performed on three different datasets while rest of the two experiments are performed on fixed dataset.

6.5.1 Deployment

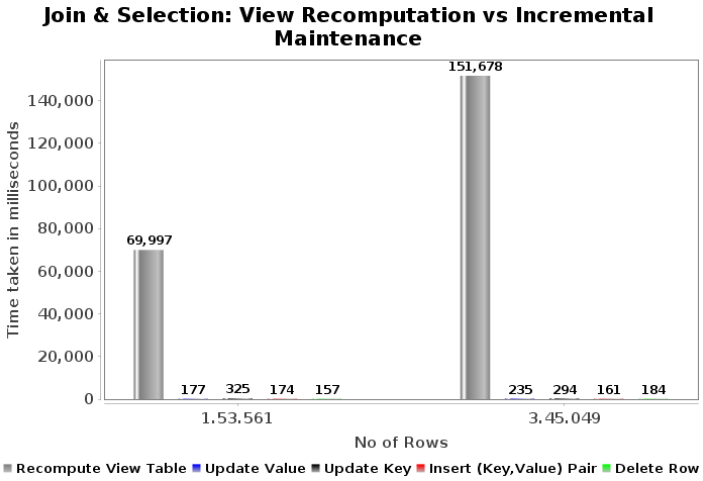
We performed our experiment on a multi node cluster of 3 nodes, two of them are slave nodes and one is master node. We installed two virtual machines for the slave nodes and the master node running as a primary operating system. The master node is running on a computer server (Ubuntu 16.04 LTS, Intel Core i5-3230M CPU @ 2.60GHz x 4, 7.7GB RAM, 364.5GB HD). Both the slave nodes were running on a computer server (Ubuntu 16.04 LTS, Intel Core i5-3230M CPU @ 2.60GHz, 2.0GB RAM, 30.3GB HD). We installed hadoop in fully distributed mode. The services like JobTracker, TaskTracker, Namenode and Datanode runs as a separate Java process in a multiple cluster. We installed hadoop version 2.6.4 and hbase version 1.1.5 for our experiments. DataNode, TaskTracker runs on the slave machine whereas Namenode and JobTracker runs on the master. We have three Region servers, each running on master and slaves respectively.

6.5.2 Table Configuration

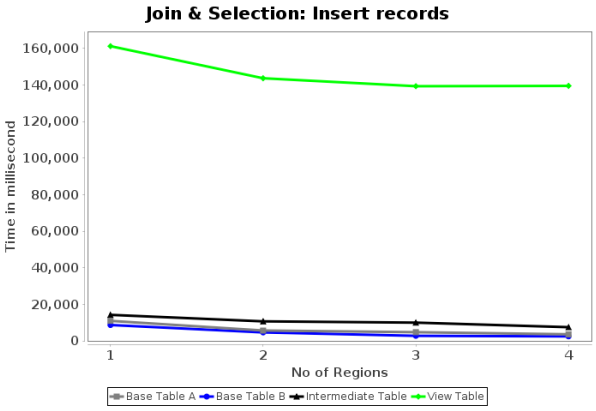
The table configuration for fully distributed is similar to pseudo distributed mode as described in sub-section 6.1.2.



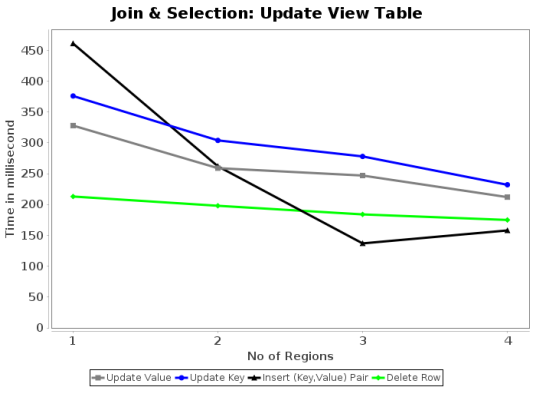
Fig(a): Rows Inserted using k-fk joins



Fig(b): View Re-computation vs Maintenance



Fig(c): Insert records



Fig(d): Update View Tables

Figure 6.3: Join and Selection Experiment Standalone

6.5.3 Control Parameter

There are certain control parameters defined for our experiments to determine performance.

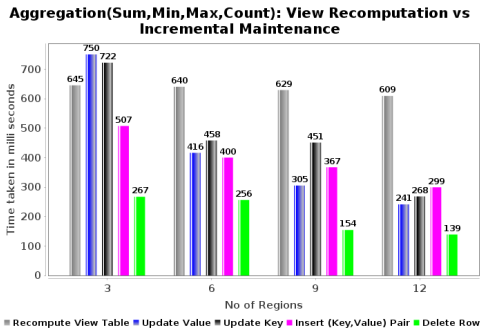
- $noOfRegionServers$: The number of Region Servers. Each Region Server can have one or many regions
- $noOfRegions$: The number of regions within a Region Server
- $typesOfOperation$: The type of operation performed by the client. In our experiment, we've performed insert, update and delete operation.
- $typesOfViews$: This parameter defines the types of views we have implemented in our experiments such as $Join$, $Selection$, Sum, $Count$, Max, Min.
- $timeInMillis$: This parameter defines the time taken in milliseconds to perform certain operations.

6.6 Experiment 4 (Aggregation)

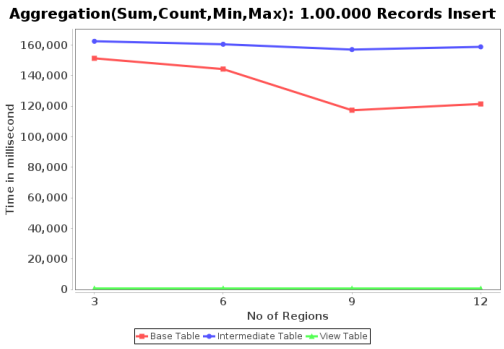
In this experiment, we perform three different sub set of experiments to see how we can improve performance by incrementally maintaining consistency between base table and view table. In this experiment, we have three region servers, two region servers running on slave machines and one region server running on master.

6.6.1 Aggregation: View Re-computation vs Maintenance

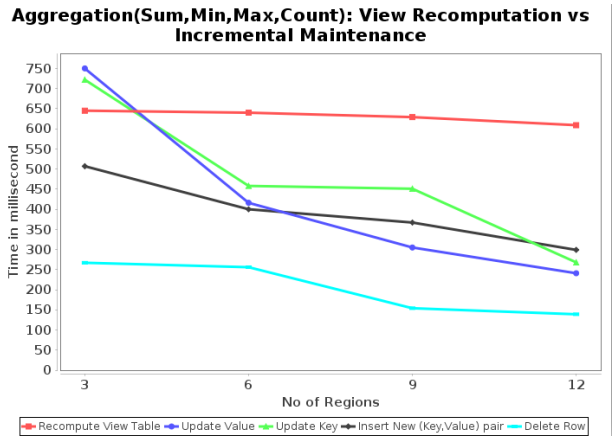
To evaluate performance in View Re-computation vs Maintenance, we first create a base table with 1.00.000 rows and generate corresponding intermediate and view tables. The view table consists of four different aggregations of the base table data, namely Sum, $Count$, Min and Max. For operation in a base table, view table can be updated to reflect changes in two different ways, (a) re-compute (b) incremental update. In our experiments, we compare the execution time for these two methods. Further, we perform the experiments after splitting table across multiple regions. In our experiments, we report the execution time in milliseconds. In the figure 6.4, in x-axis, we show no. of regions and in y-axis we show time taken to insert records in millisecond.



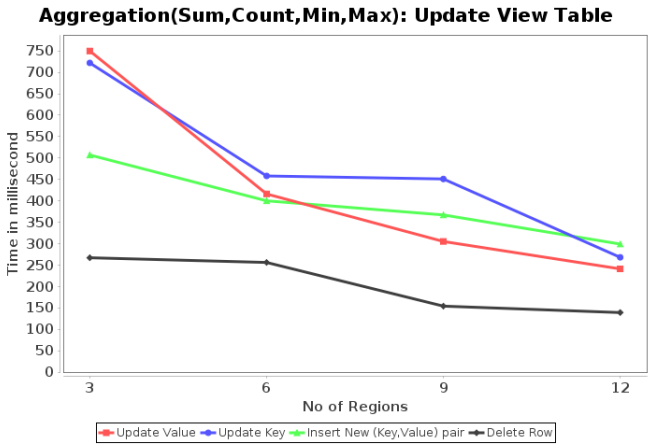
Fig(a): View Re-computation vs Maintenance



Fig(c): Insert 1.00.000 records



Fig(b): Re-computation vs Maintenance: Line Chart



Fig(d): Update view tables incrementally

Figure 6.4: Aggregation Experiment Distributed

From the Fig(a) and Fig(b), we proved that incremental maintenance takes less time than re-computing view table to maintain consistency between base table and view table. The reason for the significant improvement in performance of incremental update over the baseline is due to read-write and recompute overhead. We ran our experiment in distributed cluster environment, from one region per region server to four regions per region server. The reason for this improvement is that the table is split horizontally across all the regions. We implemented $HexStringSplit$ algorithm to partition the table evenly across all the regions, as a result of which the load is distributed equally among all the regions.

6.6.2 Aggregation: Insert Records

In this experiment we wanted to prove that splitting tables in multiple regions across multiple region servers in a cluster also improves the performance. We performed the experiment on a dataset with 100,000 records in a clustered environment. In the first scenario, we had one region in each of the region servers making three regions in total. First we inserted 1.00.000 records and compute intermediate and view table and noted the time take for each operation. We ran same experiment for two, three and four regions per region server and noted time take for each operation to be completed. We plotted a graph and from the Fig(c) in the figure 6.4, we proved that having more regions yields to the better performance. The key reason for this performance optimization is having load balancer to evenly distribute load to all the available regions.

6.6.3 Aggregation: Update View Table

In section 6.6.2, we proved that we can improve performance if we split our tables in multiple regions across multiple region servers. In this experiment, we wanted to see how it affects the performance if we split $View\ Table$ in multiple regions across multiple region servers. In the figure 6.4, $Fig(d)$, in x-axis we show no. of regions and in y-axis, we show time taken to update view table incrementally in millisecond. We first insert 1.00.000 records in base table and pre-compute view tables accordingly. For each of the operation in base table, we evaluate changes in the view table incrementally and plot the graph. Having equally distributed load among the regions, we proved that we can improve the overall performance.

6.7 Experiment 5 (Join and Aggregation)

For $Join$ & $Aggregation$, we create two base tables as it involves k-kf joins and generate corresponding intermediate and view tables. We insert 2.000 rows in both the base table, apply k-fk joins and calculate sum for k-fk joins and insert result in the view table. We then run experiments to see how we can improve performance as we split our tables in multiple regions across multiple region servers.

6.7.1 Join and Aggregation: Compute base tables and view table

In this experiment, we insert 2.000 records in both base tables. We then apply k-fk join and compute intermediate table and view table. We calculate sum based on the result of k-fk join and store it in the view table. In such case, we don't know the no. of rows generated based on the k-fk join so we analyze no. of rows generated as a benchmark for rest of the experiments. We pre-split tables into multiple regions and run the experiment for each of the scenarios. Based on the k-fk join on the base tables, we generated 1.53.000-1.55.000 rows in the view table.

6.7.2 Join and Aggregation: View Re-computation vs Maintenance

To evaluate performance in View Re-computation vs Maintenance, we first create two base tables with 2.000 rows and generate corresponding intermediate and view tables. The view table consists of a aggregation(sum) of k-fk join between the two base table data. We wanted to show that incrementally maintaining our view tables yields in significant improvement on the performance. To prove our assumption, we ran our experiment on the basis of rows generated in section 6.7.1. From the Figure 6.5, Fig(a) shows time taken to re-compute view table vs time taken to incrementally update view table. Here we observe that incremental method yields up to 98% improvement over the baseline. The reason for the significant improvement in performance of incremental update over the baseline is due to read-write and recompute overhead. In the re-compute method, we need to read the whole base table and reconstruct the view table. However, in the incremental method, we do not reconstruct the view table but only update the rows that are affected by the operation on the base table.

6.7.3 Join and Aggregation: Insert Records

In this experiment we first create two base tables with 2.000 rows and generate corresponding intermediate and view tables. The view table consists of aggregation(sum) of k-fk join between the two base table data. In this experiment we wanted to prove that evenly distributing load to multiple regions can yield to a significant performance optimization. For that we evenly distributed base tables, intermediate table and a view table across all the regions on a multiple region servers. To prove our assumption, we ran our experiment on the basis of rows generated in section 6.7.1. In the x-axis, we showed no. of regions and in y-axis we showed time taken in millisecond to complete the operation. From the figure 6.5, Fig(c), we can see that splitting tables into multiple regions and using load balancing can significantly reduce operation time.

6.7.4 Join and Aggregation: Update View Table

In section 6.7.3, we proved that we can improve optimization if we split our tables in multiple regions across multiple region servers. In this experiment, we wanted to see how

it affects the performance if we split $View\ Table$ across multiple regions. For that, we split $View\ Table$ across multiple regions and ran this experiment. In the figure 6.5, $Fig(d)$, we show time taken to update view table incrementally in millisecond. In the first experiment set, we pre-split $View\ Table$ into three regions. Then we evaluate time taken to update view table incrementally for each of the update operations on the base table. We run this experiment again after pre-splitting $view\ table$ into multiple regions and evaluate time taken to update view table incrementally. From the figure 6.5, $Fig(d)$, we again proved that the more table is split into more no. of regions, the less time is required to update view table incrementally.

6.8 Experiment 6 (Join and Selection)

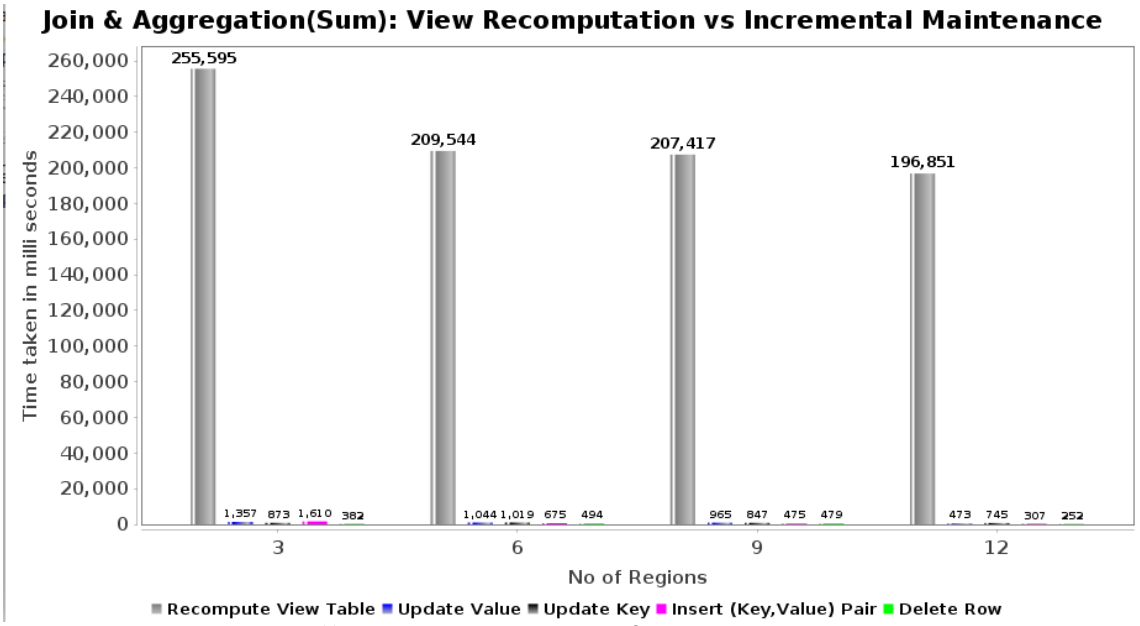
For $Join$ & $Selection$, we create two base tables as it also involves k-kf joins, an intermediate table and a view table. We insert 3.000 rows in both the base table, apply k-fk joins and select values for those joins and insert result in the view table. We then run experiments to see how we can improve performance as we split our tables across multiple regions.

6.8.1 Join and Selection: Compute base tables and view tables

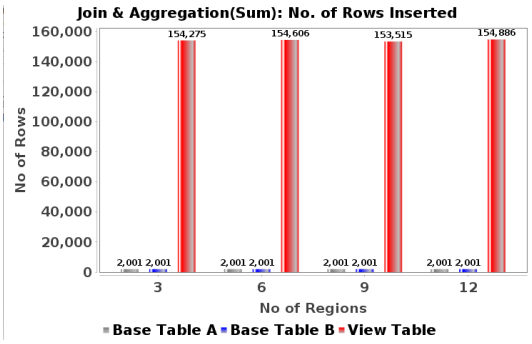
In this experiment, we insert 2.000 records in both base tables. We then apply k-fk join and compute intermediate table and view table. We then select values based on the result of k-fk join and store it in the view table. In such case, we don't know the no. of rows generated based on the k-fk join so we analyze no. of rows generated as a benchmark for rest of the experiments. We pre-split tables into multiple regions and run the experiment for each of the scenarios. Based on the k-fk join on the base tables, we generated 1.53.000-1.54.000 rows in the view table.

6.8.2 Join and Selection: View Re-computation vs Maintenance

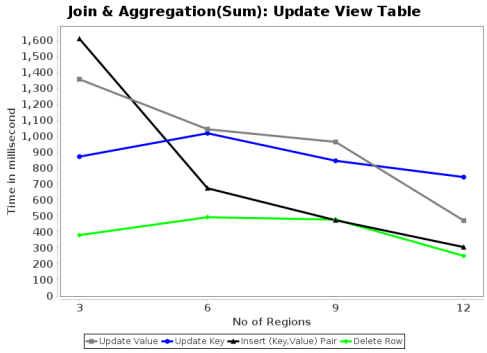
To evaluate performance in View Re-computation vs Maintenance, we first create two base tables with 2.000 rows and generate corresponding intermediate and view tables. The view table consists of a selection of k-fk join between the two base table data. We wanted to show that incrementally maintaining our view tables yields in significant improvement on the performance. To prove our assumption, we ran our experiment on the basis of rows generated in section 6.8.1. From the Figure 6.6, Fig(a) shows time taken to re-compute view table vs time taken to incrementally update view table. Here we observe that incremental method yields up to 98% improvement over the baseline. The reason for the significant improvement in performance of incremental update over the baseline is due to read-write and recompute overhead. In the re-compute method, we need to read the whole base table and reconstruct the view table. However, in the incremental method, we do not reconstruct the view table but only update the rows that are affected by the operation on the base table.



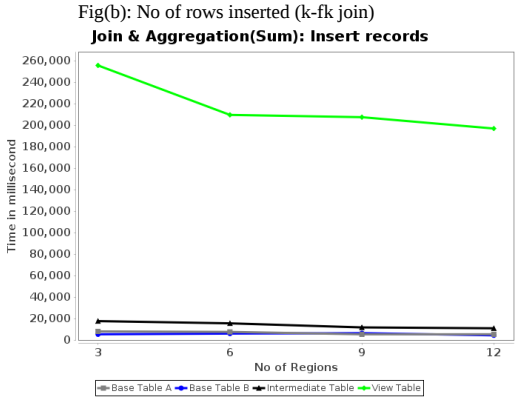
Fig(a): View Re-computation vs Incremental Maintenance



Fig(b): No of rows inserted (k-fk join)



Fig(d): Update View Table



Fig(c): Insert records (k-fk join) Line chart

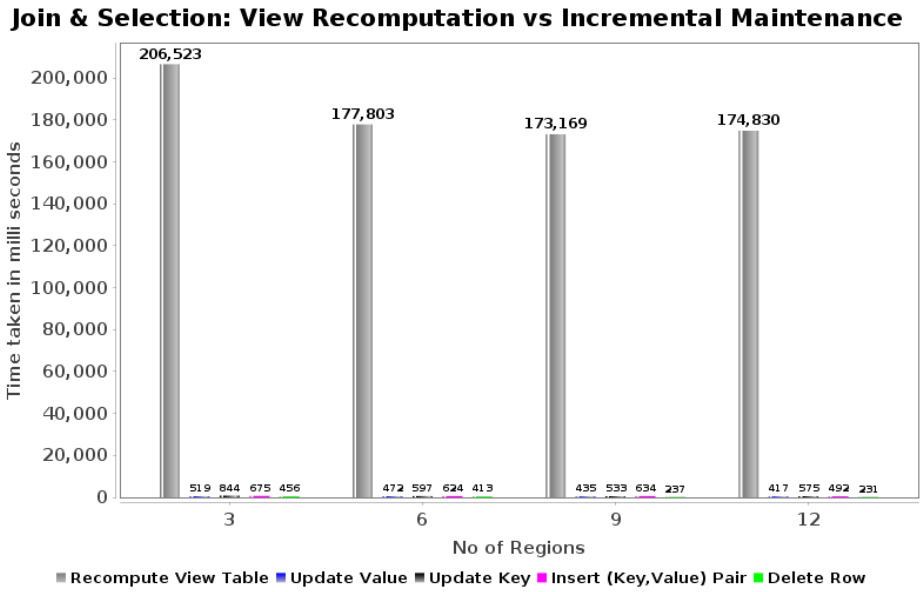
Figure 6.5: Join and Aggregation Experiment Distributed

6.8.3 Join and Selection: Insert Records

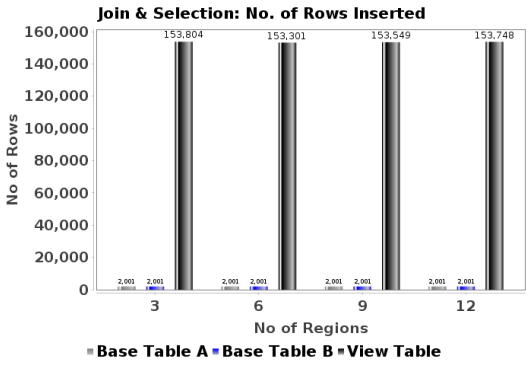
In this experiment we first create two base tables with 2.000 rows and generate corresponding intermediate and view tables. The view table consists of selection of k-fk join between the two base table data. In this experiment we wanted to prove that evenly distributing load to multiple regions can yield to a significant performance optimization. For that we evenly distributed base tables, intermediate table and a view table across all the regions on a multiple region servers. To prove our assumption, we ran our experiment on the basis of rows generated in section 6.8.1. In the x-axis, we showed no. of regions and in y-axis we showed time taken in millisecond to complete the operation. From the figure 6.6, Fig(c), we can see that splitting tables into multiple regions and using load balancing can significantly reduce operation time.

6.8.4 Join and Selection: Update View Table

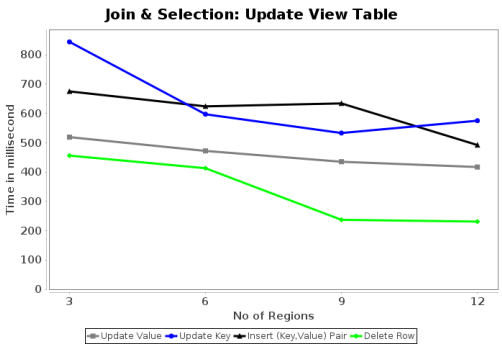
In section 6.8.3, we proved that we can improve optimization if we split our tables in multiple regions across multiple region servers. In this experiment, we wanted to see how it affects the performance if we split $View\ Table$ across multiple regions. For that, we split $View\ Table$ across multiple regions and ran this experiment. In the figure 6.6, $Fig(d)$, we show time taken to update view table incrementally in millisecond. In the first experiment set, we pre-split $View\ Table$ into three regions. Then we evaluate time taken to update view table incrementally for each of the update operations on the base table. We run this experiment again after pre-splitting $view\ table$ into multiple regions and evaluate time taken to update view table incrementally. From the figure 6.6, $Fig(d)$, we again proved that the more table is split into more no. of regions, the less time is required to update view table incrementally.



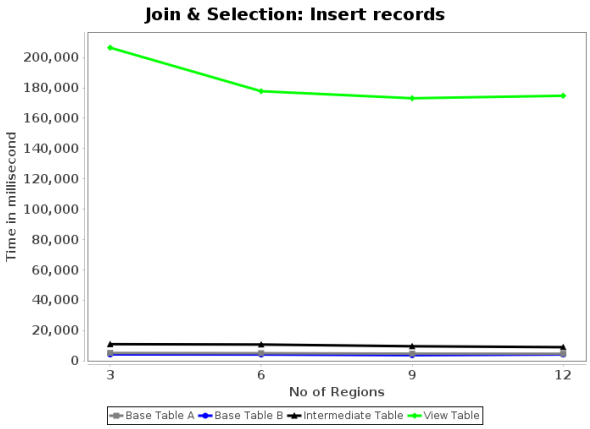
Fig(a): View Re-computation vs Incremental Maintenance



Fig(b): No of Rows Inserted (k-fk join)



Fig(d): Update View Tables



Fig(c): Insert Records (k-fk join) Line Chart

Figure 6.6: Join and Selection Experiment Distributed

7 Conclusion

We have presented an approach to maintain consistency between base table and view table incrementally using HBase Coprocessor. We have provided a solution for incremental maintenance of view table without re-computing view table for every updates on base table. We managed to load coprocessor in the base table. For every updates on the base table, we managed to trigger required updates on the view table using coprocessor and apply them in the view table. This solution addressed the problem of having read-write overhead for re-computing view table even for small changes in the base table. We introduced an intermediate table where all the values for a particular key were plotted in a single row. So for every update for a particular key on base table, we only need to scan one row in intermediate table to determine updates required in view table. This architecture overcomes read overhead i.e. we only need to scan one particular row instead of scanning whole base table to determine changes required for view table. As we overcome read-write overhead, we managed to improve performance of the system up to 98% in the best case scenario. To prove our hypothesis, we ran experiments on different datasets ranging from 10.000 rows to 2.50.000 rows. We tested our solution for different view types namely Sum, $Count$, Min, Max and $k - fk\ join$ views. We performed several experiments both on pseudo-distributed mode and fully distributed mode. In both environments, we were able to optimize performance significantly.

8 Future Work

We have used HBase Client API and HBase shell to read and write data into the HBase tables. Many of us are more familiar with SQL queries, we also can use SQL like queries to read from and write into hbase tables. Apache Spark provides Spark DataFrame Data-Source which helps to integrate Spark SQL with HBase and write SQL alike queries to read and write into hbase tables. Apache Hive also provides HiveQL for SQL like query to the HBase tables.
We ran our experiment in a laptop with two virtual machine acting as a multi-node cluster. It would be more interesting to run our experiment on cluster of more than 20 nodes with a dataset of few hundred millions records and see the results.

Appendix

Bibliography

[1] Apache hbase reference guide, Apr 2011.

[2] Quick start, Apr 2014.

[3] Serge Abitebouly, Jason McHughz, Michael Rysz, Vasilis Vassalosz, and Janet L. Wienerz. Incremental maintenance for materialized views over semistructured data. 1998.

[4] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. *SIGMOD Rec.*, 26(2):417–427, June 1997.

[5] Matt Allen. Relational databases are not designed for scale, November 2015.

[6] GAURAV BHARDWAJ. The how to of hbase coprocessors, Apr 2014.

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

[8] Robert Chansler, Hairong Kuang, Sanjay Radia, Konstantin Shvachko, and Suresh Srinivas. The hadoop distributed file system, 2011.

[9] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianosz, and Kyuseok Shim. Optimizing queries with materialized views. June 1995.

[10] Oracle Corporation. Materialized views, 1999.

[11] TK Das and P.Mohan Kumar. Big data analytics: A framework for unstructured data analysis. 5(1):152–153, Feb-Mar 2013.

[12] Big Data and Hadoop. Apache hadoop hdfs architecture, May 2013.

[13] Nick Dimiduk and Amandeep Khurana. Hbase in action. 2013.

[14] Nishant Garg. Hbase essentials. page 164, Nov 2014.

[15] Mark Grover. Zookeeper fundamentals, deployment, and applications, Aug 2013.

[16] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. June 1995.

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

[18] Dan Han and Eleni Stroulia. Hgrid: A data model for large geospatial data sets in hbase. 2013.

[19] Cloudera Inc. Cloudera installation and upgrade. page 164, Apr 2016.

[20] Edureka Inc. Insights on hbase architecture, Jan 2014.

[21] Mingjie Lai, Eugene Koontz, and Andrew Purtell. Coprocessor introduction, 2012.

[22] Sayed Mahbub and Hasan Amiri. Advantage & disadvantage of relational database, Apr 2016.

[23] Carol McDonald. An in-depth look at the hbase architecture, Aug 2015.

[24] Rohit Menon. Introducing hadoop - part ii, Jan 2013.

[25] Mukesh Mohania, Shin'ichi Konomi, and Yahiko Kambayashi. Incremental maintenance of materialized views. 1308:551–560, June 2005.

[26] Madhukara Pathak. Secondary namenode - what it really do?, Dec 2013.

[27] Daniel Price. Surprising facts and stats about the big data industry, March 2015.

[28] Abraham Silberschatz, Henry F. Korth, and Shashank Sudarshan. Database system concepts. 4.

[29] Hadoop Team. Namenode and datanode - hadoop in real world, July 2015.