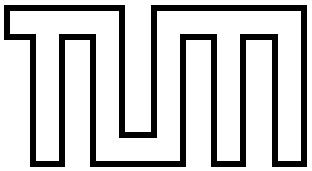


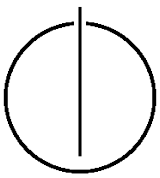
FAKULTÄT FÜR INFORMATIK

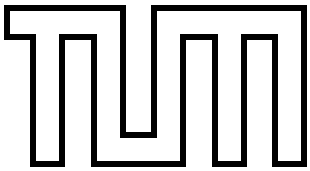
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Materialized views with Apache Spark

Saroj Gautam





FAKULTÄT FÜR INFORMATIK

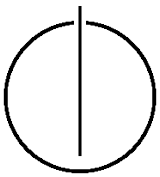
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

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

Whenever we see our friends posting pictures on Facebook or Instagram, we like them or comment on them. Whenever we feel like sharing our thoughts, we either update status on Facebook or just tweet about it. If we need some relevant information, we just google it. The amount of data generated in such a fashion has to be stored somewhere. Companies like Facebook stores 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]. 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.

HBase is an open source sortedMap Datastore from Apache Software Foundation which is used as a database to store huge volume of data. HBase supports horizontal scalability, i.e. parts of a table can be put on several machines. This way a table is broken down into multiple pieces, thus making computation fast. But when we are talking about petabytes of data, scanning each part of the table for a single user query is still considered to be expensive regarding processing time. There are several techniques to reduce this effort, but we will be talking about $Materialized\,Views$ approach.

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 modelled 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.

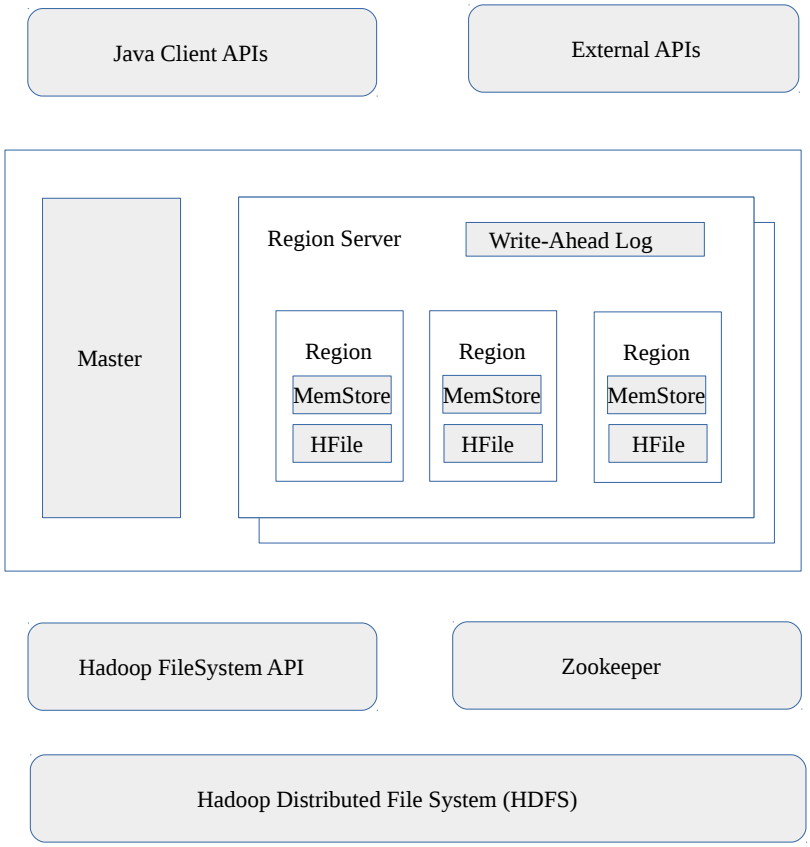


Figure: HBase Architecture

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.

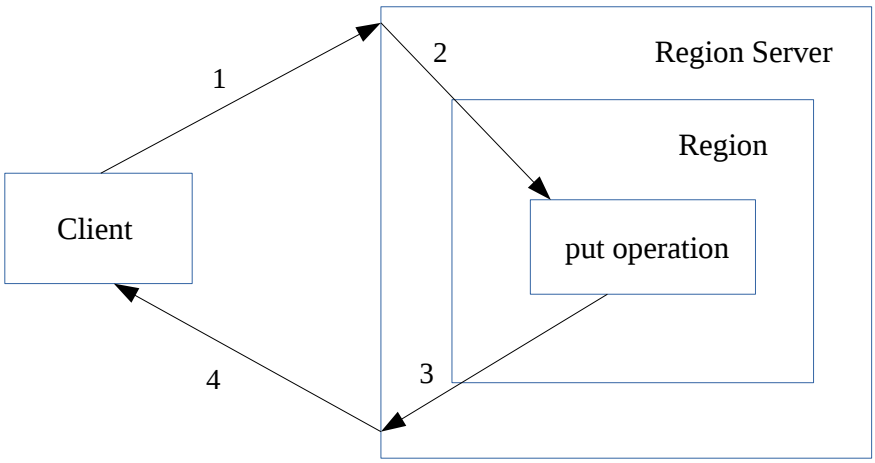


Figure: Lifecycle of a put() request in HBase

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

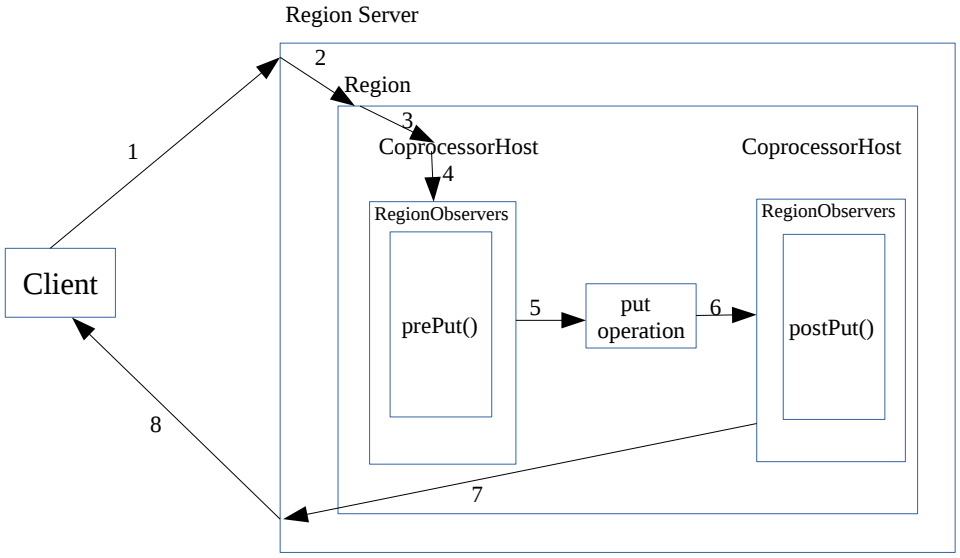


Figure: Lifecycle of put request with Observer coprocessor

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].

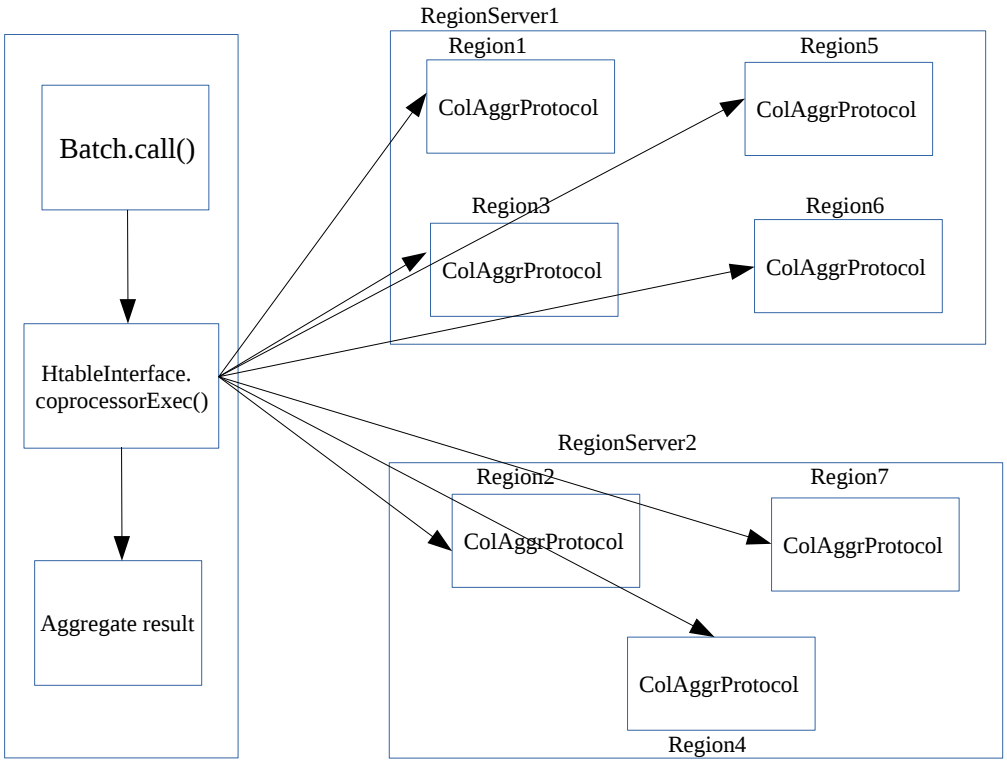


Figure 2.5: EndPoint Coprocessor

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 Implementation

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

4.1 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].

4.1.1 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.

4.1.2 Static Unloading of coprocesssor

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

4.1.3 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>)
```

4.1.4 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.

4.2 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 intermediate table in order to restrict the 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.

Creation of Intermediate table

If there are two base tables, then we merge column families of both tables into the intermediate table. If there is only one base table, then we have the same column family in our intermediate table. The key from base table becomes row key for the intermediate table, $rowkey$ of the base table becomes $column\ qualifier$ in the intermediate table. So the value for key and row key from base table is now plotted in intermediate table for that particular key. So whenever there is a CRUD operation for a particular key, we can scan row for the particular key instead of scanning the whole table. The figure below explains the transformation of base table into an intermediate table in more detail.

AggrTable

	HColumnA	
	Key	Value
x1	k1	10
x2	k2	20

AggrIMTable

	AggrColFam	
	x1	x2
k1	x1,10	
k2		x2,20

Figure 4.1: Intermediate Table

4.2.1 Aggregation

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. The base table contains key,value pairs. We construct an intermediate table from the base table. The reason behind constructing intermediate table is to restrict scanning of the whole base table for an update/delete trigger for a particular key in a base table. We take the unique keys and map them as a row key in the intermediate table, and accordingly the values are plotted. Once all the values are plotted in the intermediate table, we then construct view table. The view table contains aggregate functions like Sum, Count, Min and Max for each row Key i.e. for each unique keys of the base table.

Once we have a base table, intermediate table and view table, and successfully loaded coprocessor 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

Whenever a new row is inserted in a base table with (key,value) pair, the (key,value) pair has to be inserted in the base table and we have to plot the new (key,value) pair in the intermediate table and also view table has to be updated accordingly. 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 being inserted, we let the request to insert new (key,value) pairs to be inserted 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 plot the (key,value) pair in the intermediate table and then update aggregation functions in our view table. The figure below 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 mark the changes that are happening on base table, intermediate table and 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 4.2: 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 row key 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 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. The figure below 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.

Base Table

AggrTable

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

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	30	2	10	20
B	60	2	20	40
C	60	1	60	60

AggrViewTable

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

Figure 4.3: Update value for a existing row key

Existing key of a row is updated

Whenever there is a trigger for key of the particular row key to be updated, we first catch the request via prePut() method. In this scenario, we first have to find out the key to be updated and delete the plotting from intermediate table. In the prePut() method, we find the (key,value) pair for a key to be updated and store it somewhere in memory. Then we release the request and the key is updated in the base table. In this case, now we have $old\ key$ and the $new\ key$.

In the postPut() method, first we find the column to be deleted from the intermediate table. Then we delete that particular column and update our view table accordingly. After the process is complete without any interruption, the process is similar as of inserting new (key,value) pair. We plot the $new\ key$ and $value$ in our intermediate table and update the view table accordingly. This is the most complex scenario because it might affect more than one row in our view table. In the figure below, 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

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	2,20			
B			3,20	4,40	
C					5,60

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	30	2	10	20
B	60	2	20	40
C	60	1	60	60

AggrViewTable

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

Figure 4.4: Update Key for a existing row key

Existing row is deleted

When an existing row is deleted in the base table, in the postPut() method of observer coprocessor, we delete the plotting for that particular key. In this case, there are two scenarios. 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 only has a single plotting, then we delete that plotting from an intermediate table and then also delete the entry for that key from the view table.

In the figure below, 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 4.5: Delete an existing row

4.2.2 Join and Aggregation

In this subsection, we have implemented Join and Aggregation functionalities. For this, we have joined two base tables and in the view table, we have the sum of the values for a same key. We also have an intermediate table where both the base tables are merged by a particular key and values are plotted in the intermediate table base on that key. As described in 4.2.1, we have implemented sum for same key in both the base tables.

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

Whenever a new row is inserted in a base table with (key,value) pair, the (key,value) pair has to be inserted in the base table and we have to plot the new (key,value) pair in the intermediate table and also view table has to be updated accordingly. Here, we create an intermediate table by merging two base tables and plotting values accordingly. The algorithm for plotting values has already been discussed in sub section 4.2. In the view table, we implement sum function for same $keys$ in both base tables. The figure below 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 4.6: New row is inserted for Join and Aggregation

Existing value of a row is updated

When a value of a key is updated in the base table, the consistency should also be maintained in the view table. So when the base table is updated, the request is triggered by the postPut() method and intermediate table and view tables are updated accordingly.

Existing key of a row is updated

As described in 4.2.1, when key is modified in the base table, we first have to delete an entry from the intermediate table and view table if the row exists for that particular key. Once delete operation is completed without interruption, then we insert the updated key in the intermediate table and view table.

Existing row is deleted

The implementation for this scenario is same as described in subsection 4.2.1. The entry is first deleted from the base table, and if an entry exists in the intermediate table and view table, the entries are deleted from those table accordingly.

4.2.3 Join and Selection

This approach is similar to approach described in subsection 4.2.2. We do not use any aggregation functions but instead just select the values and put them in the view table. As already described in 4.2.2, there are also four possible scenarios where we update view table incrementally.

5 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.

5.1 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.

5.1.1 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.

5.1.2 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 4.2.1. 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 4.2.2 and 4.2.3 respectively. After we have our Intermediate table ready, we apply k-kf join and insert data into view table accordingly.

5.1.3 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.

5.2 Experiment 1 (Aggregation)

For $Aggregation$ view type, we have performed three different types of experiments. The first experiment $View\ Re\text{-}computation\ vs\ Maintenance$ is performed on three different datasets on a single region and the remaining two experiments are performed on fixed dataset of 1.00.000 records with varying number of regions on a region server.

5.2.1 View Re-computation vs Maintenance

In the first set, we insert 10.000 rows in the base table and we compute intermediate table and view tables accordingly. In the view table, we compute four different aggregation functions Sum, $Count$, Min and Max. When client issues any update operation, either we re-compute view table from scratch and compute aggregation functions or we update view table incrementally. The graph shows time taken to re-compute view table vs time taken to update view table incrementally for each type of update operations. We conduct same experiment for 1.00.000 and 2.50.000 records. From the graph, we see that updating view table incrementally saves significant amount of time than re-computing view table for every client operation.

6 Discussion and Conclusion

7 Future Work

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.