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 where 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 big volume of data. Even though data are stored in a distributed storage systems, still the huge size of data are bottleneck for performance optimization. Scanning tens of millions rows and few million columns each time are expensive in terms of execution time and processing power. $Materialized\ Views$ solve this problem by precomputing expensive queries and storing result in a physical table or disk. One of the bottleneck for this approach is constantly maintaining consistency between base table and view table. 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 generally 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[13], including 2.7 billion likes and 300 million photos. As of 2012, Facebook already has 100 petabyte of photos[13]. Google, on the other hand processes 3.5 billion request per day [13]. In Early 2000s, where there were less data shared on social media, data were stored in a relational database. Relational database were designed in such a fashion to store small amount of data and maintain integrity between them[4]. 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)[4]. 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 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 to multiple pieces, thus making computation really fast. But when we are talking about petabytes of data, scanning each part of table for a single user query is still considered to be expensive in terms of 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 Materialized Views

Materialized views are 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, in terms of execution time and processing power. Materialized views speeds up the query processing time by pre-computing joins and aggregations prior to execution and stores these results in a table or disk[6].

2.2 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 whole Materialized View from the scratch or using the heuristic of inertia[9] approach i.e. incremental maintenance with respect to the base table.

2.3 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.

2.3.1 Aggregation

In Aggregation view type, the data of 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.

2.3.2 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.

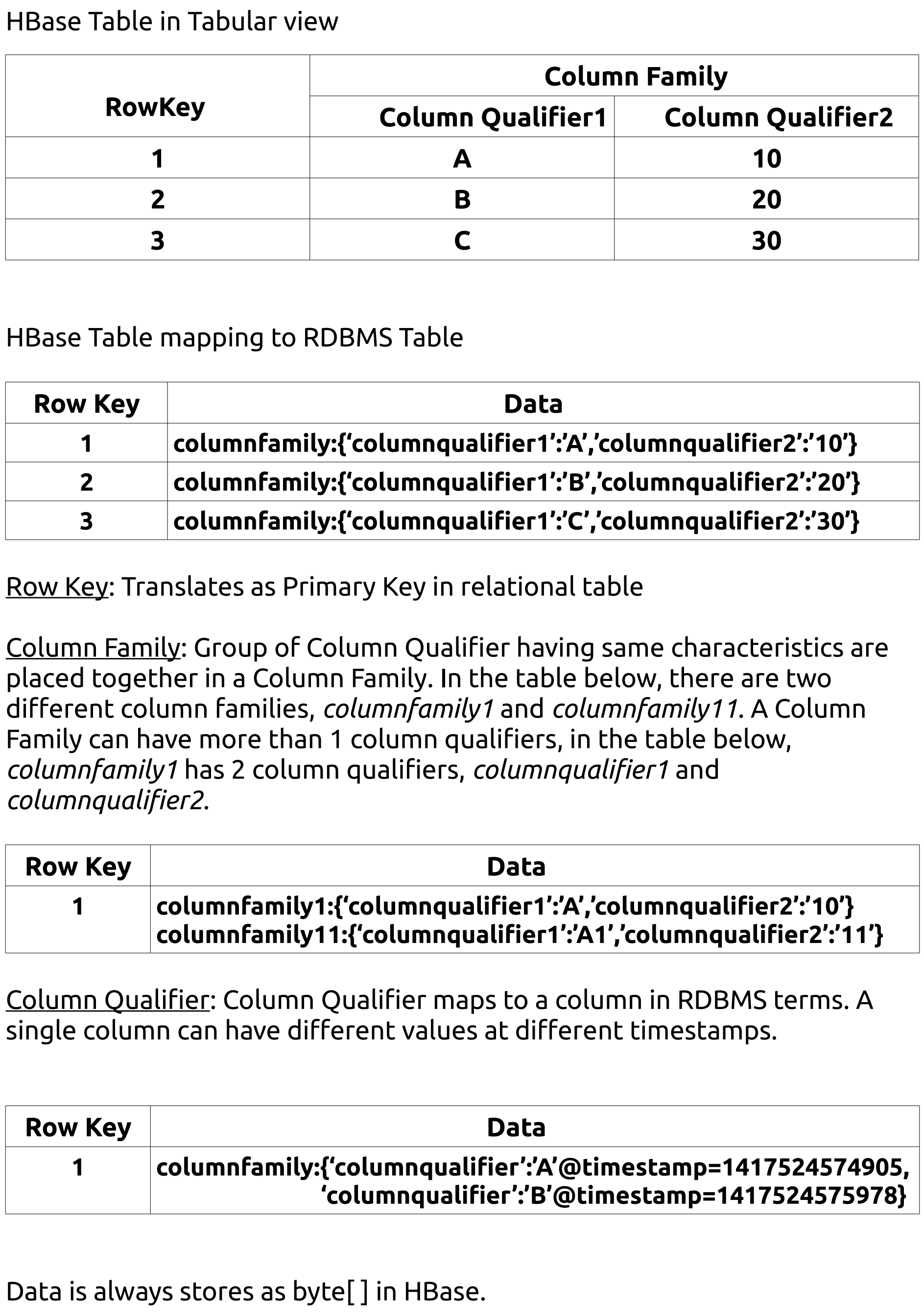
In the intermediate table, the unique keys from both the base tables act as the row key, both column families from base table are merged in the intermediate table. Now for a particular row key, the values are selected from base table and plotted in the intermediate table. Now join is applied between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

2.3.3 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.

2.4 HBase

HBase is an open source sortedMap Datastore from Apache Software Foundation. HBase is modeled after Google's BigTable framework. A basic table structure of HBase consists of Row Key, which is similar to the primary key in relational database table, Column Family and Column Qualifier.



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 Table

2.5 Coprocessor

HBase Coprocessor framework provides 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[10]. There are two types of coprocessor, Observer coprocessor which acts more like relational database triggers and Endpoint coprocessor that resembles stored procedures of RDBMS[12]

2.5.1 Observer coprocessor

Observer coprocessor as stated earlier, are more like database triggers that executes our code when certain events occurs. In the figure below, we first try to explain a simple life cycle of put() operation as an example[7]. 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[7]. 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[7]. There are four types of Observer Interfaces provided as of HBase version 1.1.3[8].

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[8] such as pre and post observers. For instance, preGetOp() and postGetOp() provides hook for manipulating get request.

2. RegionServerObserver: Likewise in RegionObserver, RegionServerObserver provides 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 hook for Write-Ahead-Log(WAL)[8] 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 hook for data manipulation for DDL events such as table creation, table deletion or table modification[11]. For in-

stance, if the secondary indexes needs to be deleted when primary table is deleted, we can use postDeleteTable(). The MasterObserver runs on master node.

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

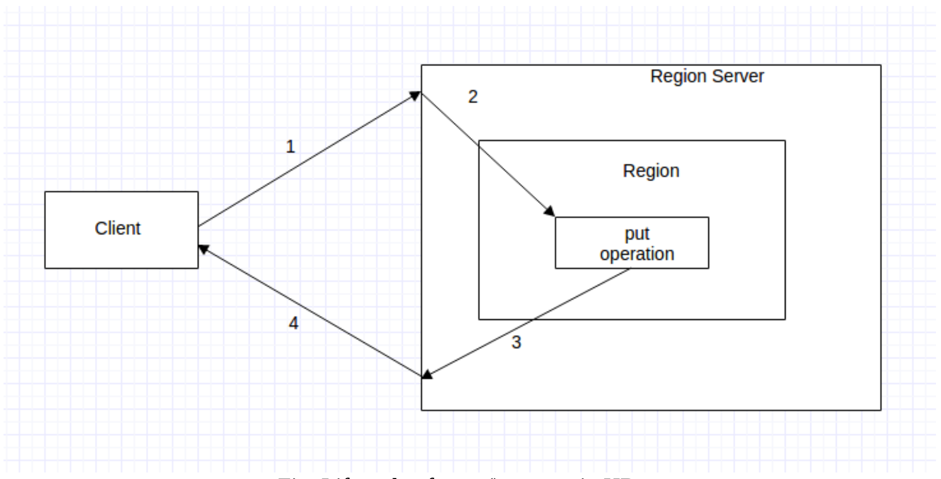


Fig: Lifecycle of a put() request in HBase

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 2.2: Life cycle of put request

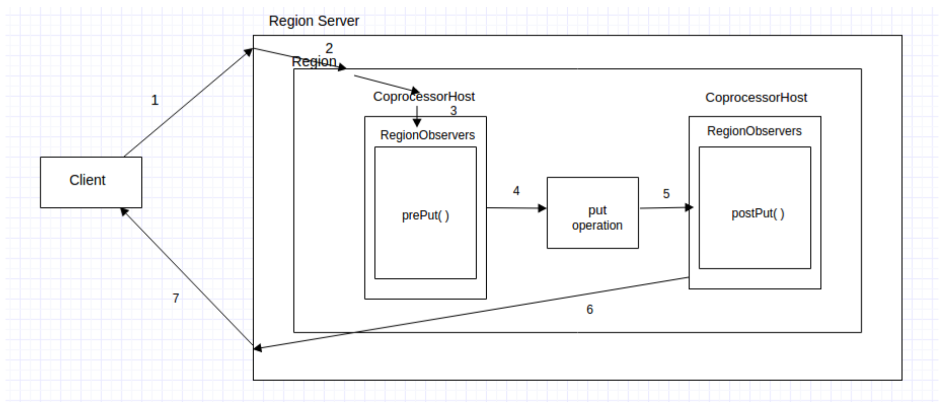


Figure: Lifecycle of put request with Observer coprocessor

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

Figure 2.3: Life cycle of put request with observer coprocessor

2.5.2 Endpoint coprocessor

Endpoint coprocessor are 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[5]. 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[7]. The figure below explains the Aggregation example[7].

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[7].

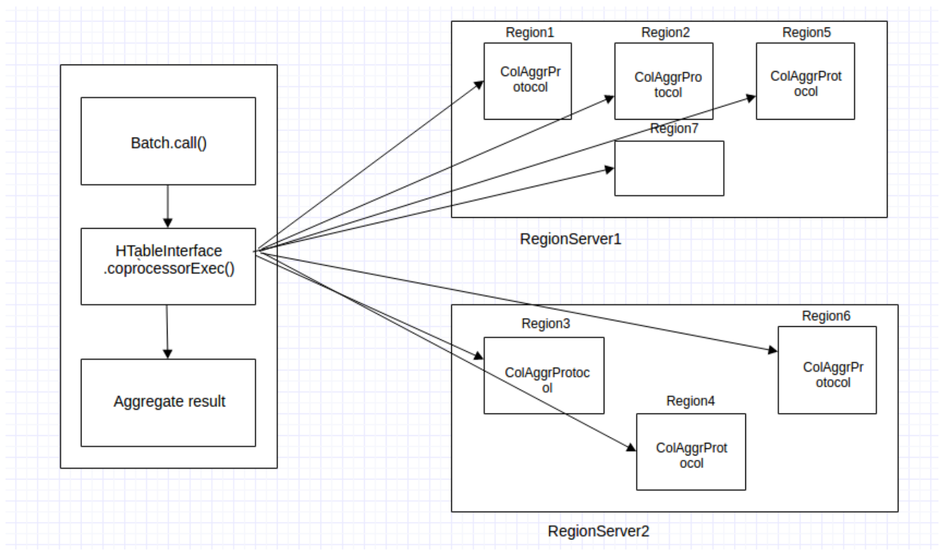


Figure: EndPoint Coprocessor

Figure 2.4: EndPoint Coprocessor

3 Implementation

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

3.1 Prerequisite

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

3.1.1 Static Loading of coprocessor

We have to define coprocessor properties in a *hbase-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 jar to HBase classpath. For that, we have to export /path/to/jar in *hbase-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.

3.1.2 Static Unloading of coprocesssor

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

3.1.3 Dynamic Loading of coprocessor

In this approach, rather than loading coprocessor to all the tables in a Region, the coprocessor are 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/saroj-gautam/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>)
```

3.1.4 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.

3.2 Proposed Method

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

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

One of the most important feature in our implementation is the introduction of intermediate table. We have introduced intermediate table in order to restrict the scanning of 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 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 transformation of base table into 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 3.1: Intermediate Table

3.2.1 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 whole base table for a update/delete trigger for a particular key in base table. We take the unique keys and map them as a row key in 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. The figure below explains how we map base table to a view table.

AggrTable

	HColumnA	
	Key	Value
1	A	10
2	B	20
3	A	30
4	C	30
5	D	20
6	D	50
7	A	70
8	E	20
9	B	60

AggrIMTable

	AggrColFam								
	1	2	3	4	5	6	7	8	9
A	1,10		3,30				7,70		
B		2,20							9,60
C				4,30					
D					5,20	5,50			
E								8,20	

AggrViewTale

	AggrColFam			
	Sum	Count	Min	Max
A	110	3	10	70
B	80	2	20	60
C	30	1	30	30
D	70	2	20	50
E	20	1	20	20

Figure 3.2: Aggregation

Once we have 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 are 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 about put() request life cycle in 2.5.1, 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 new row is inserted. The left side tables are the default tables and right side tables explains the behavior when a new row is inserted. The text displayed in red mark the changes that is 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 3.3: 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 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 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 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 a 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 3.4: 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 3.5: 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 co-processor, 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 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 3.6: Delete an existing row

4 Discussion and Conclusion

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

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

[6] Oracle Corporation. Materialized views, 1999.

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

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

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

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

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

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

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