Project Report

Members:

* Pranav Chaudhary, 193059004

* Shailendra Kirtikar, 193059007

Topic: Secondary Indices in HBase

Subject: CS631 Implementation Techniques for Relational Database Systems

Github Link:

https://github.com/ShailendraKK/SecondaryIndexHbase.git

Project Goal

Apache HBase is a key-value store. As such it does not include either secondary indices, or materialized views. However, it supports Coprocessors which is analogous to stored procedure systems in RDBMS which can be leveraged to implement these features. Thus, we plan to implement secondary indices using coprocessors

Project Plan

We plan to use Observer Coprocessors to create a secondary index table. Observer coprocessors are triggered either before and after a specific event occurs. To use Observer Coprocessor, we plan to implement the RegionObserver interface or extend the class BaseRegionObserver and override its methods such as prePut and preGetOp. So before inserting any data in the table we can insert the <secondary_index_column,row_key> data in the secondary index table by overriding the prePut method where the row key would be the column on which we want to build the secondary index instead of the primary key. Before Retrieving the data, if the query is on the predicate of the column on which the secondary index is created, then we can directly retrieve the row_key (primary key) using the secondary index column as the row_key in the preGetOp method and return the data by using the row key. When we delete the data, we will also delete the corresponding data in the secondary index by overriding the postDelete method. We plan to use the static loading of the coprocessor by modifying the HBase configuration file hbase-site.xml or dynamic loading using the HBase shell depending on the use case.

Implementation

We have implemented the Secondary Index by implementing a RegionObserver and RegionCoprocessor interface. The RegionOvserver interface is important and it gives us the following hooks to execute custom code at specific events

- **start():** → Execute our code when the coprocessor is loaded

We have overridden the start method to create the Secondary Index Table using the details such as Source Table, Column Family, Index Column, and Source Column of the Secondary Index which are passed through the parameters during Coprocessor Loading. Coprocessors are loaded dynamically for a particular table using java program to which we can pass the required parameters for creating a Secondary Index.

- **postPut():** → Execute our code after the put in the main table is completed

We have overridden this method to update our index whenever new relevant data is added to the main table.

- **postDelete():** → Execute our code after the delete operation is completed on main table

We have used this hook to delete the relevant data from the Secondary index table after it is deleted from the main table.

- **preScannerOpen()**: → Execute our code before the scanner is opened for the main table

In preScannerOpen we capture the scan incoming scan request, then convert it into JSON and then determine whether the scan needs to be intercepted and directed through the index table. We store the scan request into JSON as it arrives so that we can extract the information regarding the requested scan later. The scan request JSON contains all the information of the scan that was requested and it looks like the following example.

{"loadColumnFamiliesOnDemand":true,"filter":"ValueFilter (EQUAL, Anna)","startRow":"","stopRow":"","batch":-1,"cacheBlocks":true,"totalColumns":1,"maxResultSize":"2097152","families":{"cf":["first_name"]},"caching":2147483647,"maxVersions":1,"timeRange":["0","9223372036854775807"]}

In our Project, we have intercepted the scans with ValueFilter and SIngleColumnValueFilter on our Secondary Index attribute to go through our index and the rest of the scans proceed as usual. To achieve this we replace the scan object with a dummy scan object that scans on the rowkey value instead of the column value on a non-existent key. Since the scan is on the rowkey, Hbase will efficiently compute it and since the key is non-existent there won't be any result generated for the original scan. This is how we have effectively replaced the actual scanning task with another task involving secondary index lookup.

- **postScannerNext()**: → Execute our code after the client ask for the next rows on scanner

Using the JSON which we stored in the previous step, we query the secondary index for the value for the column that the scan was requested on and we get the corresponding rowkeys of the main table that satisfy the filter on the column on which we created the secondary index. Then the relevant rows are efficiently fetched using the get operation.

Experiment:

- We have generated 1 million rows on which we have tested our secondary index.
- Every row that is inserted corresponds to the following schema:
  ID => (country, first_name, last_name, year, month, productNo, quantity, earnings)

- We have disabled the effect of the caching so that we can test our implementation against regular scanning fairly.
- We have created the index table on a single column (first_name) that corresponds to the following schema.
  first_name|ID => ID
- We have passed the Main Table name, secondary index column family, and column on which the index is to be created.

What have we achieved?

- The secondary index is created and populated automatically.
- Some predefined scans go through the secondary index instead of scanning the whole table.
- Basically, we have implemented an inverted index.
- We have not broken any existing regular scans.
- We have tested our following scans using ValueFilter and these are our results :

First_name search key	**Time in seconds with a coprocessor**	**Time in seconds without a coprocessor**
Kenny	1.45	4.69
Anna	3.62	3.79
Pranav	0.01	2.34
Abigail	0.21	2.82

Limitation and Future Scope:

- Currently, in our implementation, we have created an index on a single column of a single column_family. However, these can be generalized to create an index on multiple or every column and can be easily implemented given sufficient time.
- Our implementation makes some assumptions for simplification like only one single column family is used for all the columns.
- Usually, when a scan request is made, every region server scans its corresponding region, and then the results are combined before displaying it to the user. In our implementation, the rows resulting out of the scan operation are generated at each and every Region server. Due to this, the final result contained rows, number of regions times the actual number of resulting rows.

  This can be resolved in two ways: Elegant solution and Naive solution. We have discussed the Elegant solution at the end, and we have discussed the Naive solution as follows:

1. Mark exactly one region out of all the regions.
2. The Region server that will be allotted the marked region will be the only Region Server that will compute the result.
3. All the other Regions servers will be idle.

This is the bottleneck in our implementation as only one Region server is doing all the work and the rest are idle. We will still see performance benefits as long as the parallelism of the Region server in regular scans does not overpower the Speed benefits that we get by Secondary Index. In our implementation, we are still getting the performance benefits of using Secondary Index against the regular scan using 2 parallel Region Server.

An elegant solution to boost performance:

By default, the split policy is the IncreasingToUpperBoundRegionSplitPolicy. By changing the split policy of the main table to KeyPrefixRegionSplitPolicy, we can split the regions based on the prefix of the rowkey.

In this implementation, we will have index rowkey as first_key|reversed(row_key). We will have to define a new Region Split policy called ReverseKeySuffixRegionSplitPolicy. This policy will essentially split the index table based on the reverse of the prefix of row_key on which the main table was split. This will split the index in such a way that all the keys in one region of the index table will correspond only to the keys in one region of the main table.

Now when a specific Region server of the Secondary Index is operating on its region, it can retrieve the actual main table data from the corresponding region of the main table without contacting the other region that doesn't have that key. Similarly, when every Region server operates in this way, we will achieve the parallelism similar to the regular scans.