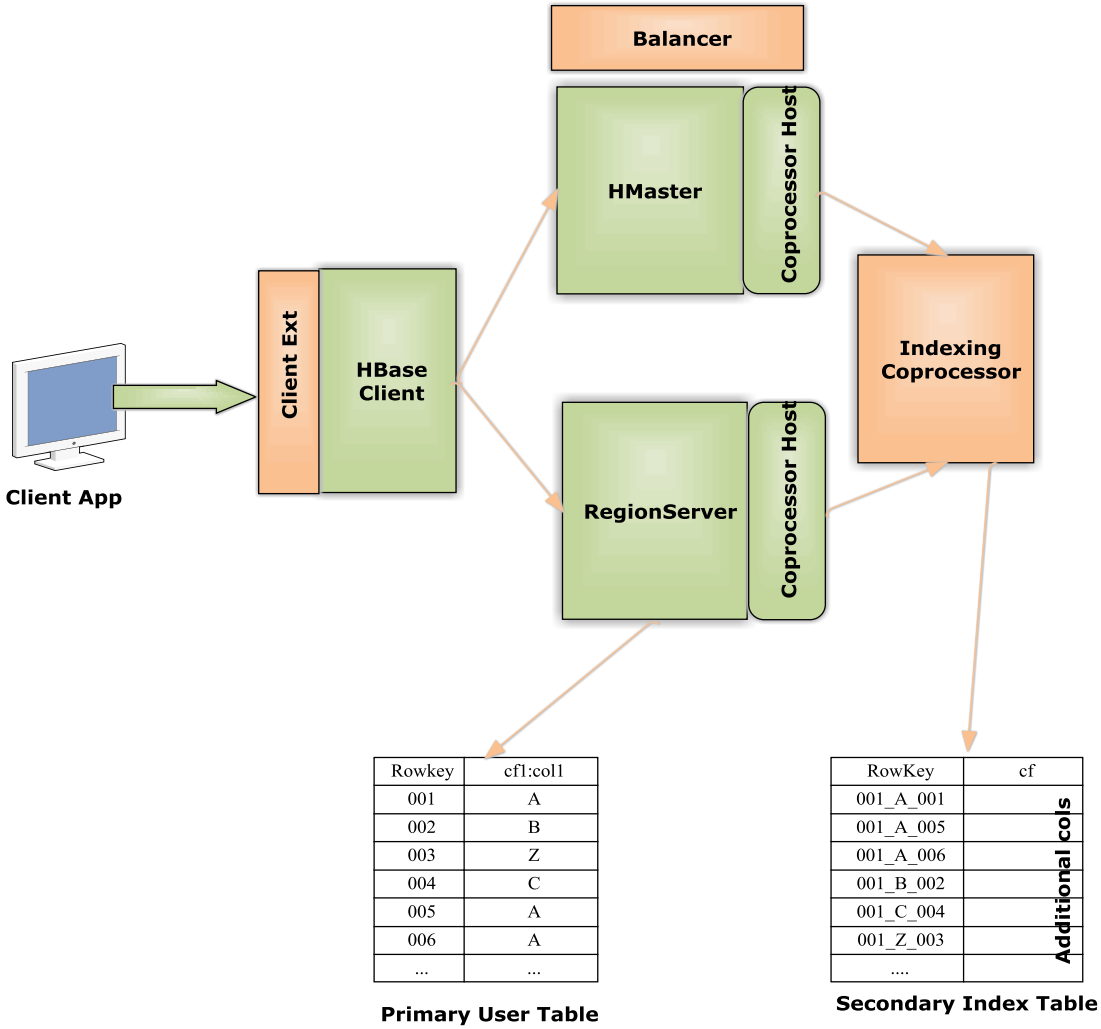
1. Introduction

Indexes in DB will accelerate the data access where data accesses based on some filter condition. Basically all the rows in the table will be having some a primary key and retrieval of data by providing this primary key will be easy. When it comes for data retrieval based on some other column's value or value range, the retrieval become expensive as it needs entire table data scan. In the world of DB index on such a column will make the retrieval very efficient and fast. The basic idea is to map that columns value with the actual row references in some way. So when the data retrieval is needed based on the indexed column value, 1st from the index we can know the actual table row references and from there we can do the actual data fetch. Here by it avoids the full table scan.

In HBase the row key is the one act as a primary key. HBase will store the data in the lexographical sorted order of this row key. Any Get(specify the rowkey) or Scan operation specifying the range of row keys (start and end row keys) won't have much overhead as it is possible to know which all regions to be contacted and in that region also the data is sorted in order of row key. So there won't be any need of the full table scan. When there is a need to fetch data from a table based on a condition on one column value, HBase supports this through the usage of different types of Filters. But this will need a full table scan which is very costly. As index is the mechanism which will help this kind of scenarios in normal DBs, we need to have similar mechanism in HBase tables also. HBase by default do not support any indexing on its tables. As we can call the row key as the primary index in HBase, we can call these as the secondary indices on the table. Even though HBase do not directly support the secondary index, it opens ways for its users to handle these situations.

Architecture



Basic solution approach will be to create another table to capture the index information about a table. From here on we will call the new table as the index table and the actual table on which indexing is done as the user table. When the data is added to the user table corresponding information will get added to the index table immediately. What gets added to index table is the indexed column's value and the rowkey. In our solution we ensure data consistency between the user table and index table always (not like eventual consistency) When data is deleted from the user table corresponding data will get deleted from the index table too. During the scan (with filters for column value condition), if the conditions are on indexed columns, we make use of the index data to avoid the full table scan. Suppose the scan is with a column value condition (like c1=10), from index table, we can get to know all the rows (in fact rowkeys) which we need to fetch.

We try to do the indexing and use index details at server side just as in normal DBs. This will make the system efficient. If the handling is done from client side it will involve more network calls for the data write and for scan, we need to scan the index table data to the client side and then issue get(s) for each of the row. It will greatly affect the data write performance (put). Also the scan performance will not improve especially when the total rows count, which need to get fetched, increases.

So what we need is hooks at the server side where we can run custom logic to deal with the index table. HBase gives such a feature named Co-Processors. We can implement co-processors for master side and region server side and plug-in the same. The server will call the hook methods along with events happening there.

Region Collocation

Another important aspect is the region collocation. As we know HBase splits data in a table into number of regions. Each region is having a start and end key (rowkey) which determines which rows it can hold. Every region will be associated with a region server. The Master process determines which region to go where. There will be region balancing happening in the cluster to make sure the load is equally distributed across all the region servers in cluster. Also when one RS is going down, to make data highly available, the regions from that will get moved to another RS. All these responsibilities will be done by the Master process.

Here by we have two tables, one user table and a corresponding index table. There will be regions for both of them and getting distributed across the RS in cluster. Now when some row is being inserted (Put) into the user table, as per the rowkey, it will be decided to which region that data needs to go. So HBase client side will contact the RS which is serving this region and data will be inserted there. Now when the table is indexed, as part of the write we need to write some data into the index table also. We will be doing this in the CP hooks. So to which region of the index table this data needs to go depends on the number of regions and start and end keys of the regions in the index table. If this index table region is in another RS, to do the write into the index table, we need to make an RPC call which is costly. This can badly affect the write throughput and higher network usage. An effort here is to make the collocation of these 2 regions (user table region and index

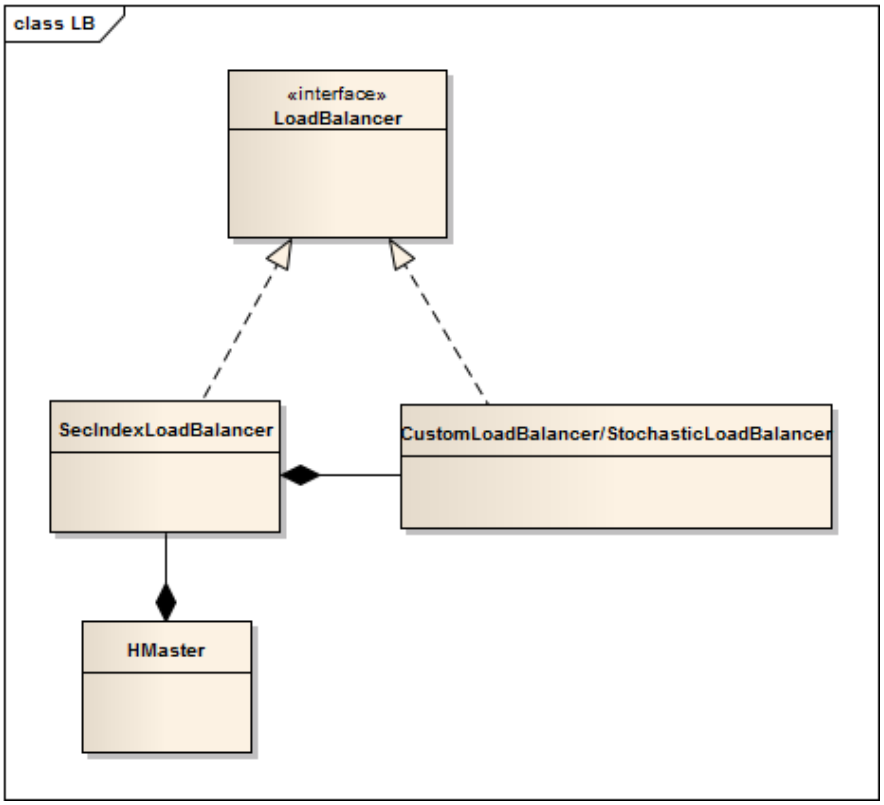
table region) in the same RS. Then there won't be any need of RPC but the CP hook can get the reference to the index table region and write data directly there. For this we need to have a clear association possible between the user table regions and index table regions. Possibly a 1-1 association would be better [We will see this later]. Other than this when master doing the region assignment he need to take care of this collocation part. We will first describe how this collocation part can be taken care. Assume that there is a 1-1 association between the user table and index table regions

Balancer

HBase master uses a load balancer module to do the region assignment across RSs. Region assignment will be needed

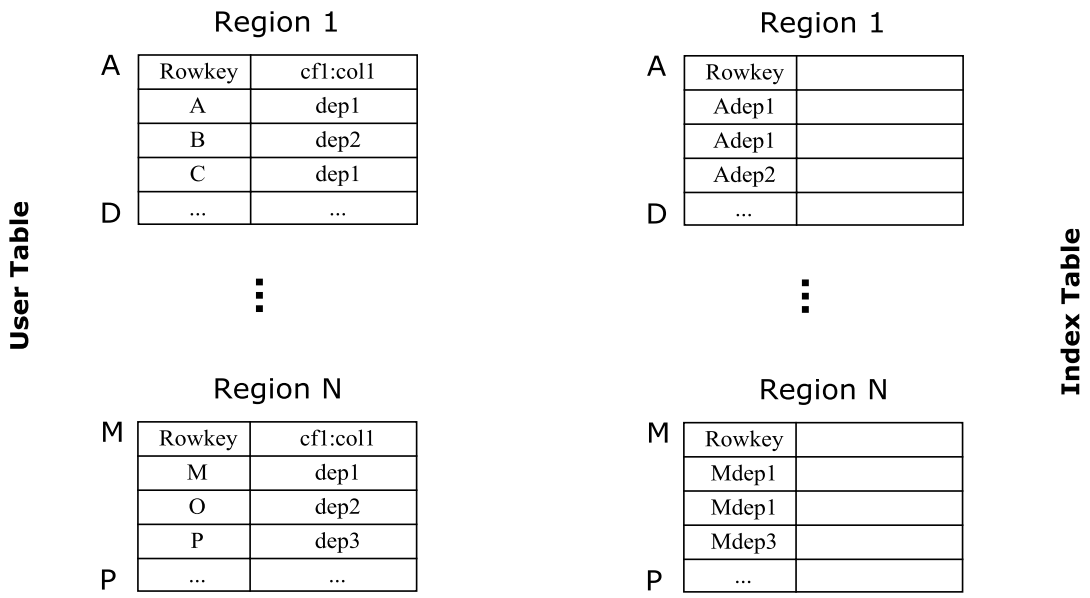
1. When a table is created with a number of pre-created regions
2. When one RS going down. Regions from that RS need to get assigned with other RSs
3. When a new RS is added to the cluster or one gets restarted
4. When a cluster is restarted
5. As part of the balancing activity. This will be running in predefined intervals in master to make sure the load is equally distributed across the cluster of RSs.

HBase allows to plugin a custom implementation for this load balancer module. As per this any user of HBase can implement his own balancer module. We can create Load balancer implementation as a wrapper over the actual load balancer. The actual load balancer will decide about the region placement for the user table regions. Our load balancer will decide the placement for the index table regions as per the corresponding user table region's location or vice versa. We can tweak the master side code to always have our wrapper being used by the master module which wraps the actual load balancer. This actual load balancer can be the default load balancer implementation from HBase or a custom implementation by the user.



User table region index table region association

We did the load balancing with the assumption that there will be a 1-1 association between the user table regions and index table regions. For this we can create same number of regions in the index table as that in the user table region. When a user table region splits let the corresponding index table region also splits into two so as to maintain the 1-1 association always. In user table the regions are split based on the rowkey for the index table data. When coming to the index table, we cannot use the same rowkey any way. We need to have the indexed column's value in the rowkey. To make this 1-1 association we can split the index table into regions with the same start and end keys as there for the user table regions. Any data written to the index table, we can prepend (prefix) the startkey of that region into the rowkey followed by the indexed column's value.



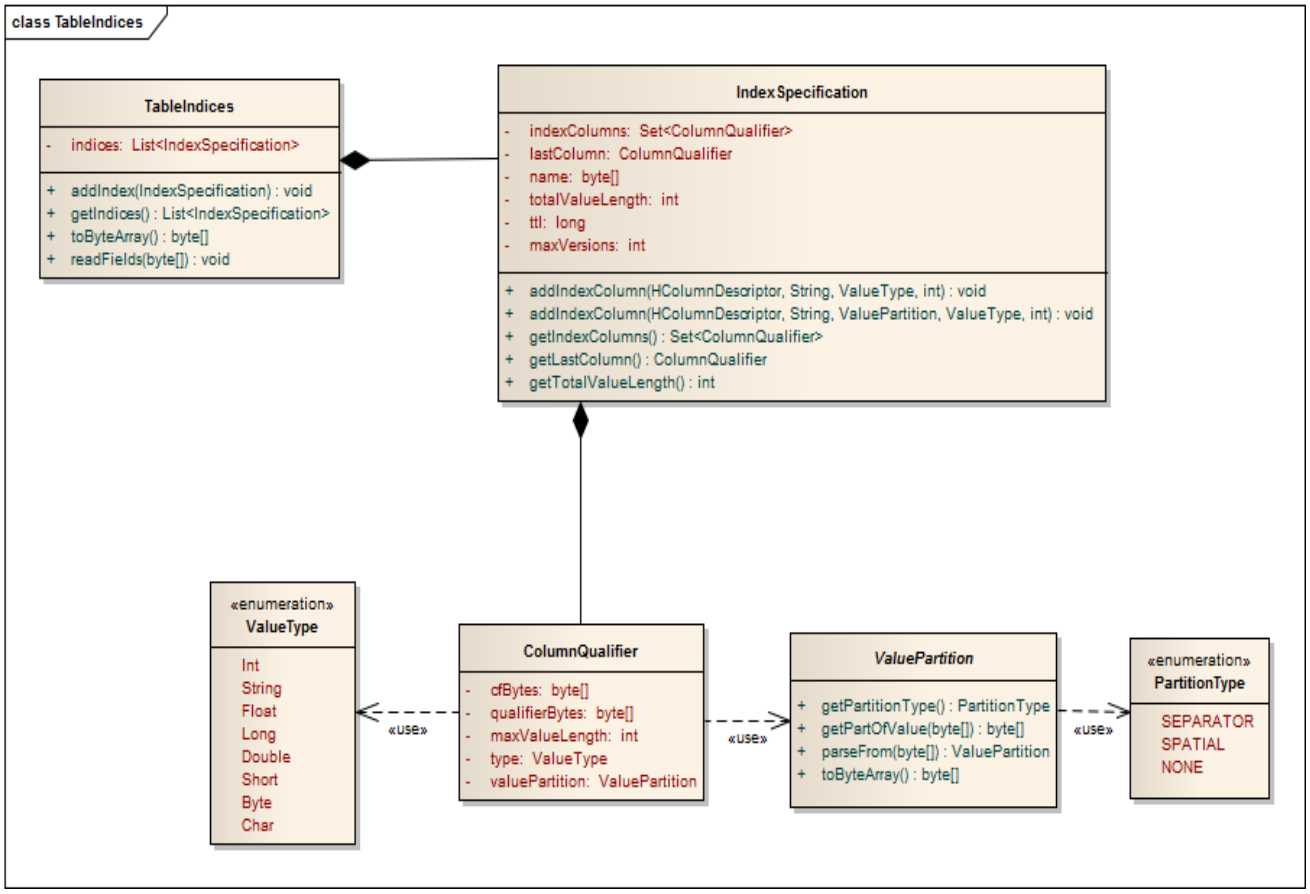
Note: There will be more things to be added in the rowkey for the index table. We will come to that later.

Here by we can see that indexing we doing per region wise. All the index details within one user table region will be there in one index table region. So our solution tries the per region indexing approach rather than the global ordered index.

Client Extension

As in arch diagram there is a need for extension in the HBase client. This is needed for the user to specify the index details while creating a table.

Specify index details while table creation



HBase client module has HTableDescriptor for the customer to specify the table details. It provides setters and getters for storing meta data as (key, value) pair of a table. TableIndices holds index details of a table. We can convert the indices into bytes and set to HTableDescriptor. There can be one or more indices on one table. An IndexSpecification should have a unique name and can be created on 1 or more columns (Here column refers to columnfamily + qualifier). For each of such column a ColumnQualifier is provided which takes the column details. This includes the cf name and qualifier name. The maxValueLength and Valuetype in ColumnQualifier, as part of the index specification will be explained below in the [Handling the Put operation] section. By default for making the index details we will consider the entire value present for the column in every row. If one want to consider only a part of that column's value it can be achieved by specifying either offset and length or separator and index. Both cannot be used together. If offset is specified, then we will consider the part of the value from position offset till offset+length. Length needs to be greater than 0. Instead of this if customer specified the separator and

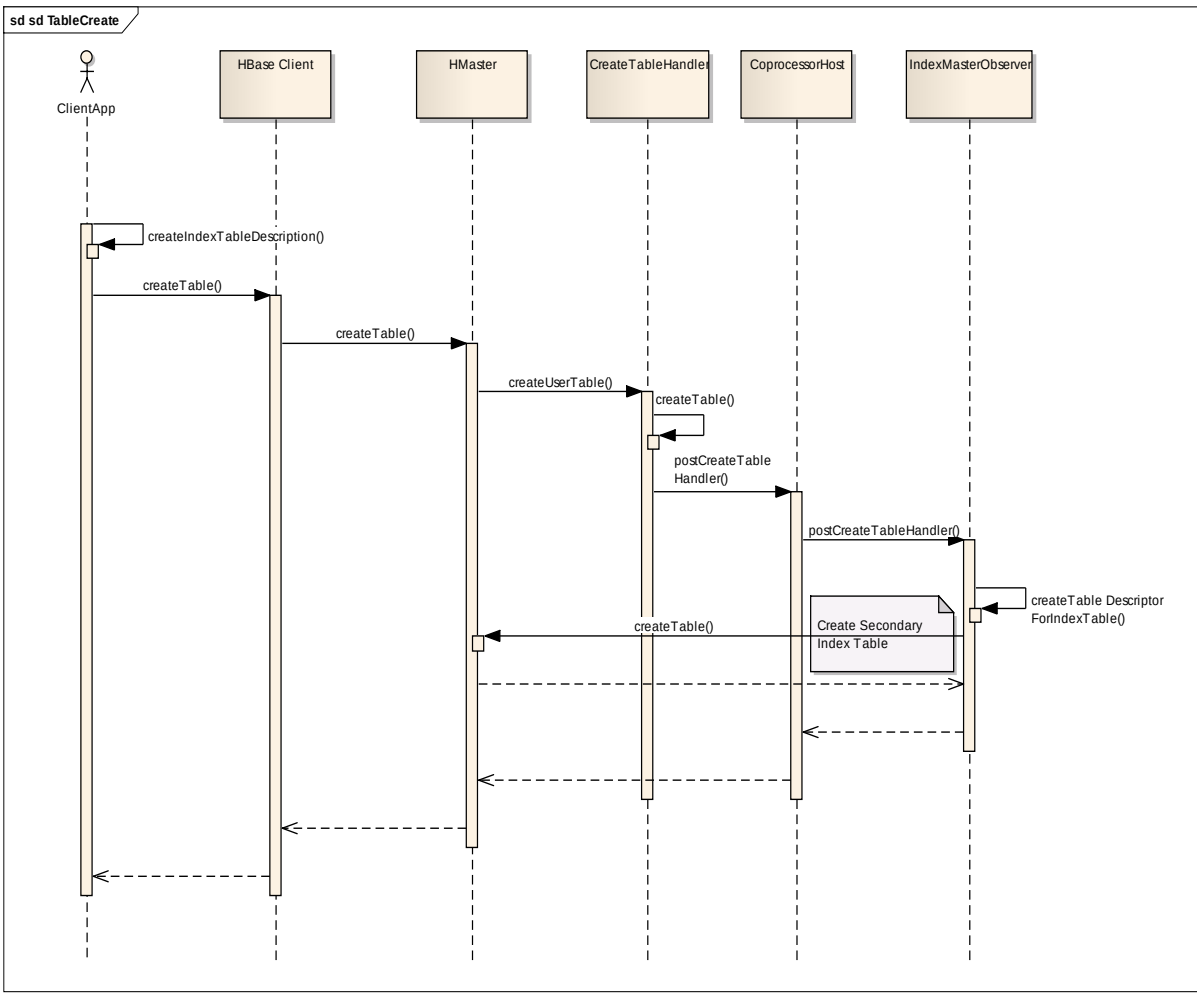
an index, we need to split the column value with the separator and consider the index position split item for the index creation on that row.

In order to dynamically add or remove an index from/to a table, let the modifyTable() API be used.

2. Sequence flows

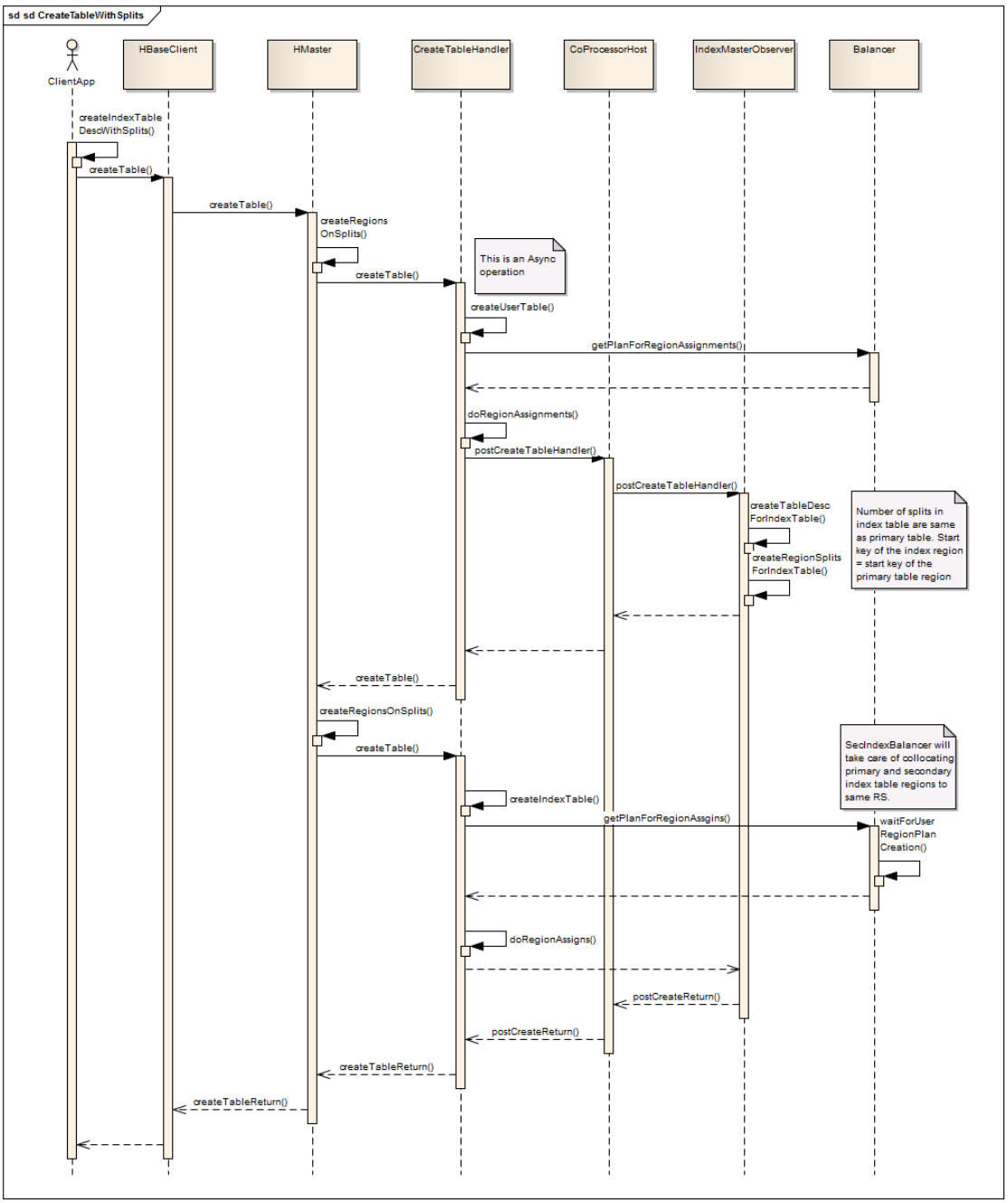
1. Creation of index table

This can be done by the Master side co processor. The co processor at Master side needs to create one secondary index table for a user table which is having one or more index specified on it. The index table can be created with a name which is the actual table name suffixed with a predefined pattern '_idx'. The index table will have only one column family.



The above flow only shows how the coprocessor will handle the table creation for the

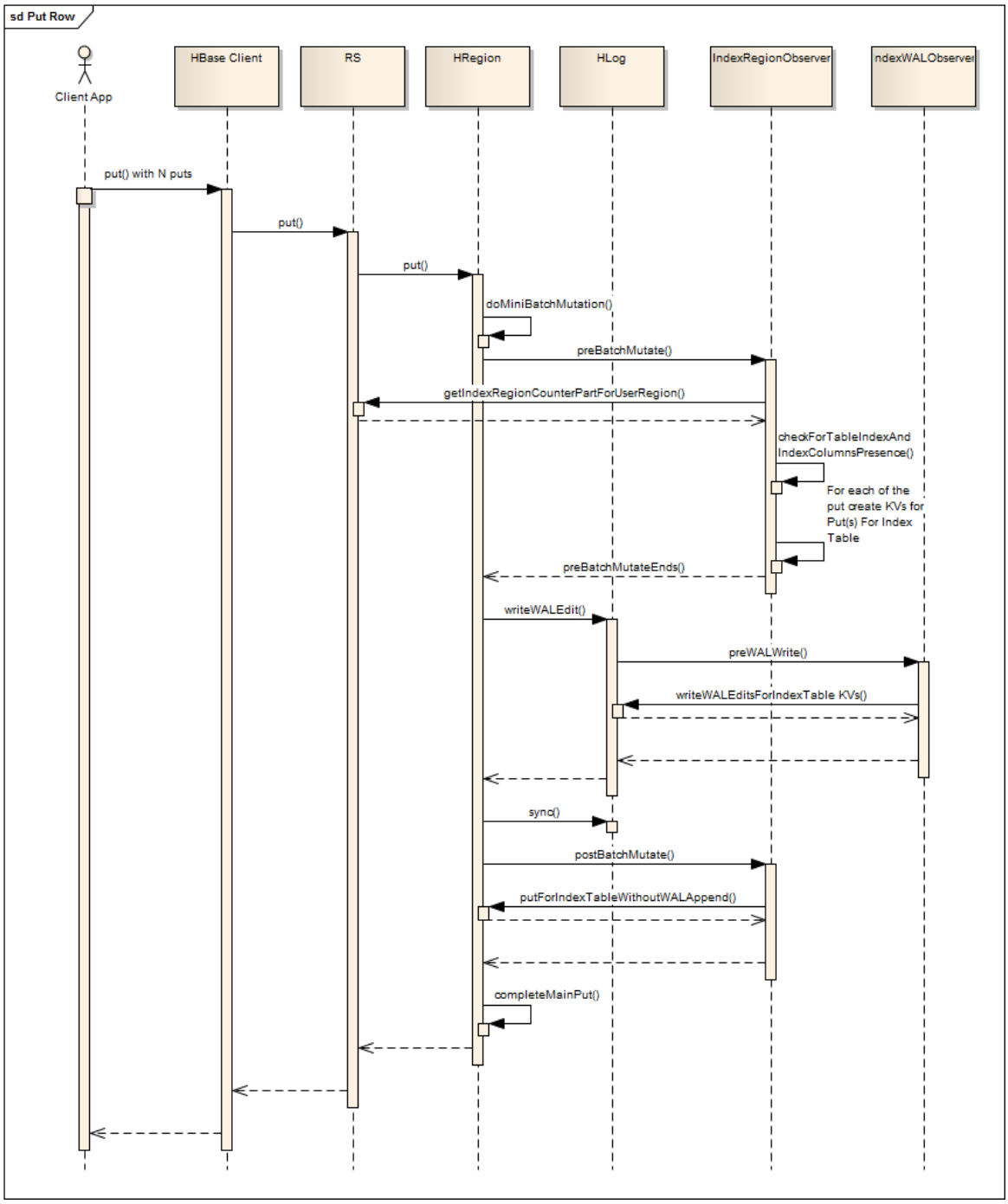
index tables. The flow diagram when the table is created with splits (pre-created regions) is shown below



2. Put Operation

Along with each put into main table, we need to put entry into the index table also if the user Put is having a value for the indexed column(s). More than one entry might be needed to be put into the index table for one put into the user table when the table has more than one index on it. We need to make use of the Region Server side co processor

(RegionObserver) to achieve this. This hook will get notified when the operations like Put, delete etc happening at the Region. As we are collocating the actual region and the index table region, the Put operation into the index table (region) need not go through the HBase client and RPC calls. We can directly access the HRegion counterpart for the actual table region and put index entry into that. IndexRegionObserver is the class which implements region level hooks.



The coprocessor implementation at put side should do this insertion operation into the index table region. When the user table is having N number of indices, N entries need to get added into the index region for this user table row. The coprocessor module needs to check and get the column values from the user table row (Put) corresponding to the index columns and create the Put(s) for the index table.

As shown in the flow diagram during the preBatchMutate() [Every put will be handled as a batch put even if putting one Put] the user table Puts will be examined and the Puts for the index table to be created. Also it will create the WALEdit corresponding to index table puts. Then the user region put will happen. As part of this 1st the data will get written into memstore and then data getting written to HLog. There is CP hooks for the HLog write step also. We will implement this so that we will write the WALEdit created corresponding to the index table also into the HLog. Later both these WALEdit writes will be synced once by the user table Put flow. Now as part of the postBatchMutate() call we will write the Puts for the index table. [while doing this put into the index region we need to specify DURABILITY to SKIP_WAL]. This will only put the value into the memstore. There will be 2 advantages for this approach

- This can avoid one extra call of sync into HDFS
- This will help in maintaining the consistency between the actual table data and the index table data. This will make the HLog write operation as part of the Put as a single unit of work. WAL data corresponding to both actual put and index data put is synced together.

Index data to be written

Now what should be written to the index table as part of the user table write. We need to create a rowkey for the Put into the index table.

Creation of the rowkey for index table

The row key needs to include the value(s) of the index column(s). Also we need to prefix it with the start key of the region (Already described). Also we are adding entries for all the indices into the same region. We must have entries for each of the index grouped together logically within the region. Later while scanning this region for one index, we should be scanning only this logical group. In HBase as we know the entries into the region

will be sorted based on the lexicographic order of the rowkey, it makes it necessary that we need to add the index identifier (index name) also into the rowkey for the entries into index table. This name will be a static value.

So the rowkey need to start with the startkey of the index region followed by the index name. After this we need to add the value of the index column into the rowkey. When the index is on more than one column we need to take values of all the columns and add them in the rowkey in the same order of columns added to the index specifier. Finally we need to append the rowkey of the entry into the user table for which this index entry we are creating.

The rowkey for the index table put can be **startkey of index region + index name + indexed column(s) value(s) + user table rowkey.**

HBase row keys are plain byte arrays, and HBase determines the row order by byte comparison. This means that to compare two byte arrays, HBase compares the corresponding bytes, from left to right. Now if we see the row key in the index table contains start key of the index region which will be constant across all the rows in that region. Next is the index name. In order for the proper order maintenance of the entries in this region, we need to make sure that the number of bytes written for the index name into all the entries is constant. The index name is given by the user. The length of this name is restricted to a max length. This index name max length should be configurable(**TODO**). The default value is 18. While creating the rowkey for the index table entry, padding is added to the index name so as to make its length equal to the max length.

The next part of the rowkey will be the index column value. And when the index is a multi column one it is the values of all these columns. The user table rowkey will come at the end. This means that the same kind of padding mechanism is needed for these values also to make sure the total number of bytes taken by this column value part is same across all the entries in the index table. For this we take the maxValueLength for an indexed column when table is created. As per this value and the actual bytes length for the column value we need to do the padding while the rowkey generation. Also where to add

this padding is another point and what needs to be the padding bytes. For this the value type field to be used. If the type is String we need to add the padding as a suffix to the value with padding bytes as 0s.

Eg : Consider the String values for a column country are coming as

Country

India

China

Australia

Here if we take 10 as the maxValueLength we need to pad the 3 values with suffixing with 0 bytes.

India00000

China00000

Australia0

Note that the final 0s are not char 0. But we need to pad with byte value of 0[All bits 0].

When the data type of the index column is integer, all the byte[] value will be of 4 bytes[] length. So here no need of any padding. But the issue will be with dealing with –ve numbers. As we know when compared in int terms -1 > -2, but when we convert both of them into byte [] and compare from byte to byte we will see -2 as greater than -1. But while creating the rowkey for the index table we cannot afford to this as we might need to perform value range lookups on this index table. In order to make integers compare correctly, we need to flip the sign bit.

When the data type is floating point it is more interesting to get the correct ordering. Java uses the float representation as defined by IEEE:

[1 sign bit][8 exponent bits][23 mantissa bits]

With this representation positive floats will compare correctly. We only have to flip the sign bit so that positive numbers will be larger than negative numbers. In case of –ve numbers simply flipping all the exponent and mantissa bits will get exactly the behavior we need.

When the data type is Character or Boolean there is no issue as both will be having fixed byte[] size (1 and 2 respectively) and there won't be issues of sign. When using indexing on a column it should be any of the above described types.

Eg : Consider an index created on 2 column each of type String. An entry is adding to the user table region [001,010) with rowkey as 007 and value for column1 as "india" and column2 as "AB". Suppose the maxValueLength for column1 was specified as 8 and that for column2 as 3. We need to add an entry into the index table region (this region will be also [001,010) for the index table). So the rowkey will be coming as *001india000AB0007*. Note that it will be the byte [] representation for the rowkeys and values will come and the padding zero is not character 0 but a byte [] with all bits as zeroes.

Creating Put for the index table

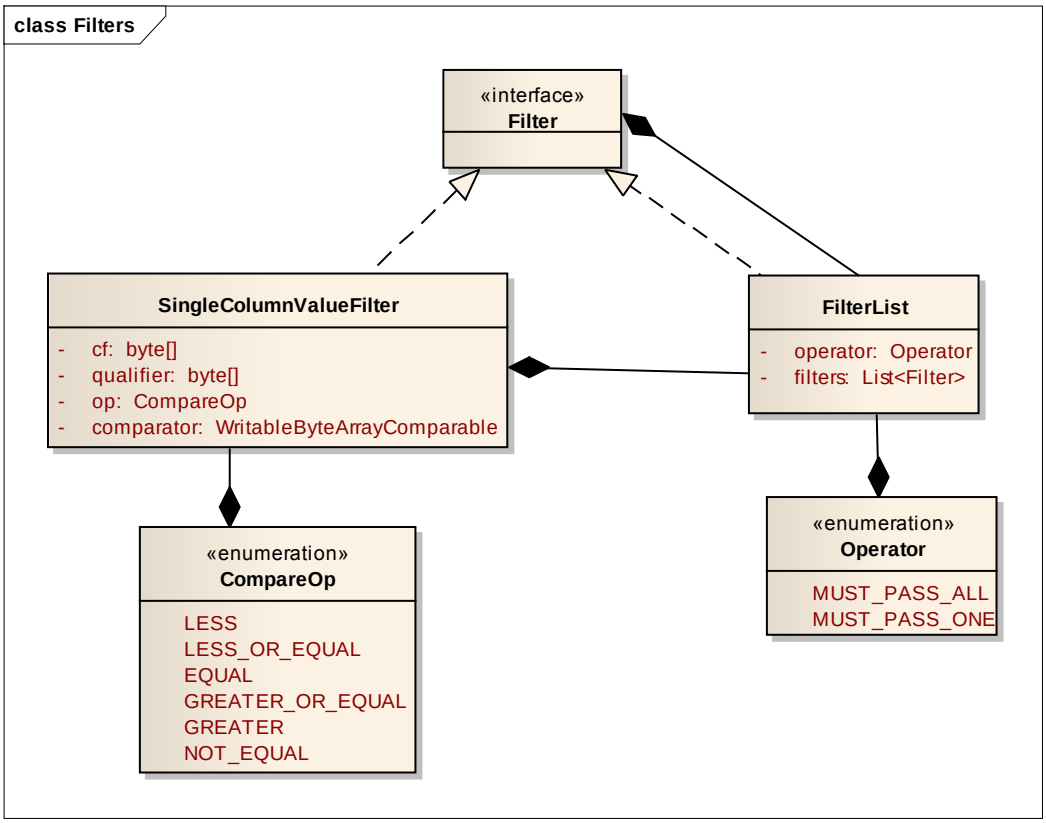
All the relevant information is capturing within the rowkey for the index table Put. We will be having only one CF for the index table. Here 4 bytes is written as data into the index Put. First 2 bytes is the length of the start key of the index region and last 2 bytes represents the starting position of the actual table's rowkey within the index rowkey. *(Why these 4 bytes needed will come later in the section of handling split of user table region)* Also the timestamp in the index table put needs to be the same as that in the user table Put.

3. Handling the Scan

HBase already provides way to scan a table based on condition on certain column's value using the filter concept. There is SingleColumnValueFilter for filtering rows based on the column value. Also customer can write custom filters of his interest. The overhead here is a full table scan is needed for these approaches which will result in lesser scan throughput.

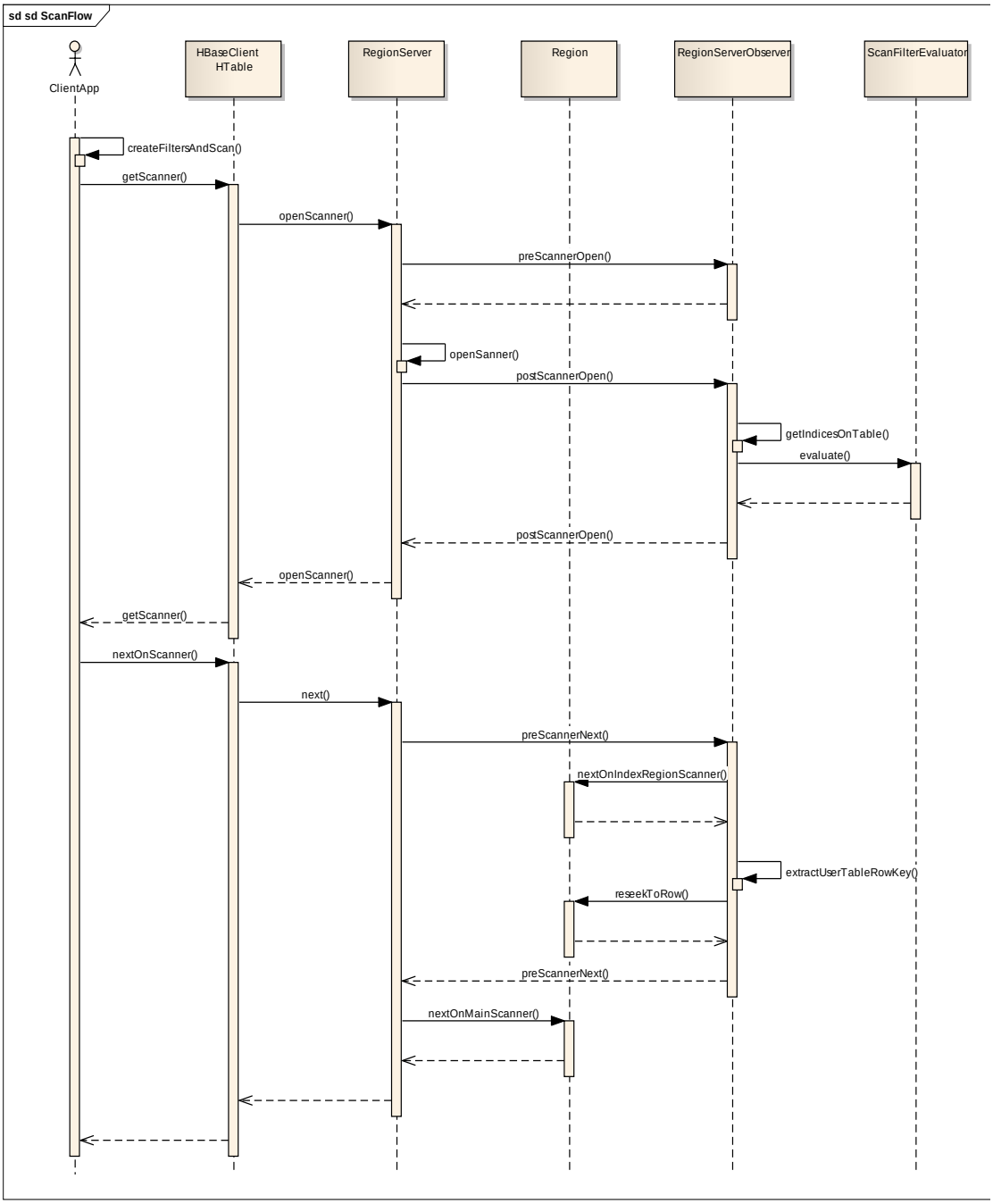
In the approach of having secondary index, an index needs to be created on the column(s) on which the filtering is required in the scan. So while the primary user table is scanned with filters on these columns, the index table can be used which can point to the rows in the user table where the column's value is the one that we are interested in.

Our solution approach here is to handle the index table lookup at the server side using co processors. Customer can specify the column value conditions using the Filter concept only.



Using FilterList more than one Filter can be added and grouped. At the coprocessor level, Filters are analyzed and decides which all index(s) to be used in the scan.

The flow diagram for scan is shown below

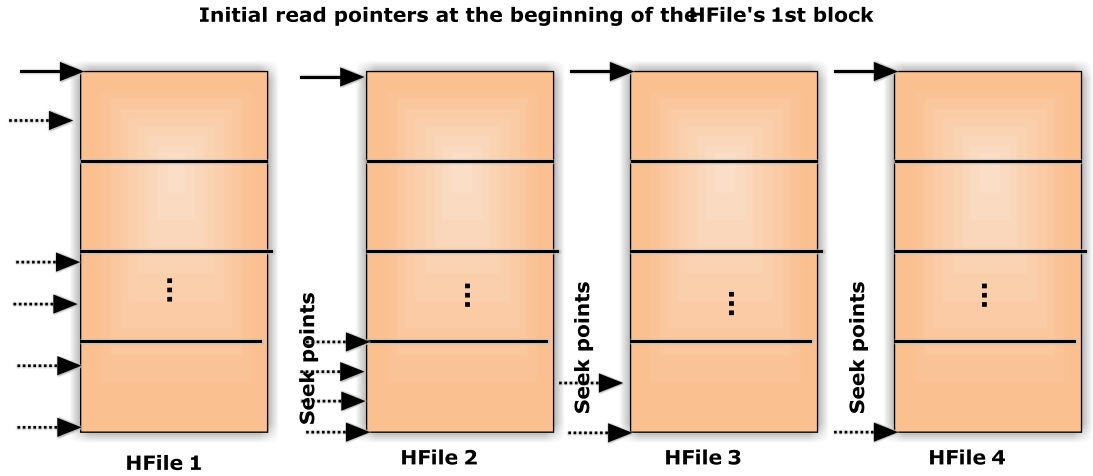


The scan will go through all the regions as per the current scan object. In one particular region, we can access the index table data corresponding to that region and based on that we can skip rows and reseek to next row as per the actual table rowkey that we get from the index table scan. So our solution will help in avoiding the full region scan. So in case when a region is not having any entry corresponding to the condition that we looking for, the entire region scan will get skipped at the 1st step itself.

How this solution will make the scan efficient

Consider scanning a region which is having 4 HFiles of nearly 1 GB size each. HBase

splits each HFile into logical blocks with 64K default size. During the scan of this region, these files will get read into RS as blocks of data. The seeking of within an HFile will allow skipping certain number of blocks and not reading some of the area of data within a retrieved block as shown below



So in some cases scanning of a full HFile or even one full region (all HFiles in that region) can be skipped as the data that we are looking for now is not present in that HFile or region at all.

4. Handling delete

In case of deletion happening from the user table, same action would need to be done in the index table too. We are making use of hooks in doMiniBatchMutation both for put and delete.

5. Creation of new index on existing table

Sometimes user might be interested in creating a new index on an existing table with data. This will be possible by modifying the user table. Steps to be done for this

a) Disable the user table HBaseAdmin# disableTable(tableName)

b) Get the table descriptor HBaseAdmin# getTableDescriptor(tableName)

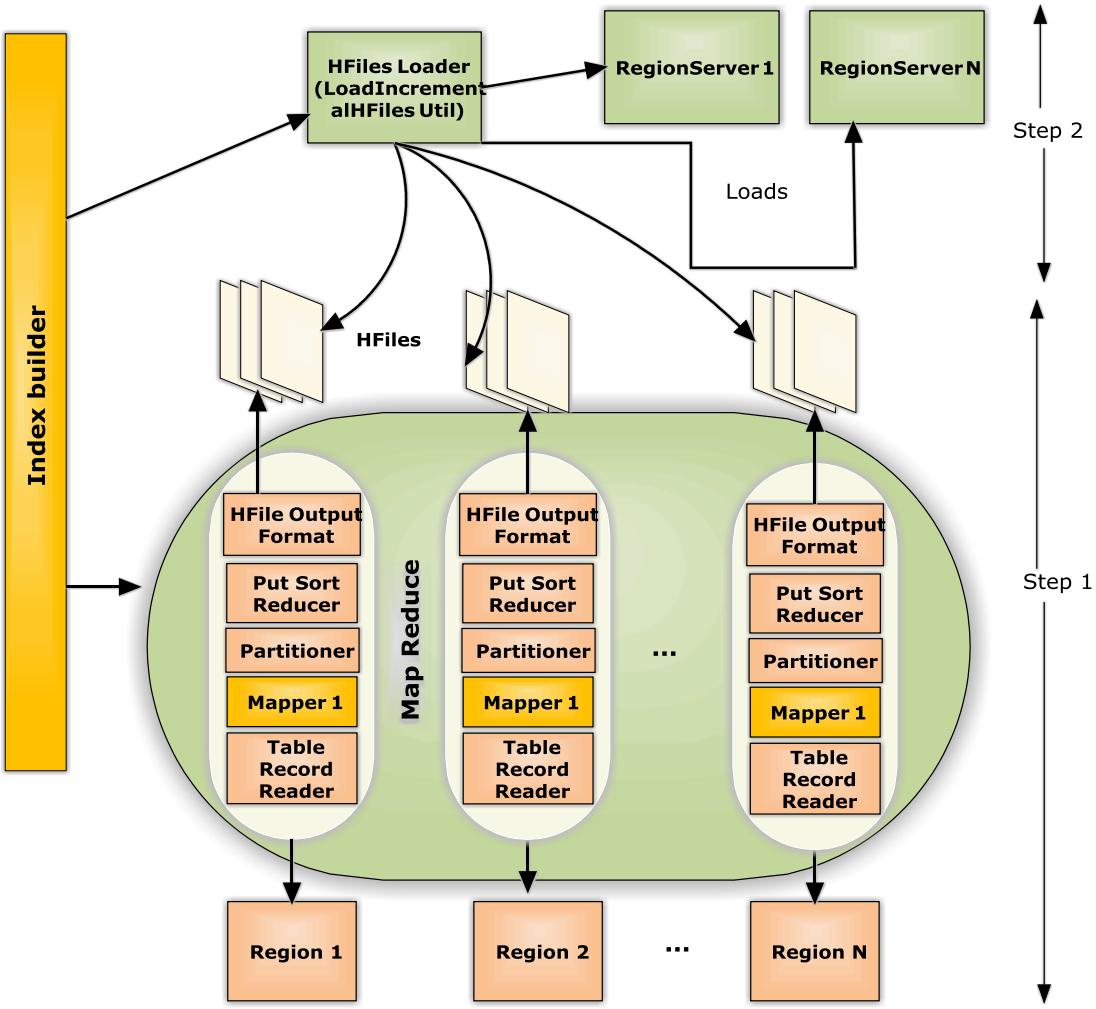
c) Modify the descriptor by adding the new index

d) Modify the table HBaseAdmin# modifyTable(tableName, htd)

e) Enable the table back

This table's column might already be having some data in it when the index is dynamically created. Using an MR job read value(s) from the column(s) of the user table

on which this index is created and make the index entries and add the same to index table. TableIndexer tool is used to run the MR job.



The 1st phase is to read the user table column data using MR job. The job will then create the Puts for the index table as per the user table column data. The output of the MR job will be a set of HFiles. As a second step all these files need to be loaded into HBase. IndexLoadIncrementalHFiles utility can be used for this. This will load the files into the index table regions in RS.

6. Bulk load data to indexed table

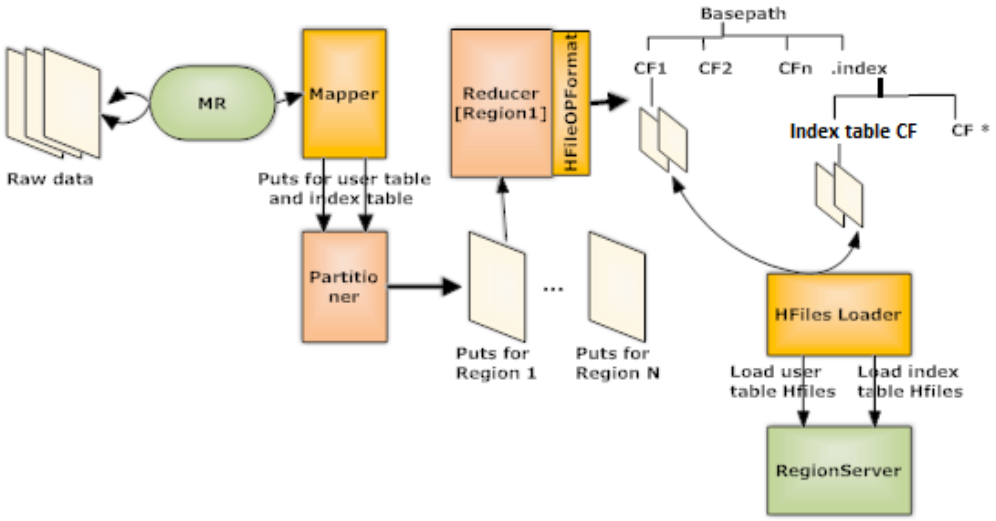
Using bulk loading feature raw data from flat files can be loaded into a table. When a table is created as an indexed table and later when data is loaded to this kind of a table using the bulk load, we need to make sure to populate the index table also along with the

user table data. IndexImportTsv tool is used to import both user table data and index table data together from TSV file. We can extend the existing map reduce support which is there for the bulk load to achieve the index details population also along with data load.

a) Create a new mapper class(IndexTsvImporterMapper) which can convert the data as plain text into the Puts into the user table. Along with this create the puts for the index table also and add to the mapper output. Now one mapper will create rows for the user table from raw data and also the corresponding index rows. The index details on the table need to be passed to the mapper through the configuration object. The Partitioner will group all the entries (Puts) to one user table region and the corresponding index table entries (Puts). The Partitioner work with region start keys and Puts rowkey. As the row key for the index table entries also begin with the start key of the actual table region, Partitioner can group all together.

b) We need to extend the HFileOutputFormat(IndexHFileOutputFormat) so as to segregate the index table rows from user table rows. Current HFileOutputFormat's RecordWriter converts a Put into KVs and write to files as per the column families of the KV. There will be one base path as the output path for the reducer. Under this per family one directory will be created and under that all KVs coming under that family will be added to files. We need to extend it such that under the base output path a special sub directory with fixed name is created. [This can be .index] From the rowkey of the Puts which the HFileOutputFormat handles it can be identified that whether the Put is for index entry or not. The index table Puts need to be separately handled and needs to be written in files under the <base op path>/.index path.

c) Provide an extended tool on top of LoadIncrementalHFiles. This should be able to check for the indexed table and find the corresponding index table. The loading need to consider the families and files coming under the <base op path>/.index path and load them into the index table. IndexLoadIncrementalHFiles is the class used to do this.

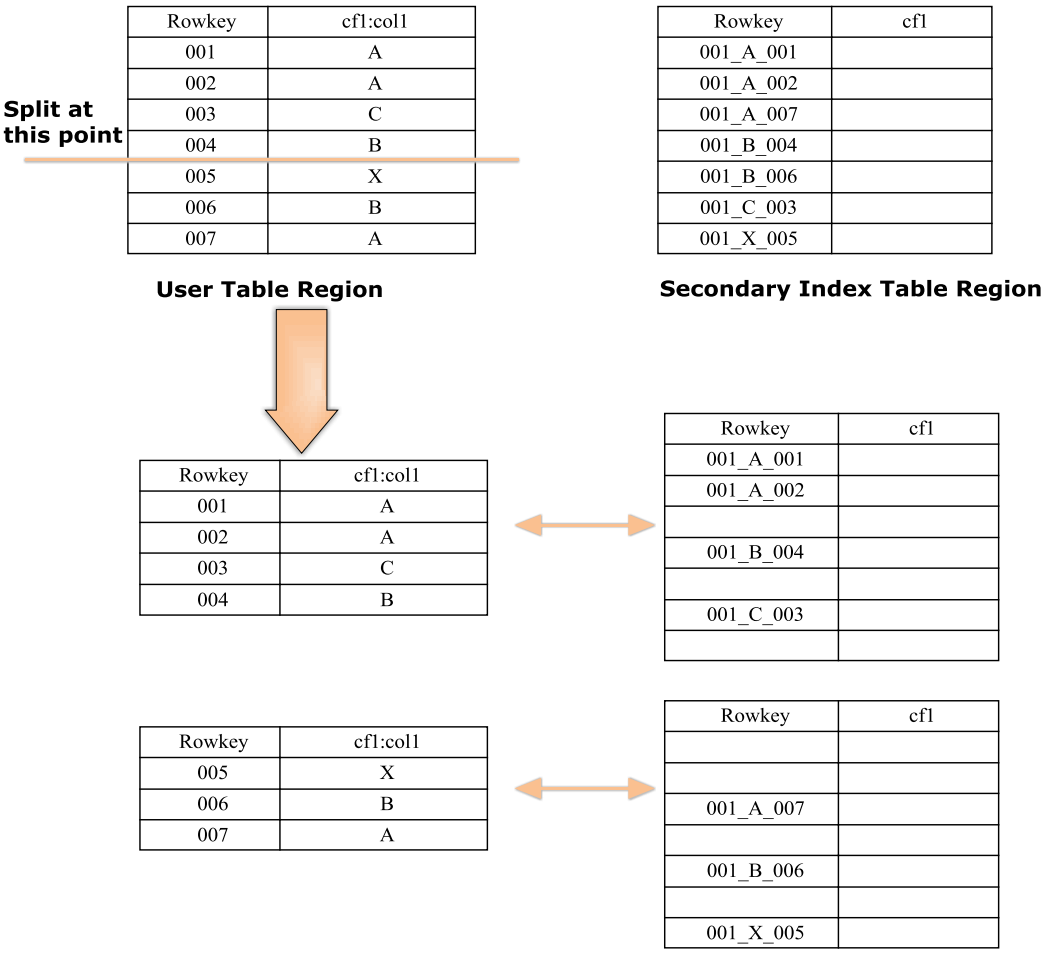


7. Drop an index from a table

For this also user can follow the same steps as mentioned in add index. There will be entries in the index table corresponding to this index. We need to delete all those entries to clean the index table, removing unwanted data. This clean up will be during the next major compaction of the index table.

8. Handling Split of user table region

When a user table region is split into 2 regions, we better split the corresponding index region also into two in such a way that index entries corresponding to the top daughter region of the user region go into the top daughter region of the index region and entries corresponding to the bottom daughter region go into bottom daughter region of the index region. As the user table and index table data is with different row keys, we won't be able to split the index region same way as user region at some middle point. This will be more like a logical point so as to associate 2 new user daughter regions with the new index regions. This separation is explained in below figure.



As per the figure we can see a user region [001-010) is split at a point 005 resulting in 2 regions [001-005) and [005-010]. But we can see that we can split the index region in the similar way. In index region, entries corresponding to rows in the parent user region are in a shuffled manner. This order very much depends on the indexed column value.

Again we need to take the split key for index region same as that for the user table region. This is because while creation of both the regions we have taken start and end keys of index region as same as that of the user region. So taking the split key for the index region same as that for the user region will make sure that the start and end key association with the user region and index region will continue to be present with the daughter regions too. This split of the index region needs to be done in postSplit () CP hook.

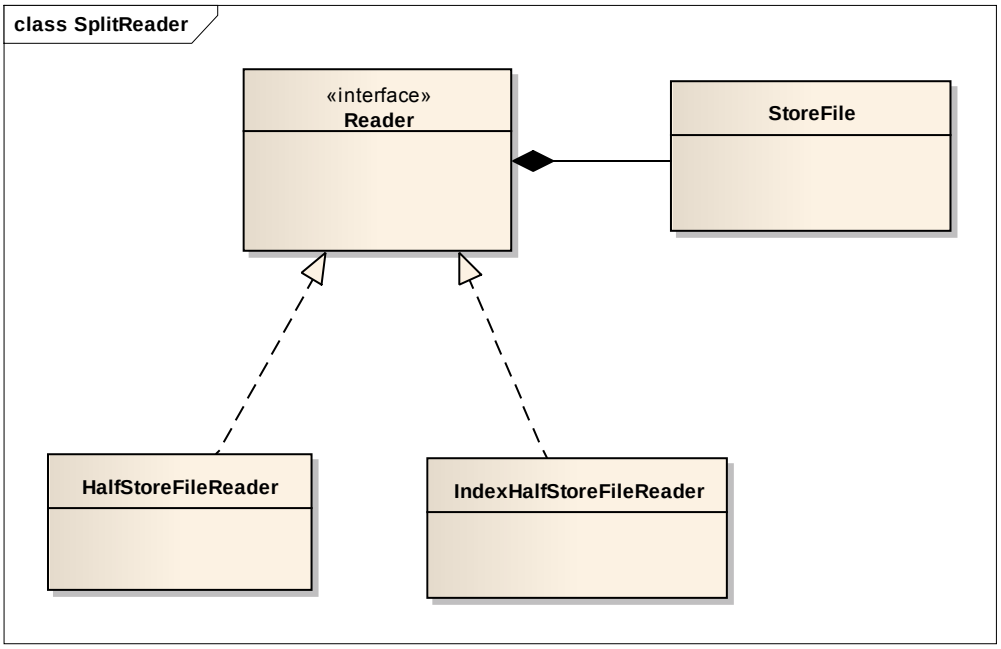
As this split is complex than the user region split how we can handle? For this first we can see what will happen when a user region split is happening. This will create 2 daughter regions and for every store file (HFile) in the parent region a reference file will get created

in both the daughter regions. Actual split of the parent region's HFile will not happen at the split operation time. This HFile will continue to be present in the HDFS as it is. Only 2 new reference files will be created one pointing to the upper half of the original HFile while the other to the lower half. When the daughter regions are initialized and the store files are being opened, a different type of reader will be created for these store files which are actually reference to other files. This is named HalfStoreFileReader and is capable of reading either the upper or lower half of the actual HFile. The HalfStoreFileReader for the bottom daughter region will start its reading from the begin of the HFile but will have check to stop reading at the split key point. This will stop the reader from reading the upper half part of the HFile. At the same time the HalfStoreFileReader for the top daughter region will start its reading from the split key position only. At first after the creation itself it will seek to the split key position. Later when the compaction of these daughter regions happen at that time only the actual split of the HFile into 2 will happen. That time also the same HalfStoreFileReaders will get used.

The split of the index region can also go through the same flow of region split, that it will create the daughter regions with references to store files. Now what is different with the split of index region is the role of HalfStoreFileReader. As the split of the index region is special and different from normal region split, we need to have a new type of HalfStoreFileReader. Both Readers for the index daughter regions need to read the full data in the physical HFiles. When the reader reads through the data it needs to check whether this data is applicable to its region. For this it need to check with the last part of the row key ie. the user table rowkey. Any way all the rowkeys in the actual HFile will be starting with the start key of parent region. As per the check for the last part of the rowkey, the reader might need to skip certain keys and go to the next one. The reader for the upper half index daughter region will consider only those entries with last part of rowkey >=splitkey as its values while the other reader those entries with last part of rowkey<splitkey. Also the upper half reader may need to change the rowkey of its entries replacing daughter regions startkey as the 1st part of the rowkey. Actually in the HFiles the index rowkeys will be starting with the startkey of the parent region. As the reader changes this while next time compaction the new rowkey will get written into the new

compacted (merged) HFiles(For upper daughter).

We need to make slight change in the HBase kernel code so as to create the appropriate HalfStoreFileReader for handling the index region splits. The kernel should make use of a factory to create this HalfStoreFileReader and the same should be pluggable. Our plugged in factory class will distinguish between the user region and index region and will create appropriate HalfStoreFileReader.



9. Tool to check for the consistency

The indexing solution is based on certain factors. These are

1. There will be one index table corresponding to every indexed table
2. There will be one region in the index table corresponding to the user table region with same start and end key. In other words there should be a 1-1 mapping between the user table and index table regions
3. The user table region and corresponding index table region to be collocated.

In the running cluster there are possibilities that some of these conditions getting broken. At that time we should provide a way for the user to detect such issues and possibly fix them. This should be a tool like HBCK tool. Different possible issues which can come are

1. Missing the index table

2. Missing some of the index regions.
3. Orphan regions in index table
4. Orphan index table itself
5. Collocation of the regions is broken
6. Data inconsistency between the user table and index table.

The tool can run on the live cluster and check these possible issues and report errors. It can fix issues like deletion of the orphan index table and/or orphan regions. Also it can fix the collocation issue by issues a new force balance () to Master. But missing the index table or region cannot be easily fixable. We can WARN the user with error. At this time we can inform user to drop the index and rebuild the same. That will make sure to recreate the index data. During this time when the index is not there, scan using index should not happen. This is something the App side needs to take care about. Also checking the 6th item ie. the data consistency check will be a heavy operation as it will need scan the entire index table and user table. We should give provision in the tool to turn this check ON or OFF. Any issue with this check also needs a rebuild of the index. Index rebuild we can implement like drop all the indices on the user table and then create the indices on the existing table with data. How to do this is already explained in this document.