Simplifying Index Update Generation for Consistent Global Indexes

Kadir Ozdemir <kadirozde@gmail.com>

The implementation of the new global index design by PHOENIX-5156 essentially introduced two coprocessors, IndexRegionObserver and GlobalIndexChecker. IndexRegionObserver is the counterpart of the existing Indexer coprocessor that the previous global indexing feature uses. It implements the indexing write path. GlobalIndexChecker implements the read verification and read repair that happens on the read path. One of the main objectives of the design behind new global indexing was to leverage as much existing indexing code as possible. This objective has been achieved greatly as the entire index table update generation code implemented by various classes (including PhoenixIndexBuilder, CachedLocalTable, NonTxIndexBuilder, IndexUpdateManager, LocalTableState, ScannerBuilder, IndexMemStore and PhoenixIndexCodec) is leveraged as it is mainly. This objective has served us well to deliver the new indexing feature quickly. The leveraged code is very complex, over engineered, and inefficient, and is not bug free. It is very hard to maintain. It is time to replace the complex set of classes with something drastically simpler and more efficient for the new design.

Objectives

1. The index update generation code during data table update and index rebuild should be easy to read, modify and maintain.
2. The index tables should support point-in-time queries and should continue to be strongly consistent for point-in-time queries. This objective requires building index updates for all versions of data table rows.
3. There should be a tool to quickly and efficiently verify an index table is consistent with its data table. IndexScrutiny is very slow and verify only one version of rows

Background

The design of consistent indexes are explained in PHOENIX-5156 Design Doc. The following are the design principles for the consistent indexes:

1. Every index row has a verification status column. An index row status is either “unverified” or “verified”. The verification status is stored in the existing empty column of index tables. The rows with the unverified status and the verified status are simply referred to as unverified index rows and verified index rows, respectively.
2. Index tables are never updated directly. They are updated as part of the transactions that update their data tables. All the (data and index table) cells that are generated by such a

transaction get the same timestamp. This timestamp is the current wall clock time of the region server handling the data table row. All the mutations in a batch get the same timestamps. The batches are serialized by the per-data-table-row lock implemented by IndexRegionObserver such that a batch is processed only after the row lock is acquired for every row to be mutated by the batch. IndexRegionObserver makes sure that if two batches need to be serialized (because they attempt to mutate the same row), they do not get the same timestamp.

3. An index row update during data table update is derived from the data table row mutation and the current state of the data table row in HBase. An Index table row is always updated fully even when the data table update (mutation) is partial. This is one of the reasons that before updating an index row, the corresponding data table row is read from HBase and the column values that are not included in the data table row update are retrieved from HBase if they exist.
4. All cells in a version of an index table row have the same timestamp. This is the side effect of generating full index table rows during data table updates. This means that the versions of an index table row do not share cells unlike their data table rows as they can be updated partially. Because of this, index table rows do not need to include cell types Delete or DeleteColumn.
5. An index row is updated in two phases, before and after data table row update. These phases are called the first (or pre-index) phase and the third (or post-index) write phase. Pre index writes generate full but unverified index rows. In the third phase, the verification status (i.e., the empty column value) is updated one more time to change the row status from unverified to verified. The same timestamp is used for all these updates including the data table row update.
6. An index row is deleted in two cases. In the first case, a data table row is deleted. In this case, an index row is deleted in two phases. These phases happen before and after the data table row is deleted. In the first phase, the verification status of the index table row corresponding to the data table row is set to “unverified” by adding an empty column (i.e., the verification status) cell to the index row and the timestamp of the cell equals to the timestamp of the data table row delete mutation. In the second phase (the post-data-row-delete phase), the index row is deleted with the same timestamp. In the second delete case, the data table row update changes one of the indexed columns. This means this update results in writing a new index row and requires deleting the existing index row for the data table row in HBase before this update. This delete also happens in two phases as in the first case. In the first phase, the index row is made unverified and in the second phase the index row is deleted. The timestamps for these phases are the same timestamp used for the data table row update.
7. When an unverified row is scanned during Phoenix queries, it is always repaired. This is called read repair. The read repair is to rebuild the unverified row from the corresponding data table row if such a row exists. If there is no data row for the unverified index row, the index row is not returned to the client and eventually deleted. The delete mutation timestamp is the same as the index row timestamp.

8. From the consistent indexing design (PHOENIX-5156) perspective, two or more pending updates from different batches on the same data row are concurrent if and only if for all of these updates the data table row state is read from HBase under the row lock and for none of them the row lock has been acquired the second time for updating the data table. In other words, all of them are in the first update phase concurrently. For concurrent updates, the first two update phases are done but the last update phase is skipped. This means the data table row will be updated by these updates but the corresponding index table rows will be left with the unverified status. Then, the read repair process will repair these unverified index rows during scans.

Index Update Generation During Data Table Updates

In Phoenix, data tables are updated in batches of mutations. A batch of mutations is generated for a set of upsert or a set delete statements.

A batch of mutations for upserts always include a put mutation and possibly a delete mutation. A delete mutation is included if an upsert statement sets a column to the null value. The delete mutation in this case includes only a DeleteColumn cell for each set null operation. If there are multiple upserts for the same row in a single Phoenix commit, these upserts are merged into one HBase put mutation and possibly one HBase delete mutation such that for a given column there is at most one put cell and/or delete cell in this mutation.

A Phoenix delete statement is for deleting a row. Thus, it maps to an HBase delete mutation such that this delete mutation includes only DeleteFamily cells to delete all versions of all the cells of the families of this row.

Based on this, it should be clear that a batch of mutation sent to the region server can include the following combinations of data table mutations for a given row:

1. One delete mutation to delete the given row
2. One put mutation to mutate an existing row or to insert a new row
3. One put mutation to mutate an existing row or to insert a new row and one delete mutation to set some column values to null

IndexRegionObserver has been responsible for implementing the data table and index table updates using the three phase write approach introduced by PHOENIX-5156. With PHOENIX-5748, IndexRegionObserver becomes also responsible for generating index table mutations. To do that it follows the following steps:

1. Read the existing for every (data) row keys in a batch of mutations. A map from a data row key to a pair of mutations is maintained. The first mutation in the pair represents the

existing data row state for the row key and the second mutation represents the next data row state which is obtained after applying new mutations (in the batch) on the current row state. Initially both mutations are set to the current row state which is obtained by scanning the data table in HBase (using a regular (i.e., not raw) scan).

2. For each data row key, apply the pending mutations from the batch on the current row state to form the next row state. If there is a pending put mutation for the row key then it is applied on the next row state in the map for this row key. As stated in step 1, the next row state is initially equal to the current row state. The “apply” here is to replace the cells with their new versions from the batch and otherwise add them. The next step is to apply the delete mutation on the next row state if it exists. Applying the delete mutation means removing cells. If all cells are removed from the next row state, the row state will be null, i.e., the map will have the null value for the next row state for this row key.
3. For each row key, generate the index put mutation from the next row state maintained in the map for this row key and generate the index row key for the current row state for this row, if the next row state is not null. If the index row key of this put mutation is different from the index row key derived for the current row state (this index row key is the row key of the current index row in HBase), then generate the delete mutation to delete the index row with the index row key derived from the current row state.
4. For each row key for which the current row state is not null but the next row state is null, generate the index row key from the current row state, then generate the delete mutation to delete the index row with the index row key.

Index Update Generation During Index Rebuild

Phoenix supports rebuilding index tables online. The state of the index table is either BUILDING or ACTIVE during rebuild for global consistent indexes. The BUILDING state is used when an index has been created but has not been fully built yet. In this state, the writes on the data tables also update index tables but index tables are not used for reads (Phoenix queries). When an index is built, it becomes ACTIVE. In this state, an index table continues being updated for writes and is also used for reads. An ACTIVE index can be rebuilt online also.

Rebuilding an index means reading a point-in-time image of a data table and reconstructing index table rows from the data table rows. PHOENIX-5748 rebuilds not only the latest version of rows but also all versions at the point-in-time image of the data table. This allows the SCN queries to be correctly executed on index tables.

With PHOENIX-5748, IndexRebuildRegionScanner is responsible for generating index table mutations. To do that it follows the following steps:

1. The data table rows are scanned with a raw scan. This raw scan is configured to read all versions of rows.

2. For each scanned row, the cells that are scanned are grouped into two sets: put and delete. The put set is the set of put cells and the delete set is the set of delete cells.
3. The put and delete sets for a given row are further grouped based on their timestamps into put and delete mutations such that all the cells in a mutation have the same timestamp.
4. The put and delete mutations are then sorted within a single list. Mutations in this list are sorted in ascending order of their timestamp. The put mutations come before delete mutations if their timestamps are the same.
5. The sorted list of mutations for a given data table row are then processed orderly starting from the first element in the list.

The process of generating index mutation is very similar to the one explained previously for the pending data table updates with one main difference. That is that for the pending data table updates, a map is maintained from a data table row key to a pair of put mutations, one for the current row state and the other next row state. For rebuild, such a map is not used, instead the current and next row state is constructed on the fly while processing the data row versions. Initially the current and next row state for a data row key are null. The very first mutation on the sorted list of data row mutations is typically a put mutation. The very first put mutation forms the first next row state for the very first put mutation. Then, for the next mutation the next row state becomes the current row state. The next row state is formed by applying this mutation on the current row state.

There can be a delete and put mutation with the same timestamp. Since the put mutation comes first on the list, the next element on the list is checked to see if it is the delete mutation with the same timestamp when a put mutation is processed. If so, the delete and put are processed together in one iteration. First, the delete mutation is applied on the put mutation and current row state. This concludes the processing of the delete mutation. And then the modified put mutation is processed.

Processing a put mutation is done as follows. An index put mutation is generated from the next row state and the index row key is derived for the current row state, if the next row state is not null. If the index row key of this put mutation is different from the index row key derived for the current row state, then the delete mutation to delete the index row with the index row key derived from the current row state is generated.

Processing a delete mutation means applying the delete on the current row state to obtain the next row state. If the next row state is null, then the index row key from the current row state is generated and using this index row key, a delete row mutation is generated to delete the index row for this index row key. If the next row is not null, then an index put mutation is generated from the next row state and the index row key is derived for the current row state. If the index row key of this put mutation is different from the index row key derived for the current row state, then the delete mutation to delete the index row with the index row key derived from the current row state is also generated.

IndexTool Verification

There are two types of verification: without repair and with repair. Without-repair verification is done before or after index rebuild. It is done before index rebuild to identify the rows to be rebuilt. It is done after index rebuild to verify the rows that have been rebuilt. With-repair verification can be done anytime using the “-v ONLY” option to check the consistency of the index table. With-repair verification means index rows will be repaired in memory only before they are verified against data table rows.

Unverified Rows

For each mutable data table mutation during regular data table updates, two operations are done on the data table. One is to read the existing row state, and the second is to update the data table for this row. The processing of concurrent data mutations are serialized once for reading the existing row states, and then serialized again for updating the data table. In other words, they go through locking twice, i.e., [lock, read, unlock] and [lock, write, unlock]. Because of this two phase locking, for a pair of concurrent mutations (for the same row), the same row state can be read from the data table. This means the same existing index row can be made unverified twice with different timestamps, one for each concurrent mutation. These unverified mutations can be repaired from the data table later during HBase scans using the index read repair process. This is one of the reasons for having extra unverified rows in the index table. The other reason is the data table write failures. When a data table write fails, it leaves an unverified index row behind. These rows are never returned to clients, instead they are repaired, which means either they are rebuilt from their data table rows or they are deleted if their data table rows do not exist.

Delete Family Version Markers

The family version delete markers are generated by the read repair to remove extra unverified rows. They only show up in the actual mutation list since they are not generated for regular table updates or index rebuilds. For the verification purpose, these delete markers can be treated as extra unverified rows and can be safely skipped.

Delete Family Markers

Delete family markers are generated during read repair, regular table updates and index rebuilds to delete index table rows. The read repair generates them to delete extra unverified rows. During regular table updates or index rebuilds, the delete family markers are used to delete index rows due to data table row deletes or data table row overwrites.

Verification Algorithm

IndexTool verification generates an expected list of index mutations from the data table rows and uses this list to check if index table rows are consistent with the data table.

The expect list is generated using the index rebuild algorithm described previously. This mean for a given row, the list can include a number of put and delete mutations such that the followings hold:

1. Every mutation will include a set of cells with the same timestamp
2. Every mutation has a different timestamp
3. A delete mutation will include only delete family cells and it is for deleting the entire row and its versions
4. Every put mutation is verified

For both verification types, after the expected list of index mutations is constructed for a given data table, another list called the actual list of index mutations is constructed by reading the index table row using HBase raw scan and all versions of the cells of the row are retrieved. As in the construction for the expected list, the cells are grouped into a put and a delete set. The put and delete sets for a given row are further grouped based on their timestamps into put and delete mutations such that all the cells in a mutation have the same timestamp. The put and delete mutations are then sorted within a single list. Mutations in this list are sorted in descending order of their timestamp. This list is the actual list.

For the without-repair verification, unverified mutations and family version delete markers are removed from the actual list and then the list is compared with the expected list.

In case of the with-repair verification, the actual list is first repaired, then unverified mutations and family version delete markers are removed from the actual list and finally the list is compared with the expected list.

The actual list is repaired as follows: Every unverified mutation is repaired using the method read repair uses. However, instead of going through actual repair implementation, the expected mutations are used for repair.