HBase MOB

1 Terminology

LOB: Large object. It usually refers to BLOB(binary large object) and CLOB(character large object). It can be a PDF document, Word document, image, multimedia object, etc. Unlike typical records, LOB can typically be several hundred KB to tens or hundreds of MB in size.

MOB: Medium object. It's not big as the LOB. Usually it's up to 10MB.

Metadata: The rest data in a record besides the MOB. Usually they're the metadata of the MOB ones, for example the title, description, etc.

2 Use case

The traditional database has the ability to save the MOB, for example, Oracle Database. This had been widely used in the world currently. As a competitor and a database option, HBase needs it as well.

It is quite useful to save the binary data like images, documents into the HBase. For example, the online applications save the uploaded pictures and documents, the ITS (Intelligent Transportation System) saves massive pictures taken by the lane cameras to track the vehicles in the purposes of monitoring the traffic and security issues.

3 Characteristics of MOB Use

MOB data is typically stored together with its metadata. For example, in ITS, the metadata (car speed, color, direction, plate number, etc.) of the picture taken by lane camera is extracted before storing MOB. And these metadata are stored together with MOB (pictures).

Write characteristics:

1. Write performance is quite critical. We need to store MOBs and their metadata efficiently.
2. The data size is quite big.
3. The MOB data are seldom updated.

Read characteristics:

1. The MOB data are accessed much less than their corresponding metadata. Typically, considering the size of MOB data, metadata are accessed for analytics; MOB data are used as archives, and accessed only when users explicitly request such data.
2. MOB data are typically accessed in a random way. User typically doesn't scan all MOB data at one batch.

4 The design goals

1. High performance.
    a) Write: Stable low latency and high throughput. Minimize the impact

to the HBase by the large objects when split, compaction, etc.

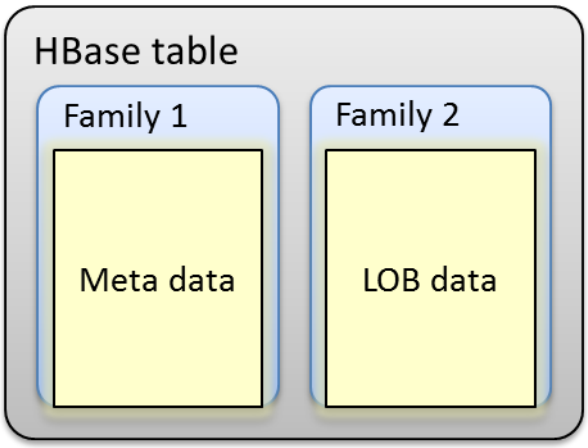
b) Read: Low read latency and good concurrent read performance. Fast scan on the metadata, and fast random read on the MOB data.

2. Consistency. HBase offers strong consistency in read/write, we should guarantee it in MOB data as well.
3. Transparency in reading and writing against the MOB data.

4.1 Existing Solutions

Let's list some existing solutions for the MOB and compare them.

4.1.1 Directly Store the MOB in the HBase Tables



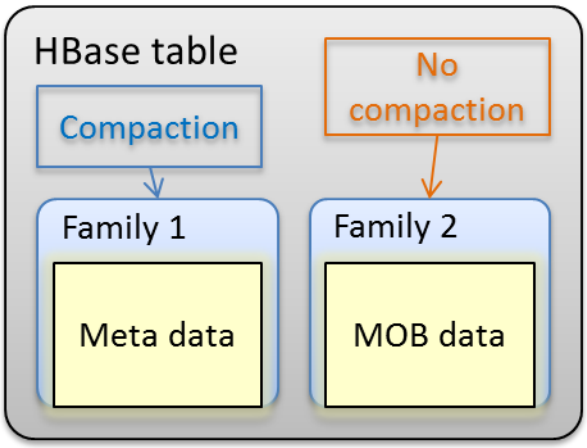
The column value of HBase is byte array, to save the binary data like images, documents could work in HBase as well. In this solution, the MOB data are saved in the HBase tables directly.

1. Save the MOB data into a separate column family
2. Save the metadata into other column families.

Typically the metadata are very small, whereas the MOB data are usually up

to tens of MB. The MOB store becomes very large easily which leads to the region split, and this brings very heavy IO. The compaction for this store encounters the same problem which leads to a slow compaction, this would probably delays the flushing, and consequently blocks the updates and unnecessary retries because of socket timeout which leads the HBase to a worse situation.

Think it differently, how about to skip compaction against the MOB store?



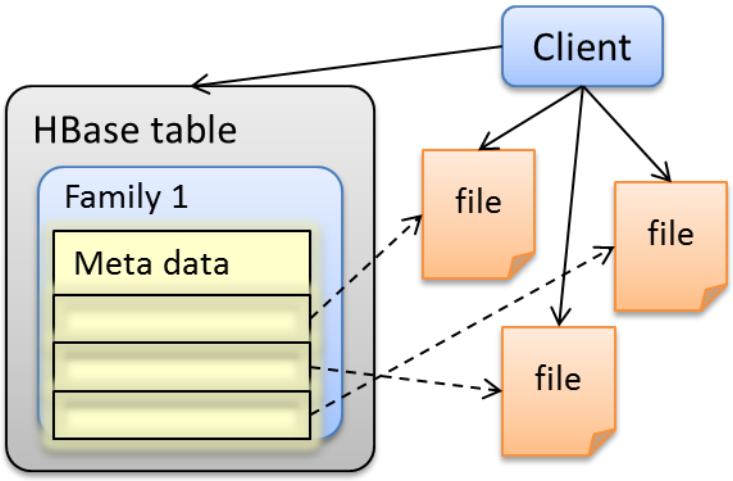
There will be lots of MOB files which lead to the slow scan and random read. And again the split of the region bring heavy IO of MOB data to slow down the HBase performance.

4.1.2 Save MOB in HDFS and Metadata in the HBase Tables

Considering the heavy IO by the MOB data in split and compaction, another solution is to move the MOB data out of the HBase, and save them to the HDFS directly.

In this solution, the metadata are saved in the HBase table, and each MOB

data are saved in a single file in HDFS. The reference which has the path of this MOB file is saved in the HBase table as well.



Pros:

1. The MOB files are out of the control from the HBase. The MOB files don't participate in the split and compaction in HBase.
2. The reference directly points to the MOB data, the random read is very fast. There's no need to do the split and compaction for these MOB files.

Cons:

1. If we save one MOB data in a single file, there will be too many small files. And storing too many small files in HDFS is insufficient. These small files occupy a large portion of the namespace whereas only a few disk spaces are utilized.

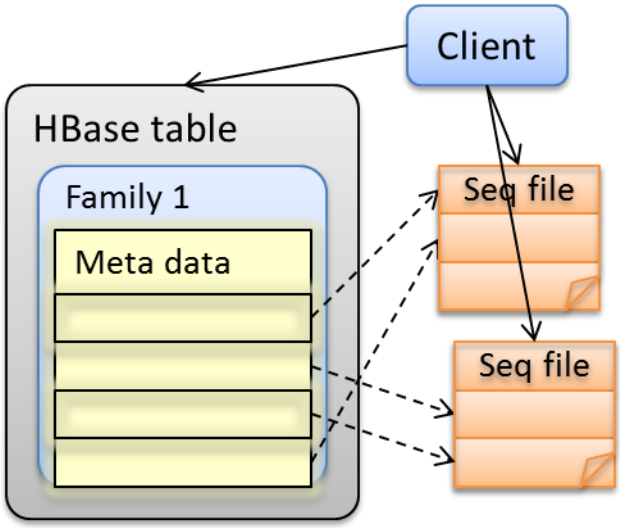
As an improvement, we group the MOB files into several SequenceFiles.

Pros:

1. We gain almost all the advantages from the above solution.
2. Avoid too many small files.

Cons:

1. No consistency guaranteed. It's hard to keep the consistency between the metadata and MOB data due to the failure of flushing and race condition.
2. It's difficult to manage the unused and out-of-date MOB data.



5 Design Overview

For the existing solutions, several problems are encountered.

1. Heavy IO in split and compaction against the MOB data which leads the HBase to the bad performance.
2. Too many small files which leads to slow read and inefficient disk utilization in HDFS.
3. Consistency issue and pool manageability if saving the MOB data out of

HBase.

a) The flush policy in MemStore guarantees the consistency when flushing. If saving the MOB data out of HBase MemStore, It's hard to keep the consistency between the metadata and MOB data due to the failure of flushing.
b) If saving the MOB directly into SequenceFile, implementing MOB compaction is difficult and will add load to HBase when updating pointers to MOB data in the HBase table.

As mentioned in the chapter *Read Characteristics*, the metadata store the information like title, description, etc. The size is much less than the MOB. Typically the metadata are accessed for scan and analysis, MOB data are used for archive, and accessed only when users explicitly request. So to save the MOB data into the region is expensive.

Considering the HBase benefits and the MOB read characteristics, we need to use the HBase as the entry to read/write MOB data, and need the MOB data to be saved into MemStore and consequently participate in the region flush, are flushed out of the region.

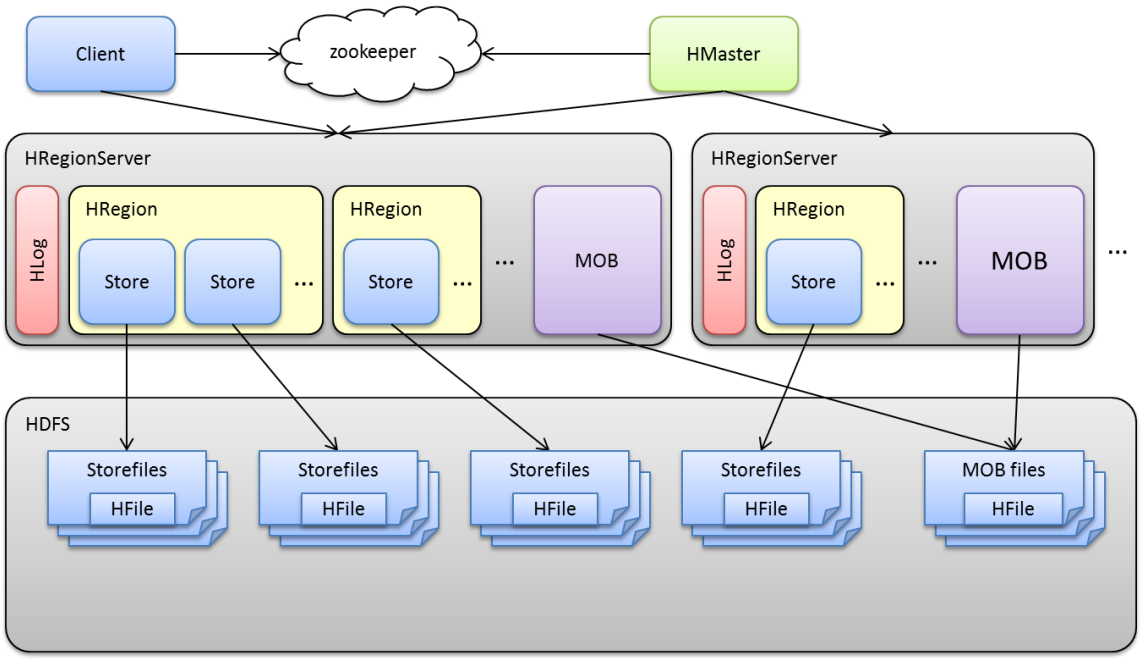
1. The metadata are directly saved in the HBase table, the MOB data are saved in the MOB files out of region in the format of HFile.
2. There's a reference column in the HBase table which links to the MOB file. A MOB file contains many MOB data.

3. The MOB data don’t participate in the split and compaction in the HBase.

In HBase, if there’re too many small files or there’s a big region, the scan and random read get slow. So we need to merge those small files into bigger ones, and split a big region into smaller ones which speeds up read and improve the cluster load as well.

Usually the MOB data are accessed randomly and the links/references are saved in the HBase tables, the chain of the random read against the MOB data is “Find the metadata, and the path of the HFile in the metadata”->”Find the HFile by the path”->”Seek the KeyValue with the key rowkey,columnFamily:column,ts in this HFile, and retrieve it”, the random read against **a single MOB file** is fast after the metadata is retrieved no matter how many HFiles collocate with it. Too many MOB files won’t impact the random read against MOB data, to split and compact the MOB files are unnecessary.

Here following is the design for the HBase MOB.



First of all, this solution is totally built on the top of Apache HBase. And the design is based on the code of Apache HBase 0.98+.

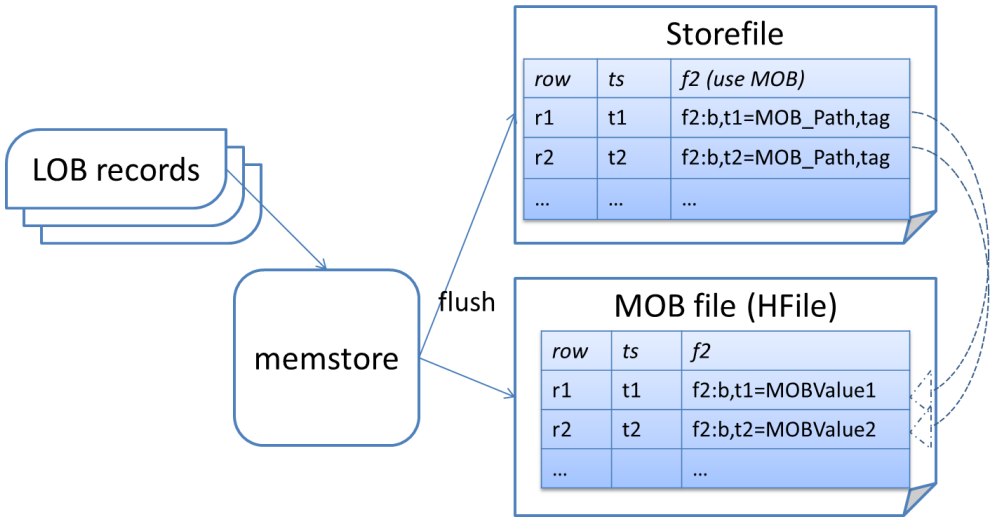
In this solution, the metadata are directly saved in the HBase table; the MOB data are saved in the MOB files out of region in the format of HFile.

The metadata and MOB data are stored in separate column families. The description of the column family contains the information of the MOB whereby we could have specific handling on the MOB data in flushing and reading.

1. In writing, the MOB data are written into the KeyValue of the MOB column, and saved to the MemStore together with the metadata. When the MemStore is full, the MOB data are flushed to the MOB files in the format of the HFiles, and the metadata are flushed to the StoreFiles, the values of the MOB KeyValues are replaced by the paths of the MOB files and flushed to the StoreFiles. These MOB files are managed by the MOB

store manager and don't participate in the region split and store compaction in HBase.

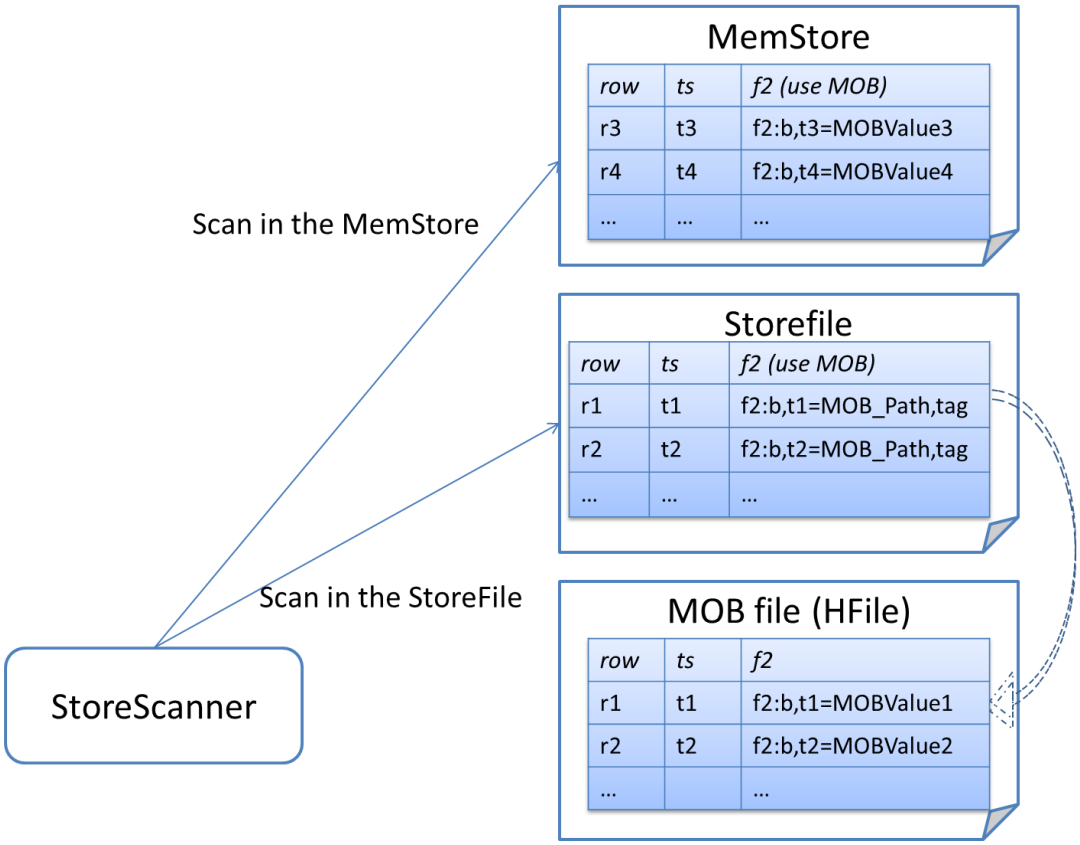
a) The MOB KeyValue in the StoreFile have a tag to mark it as a reference which links to a MOB file. This tag is very useful in both write and read.



2. In reading, the scanner scans both MemStore and StoreFiles. In the MemStore the value of the MOB column is the MOB data, whereas in the StoreFile the value of the MOB column is the path of the MOB file.

   a) Scanning the MemStore, the KeyValue of the MOB column which doesn't have the reference tag are actually the MOB data, directly return it.

   b) In the StoreFiles, the value of the MOB column is the path of a MOB file and its KeyValue has the reference tag, whereby to find the MOB file. In each MOB file, there're many MOB data, we need seek the KeyValue with the key *rowKey,CF:MOB,ts* in the MOB file to get the MOB data. After that the MOB data are returned instead.



The file path is */rootPath/tableName/.mob/columnFamilyName/${filename}*. All the MOB files which belongs to the same table and column family are saved together. Each flush of the MemStore generates a single MOB file under this directory (i.e. */rootPath/tableName/.mob/columnFamilyName*).

The file name composes of the date, the checksum of the start key, capacity and UUID, looks like {date}{checkSumOfStartKey}{capacity}{UUID}. The date is the time when the MemStore is flushed. The start key is the start key of the region where the MOB data are flushed. The capacity is the count of the KeyValues in the MOB file. The date, checksum and capacity in the file name are useful in the housekeeping.

We have a MOB files cleaner and a sweep tool to do the housekeeping. Both of them are triggered by users.

The MOB file cleaner cleans the expired MOB files. If MOB file is expired (lives longs than the TTL), it'll be deleted by the MOB file cleaner.

The sweep tool uses a MapReduce job to clean the unused MOB data.

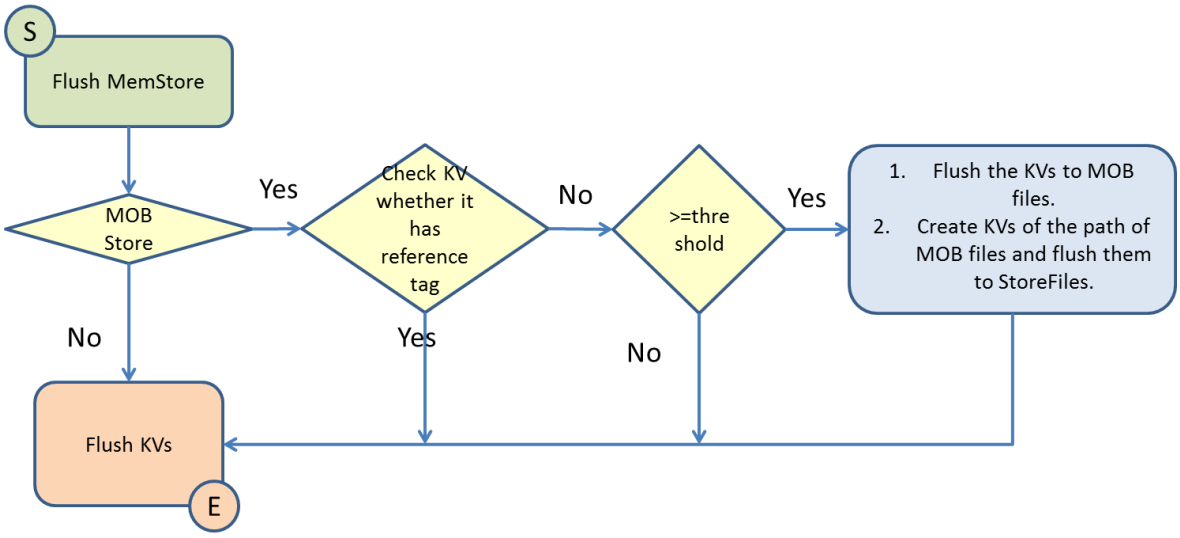
- In some situations, the MemStore is flushed before it's full which leads to small MOB files. This tool merges them into big ones to improve the HDFS utilization. It's decided by users when to use this tool, one day or even longer.

This solution achieves the design goals and even gains additional benefits.

1. High performance.
    a) Minimize the IO impact when split and compaction in HBase, and guarantees the stable low latency and high throughput.
    b) Fast scan on metadata and fast random read on the MOB data.
2. Simply use the HBase APIs
    a) Retain the HBase functions, for example the WAL and retry.
    b) Guarantee the consistency by working with the region flush policy.
    c) Multiple MOB data are flushed into one HFIle which avoids too many small files.
    d) Customize the store flusher and scanner, keep the transparency when users read and write the MOB data.

5.1 Operations

5.1.1 Write



In order to implement the transparent write, we need to customize a ***StoreFlusher*** where we could flush the KVs into different places.

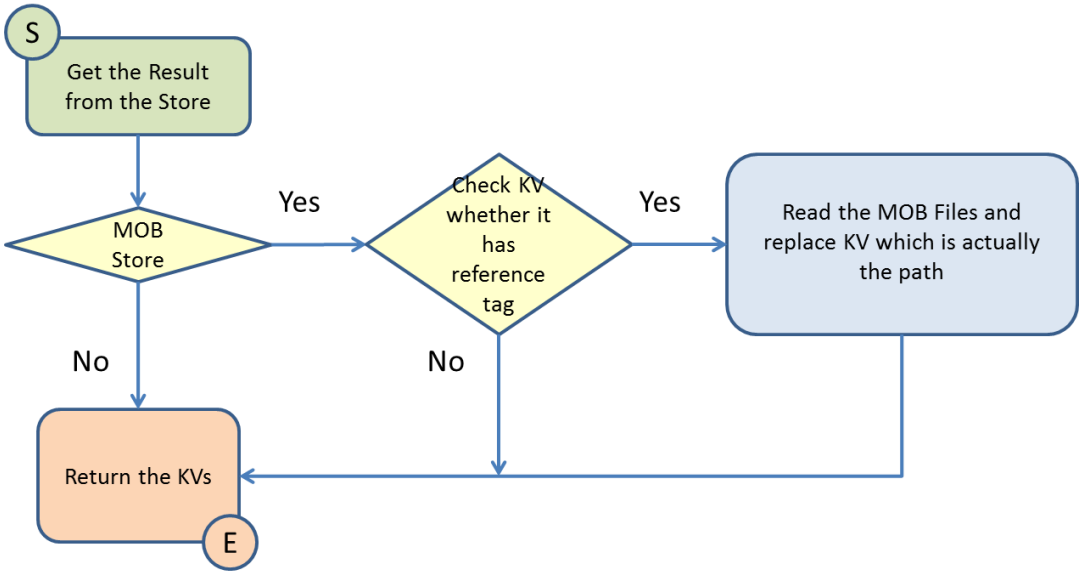
1. If the current column family is NOT a MOB one, follow the steps in the default flusher to flush the KVs.
2. If the column family is, check each KVs before flushing.
    a) If the KV doesn't have a reference tag (which indicate the value of the KV is a path of a mob file), follow the steps in the default flusher.
    b) If the KV is a PUT type, have a reference tag and the value size is not less than a user defined threshold, perform as following.
        i. Flush the KVs to MOB files.
        ii. Create KVs of the paths of MOB files with reference tags and flush them to the HBase StoreFiles.
    c) Otherwise, follow the steps in the default flusher.

The following is the code snippet for the ***DefaultMobStoreFlusher***.

```
public class DefaultMobStoreFlusher extends DefaultStoreFlusher {
  @Override
  public List<Path> flushSnapshot(SortedSet<KeyValue> snapshot, long
cacheFlushId,
      TimeRangeTracker snapshotTimeRangeTracker, AtomicLong
flushedSize,
      MonitoredTask status) throws IOException {
      //TODO flush the MOB data to the MOB files and flush the metadata
to StoreFiles }
}
```

Then users need to configure the ***DefaultMobStoreFlusher*** in the conf (*hbase.hstore.defaultengine.storeflusher.class*) as the default store flusher implementation.

5.1.2 Read



In order to implement the transparent read, we need to customize a ***StoreScanner*** (named it ***MobStoreScanner***) to read the MOB file and merge it back to the *Result*.

MOBStoreScanner reads the KeyValues from the MemStore and StoreFiles.

1. If the current store is NOT a MOB one, directly use the KeyValues.
2. If the current store is a MOB one, check each KeyValue.
    a) If it doesn't have a reference tag, directly use this KeyValue.
    b) If it has one, its value is a path of a MOB file. Find this MOB file and seek this KeyValue in this MOB file, use the KeyValue in the MOB file instead.

We could override method *getScanners()* in the ***HMobStore*** to realize this.

```
public class HMobStore extends HStore {
  @Override
  public KeyValueScanner getScanner(Scan scan,
      final NavigableSet<byte []> targetCols, long readPt) throws
IOException {
    //return the MOBStoreScanner
  }
}
```

As the entry of the ***HMobStore***, we have to customize the ***HMobRegion*** to override the ***instantiateHStore*** method which creates a customized ***HStore*** (name it ***HMobStore***). The code snippet is listed following.

1. Create the ***HMobStore*** when the column family is a MOB one.
2. Otherwise create an ***HStore***.

```
public class HMobRegion extends HRegion {
  @Override
  protected HStore instantiateHStore(final HColumnDescriptor family)
throws IOException {
    String is_MOB = family.getValue("IS_MOB");
    if(is_MOB!= null) {
```

```
      if(Boolean.parseBoolean(is_MOB)) {
        return new HMOBStore(this, family, this.conf);
      }
    }
    return super.instantiateHStore(family);
  }
}
```

Then we configure the ***HMobRegion*** in the conf (hbase.hregion.impl) as the default region implementation.

5.1.3 Cache

The MOB could use the block cache in the HBase. Users could enable it by adding an attribute in the scan.

5.1.4 Housekeeping

We provide two ways to do the housekeeping.

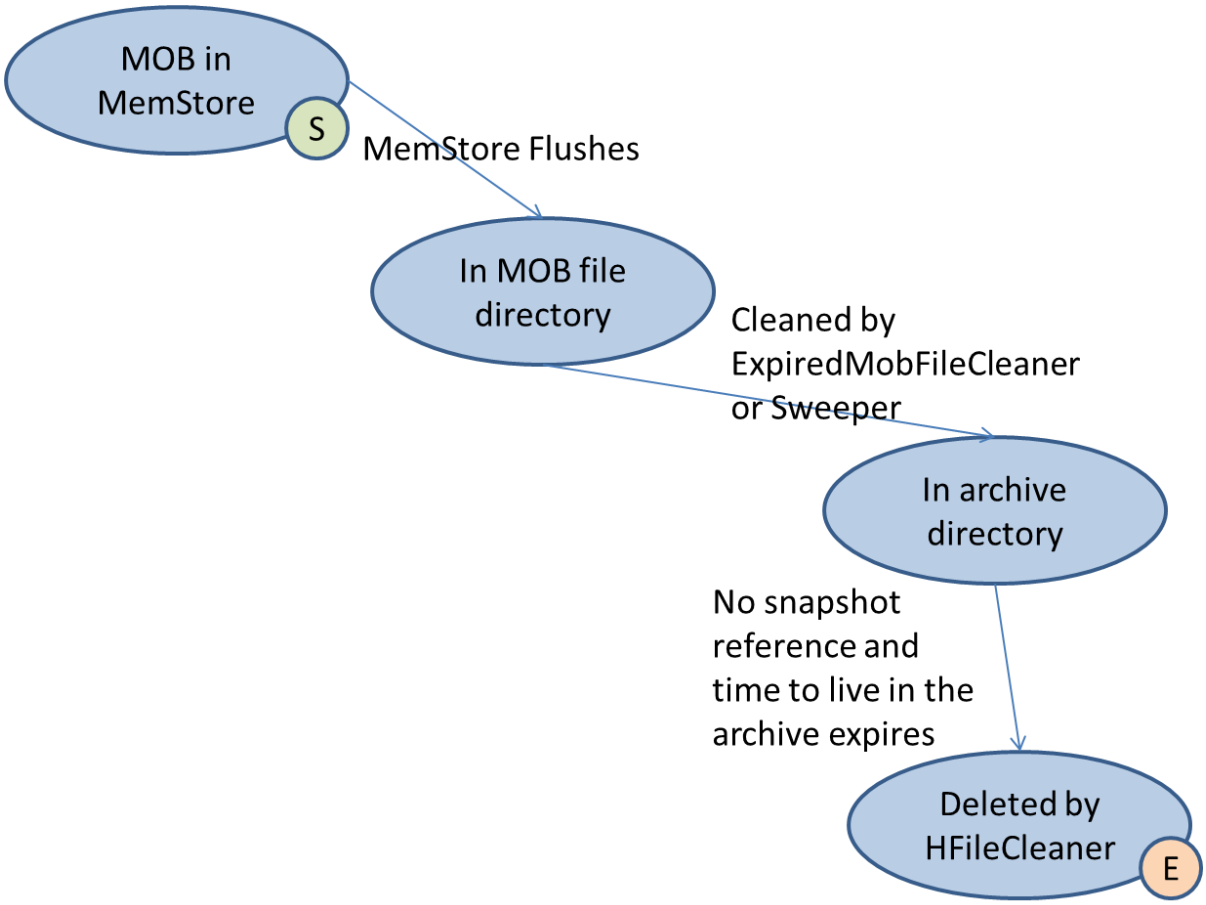
1. Delete the expired MOB files according to the TTL and minVersions in the column family. It only handles the files that are expired and the minVersions is 0.
2. Provide a user-triggered tool to clean the unused and expired MOB data.

5.1.4.1 Lifecycle of MOB Files

Before moving to the details of the tools, let's talk about the lifecycle of the mob files.

The MOB files are generated by the MemStore flushing, and eventually

deleted from the archive by the HFileCleaner.



5.1.4.2 Clean the Expired MOB Data

This is a tool triggered by users and used to delete the expired MOB files according to the TTL and minVersions defined in the MOB column family.

1. Since the names of MOB files use the date as the prefix, we could know the flush date of the MOB files.
2. Compare this date with the TTL and the minVersions defined in the MOB column family, and delete (actually archive it) the MOB file if it's expired and the minVersion is 0.

3. For those MOB files which are expired but has the minVersion larger than 0, the sweep tool will handle that.

5.1.4.3Sweep Tool

This tool performs the compaction against the MOB files. Users could run it when needed. This tool submits a MapReduce job to clean the unused and expired MOB data, and generate new MOB files.

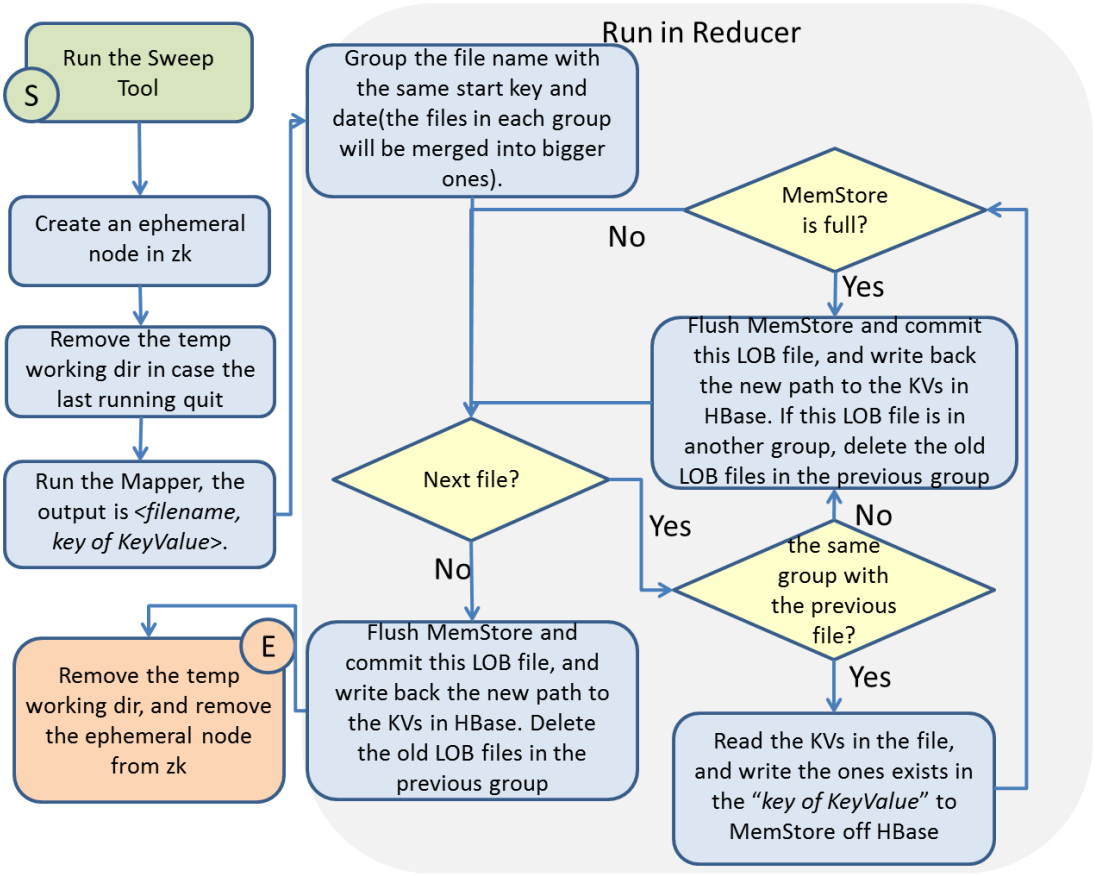
In the Mapper, the input is the HBase table (TableInputFormat), the output is <filename, keys>; In the Reducer, the input is <filename, collection<keys>>, and the output is NullOutputFormat.

In each reducer, the key of input is a file name, and the value is a collection of keys. Then we could know how many Cells actually exist in a MOB file, this is useful for sweeping.

In the reducer, the selected MOB files are merged into bigger ones. We have two policies to select the merged files.

1. If the MOB file is too small (size < configured bytes).
2. If there’re too many records are deleted from this MOB file. The name of the MOB file contains the record count in it. We could compare the existing count and capacity to know the result.

The procedures to compact the MOB files in the reducer are listed below.



1. Try to create an ephemeral node in Zookeeper. All major compactions for this column in all regions will be converted to minor ones if this node exists.
   a) Successful. Remove the temp working directory in case that the last running quit or had a failure.
   b) Failed. There's another sweeper is running. Simply quit.
2. Running the Mapper, the output is <file, key in KeyValue>.
3. The input of the reducer is <filename, keys in KeyValues>. Group the files with the same date and checksum of the start key) in the names.
4. Do merge within each group. Check each file whether it needs to be merged, the rules are introduced above.

a) Yes. Read KeyValues from that MOB file, and if its metadata exist in HBase, write the MOB data into a MemStore, flush the data to a temporary directory if MemStore is full and commit it to the MOB directory. The MOB file has the same prefix (date and checksum of the start key) with the old one (If the old one is *20140714startKey#1Capacity#1UUID#1*, then the new one is *20140714startKey#1Capacity#2UUID#2*), after that write the KeyValues back to the HBase by using HTableInterface.put.

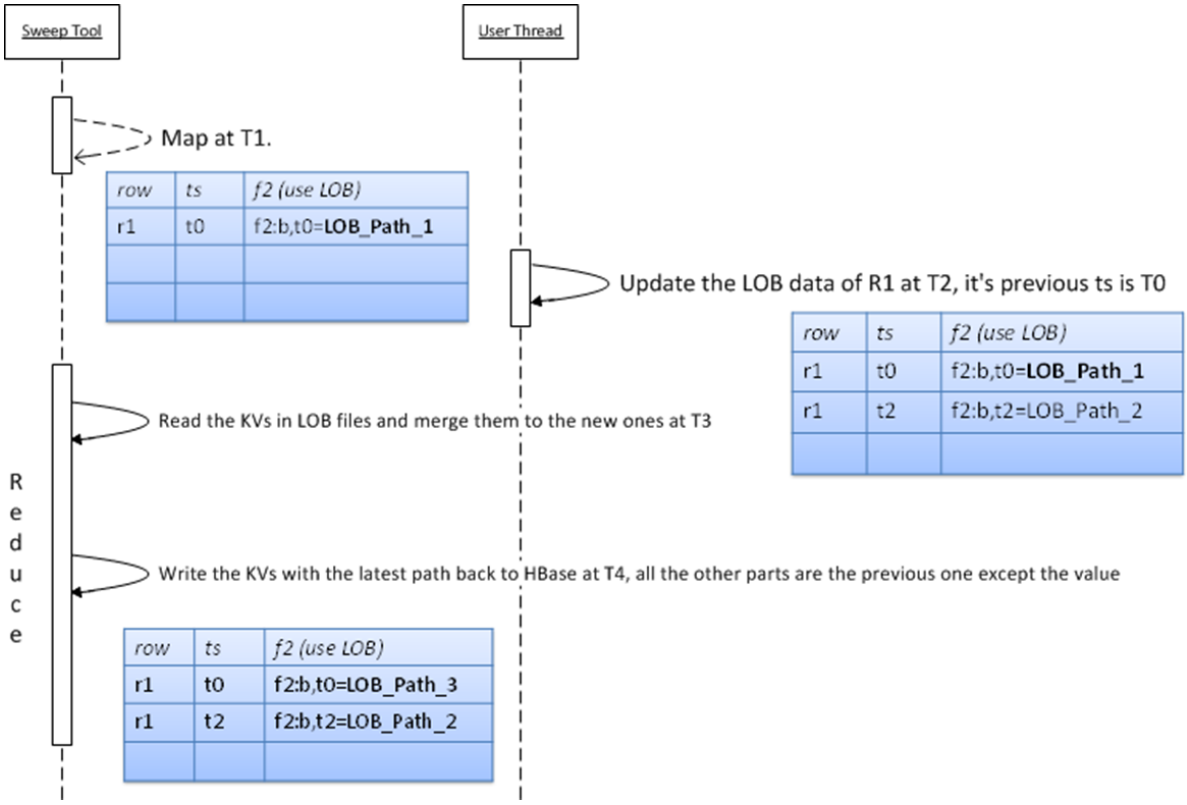
b) No, skip it.

5. Archive all the selected old files and unused file by HBase.
6. At last, remove the working temp directory and remove the ephemeral node from Zookeeper.

Do the updates during the sweeping impact the data consistency?

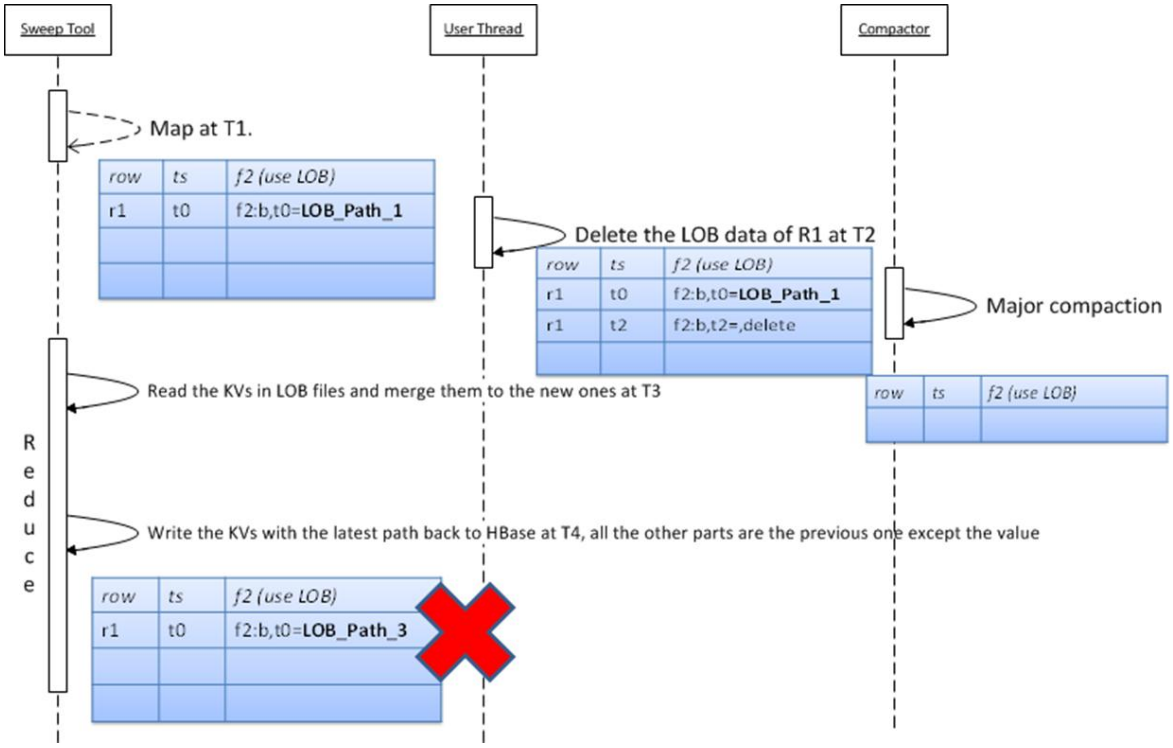
Let's see the following two cases.

1. Updates, not delete



If an update happens between the Map and Reduce, the latest KeyValue in HBase isn't replaced by the KeyValue produced by the Reducer. The KeyValue produced by the Reducer has the old timestamp, only the value which is the path of the new MOB file is changed, it don't replace the latest updates.

2. Delete

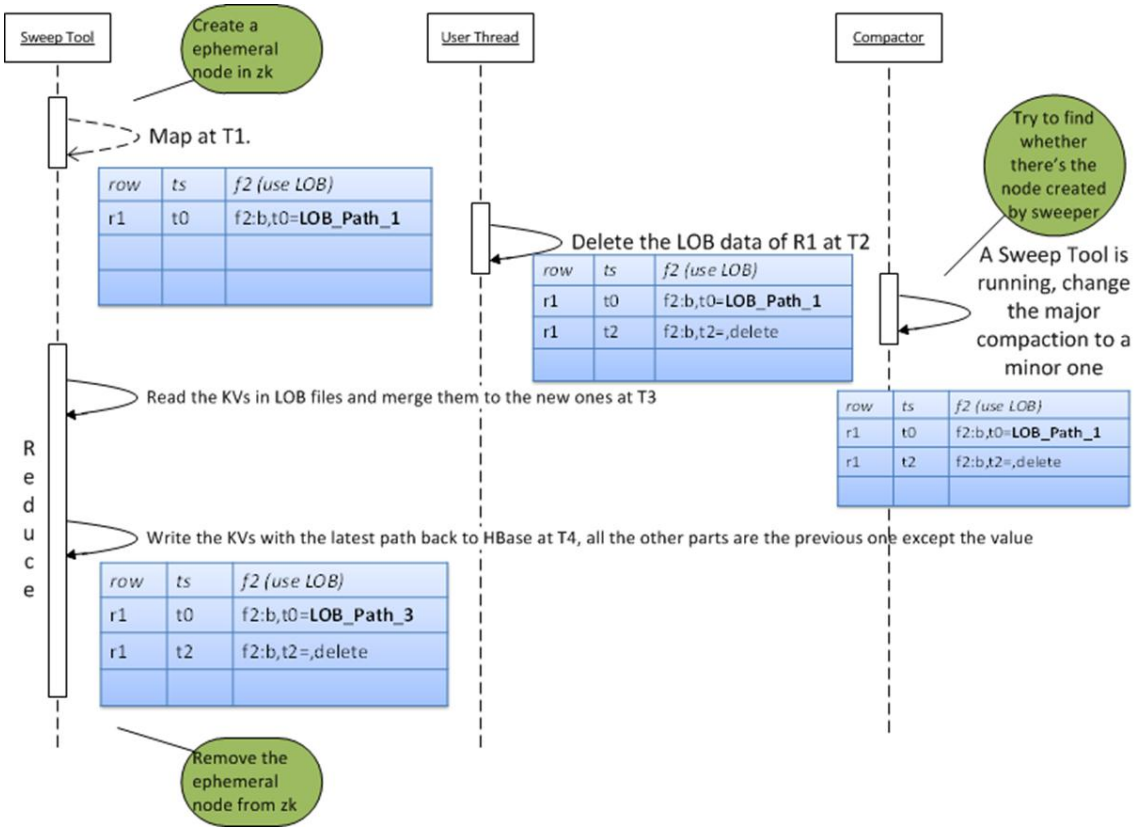


The above image shows a potential race condition issue when there's a delete and major compaction during the Sweep Tool runs. A deleted KeyValue is written back to HBase.

In order to avoid this, we need synchronize the Sweep Tool and major compaction in this store by the locks of Zookeeper. The running of the Sweep Tool and major compaction is mutually exclusive, they could not run at the same time.

1. During the major compaction, the startup of the Sweep Tool quits.
2. During the running of the Sweep Tool, the major compaction will be changed to a minor one (but all the files are counted in this compaction) by setting the scan as a raw one.

   The following image shows how it works.



What if a Sweep Tool is killed or has a failure, will the data be lost?

The data won't be lost if the Sweep Tool is killed or has a failure. As seen in the above description, for each newly flushed MOB file, we write back the new paths to the KeyValues in HBase by HTableInterface.put. Whenever the Sweep Tool quits, the KeyValues in HBase point to the right MOB files, and the pointed MOB files are either the new ones, or the old ones.

5.1.4.3.1 Mutex

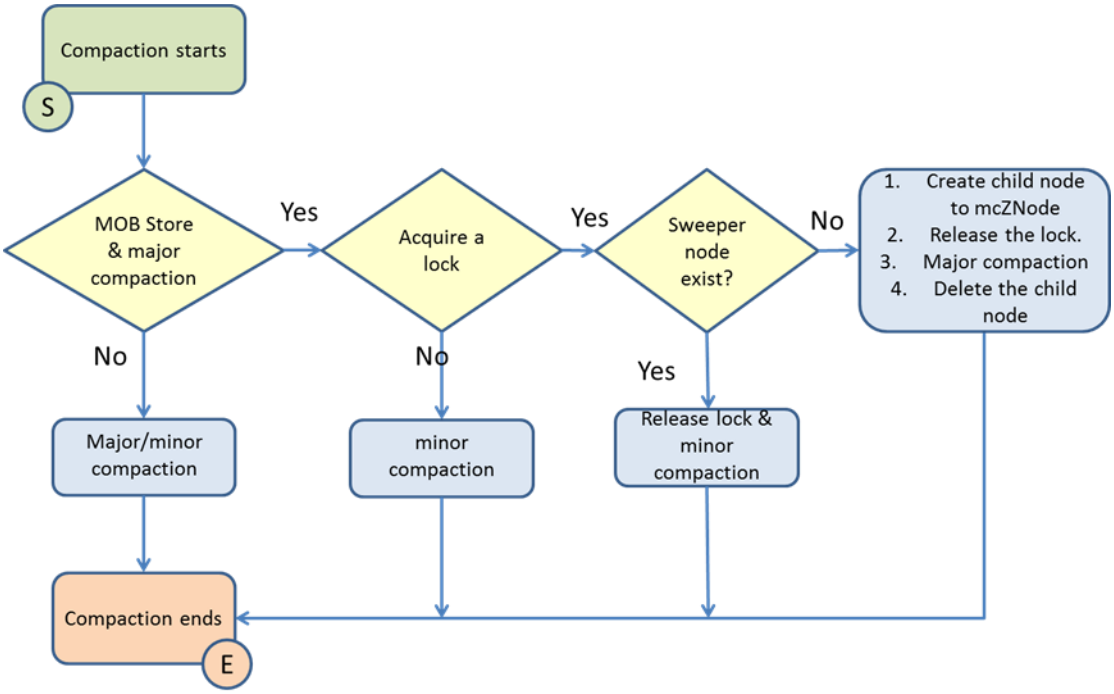
As described above, the major compactions on the MOB column family in all regions and sweeper on this column family are mutual exclusive. Use Zookeeper to realize this.

1. The major compactions for the same column family are not mutual

exclusive.

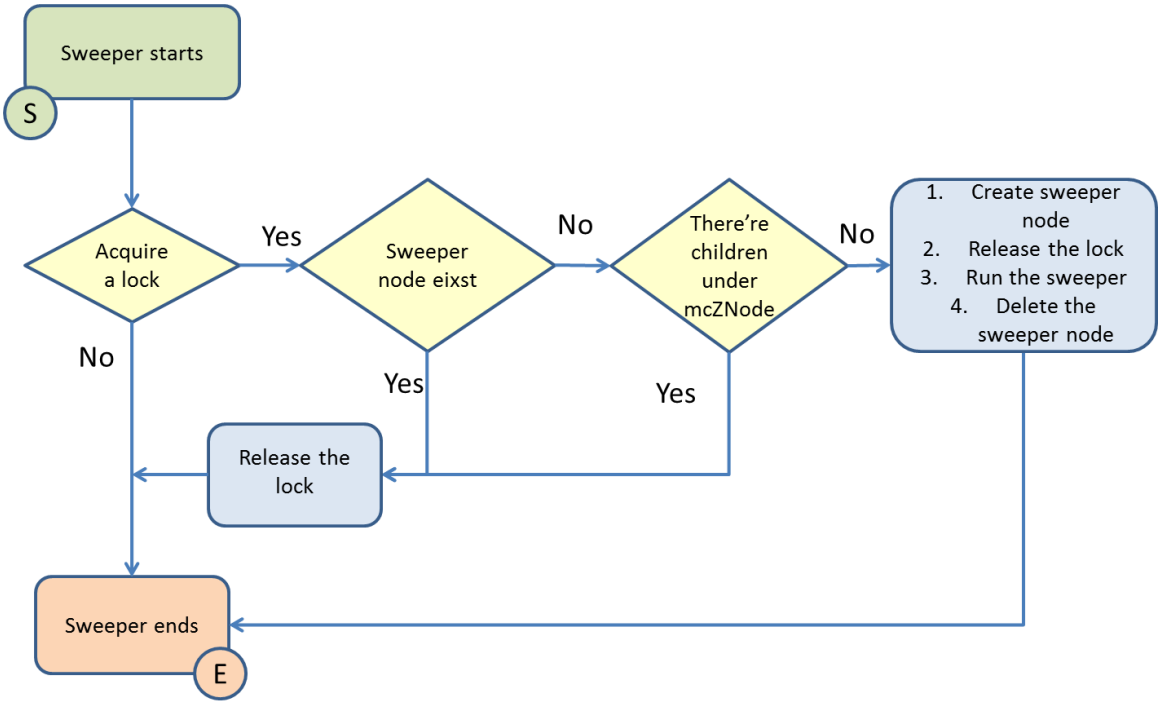
2. The compaction and sweeper for the same column family are mutual exclusive.
3. The sweeper and sweeper for the same column family are mutual exclusive.

The major compactions and sweeper maintain different Zookeeper nodes. The sweeper maintains a node named *sweeprZNode* (*table:CF-sweeper*), and the compactions maintain another node named *mcZNode* (*table:CF-majorCompactions-UUID*).



In the major compaction side, before the major compaction occurs, it checks whether the sweeperZNode exists.

1. Yes, run the compaction as a minor one.
2. No, add a child node to mcZNode, and run the major compaction.



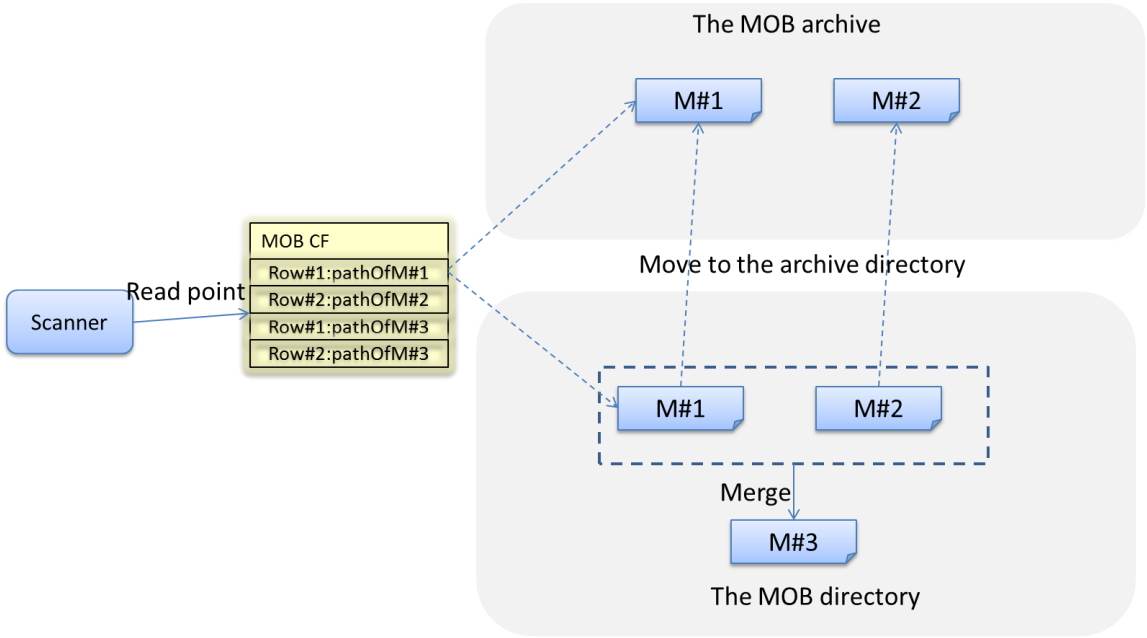
In the sweeper side, it checks whether there's a sweeperZNode, if no it continues to check whether there're children under the mcZNode.

1. Yes, skip the running.
2. No, create the sweeperZNode and run the sweeper.

There might be race condition between the checking in major compactions and sweepers. So before checking, they need to acquire a lock, and do the remains after.

5.1.4.3.2 Scan during Sweeping

What if a scanner is opened before the selected old MOB files are archived, and read MOB Cells from the selected old ones after the old ones are archived?



Since the sweeper runs out of the HBase, we could not do things like what are done after each compaction. In this case, the scanner could not get the latest path, it will read the file doesn't exist in the MOB directory. At that moment the scanner against the old files will be closed and a new scanner against the file with the same name in the archive will be opened instead.

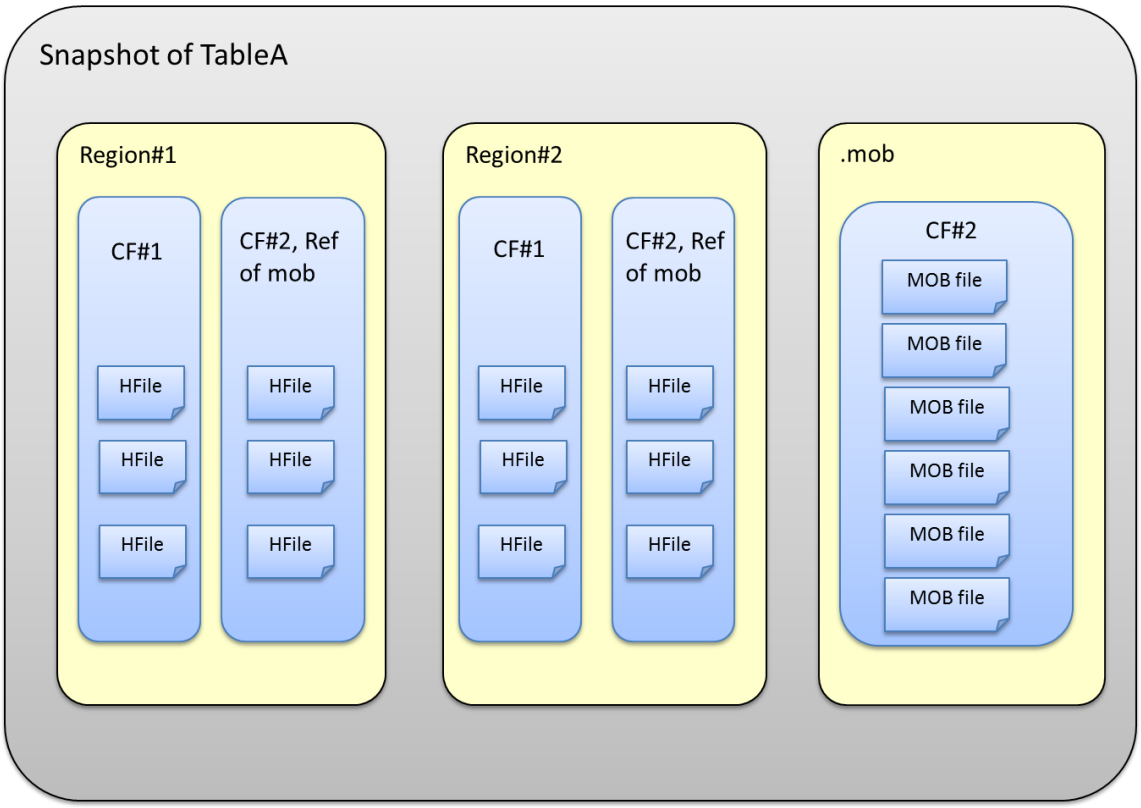
5.1.5 Snapshot

When users take a snapshot for a table, the snapshot of the MOB files should be done as well.

5.1.5.1Save Snapshot

The snapshots are saved in the same directory with the snapshot of other regions, it's regarded as a new region, we could name it .mob in the snapshot directory. Since the MOB files for all regions are saved in the same directory,

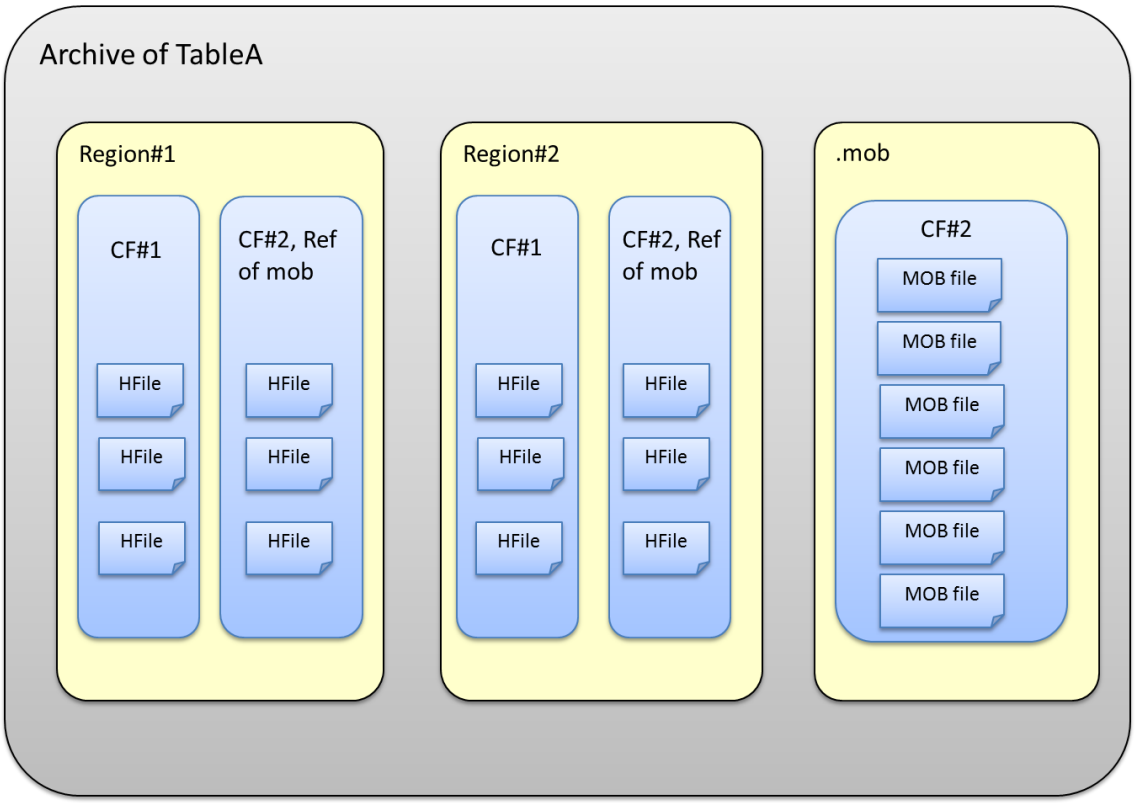
we need only take the snapshot once. We do it before the first region (which has an empty start key) does.



5.1.5.2Snapshot Archive

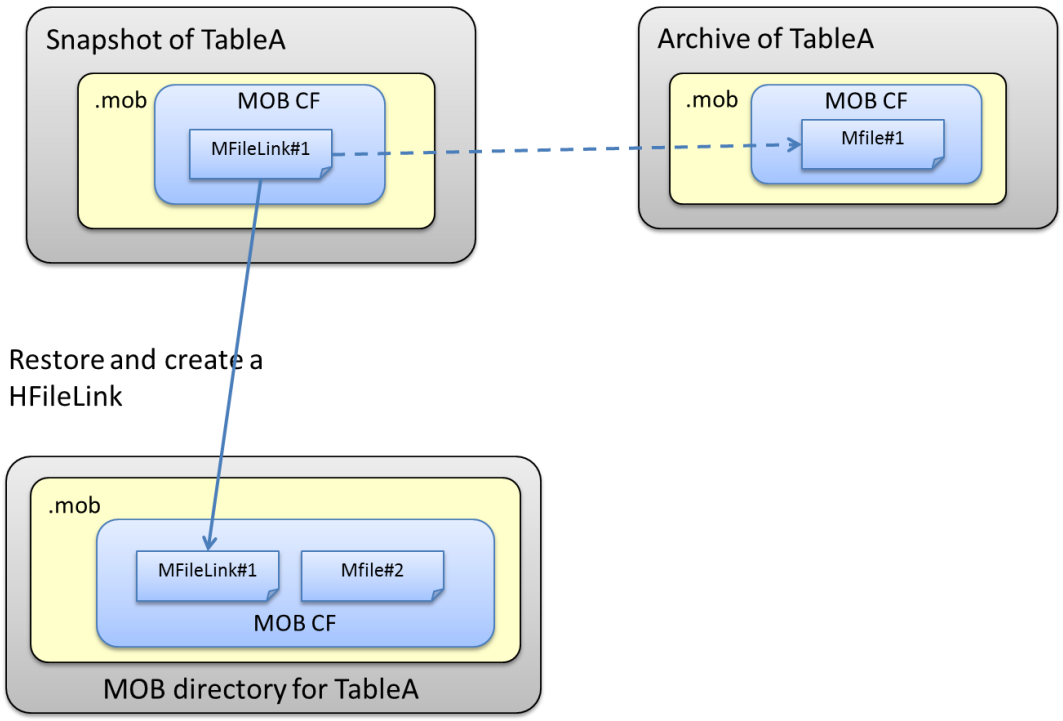
There’re two places in MOB to delete the MOB files, one is to delete the expired directory after each HBase major compaction, the other is in sweep tool. Instead of deleting the mob files, we archive them.

The archives of MOB files are saved in the same directory with archives of other regions. This should be done before the first region archives.



5.1.5.3Restore/Clone Snapshot

Similar with restoring/cloning the snapshot for the HFiles, the HFileLinks for the MOB files are created in the MOB directory when restoring and cloning. Restoring the snapshot of MOB files once before the HFiles of the first region are restored.

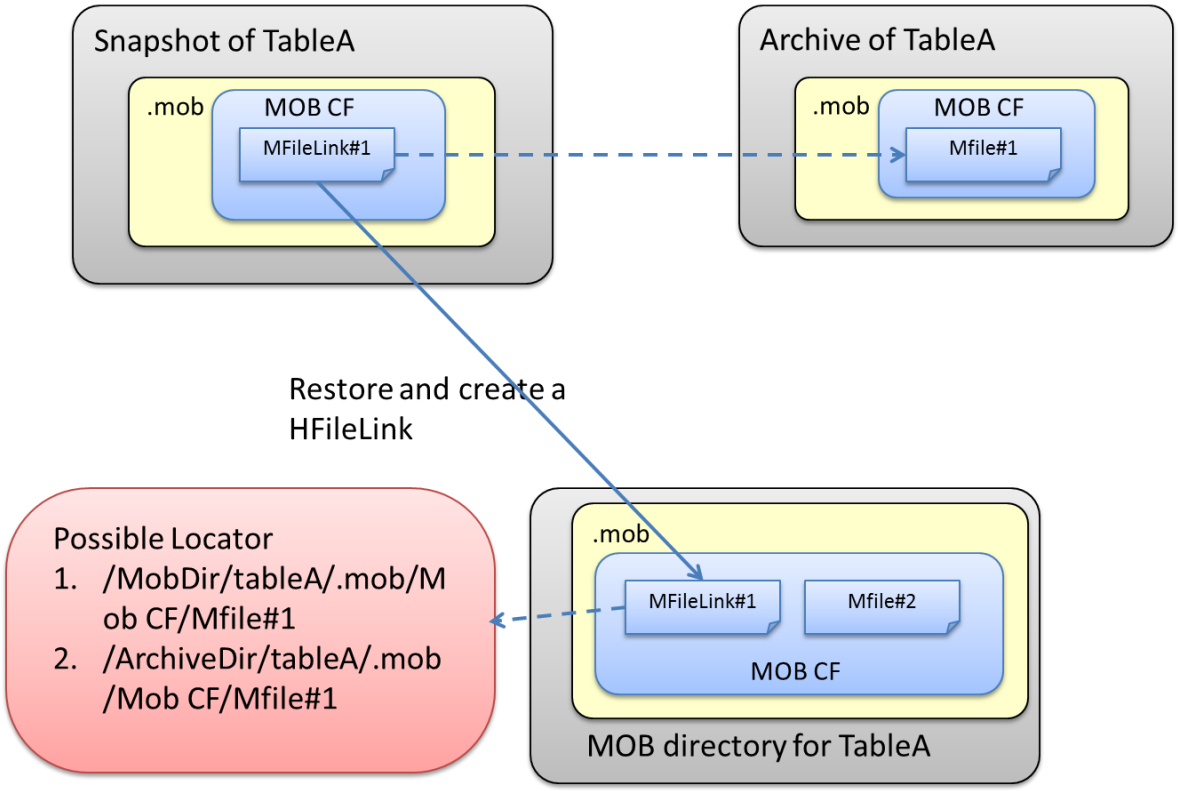


5.1.5.4Scan after Restoring Snapshot

After restoring or cloning a table, there might be HFileLinks in the region directories, as well in the MOB directories. Before opening the scanner on MOB file, we need to make sure it's a real MOB file not a HFileLink. If it's a HFileLink we need to find the real MOB file by it.

Decide whether it's a HFileLink by the file name. Check the file name under MOB directory.

1. If it's a HFileLink (the name follows the rule "tableA=.mob-Mfile#1"), check whether there's a file named Mfile#1 in the possible locations, one is under the MOB directory, the other one is in the archive directory. Use it if it's found.
2. If It's not, directly use this MOB file.



5.1.5.5HFileCleaner

The archives of MOB need be cleaned by the HFileCleaner. We need define a new FileCleanrDelegate to handle the MOB archives. If an archive satisfies both of the following conditions, it'll be deleted.

1. It's not referenced by the snapshot.
2. Users could define a TTL for the archive (not a TTL in column family, but a TTL for archive). If the archive lives too long than the TTL, it'll be deleted.
    a) Why do we need this TTL? Since the case introduced in 5.1.4.3.2, the scanner might read the MOB files in archives, we should provide enough time to let those scanners be finished.

5.2 Limitations

1. The size of the MOB data could not be very large, it better to keep the MOB size within 100KB and 5MB.
    a) Since the MOB data are inserted into the MemStore before flushing, too big MOB data will trigger the frequent flushing and even exceed the MemStore capability.
    b) The frequent flushing leads to the too many small metatada HFiles, and consequently frequent minor compactions.
2. A bigger MemStore, this avoids the frequent flushing.
3. The MOB data and metadata are saved in different column families.

5.3 Appendix: Patches for HBase Core Code

5.3.1 MOB Replica in File System

Typically the MOB data are used as archive, and accessed only when they're explicitly requested. The HA is not a strong requirement for the MOB data. Considering their relatively big size, we could set a less replication than usual for them, for example 2, even 1.

In Apache HBase, the block replication relies on the one in the underlying file system. This value works for all the HBase data. We want to enhance it to allow the replication to be configured by the path.