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: Moderate 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 meta information of MOBs, for example the title, description, etc.

2 Purpose

Nowadays, some of the users have created a growing demand for the ability to store documents, images and many other moderate objects (MOB) in HBase, and want to read and write these objects with low latency. The users may be the banks who want to store the scan copies of signed documents of customers, they might be the transport agencies who want to store the snapshots of traffic and moving cars.

The operations on MOBs are usually write-intensive with rare update or deletion, but with relatively less-frequent readings.
MOBs are usually stored together with their metadata. Metadata are the meta information of the MOBs, for instance the car number, speed, color, etc., they are very small relative to the MOBs. Typically considering the size, the metadata are usually accessed for analysis, and MOBs are usually randomly accessed only when they are explicitly requested with row keys.

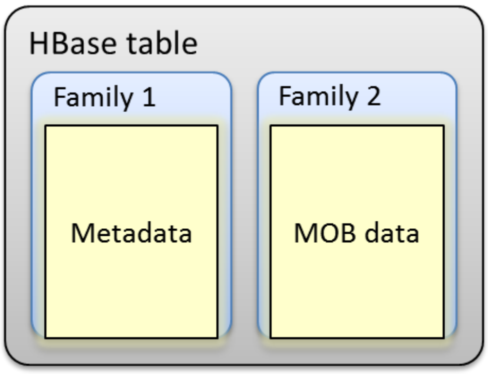
3 The design goals

1. Better and more predictable performance.
2. Strong data consistency as what is provided by HBase.
3. Minimal changes to APIs and operations.
4. Work with the exiting HBase features, for instance snapshot, bulk load, replication, security mechanism, etc.

3.1 Existing Solutions

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

3.1.1 Directly Store the MOB in the HBase Tables

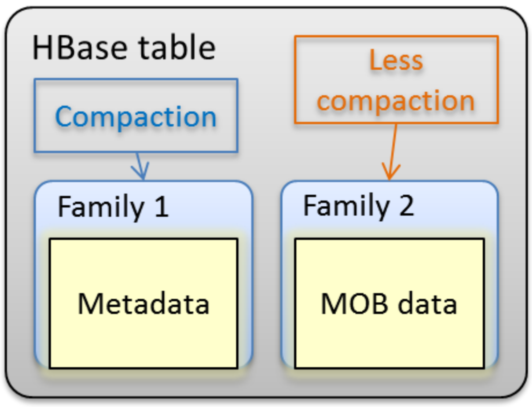


The first approach is to store the MOBs directly in HBase.

1. Store the MOBs into a separate column family.
2. Store the metadata into other column families.

Typically the MOBs are usually up to hundreds KBs or several MBs, the memstore is flushed more frequently, and consequently much more compactions are triggered. Along with the accumulation of the MOBs, I/O created by compactions is huge, which will slow down the compaction, and further block the memstore flushing, and eventually block the updates. A big MOB store will trigger the region split more frequently which makes the regions offline and block the updates too.

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

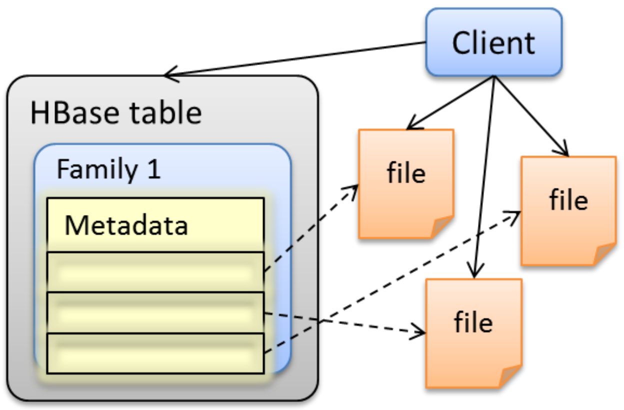


Along with the increasing number of the store files, there will be too many opened readers in a single store, even more than that is allowed by OS, and a lot of memories are consumed by them. The read performance is impacted badly.

3.1.2 Write MOB in HDFS and Metadata in the HBase Tables

Another approach is to use HBase + HDFS model to store the metadata and MOBs separately.

In this solution, the MOB is stored in a single file in HDFS, and its link is stored in the HBase.

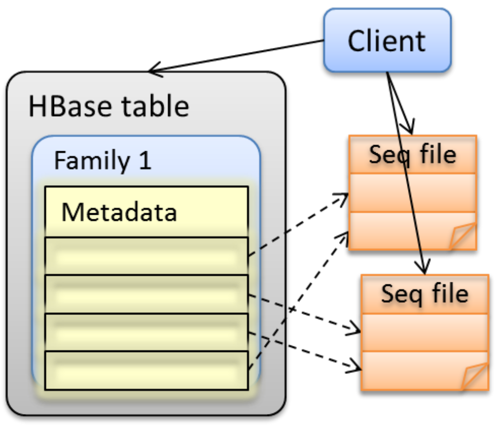


This is a client solution, and the transaction is controlled by the client. No HBase side memories are consumed by MOBs, the reading and writing are fast.

This approach is fine for LOBs which are larger than 10 MB, but for MOBs there are too many moderately sized files which leads to the inefficient usage in HDFS since the default HDFS block size is 64 MB. For example a NameNode has 48 a GB memory, each MOB file is 500 KB and each file takes more than 300 bytes in the memory. The 48 GB memory can hold less than 160 million files, and the HDFS can only store 80 TB MOB files in total.

As an improvement, we can assemble the small MOB files into a bigger one. It means we have to store the offset and length in the HBase table.

This is hard to keep the consistency for the MOBs, and difficult to manage the deleted MOBs and possible small MOB files.



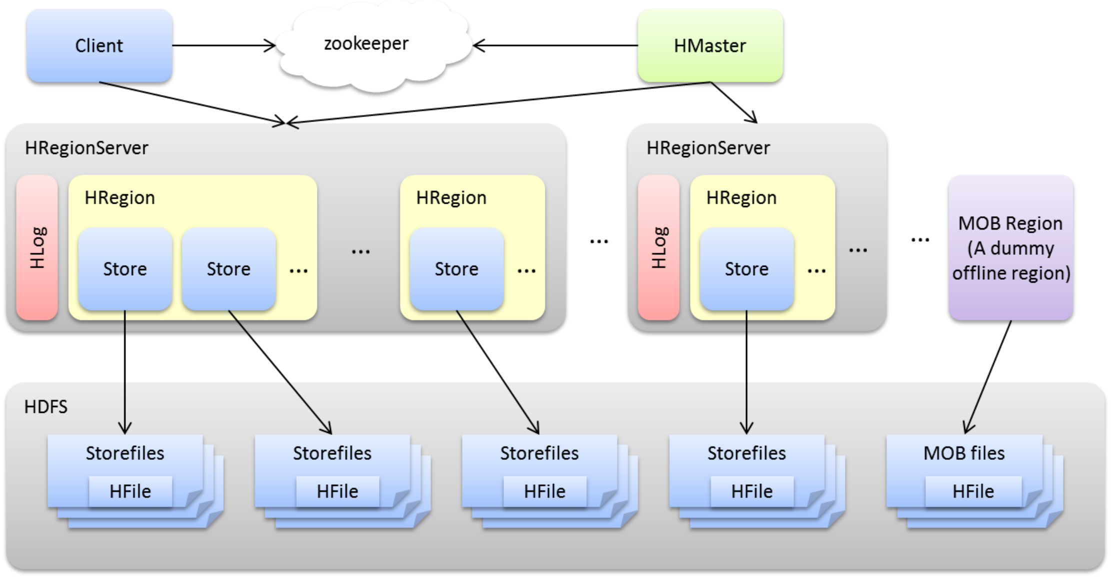
If we were to use this approach, we'd have to consider new security policies, lose atomicity properties of writes, and potentially lose the backup and disaster recovery provided by replication and snapshots.

4 HBase MOB

Most of the I/O is created by compactions, we need to move the MOBs out of the management of normal regions to avoid the region splits and compactions by them.

For the existing solutions, several problems are encounter.

1. Write amplification created by compactions impact the write performance.
2. Too many small files leads to slow read and inefficient HDFS usage.
3. Data consistency issue and pool manageability of MOB files.
    a. The flusher of memstore guarantees the data consistency in flushing. If moving MOBs out of memstore, it is hard to guarantee that.

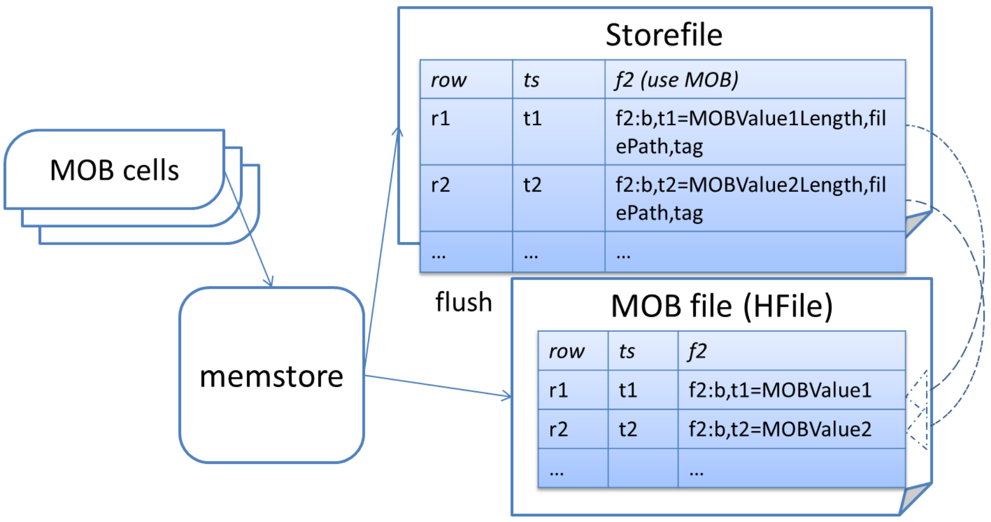


HBase MOB is pretty similar with the HBase + HDFS approach, it stores the metadata and MOBs separately too. But the differences are this a total server side design, we use memstore to cache MOBs before they are flushed to disk, and the MOBs are written into a HFile that is called MOB file in each flushing, and the MOB file has multiple entries instead of a single file in HDFS for each MOB. This MOB file is stored in a special region that is out of the management by HBase. All the read and write can be used by the current HBase APIs.

The read path of the HBase MOB is "Seek cell in HBase Table and find the MOB file path" - > "Find the MOB file by path" - > "Seek the cell by row key in the MOB file". The read from the MOB file is against a single file, it is fast no matter how many MOB files collocate with it. Too many MOB files will not impact the read, to split and compact MOB files are not necessary.

4.1 Write and read

We have a threshold for MOB, if the value length of a cell is larger than this threshold, this cell is regarded as a MOB cell.



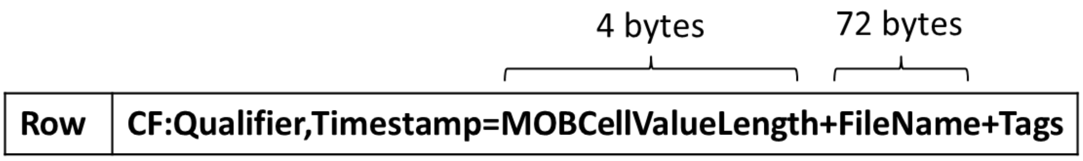
MOB cells
memstore
flush
Storefile
row | ts | f2 (use MOB)
r1 | t1 | f2:b,t1=MOBValue1Length,filePath,tag
r2 | t2 | f2:b,t2=MOBValue2Length,filePath,tag
MOB file (HFile)
row | ts | f2
r1 | t1 | f2:b,t1=MOBValue1
r2 | t2 | f2:b,t2=MOBValue2

When the MOB cells are updated into the region, they are written to the WAL and memstore, just like the normal cells. In flushing, MOB cells are flushed to MOB files, the metadata and the paths of this MOB files are flushed to store files. The data consistency and HBase replication features come inbuilt with this design.

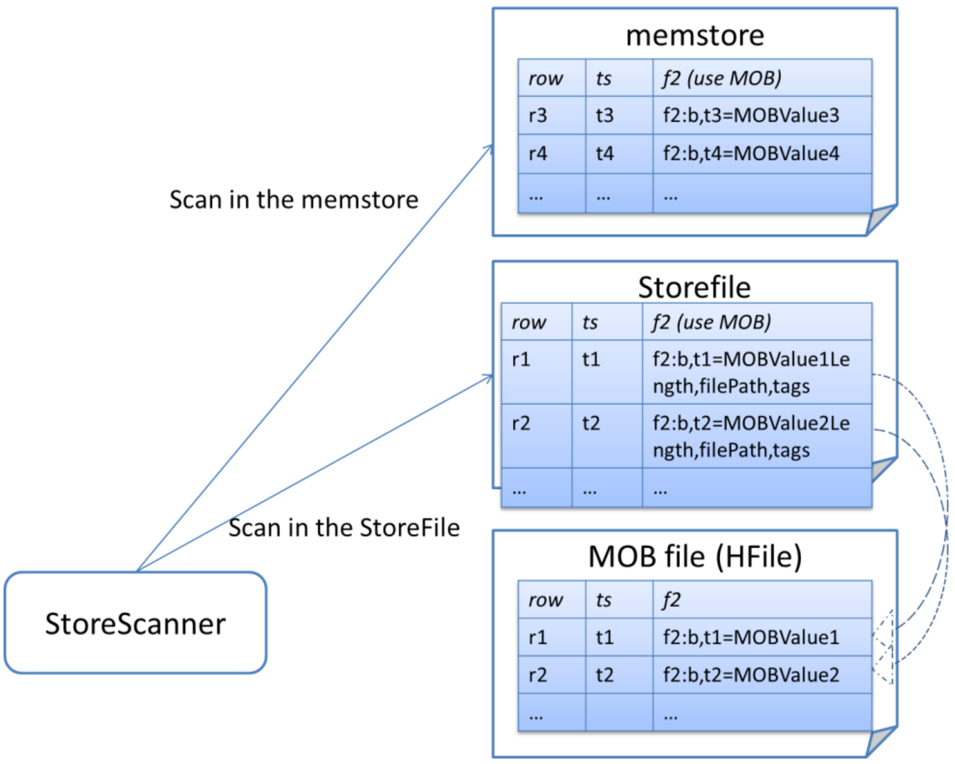
The MOB files are stored in a special and dummy region, the file path looks like */mobRootDir/data/namespace/tableQualifierName/dummyRegionName/columnFamilyName/${filename}*. All the MOB files belongs to the same table and column family are saved together.

The file name is readable, and consists of three parts, the MD5 of the start key, the latest date of cells in this MOB file, and UUID. The first part is the start key of the region where this MOB file flushed from. Usually the MOBs have a user-defined TTL, we could find and delete the expired MOB files by comparing the second part with TTL.

The cells that contain the paths of MOB files are called referenced cells, the format looks like the following.



The tags are retained in the reference cells by which we could still rely on the HBase security mechanism. Especially it has a reference tag whereby to different the normal cells and reference ones. A reference cell implies it has a MOB cell in a MOB file, we need to do further resolving in reading.



memstore

row	*ts*	*f2 (use MOB)*
r3	t3	f2:b,t3=MOBValue3
r4	t4	f2:b,t4=MOBValue4
…	…	…

Storefile

row	*ts*	*f2 (use MOB)*
r1	t1	f2:b,t1=MOBValue1Length,filePath,tags
r2	t2	f2:b,t2=MOBValue2Length,filePath,tags
…	…	…

MOB file (HFile)

row	*ts*	*f2*
r1	t1	f2:b,t1=MOBValue1
r2	t2	f2:b,t2=MOBValue2
…		…

In reading, the store scanner open scanners to both memstore and store files. If a reference cell is met, the scanner reads the file path from the cell value, and seeks the same row key from that file. The block cache can be enabled for the MOB by setting an attribute in scan that can accelerate the seeking.

4.2 Cache

We have two caches in MOB. One is the block cache of region server, the other is a cache for MOB files.

The MOB could use the block cache in reading just like the way for data blocks of HFiles by setting an attribute in scan.

The MOB file cache does not cache the data of MOB files, it caches their opened readers.

When to read mob files, the readers need to be opened. These readers should be cached within a pool which is owned by the region server to avoid they are opened so frequently. This pool size is fixed, if there are too many opened readers, part of them are closed and evicted by LRU.

4.3 Housekeeping

Note that section 4.3 Housekeeping has been obviated by "Disributed MOB Compaction".

We provide two approaches to do the housekeeping. Both of them can be triggered by users and run periodically in HMaster.

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. The MOB file compaction. The small files are merged into bigger ones, and meanwhile deleted ones are dropped.

4.3.1 Clean the expired MOB data

HBase MOB has a chore in master, it scans the MOB files, find the expired MOB files determined by the date in the file name, and delete them. Thus the disk space can be reclaimed periodically.

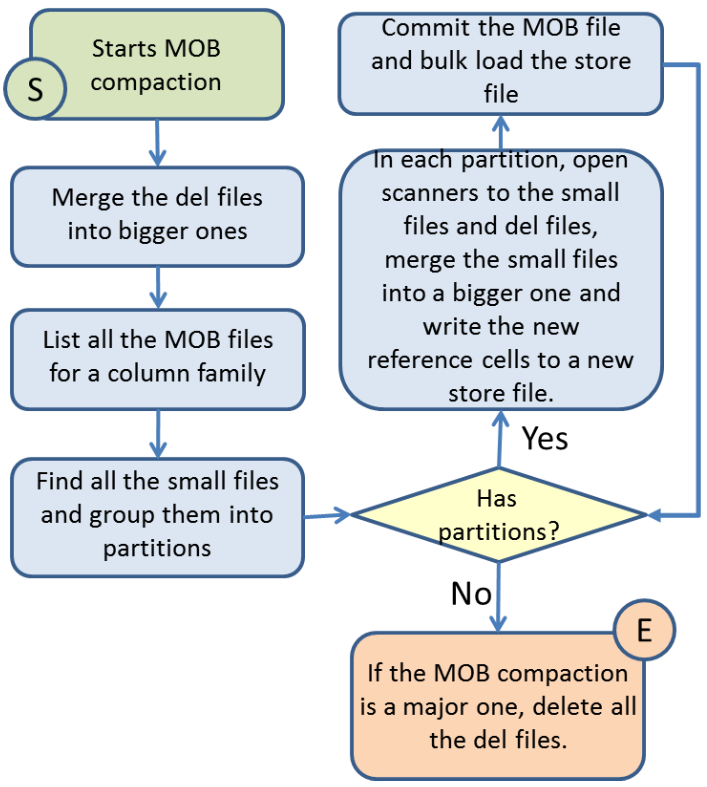
We can determine the expired MOB files by comparing the date in the file and TTL. The MOB files are expired only when the date is older than (*current time* - *TTL*) and the minVersions is 0.

4.3.2 MOB compaction

MOB files might be relatively small compared to a HDFS block if we write rows where only a few entries qualify as MOBs, also there might be deleted cells. We need to drop the deleted cells and merge the small files into bigger ones to improve the HDFS utilization. The MOB compactions only compact the small files and large files are not toughed, which avoids the infinite compaction to the large files and makes the write amplification more predictable.

We need to know which cells are deleted in the MOB compaction. So in every HBase major compaction the delete markers are written to a del file before they are dropped.

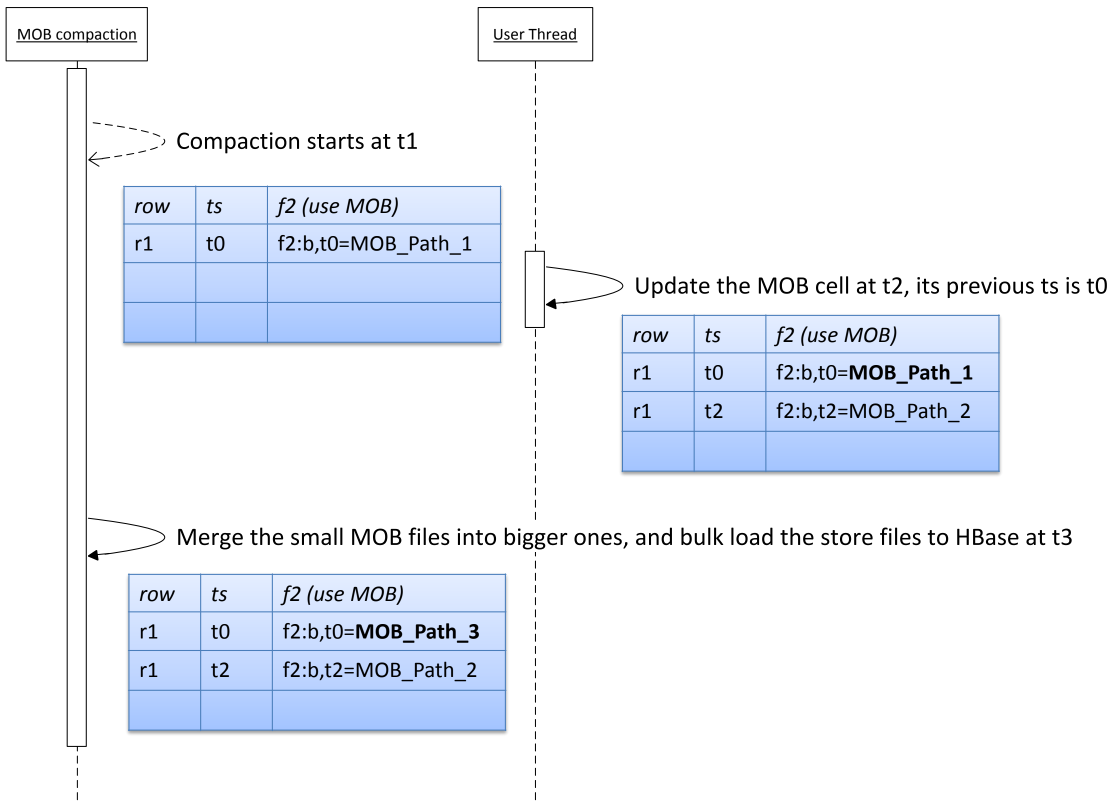
The procedures to compact the MOB files are listed below.


S
Starts MOB compaction
Merge the del files into bigger ones
List all the MOB files for a column family
Find all the small files and group them into partitions
Has partitions?
Yes
No
In each partition, open scanners to the small files and del files, merge the small files into a bigger one and write the new reference cells to a new store file.
Commit the MOB file and bulk load the store file
E
If the MOB compaction is a major one, delete all the del files.

1. Reduce the number of del files, merge the del files into bigger ones.
2. Find all the small MOB files and group them into partitions by the name prefix (the encoded name of start key and date).
3. For each partition, open scanners to the small MOB files and all the del files, write the valid cells to a new bigger MOB file, and meanwhile write the new reference cells to a new store file.
4. Commit the new MOB file, and bulk load the new store file to HBase, and delete the old MOB files.
5. If the MOB compaction is a major one, delete all the del files after the compactions are done in all partitions.

Do the updates during the MOB compactions impact the data consistency?
Let's see the following two cases.

1. Update without delete.



Compaction starts at t1

row	ts	f2 (use MOB)
r1	t0	f2:b,t0=MOB_Path_1

Update the MOB cell at t2, its previous ts is t0

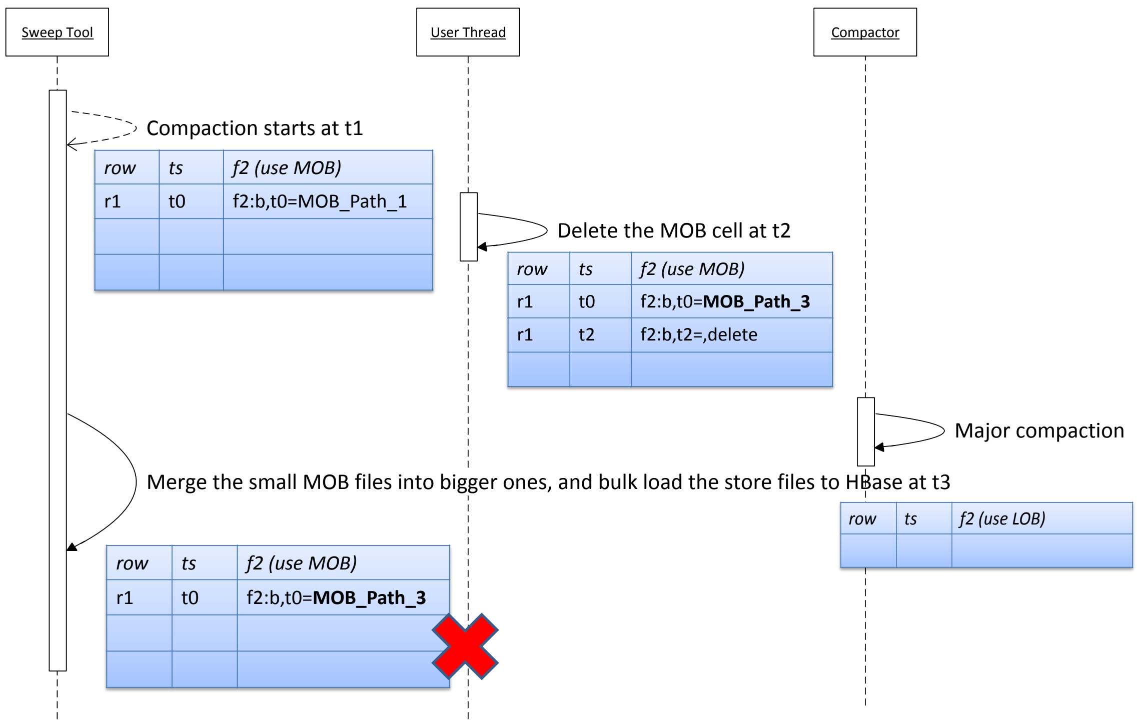
row	ts	f2 (use MOB)
r1	t0	f2:b,t0=**MOB_Path_1**
r1	t2	f2:b,t2=MOB_Path_2

Merge the small MOB files into bigger ones, and bulk load the store files to HBase at t3

row	ts	f2 (use MOB)
r1	t0	f2:b,t0=**MOB_Path_3**
r1	t2	f2:b,t2=MOB_Path_2

In the MOB compaction, when we create a new reference cell, we retain the old timestamp in it, and only the cell that has the same timestamp would be replaced. Meanwhile in commit phase, we firstly commit the new MOB file, then bulk load the store file. This could guarantee the data consistency and make the changes visible at the same time.

2. Update with delete



Compaction starts at t1

row	ts	f2 (use MOB)
r1	t0	f2:b,t0=MOB_Path_1

Delete the MOB cell at t2

row	ts	f2 (use MOB)
r1	t0	f2:b,t0=**MOB_Path_3**
r1	t2	f2:b,t2=,delete

Major compaction

row	ts	f2 (use LOB)

Merge the small MOB files into bigger ones, and bulk load the store files to HBase at t3

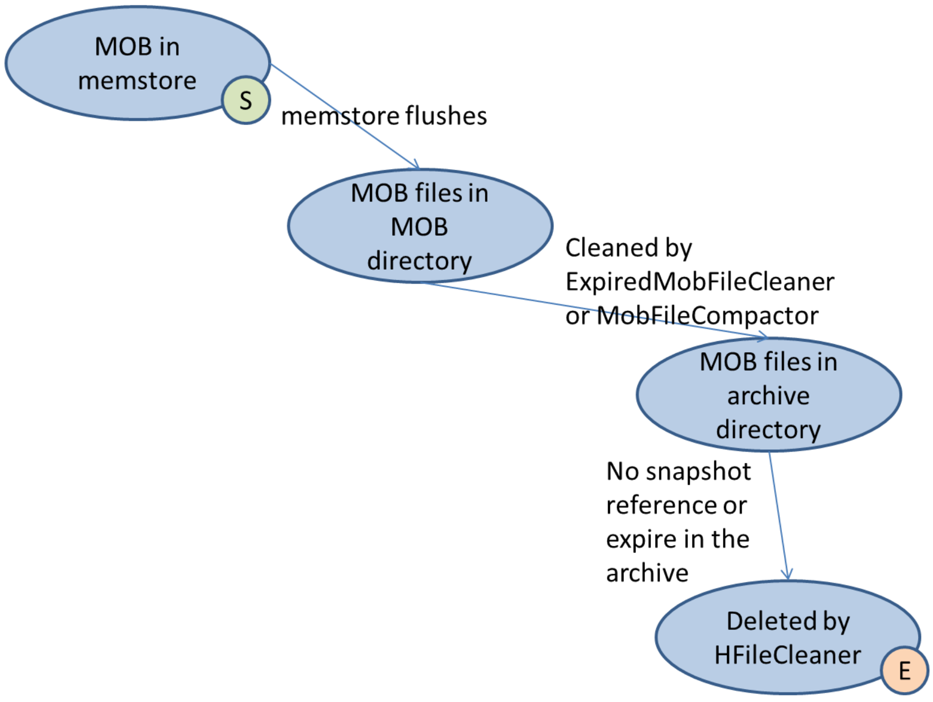
row	ts	f2 (use MOB)
r1	t0	f2:b,t0=**MOB_Path_3**

The above image shows a potential race condition issue when there is a delete marker and a major compaction when the MOB compaction runs. The deleted cells come back to HBase!

In order to avoid this, we need to add locks to the MOB compaction and region compaction on the MOB-enabled column family. In each of these operation, it has to acquire a lock before it can proceed. They are mutually exclusive and cannot run at the same time.

4.3.3 Lifecycle of MOB files

When the MOB files are deleted, they are not deleted indeed, they are archived instead. The life cycle of MOB files looks like the following, they are created when memstore is flushed, and deleted by HFileCleaner from the file system when they are not referenced by the snapshot or expired in the archive.


MOB in memstore
S
memstore flushes
MOB files in MOB directory
Cleaned by ExpiredMobFileCleaner or MobFileCompactor
MOB files in archive directory
No snapshot reference or expire in the archive
Deleted by HFileCleaner
E

4.3.4 Archive of MOB files

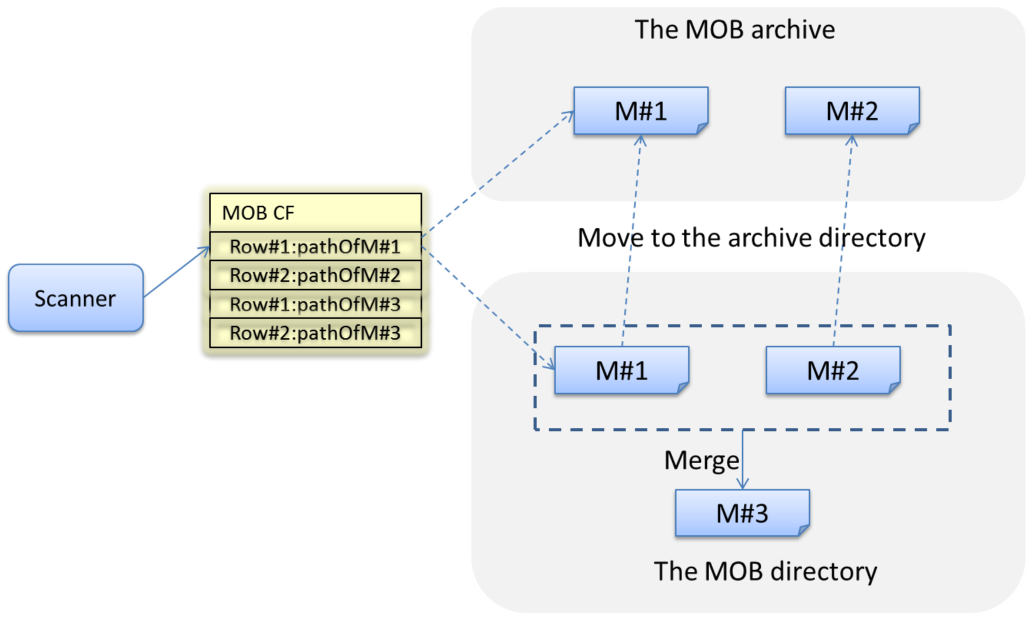
The MOB files are archived when they are deleted, just like the way in store files.

Store files are archived into the directory */{hbase.rootdir}/archive/data/ns/tn /region/cf* when they are deleted. Thus the snapshot feature can work with them.

Likewise, the MOB files are archived in the same way when they are deleted. The difference is the MOB files use a special region name, its path is */{hbase.rootdir} /archive/data/ns/tn/{mobRegion}/cf.*

4.3.5 Scan during MOB compaction

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



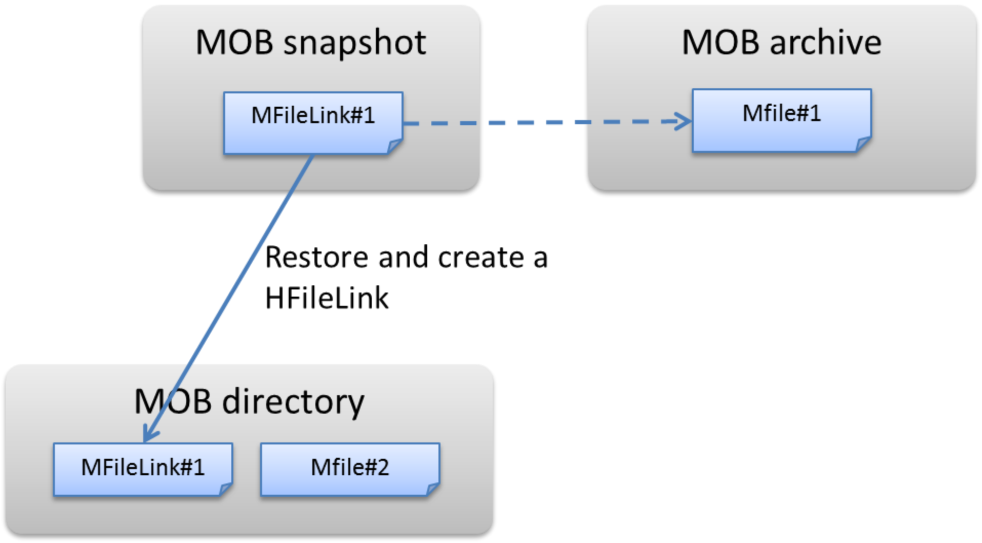
Since the MOB compaction runs out of the regions, we could not do things like what are done after each region compaction (for instance, reopen store file scanners). In this case, the scanner could not get the latest path, it will read the file doesn't exist in the MOB directory. At that moment the scanner is switched to read the MOB file in the archive directory.

4.4 Snapshot

When a user takes a snapshot for a table, HBase adds the names of existing store files into the manifest for each region, it does the same for the MOB files. In the snapshot, MOB is regarded as a special region. This region is stored in the same manifest together with normal regions in snapshot.

4.4.1 Restore/Clone Snapshot

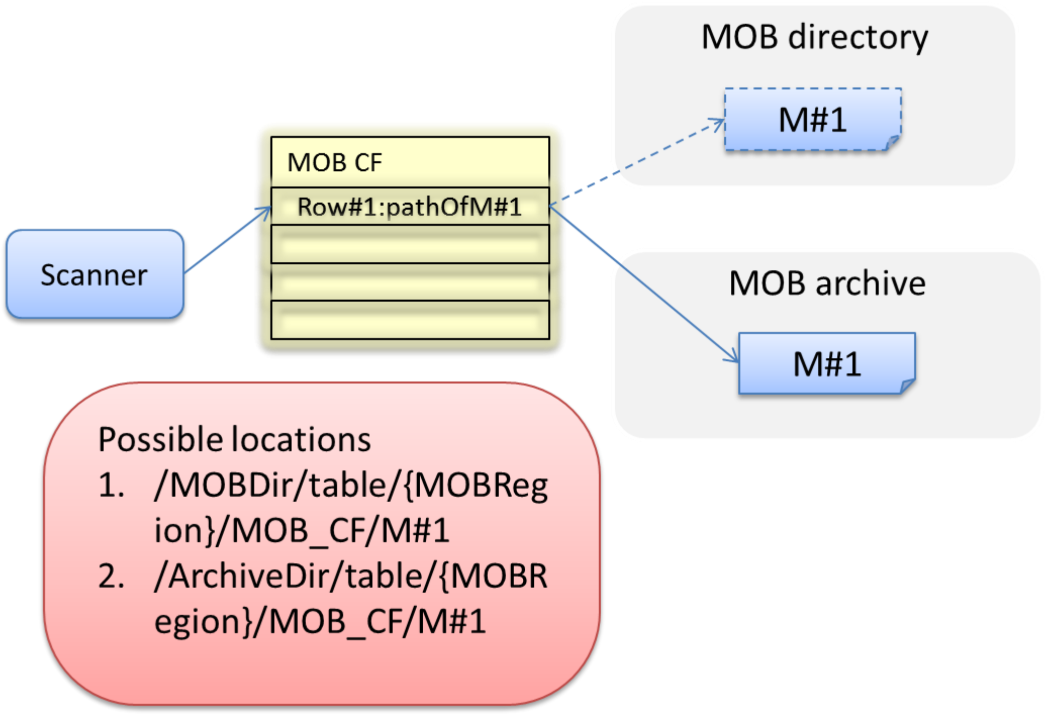
When restoring/cloning the snapshot for the HFiles in normal regions, the HFileLinks are created in the region directories. Likewise, the HFileLinks are created in MOB directory at that time. The snapshot of MOB files is restored/cloned before the first normal region (that has the empty start key) is restored/cloned.



4.4.2 Scan after Restoring Snapshot

After restoring or cloning a table, there might be HFileLinks in the MOB directory. In normal regions, the file name pattern tells where to find the real file. For MOB, due to the different read path, we need to search the files from multiple locations.

1. The MOB directory.
2. The MOB archive directory.


MOB directory
M#1
MOB CF
Row#1:pathOfM#1
Scanner
MOB archive
M#1
Possible locations
1. /MOBDir/table/{MOBRegion}/MOB_CF/M#1
2. /ArchiveDir/table/{MOBRegion}/MOB_CF/M#1

4.5 Handle MOB in compaction

The HFiles contains MOB cells could be bulk loaded into HBase, the MOB cells reside directly in HBase. How do we handle these MOB cells? Furthermore, what if the threshold is changed?

This could be handled in region compactions, for each cell in the compaction, we compare the value size with threshold, move the cell larger than the threshold to the MOB file, otherwise move the cell to a store file.

1. A cell doesn't have a reference tag (It's a normal cell in HBase), we compare the value size with the threshold.
   a) Larger than the threshold, write this cell to a MOB file, and write the reference cell to a store file.
   b) Otherwise, write it directly in a store file.
2. A cell has the reference tag, we compare the value size of the MOB cell referenced by it with the threshold.
   a) Larger than the threshold, write this MOB cell in a MOB file, and write the reference cell in a store file.
   b) Otherwise, move the cell from a MOB file to a store file.

4.6 Limitations

1. The filters related with column value don't work with the MOB scanners.
2. The size of the MOB data could not be very large, it better to keep the MOB size within 100KB and 10MB. Since MOB cells are written into the memstore before flushing, large MOB cells stress the memory in region servers.
3. The MOB cells and metadata are stored in different column families.