HBase MOB 2.0

MOB 2.0 design goals 3

MOB 2.0 non-goals 3

Unified compactor 3

MOB compaction 4

Introduction 4

Regular MOB compaction algorithm 5

I/O optimized MOB compaction algorithm 5

Algorithm analysis 5

Maximum I/O amplification 5

Calculating maximum MOB file size 6

Scalable MOB compactions 6

Batch mode 6

Code simplification and reduction 6

Important code and algorithm changes 7

DefaultMobStoreFlusher changes 7

New MOB File Cleaner Chore 7

Upgrade plan 8

Requirements 8

Performance considerations 8

Current MOB 2.0 8

MOB 2.0 compaction for small MOB (below 50KB) 9

MOB 2.0 metrics 9

MOB 2.0 testing 9

Stress tool with fault injections 9

Developer qualification 10

QA qualification 10

Upgrade testing 10

Documentation 10

Conclusion 10

Appendix A. What is wrong with MOB 1.0 or chronology of unfortunate events **12**

References **13**

Revision history

Version	Editor	Description	Date
1.0	vrodionov@cloudera.com	Initial draft	7/26/2019
2.0	vrodionov@cloudera.com	Added partial MMC section, potential optimization for small MOB (10K-50K).	7/29/2019
2.1	vrodionov@cloudera.com	Added clarification on CompactType.MOB support	7/29/2019
2.2	vrodionov@cloudera.com	New MOB compaction algorithm	9/4/2019
2.3	vrodionov@cloudera.com	I/O optimized compaction algorithm	11/13/19
3.0	vrodionov@cloudera.com	Minor changes, bumped version to 3.0	11/14/19

MOB 2.0 design goals

There are several drawbacks in the original MOB 1.0 (Moderate Object Storage) implementation, which can limit the adoption of the MOB feature:

1. MOB compactions are executed in a Master as a chore, which **limits scalability** because all I/O goes through a single HBase Master server.
2. Yarn/Mapreduce framework is required to run MOB compactions in a scalable way, but this won't work in a stand-alone HBase cluster.
3. Two separate compactors for MOB and for regular store files and their interactions can result in a data loss (see Appendix A. for details)

The design goals for MOB 2.0 were to provide 100% MOB 1.0 - compatible implementation, which is free of the above drawbacks and can be used as a drop in replacement in existing MOB deployments. So, these are the design goals of a MOB 2.0:

1. Make MOB compactions scalable without relying on Yarn/Mapreduce framework
2. Provide unified compactor for both MOB and regular store files
3. Make it more robust especially w.r.t. to data losses.
4. Simplify and reduce the overall MOB code.
5. Provide 100% compatible implementation with MOB 1.0.
6. No migration of data should be required between MOB 1.0 and MOB 2.0 - just software upgrade.

MOB 2.0 non-goals

Matching or exceeding the performance of the MOB 1.0 mapreduce MOB compaction implementation is not a goal. We are not going to compare MOB 2.0 compaction performance vs. older mapreduce implementation.
Another non-goal, at least for initial MOB 2.0 release is the optimization for small MOB values (below 50K). This optimization is possible, but we will leave it for the next MOB 2.0 release and only if there will be requests for such optimization.

Unified compactor

Our new MOB unified compaction design approach is the following:

1. Both MOB files and regular store compactions are executed by **DefaultMobStoreCompactor** now. There is no more special compactor for MOB files.

2. MOB compactions are triggered only by user's request, either from hbase shell or via Admin API. All system-originated compactions do not touch MOB data at all (they do not compact MOB files but can create a new one).
3. To run MOB compaction, User must submit **major_compact** with type=CompactType.MOB request on a table or table's region.
4. **DefaultMobStoreCompactor** does not archive MOB files and leave this duty to a new **MobFileCleanerChore** implementation (see below).
5. During any compaction (User or system, minor or major), **DefaultMobStoreCompactor** collects list of MOB files, which store file, being created, has references to and writes this list into the metadata section (***MOB_FILE_REFS***) of a newly compacted store file. As a result, every store file has a list of MOB files it will require to resolve all MOB references. This data will be used by MOB File Cleaner Chore to decide which MOB file can be safely archived.
6. **CompactType.MOB** support. Both: **CompactType.MOB** and **CompactType.NORMAL** are supported. User-originated, major compact request with type **CompactType.MOB** will compact both: MOB and regular store files. To compact only regular store files, user must set type to **CompactType.NORMAL.**

MOB compaction

Introduction

The MOB compaction is needed mostly for two reasons :

1. To decrease storage space amplification due to frequent deletes and updates of MOB cells.
2. To keep the total number of MOB files under control, which can be very important for some distributed file systems, such as HDFS.

The reason #3 - improving scan performance is not applicable (again, in the majority of use cases) for MOB data, because it is usually accessed by Get and not by a Scan operation.
In most use cases, reason #1 is not that important as well due to the static nature of a MOB (large) values. Usually, MOB data is inserted once and will never be updated or deleted until it is expired. These observations are the foundations of the I/O optimized MOB compaction algorithm.

Regular MOB compaction algorithm

This is the default MOB compaction mode. It is designed to be suitable for use cases **with frequent updates or/and deletes of a MOB data**. During regular MOB compaction of a table's region all the

MOB files, belonging to this region, are rewritten into a single MOB file. The result of regular major MOB compaction request for a region are two files: one MOB and one regular store file. During regular compaction all deleted MOB data get purged and all updates are coalesced into a single one value for a MOB key.

I/O optimized MOB compaction algorithm

For use cases with infrequent or no updates or deletes of a MOB data, the I/O optimized algorithm is proposed. This algorithm allows to minimize the overall read and write I/O amplification during MOB compaction by limiting the maximum size of a MOB file, created during compaction. The maximum MOB file size is defined by configuration parameter hbase.mob.compactions.max.file.size,
Which is, by default - 1GB. The value must be set in bytes. The algorithm works as follows:

1. No single MOB file, produced during MOB compaction, can significantly exceed the maximum size limit. If system, during MOB compaction, detects that current MOB file, being produced, exceeds the maximum size limit, the system closes current MOB file, open a new one and continues compaction.
2. During MOB compaction, only MOB files, whose size is below the maximum limit will be compacted. It means, that all MOB files with sizes above the limit will stay untouched and all MOB files with sizes below the limit will be compacted into one or (potentially) more MOB files.
3. If overall size of all small MOB files for a region exceeds the maximum size limit - more than one MOB file will be produced during compaction (see 1.)

Algorithm analysis

Maximum I/O amplification

The algorithm provides an upper bound for a write amplification, which can be approximately estimated as:

WA = logK (M/S)

where K - average selection size - number of files in compaction selection, M - threshold defined by **hbase.mob.compactions.max.file.size** and S - average size of a MOB file after memstore flush. The read amplification has approximately the same upper bound.

Example:

K = 5, M = 1GB, S=10MB, write amplification is $log_5(1GB/10MB) = log_5(100) = 2.86$

Calculating maximum MOB file size

As overall data size grows, the majority of MOB files will be at max limit. So we can propose a simple formula to select maximum file size:

M = P/N -

(where M - maximum MOB file size, P - maximum data size expected and N - recommended maximum number of MOB files in a system). WIth P = 1PB and N = 1M - we have default maximum size - 1GB.

Scalable MOB compactions

Periodic MOB file compaction Chore will be added to run MOB compactions periodically without user's interventions. This Chore will scan all tables, find those with MOB-enabled column families and will issue **major_compact** request with **type= CompactType.MOB** on those column families, using the standard HBase Admin API. That chore is present in the original MOB (although it does only cleaning of a TTL-expired MOB files)**, but now MOB compactions can run in parallel and are not limited by a single Master only.**

Batch mode

To avoid possible compaction storms, the new configuration parameter has been added - hbase.mob.compaction.batch.size - the maximum number of regions, which can be compacted in parallel. Default value is 0 - means no limit. If this parameter is set to a non-default value N, the MobFileCompactionChore will group regions in batches of size N and will compact batches serially. The batch mode is implemented in **MobFileCompactionChore**, therefore it can be run only in automatic mode as a chore in HBase master. The batching is happening on a HBase Master. **MobFileCompactionChore** collects all regions for a table, shuffles them, then issues compaction requests in batches, then waits until the batch is complete, then issues next batch requests, etc. When user requests MOB compaction either via HBase shell (major_compact command) or in an application - the batch mode is not available, all regions of a table will be compacted simultaneously.

Code simplification and reduction

The overall code size will be reduced quite significantly:

1. The **mob.compactions** and **mob.mapreduce** packages and corresponding test packages will be gone completely.
2. Both: MOB and regular compactions will be unified in a single place.
3. No more special treatment for Delete MOB cell operations, no more separate **_del** files and all the code to support Delete MOB functionality will be gone too, because now all deletes are handled by a default store compactor.

Important code and algorithm changes

DefaultMobStoreFlusher changes

MOB store flusher creates two files during memstore flush: MOB file and corresponding store file. The new implementation adds metadata (***MOB_FILE_REFS***) to a store file to keep the MOB file name (see above compaction section). This is the difference between original **DeafultMobStoreFlusher** and redesigned **DeafultMobStoreFlusher** 2.0.

New MOB File Cleaner Chore

Original MOB cleaner chore in MOB 1.0 does only one task: it checks for TTL-expired MOB files and archives them if finds such a files. Our new MOB cleaner must take care of obsolete (after compactions) MOB files. These files must be archived as well. This is what the new MOB cleaner chore does:

1. It still does TTL expiration check, but after that:
2. It collects all live MOB file names from metadata of a MOB table's column family store files in a single run.
3. If a store file does not have metadata with the key (**MOB_FILE_REFS**) and has metadata with the key (**MOB_CELLS_COUNT**) - it means that a store file has been created by older version of MOB, the chore logs the message that it can not proceed because old store file still exists and the chore aborts the overall operation. This is done to support seamless migration from original MOB 1.0 to a new MOB 2.0. The overall cleaning is postponed until all the regions of a table are MOB-compacted and hence have corresponding metadata key (**MOB_FILE_REFS**).
4. If a store file does not have both: **MOB_FILE_REFS and MOB_CELLS_COUNT** - it is either store file after compaction with no MOB cell references, a bulkoaded file or a MOB file, which was just flushed and has not gone through a single compaction cycle yet. To distinguish between bulkloaded file and freshly flushed MOB file we check **BULKLOAD_TASK_KEY.** The Chore aborts if it finds freshly flushed MOB file (no bulkload key, no mob reference key, no mob cell count key).
5. After compiling a list of all live MOB files for a MOB column family, the chore scans MOB files in MOB directory and archive those not in the list.

Upgrade plan

The migration of data is not required after upgrading system to a MOB 2.0. The new compaction and cleaner chore in MOB 2.0 are smart enough to support both: old and new data. MOB 2.0 will never

archive old MOB file until all table regions are gone through the new MOB compactions and hence no data loss will be possible.

Requirements

During post-upgrade process, before all MOB tables will be major compacted, an additional storage space, of at least of the size of the largest MOB table, will be required. So the upgrade process should be planned accordingly.

Performance considerations

Current MOB 2.0

We will measure the impact of the implementation of MOB compactions in both: general performance (how long will it take to fully compact MOB table) and I/O overhead (how many bytes were read and written during compaction). We will compare these numbers with MOB 1.0 version.
From a preliminary estimate, we can predict that the new compactor in MOB 2.0 will vastly outperform, in general performance, MOB 1.0 compactor, orchestrated by Master, it will have less I/O overhead in a partial major MOB compaction mode, than old MOB 1.0 compactor and the new compactor will have comparable I/O overhead in default (non-optimized) mode.

One major difference between MOB compactors in 2.0 and 1.0 is the former compactor reads MOB cells issuing random I/O requests, when the latter one reads MOB data sequentially. This should not hurt the performance of MOB 2.0 compactor because of the following factors:

1. The read request size is at least 100KB (this is default MOB threshold and lower size bound for MOB value in original MOB design). Even a single HDD can serve 150 random requests per sec, which translates to 15MB/sec compaction throughput for a single compaction thread.
2. Increasing number of compaction threads will improve performance linearly (in a multi-HDD storage setup) in a single Region Server. As an example, with 4 HHDs per Data Node a compaction thread pool with 4 threads will increase throughput to 60MB/sec (with 100KB average MOB value size), which is very good throughput by any measure.
3. When SSDs are used as a storage media - this is not the issue at all.
4. MOB 2.0 compactor will outperform 1.0 anyway because it scales linearly with cluster size and MOB 1.0 does not (when Master-orchestrated)
5. The preliminary analysis of a data access pattern in MOB files in MOB 2.0 compactor shows that I/O may not be that 100% random and OS can take advantage of prefetching data, which should improve read performance as well.

We are pretty confident that the current MOB 2.0 compaction algorithm will be more than adequate for MOB deployments with MOB threshold equals to or above the default MOB threshold of 100KB, but to support lower values of a MOB thresholds 10 - 50KB (although it is against the original MOB

design approach, where MOB value starts with 100KB) additional optimization may be required - converting random MOB read requests into sequential ones.

MOB 2.0 compaction for small MOB (below 50KB)

This is not the goal for the first release of MOB 2.0, but we can improve MOB compaction performance for small MOB values, reading MOB cells during MOB compactions using the same **StoreScanner** machinery, which HBase provides to read key-values orderly from a multiple store files. We will leave the details of this optimization for the next MOB 2.0 release and only if there will be requests for such an optimization.

MOB 2.0 metrics

No special to MOB 2.0 metrics are required. The existing MOB compaction metrics will suffice.

MOB 2.0 testing

Stress tool with fault injections

Stress tool, which can be run from a command-line, will be used in both: developer and QA qualification to test the new MOB 2.0. The tool will execute the following tasks:

- Loading MOB and not MOB data, using write thread pool.
- Running, in parallel with data loaders, MOB compaction chore (default interval is 2 min).
- Running, in parallel with data loaders, MOB file cleaner chore (default interval 2min).
- Running HBase archive cleaner in aggressive mode (with intervals 60-120 sec)

After data loading is complete, the dedicated Read thread pool will scan all the MOB table and compare MOB values, read from a table, against expected values.

The new **FaultyMobStoreCompactor** will be used by the tool to introduce random **IOExceptions** during MOB compactions.

Developer qualification

Includes running stress tool in **dev** environment with number of rows inserted up to 30M. The test with 30M rows run takes up to 8 hours to complete (single threaded, w/o write and read thread pools). Adding write and read thread pools must improve testing performance significantly.

QA qualification

Includes running stress tool in a cluster environment with up to 100M rows. QA will also test upgrade from MOB 1.0 to MOB 2.0.

Upgrade testing

Upgrade testing procedure steps:

1. Load data using existing MOB 1.0 (pre-upgrade step)
2. Shutdown the cluster.
3. Dump the content of MOB table's mob directory (list of MOB files).
4. Upgrade cluster to MOB 2.0
5. Set MOB file cleaner chore interval to a low value , say X min (X is low minutes)
6. Start up cluster
7. Run MOB compaction on a MOB table from hbase shell, wait until it completes.
8. Wait X min
9. Verify all data are in place
10. Verify that MOB Cleaner Chore started working (old data must be moved to archive) and no old files from a list compiled in step 3 are present in a mob directory.

Documentation

The HBase documentation will be updated with the design approach, upgrade procedure and configuration changes (new configuration parameters). The obsolete MOB 1.0 configuration parameters will be removed from the documentation.

Conclusion

The new MOB has the following advantages over the current implementation:

1. MOB compactions are now scalable, because they run in parallel across a cluster and not through a single Master server.
2. Mapreduce/YARN is not required anymore to achieve comparable to MOB 1.0 in mapreduce mode scalability.
3. The code size is reduced - two subpackages have been removed, some other utility classes - MobUtils has been reduced as well.
4. Both MOB and regular compactions are now unified and are executed by a single **DefaultMobStoreCompactor.**
5. The code itself is more self-contained and does not rely on some low level details or peculiarities of other HBase subsystems, such as correct bulk loading handling including error handling (see Appendix A for details).
6. And it is free of a race condition, which we describe in the next section of this document.

Appendix A. What is wrong with MOB 1.0 or chronology of unfortunate events

Our investigation of a MOB and MOB - related data losses started some time ago, more than a month before the corresponding ticket has been opened in Apache HBase (HBASE-22075). The customer of ours has complained about unexpected data loss in a MOB - enabled tables. This always happened after MOB compaction, which they run periodically, once a week. The HBase version they use is based on 1.1.2 (HDP 2.6.5). The MOB (medium size objects) itself was introduced in HBase v2.0 and has been backported to our HDP 2.5 release line by a colleague of ours at HW.

So, the initial hypothesis was - something was wrong with backporting. We did some (quite extensive) investigation, it took time to get to the “root cause” (as we thought at that time) and it turned out that the issue is present in all HBase 2.x and in the master branch as well. This resulted in HBASE-22075 (Potential data loss when MOB compaction fails). Here: HBASE-22075.

The issue #1

The first case of improper error handling occurs at the end of a compaction of the backing files that hold MOB values. The final step of compacting these files is a bulk load of updated reference cells for the regions that have values over the MOB threshold. In certain edge cases the bulk load can fail in such a way that some regions have successfully been updated to reference the newly compacted files while some still refer to the original files. The existing MOB compaction code does not distinguish this partial failure from a total failure where none of the regions were updated. Thus it acts as though none of the newly created files are needed and removes them, assuming any existing references must be to the original backing files. Those regions that successfully applied the bulk load will subsequently report a failure to find the removed files when attempting to read reference cells.

The patch was small, we were able to reproduce the issue (data loss with FileNotFound exceptions) without patch and were not able to reproduce it with the patch and the patch, along with the updated jar files, were sent over to our customer for internal testing. But, unfortunately, a couple weeks later, our customer returned back with the same complaint - the same issue again and again - after MOB compaction runs. The same data loss, the same FileNotFound exceptions for some missing MOB files. So there was something else, what caused data loss.

We have spent a lot of time, trying to reproduce the issue in our own lab and, long story short, finally, came to the following conclusion: there is a race condition between MOB and regular compactions,

in some rare cases. This was a plausible explanation, but, unfortunately, we were not able to catch and fix this race condition.

The issue #2

The data loss issue occurs due to a race condition between Major Compactions of normal regions and a MOB compaction that runs concurrently. In this case, a given Region Server creates a newly compacted HFile with reference cells that point at the MOB files from prior to the MOB compaction. In parallel, a MOB compaction running on the Master replaces these backing MOB files. If the race results in the MOB compaction finishing its reference update followed by the Major Compaction completing then the end state will be reference cells that point to files that are no longer tracked as needed by the MOB system. Once the regular process of cleaning out old files completes impacted regions will report a failure to find the removed files when attempting to read reference cells.

Having spent an enormous amount of time on debugging MOB, we were about to give up: we have almost lost our confidence in finding the root cause in the existing MOB code. Something was definitely wrong in a MOB compaction code, this was for sure, so we decided to rewrite MOB compaction code completely. The idea was - if two compactions, MOB and regular, can produce unexpected results, such as a data loss, let us get rid off MOB compactions completely and compact MOB files during regular major compactions. This is how the **unified compactions** and MOB 2.0 were born.

References

1. HBASE-22075 Potential data loss when MOB compaction fails
2. HBase MOB Design