# Date Tiered Compaction

Clara Xiong
Flurry/Yahoo!
clarax98007@gmail.com

## Problem Statement

ExploringCompactionPolicy is the default policy for minor compaction for HBase. Figure 1 illustrates the algorithm with the default settings.

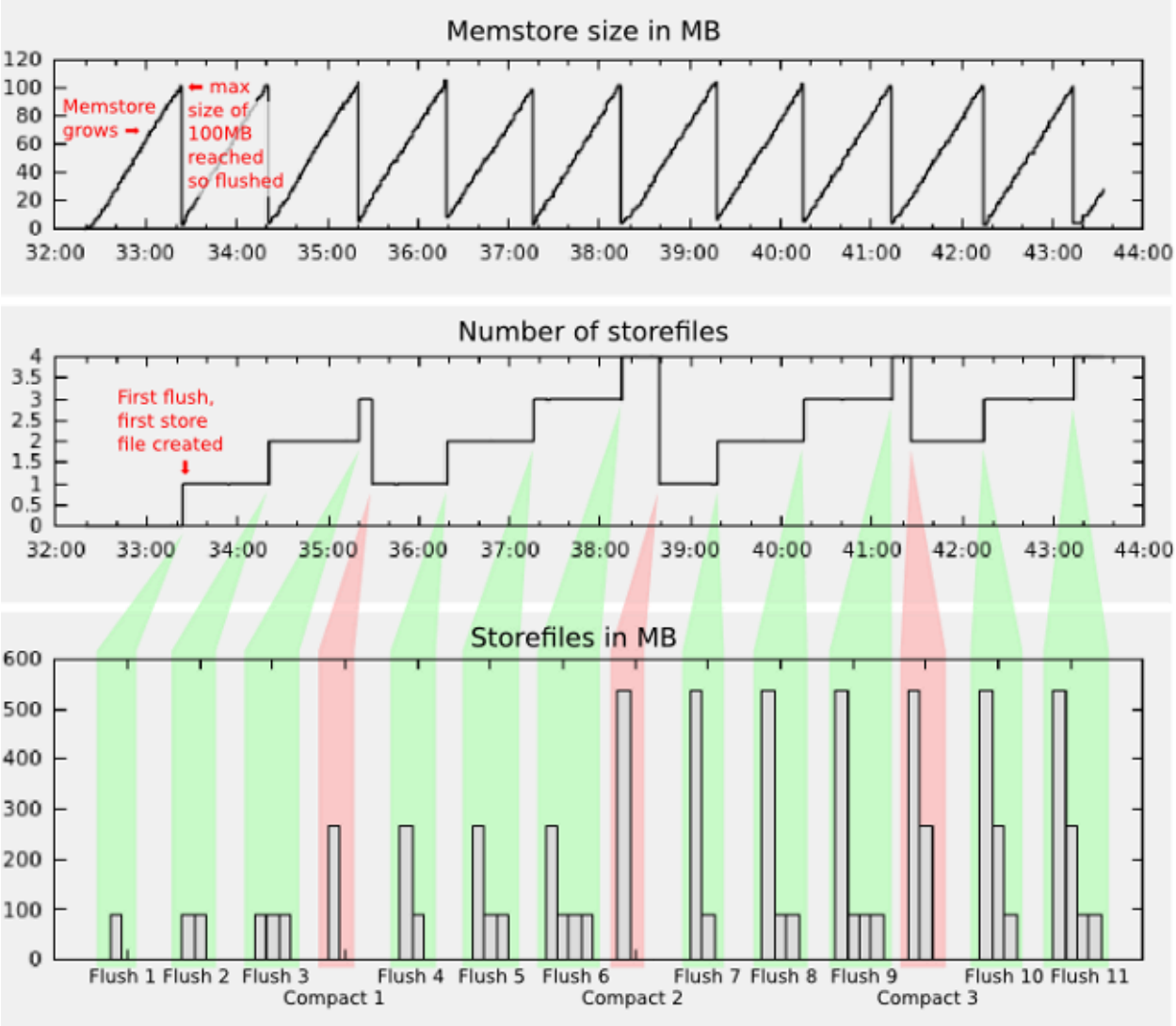


Figure 1: Illustration of store files with threshold = 3 and compaction ratio = 1.2 (default and our setting too) from http://www.ngdata.com/visualizing-hbase-flushes-and-compactions/

The resulted file layout is actually very similar to fixed-window tier compaction mentioned below, except we have no fine control of the trunk size or tiers. And the upper bound is determined by the major compaction interval. Major compaction compacts all files into one file.

We have a use case here that the write access pattern is mainly sequential writes by the time of data arrival to the backend and the read access pattern is mainly time-range scans of certain column families. Most of the scans are based on look-back windows.  The store file layout doesn't allow us to fully leverage the scan api feature to skip store files with data out of the time range.

We want to build date-based tiered compaction for the following benefits:

- Better granularity beyond major compaction intervals for efficient and consistent timespan-based scans
- Reduce IO cost of compactions
- Efficient data retention management

# Compaction Strategy

The cassandra design is explained in https://labs.spotify.com/2014/12/18/date-tiered-compaction/. We compute exponential time windows between Unix epoch and now.These windows don't slide with the passage of time. Instead, as time passes, new time windows appear and old ones get merged into exponentially larger windows, as shown in figure 2.

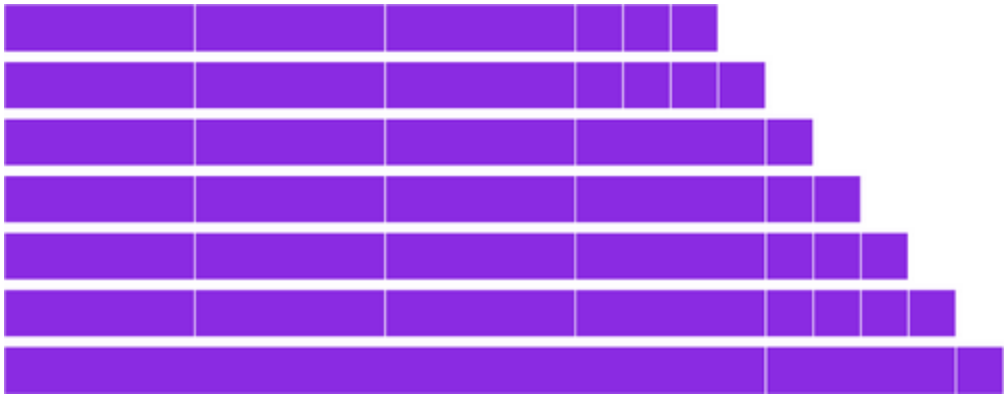

Figure 2. base window = 1 hour, windows per tier = 4

The current point in time is always located in the latest time window.The moment 4 windows of one tier get accompanied by yet another same-sized window, the aforementioned group of windows merge into one. This can have a domino effect as seen in the bottom lane of the image, where a fifth 1-hour window triggered the creation of a fifth 4-hour window, triggering the creation of a 16-hour window.

The sizes of windows are defined as purely exponential to simplify implementation and tuning, instead of using hour, day, month or year time units. Therefore there is no need to consider

calendar boundaries or sliding boundaries. However, when we tune the configuration, we have to consider the read access pattern so minimal disk seek is achieved for typical access for hour, day, month intervals.

We need to configure the following:

- base window: The smallest time window for first tier.
- windows per tier: The scale factor of the window sizes from a tier to the next.
- max storefile age: how old does it have to be before we stop compacting. This is the biggest tier as parted of the tier design.
- incoming window threshold: how many files in the incoming window before we compact to the first tier.

Pros of more tiers:

- Better granularity. Smaller data range and fewer files for most common scans.

Cons of more tiers:

- Disk seek and store file merge cost for GET and scans.
- The # of tiers is the # of compaction every key-value pair goes through.

## Questions and solutions

1. Can we make sure we have at least one file per tier for optimal scan performance?
   The algorithm will guarantee there is at least one window allocated per tier. But if there is no data written during this time, there is no file. User need to look at data write pattern to choose the right base window size.

2. How do we deal with data out of the window boundaries?
   We might have late-arriving data, either by replication lag or user specification, and bulk loaded data. The store files could have overlapping min-max time intervals, which will interfere with the ordering and determination of tiers.

   We use max timestamp to determine the order of files and the compaction window to optimize for the common scenario that user never write future data. Files don't have to follow any particular timespan size to determine which tier they belong to. Their tier numbers are determined by the max timestamp. Unusually large files on a tier make scan performance suboptimal but it will get better when time passes. If user don't write future data, the worst scenario is that file on the lower tier have long tails instead of the data goes to higher tier. The additional cost of long tail is the cost to scan newer and smaller files. Given the tiered design, we only need to scan additional data at most the tail size + current window size.

   This compaction policy is not recommended for use cases that need to write future timestamp for business logic. It will fall back as exploring compaction.

3. How do we guarantee data consistency?
   As explained by Enis Soztutar: “If two puts happen with the same timestamp, we are ordering them using the sequenceId so that the "latest" one is returned always. This allows the user to override a previously set value for example in some cases. The problem with non-contiguous compactions is that, we do not keep the seqids of cells forever. After some time, we remove per-cell seqIds and only keep 1 sequenceId per hfile. Thus if we end up with two different puts having different seqIds in files, but with same timestamp, then allowing non-contiguous compactions may break the ordering.”

   I added a tweak to the original design: scan the store files from the oldest to the newest based on sequence id. If the current file has a maxTimestamp older than last known maximum, treat this file as it carries the last known maximum. This way both seqId and timestamp are in the same order. If files carry the same maxTimestamps, they are ordered by seqId.

   Here is an example: I have (seqId, timestamp) for a list of files that are already ordered by seqId from the oldest to the newest:
   1,0), (2, 13), (3,3), (4,10), (5,11), (6,1), (7,2), (8,12), (9,14), (10,15)
   After the scan and update , I have:
   (1,0), (2, 13), (3,13), (4,13), (5,13), (6,13), (7,13), (8,13), (9,14), (10,15)

   I then reverse the list so they are ordered by seqId and maxTimestamp in descending order and build the time windows the same way as original design. The tweak will put all the out-of-order data into the same compaction windows, guaranteeing contiguous compaction based on sequence id. Scan performance will be worse than the original design because now the compacted store file will have to include any files in between. Two offenders I considered are: 1. seqId and ts are in completely opposite orders. 2. Bulk load files carry -1 as seqId when user explicitly turn off "hbase.mapreduce.bulkload.assign.sequenceNumbers". They will fall back to exploring compaction.

4. How do we handle bulk-load files gracefully?
   By default, bulk load files gets assigned the sequence Id at the time of load which will likely trigger compaction of files between maxTimestamp of the bulk load files to the creation time. If "hbase.mapreduce.bulkload.assign.sequenceNumbers"
   is explicitly turned off, bulk-load file will be treated as carrying -1 for seqId, which will trigger a compaction with an impact similar to a major compaction. So we don’t recommend it.

   Time-series data that are loaded periodically with minimal time range overlap will perform perfectly in this case with base window set to cover the interval. Some users may have occasional bulkload data that could be out of proportion of the files on the same tiers and they will need to pay some scan performance penalty. As time passes, they move to higher

tier, the penalty will diminish. If the files are too big, major compaction is more desirable in this case.

# Trigger Mechanism

Currently compaction is triggered by the number of store files and file size ratio in CompactionPolicy.needCompaction(). CompactionChecker keeps calling this check routinely by default every 10  seconds x hbase.server.compactchecker.interval.multiplier(4).

We will compute the tier windows bottom up. In the incoming window, we only compact if the flush  file count exceeds incoming window min to avoid re-compaction. For other tiers, we apply the exploring compaction using a small file count which is configured as "hbase.hstore.compaction.min".

## Questions and solutions:

5. Currently scan API opens all files to check min-max time range. Should we do some optimization by picking only the relevant files?
   No. The timestamp range is currently persisted in meta data and get loaded into memory every time an HStore is opened.
6. How to avoid all the regions hitting the time mark of compacting for a large tier?
   Use PressureAwareCompactionThroughputController.
   http://www.cloudera.com/content/www/en-us/documentation/enterprise/latest/topics/admin_hbase_compaction_throughput_configure.html
   Notes:
   a. Making compaction longer could lock the region for region split.
   b. Compaction pressure is determined by (storefileCount - minFilesToCompact) / (blockingFileCount - minFilesToCompact). So blockingFileCount setting is important.

# TTL

TTL can take advantage of tiered compaction by drop the whole store files that has oldmax cell timestamp, which is already done.

# Old files Before New Compaction Policy Is In Place

We will not split older store files which are created before the new compaction in place. They will gradually be less frequently accessed and eventually expire. ExploringCompactionPolicy per bucket will prevent continuous re-compaction of the big file with other small files.

# Multiple Output for Date Tiered Compaction

When we compact, we can output multiple files along the current window boundaries. There are two use cases:

1. Major compaction: We want to output date tiered store files.
2. Bulk load files and the old file generated by major compaction before upgrading to DTCP.

Pros:
1. Restore locality, process versioning, updates and deletes while maintaining the tiered layout.
2. The best way to fix a skewed layout.

A few design decisions here:

1. We only want to output the files along all windows for major compaction. We also want to maintain the window size of highest tier for older files. Once a file is old enough to be out of the range we do tiered minor compaction, we don't compact them any further. So they retain the same timespan as they were compacted last time, which is the window size of the highest tier. Major compaction will touch these files and we want to maintain the same layout.

2. For minor compaction, we don't want to output too many files, which will remain around because of current restriction of contiguous compaction by seq id. I will only output two files if all the files in the windows are being combined, one for the data within window and the other for the out-of-window tail. If there is any file in the window excluded from compaction, only one file will be output from compaction. When the windows are promoted, the situation of out of order data will gradually improve.

3. We have to pass the boundaries with the list of store file as a complete time snapshot instead of two separate calls because window layout is determined by the time the computation is called. So we will need new type of compaction request.

4. we cannot safely assign distinct seq id on the output unless we have enough space between 0 and HRegion.sequenceId. This is not only for Date Tiered Compaction but also for Stripe Compaction. Going into negative breaks the assumption of sequence id being non negative except for bulk load files with assigning seg id turned off. One way is to bump up HRegion.sequenceId to max output count we need for compaction every time we request compaction so that the future flushes will carry larger seq id when we are figuring out whether we need to compact.  Another way is to force every flush/bulk load to set HRegion.sequenceId to max(HRegion.sequenceId + 1, max output count). I prefer the second way. But either way requires quite some major changes and the benefit is limited. We know correctness will be

guaranteed among output files for stripe and date tiered compaction because there is either no dup key or no dup timestamp among different files.

5. Since we will assign the same seq id for all output files, we need to sort by maxTimestamp subsequently. Right now all compaction policy gets the files sorted for StoreFileManager which sort by seq id and other criteria. I will use this order for DTCP only, to avoid impacting other compaction policies.

5. We need some cleanup of current design of StoreEngine and CompactionPolicy.

# Region Split and Merge

During a region split, RegionServer splits the store files, in the sense that it creates two reference files per store file in the parent region. Those reference files will point to the parent regions files. After the split, meta and HDFS will still contain references to the parent region. Those references will be removed when compactions in daughter regions rewrite the data files. Garbage collection tasks in the master periodically checks whether the daughter regions still refer to parents files.  If not, the parent region will be removed.

Region merge is the other way around. The only concern is when store files from both regions fall into the same compaction window, it will trigger compaction. In the worst case scenario, all the store files within max_age will have to be compacted, similar to a major compaction. Throttling will help us to avoid IO spikes and contention.

Changing from the default exploring compaction policy to tiered compaction policy won't change the internal compaction process and won't break the existing logic.

# Configuration

## Tiered Compaction Specific Configuration

We will be able to set the configuration in hbase-site.xml per node as in following table 1. We will be able to also override the settings per-table/family by using hbase shell "alter" commands.

It is recommended to only turn on Dated Tied compaction for user table by using hbase shell instead of turning it on cluster-wide, which will put meta table on date tiered compaction and impact random lookup and writes.

| Configuration Key | Default | Note |
|---|---|---|

| | | |
|---|---|---|
| hbase.hstore.compaction.date.tiered.max.storefile.age.millis | Long.MAX_VALUE | Files with max-timestamp smaller than this will no longer be promoted to the next tier to be combined into large sizes. |
| hbase.hstore.compaction.date.tiered.base.window.millis | 2160000 | base window size in milliseconds. 6 hours. |
| hbase.hstore.compaction.date.tiered.windows.per.tier | 4 | Number of windows per tier. |
| hbase.hstore.compaction.date.tiered.incoming.window.min | 6 | Minimal number of files to compact in the incoming window. Set it to expected number of files in the window to avoid wasteful compaction. |
| hbase.hstore.compaction.date.tiered.window.policy.class | org.apache.hadoop.hbase.regionserver.compactions.ExploringCompactionPolicy | The policy to select store files within the same time window. It doesn't apply to the incoming window. |
| hbase.hstore.compaction.ratio | 1.2 | Ratio to be applied per window. |
| hbase.hstore.compaction.min | 6 | Per window. Keep this number small to reduce the total file count. Recommend to set to 2. |
| hbase.hstore.compaction.max | 12 | Per window. If want to trigger major compaction, recommend to set to larger numbers as hbase.hstore.blockingStoreFiles. |
| hbase.hstore.compaction.min.size | 134217728 | Per window. |
| hbase.hstore.compaction.max.size | Long.MAX_VALUE | Per window. |
| hbase.hstore.compaction.date.tiered.single.output.for.minor.compaction | true | Set it to false if you expect long tails |

Table 1

# Tiered Compaction Independent Configuration

Table 2 shows the recommended settings for the existing configuration to use with tiered compaction:

| Configuration Key | Recommended Value | Note |
|---|---|---|
| hbase.regionserver.thread.compaction.throttle | Size of compaction for higher tier | To avoid IO spikes when all the RS starts compacting |
| hbase.regionserver.throughput.controller | org.apache.hadoop.hbase.regionserver.compactions.PressureAwareCompactionThroughputController | |

| | | |
|---|---|---|
| hbase.hstore.compaction.throughput.higher.bound | | |
| hbase.hstore.compaction.throughput.lower.bound: | | |
| hbase.hstore.blockingStoreFiles | Set it generously so flushing is not blocked and large compaction is always throttled | Max store file count before we block memstore flush.<br><br>Compaction pressure is determined by (storefileCount - minFilesToCompact) / (blockingFileCount - minFilesToCompact). So blockingFileCount setting is important. If compaction pressure > 1, there won't be any limit. |
| hbase.mapreduce.bulkload.assign.sequenceNumbers | true(default) | keep the default to optimize compaction/scan performance for bulk load data |

Table 2