Context

Following the introduction of the cell timestamp based priority logic for bucket cache detailed in this design doc, we should aim for supporting custom based priority logic for bucket cache.

This is a major improvement for use cases where a huge portion of the dataset got ingested together at a single point in time, yet the dataset has its own time based tracking data stored as a user defined cell, or even contained within portions of specific cell values or as part of the row key definition.

At Cloudera, we have at least one major customer operating a single HBase cluster with a dataset of a few petabytes. This deployment uses cloud storage for the underlying filesystem and benefits from a file based bucket cache over local SSD disks for faster read access. It relies heavily on caching for achieving its applications SLAs, and has prefetch, cache on read/write/compaction already enabled. To minimise cache warmup time (prefetch) on eventual restarts/table enabling, it also uses the bucket cache persistent cache capability.

The current total cache capacity for the cluster is around 900TB. The dataset is time based, within each row relating to a specific date time, with rows under a certain age rarely accessed, which make those potential candidates to be kept off cache.

As an example (the time windows are not necessarily accurate in this description):

- They need to keep all records for the last 10 years in the cache;
- Each row in their dataset has a datetime cell with values spanning over the last 20 years;
- The date of interest is also part of the row key.
- The first ingestion happened 5 years ago. This means all data older than 5 years has roughly the same timestamp of 5 years ago.

If we rely on the cell timestamp based priority for bucket cache, simply setting the hot age to 10 years would deem the whole dataset as hot.

The already existing LRU priority would eventually succeed in keeping the most recent data in cache, as these are the most accessed, but the time for such cache state to be achieved is not well predictable, and additional ingestion of “old” data, as well as major compaction, splits, merges, disabling/enabling, region moves, and region server restarts could all reset the cache state and potentially lead to older data being cached, instead of the most recent data that should have higher priority,

This could be properly sorted by a custom, pluggable implementation for providing the value to be used in the priority comparison.

Dependencies

Similarly to what has been described in HBASE-28463 and documented in the original design doc, this would require some grouping of cells by the specified value into specific files. For the simple cell timestamp based tiering, this is already implemented by the Date Tiered Compaction feature.

A custom comparison logic would require an additional compaction implementation for producing files grouped by whatever value is used by the pluggable implementation. This can be done as an extension of the Date Tiered Compaction.

Some scenarios may not have the value self contained in all cells of a given row, for example, when using a specific qualifier value, so *Compactor* should define a *decorateCells* method to be overridden by priority strategies that need to propagate the comparison value to all cells (using tags, for example).

With the above implemented, minor adjustments to the framework proposed in HBASE-28465 would be needed to integrate the custom priority logic.

Scope

The implementation should be able to work seamlessly on pre-existing data.

Should be configurable on column family level. After enabled, major compaction should be able to apply the given tiering logic over existing data.

This should define the pluggable framework for “tiered compactions", as well as a built-in implementation that uses a configurable qualifier value as the one for the age to be used in the priority comparison.

Initially, should be implemented at the compaction and bucket cache components only. Flushes could be covered in a posterior phase.

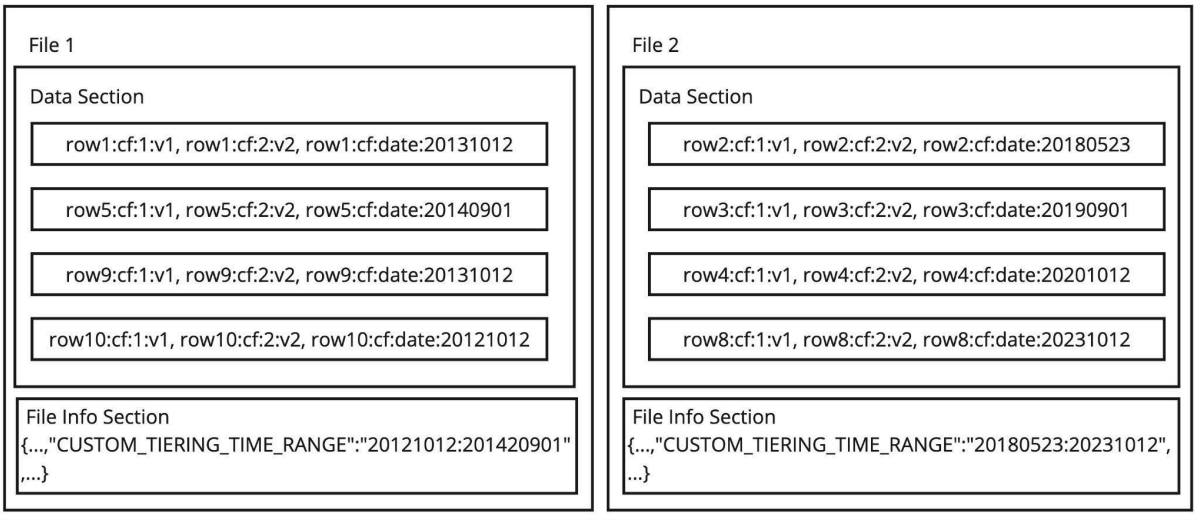
Differently from the existing Date Tiered Compaction, this doesn't yield multiple tiers or files, but rather provides two tiers based on a configurable “cut-off” age. All rows with the cell tiering value older than this “cut-off” age would be placed together in an “old” tier, whilst younger rows would go to a separate, “young” tier file.

High Level Requirements

1) Extend DateTieredCompactor with a CustomTieredCompactor that uses a pluggable value provider for extracting the value to be used for comparison in this tiered compaction

2) Define a built-in value provider that uses a configurable qualifier value for comparison in the tiered compaction. Using a qualifier value for grouping data may require propagating this value to all other cells within the same row key:
    a) We can use cell tags to append the *tiering value* to all other cells within a row. This might be needed by the multi file writer, as cells are appended individually, and the multi file writer must know to which tier file the appended cell must be forwarded.
    b) Finding the *tiering value* for each row requires going back and forward the cells of a given row. This is needed in order to figure out the *tiering value* before starting writing the row to new files.
    c) If a given row doesn't have the *tiering value,* just treat it as high priority and *tag* its cells to be tiered within the highest priority group.

3) The *tiering boundary values* (min/max) of each tiering group should be persisted to the related store file as a KV pair in the file info portion of the file. For example, considering the *tiering value* is actually defined by the "cf:date":



See file format for more information.

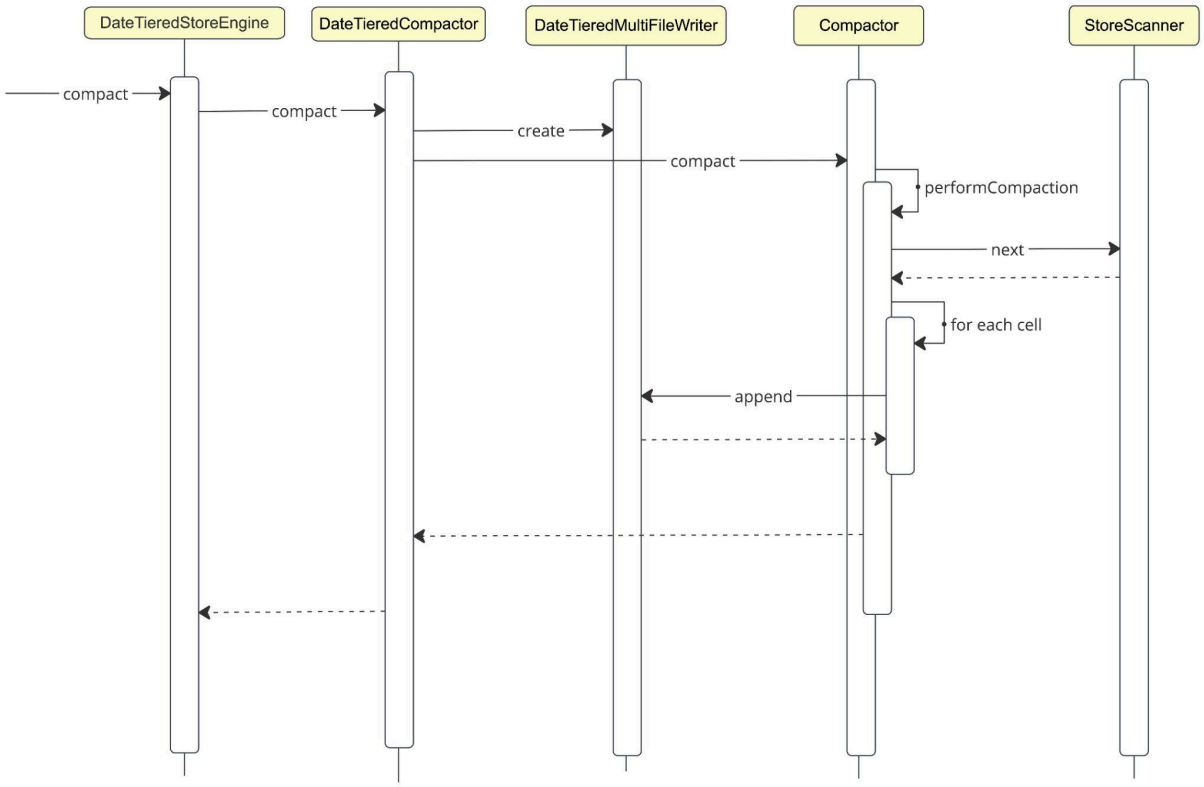
4) Use the *tiering boundary values* recorded for each file to classify the file tiering group for caching (either HOT or COLD).

Implementation Details

Date Tiered Compaction Modifications

Date Tiered Compaction

Date Tiered Compaction calculates windows of time for the different tiers files should be written to during compaction. It then creates multiple file writers for each of these windows. These writers are all encapsulated by DateTieredMultifileWriter, which indexes the writers with the timestamp of each window. Then, when Compactor sends a cell to the DateTieredMultiFileWriter, the specific writer for that cell timestamp is chosen:

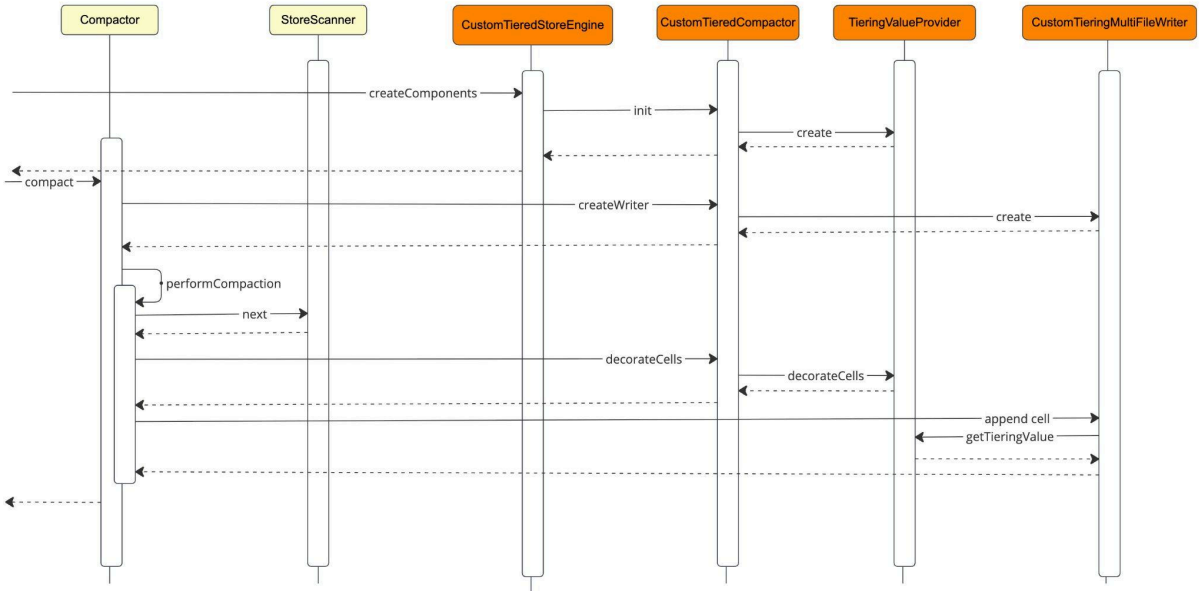


We need to extend the way DateTieredMultifileWriter indexes writers, ideally allowing pluggable logic. We can introduce a tieringFunction lambda to be specified by the caller and this would be used inside DateTieredMultiFileWriter.append. DateTieredMultifileWriter is created in DateTieredCompactor. We can define a protected method for creating the multi file writer in DateTieredCompactor, which by default creates the DateTieredMultiFileWriter but can be overwritten in extensions of DateTieredCompactor.

Currently main compaction work happens inside Compactor.performCompaction. It uses StoreScanner for iterating through all the selected files content. It relies on the StoreScanner.next(List<Cell>, ScannerContext) to read all cells from each row at once. It then iterates through the batch of cells returned for the row, and sends each cell to the writer (in this case, the DateTieredMultiFileWriter).

CustomTieredCompactor

For custom priority, the value used for comparison is arbitrary. It may or may not be present at all cells already, so we need to provide a mechanism to propagate the value in case of the latter.
We can define a no-op, decorateCells method on Compactor class, so that it can be extended in a CustomTieredCompactor class (extending DateTieredCompactor), to implement the tiering value propagation logic in its decorateCells override, if needed.



The pluggable logic for finding the tiering value and decorate cells if needed should be provided as implementations of the TieringValueProvider.
We also need the CustomTieringMultiFileWriter extension of the DateTieredMultifileWriter, so that the tiering time range can be appended in the file info section, according to the custom tiering logic.

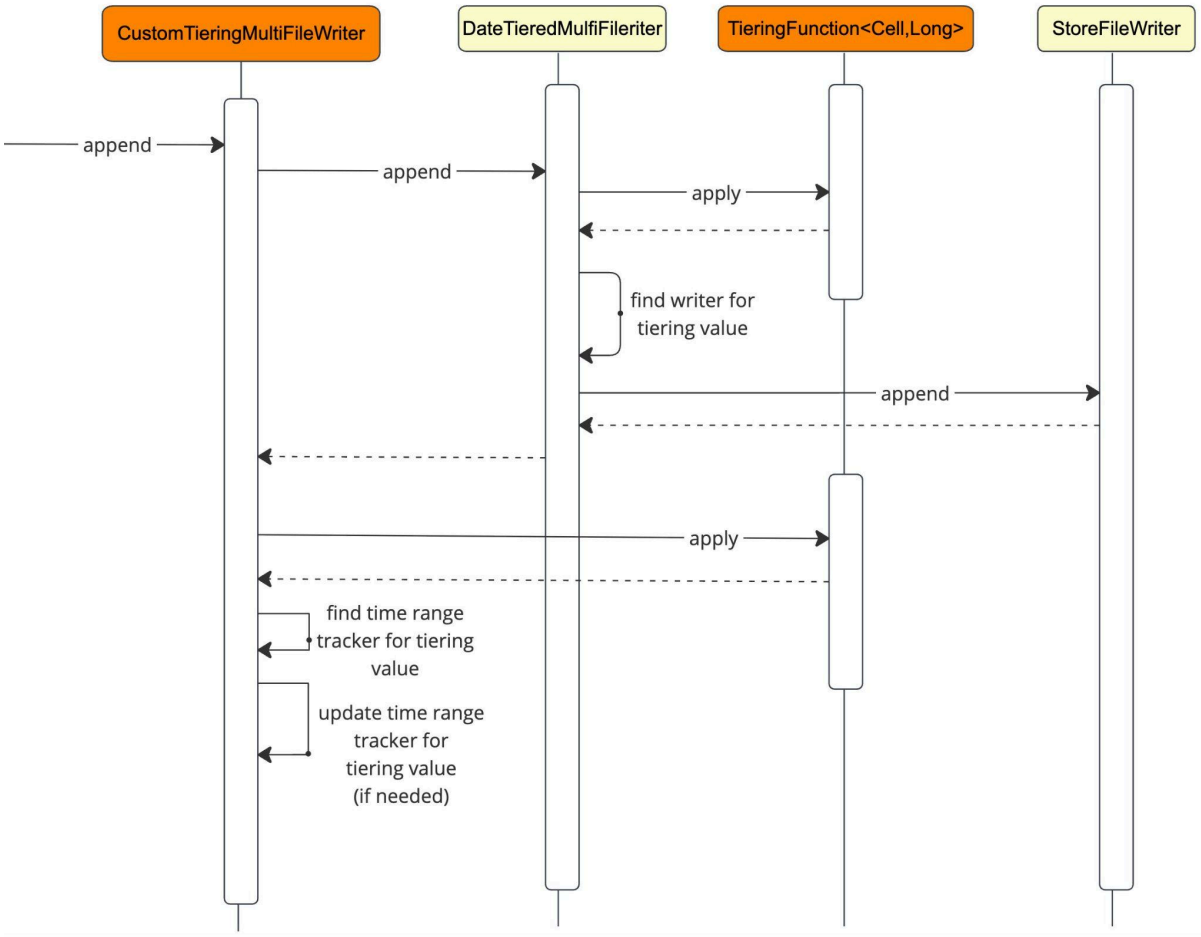
CustomTieringMultiFileWriter

CustomTieringMultiFileWriter wraps two separate store file writers, one for each tier output file. Based on the configured age limits, it decides to which writer the appended cell should be sent, using the TieringValueProvider implementation to decide on the comparison value. CustomTieringMultiFileWriter also implements time range tracker logic in the append method, in order to record the min/max values in each tier file. The related TimeRangeTracker with the min/max times of each file is needed at two different points:

- HFileWriterImpl: when caching from compaction writes is enabled, HFileWriterImpl needs to pass the TimeRangeTracker to the cache implementation, which will then decide based on these min/max times if the block should be cached. CustomTieringMultiFileWriter updates the TimeRangeTracker of each of its wrapped HFileWriterImpl in the append method.

- CustomTieredCompactionPolicy: This is where the date boundary is calculated, at the beginning of a compaction. For each file in the list of files selected to be compacted, it reads the TimeRangeTracker containing min/max values for the custom cell, finds out the min and max across all files, then calculates the boundary by comparing the min/max against (current date - hbase.hstore.compaction.date.tiered.custom.age.limit.millis).

CustomTieringMultiFileWriter writes the TimeRangeTracker of each of its wrapped writers in the commitWriters method. The TimeRangeTracker is written to the file info portion of the file.



CustomTieringMultiFileWriter uses the boundary calculated in CustomTieredCompactionPolicy to determine the number of writers it needs to wrap (either one or two) and how to redirect cells to each writer. When the min is less than (current date - hbase.hstore.compaction.date.tiered.custom.age.limit.millis), and max is greater than (current date - hbase.hstore.compaction.date.tiered.custom.age.limit.millis), then compaction will produce two files. Note that this approach produces at most two files, as it would only lead to two tiers at most: one containing all rows older (according to the tiering value) than the configured cut-off date and the other containing the rows that are younger than the cut-off age.

The tiers and boundary calculations need to rely on the extra min/max custom tiering values from the file info section. However, this would only be appended in a hfile if either minor or major compaction has run at least once, after the feature was enabled. This means that the first compaction run after enabling the feature would not succeed in separating the rows into

tiers even if the files do contain rows with tiering values older than the cutoff date. A second compaction run would be required to effectively separate the rows between two different tiers.

Custom Tiering Compaction Configurations

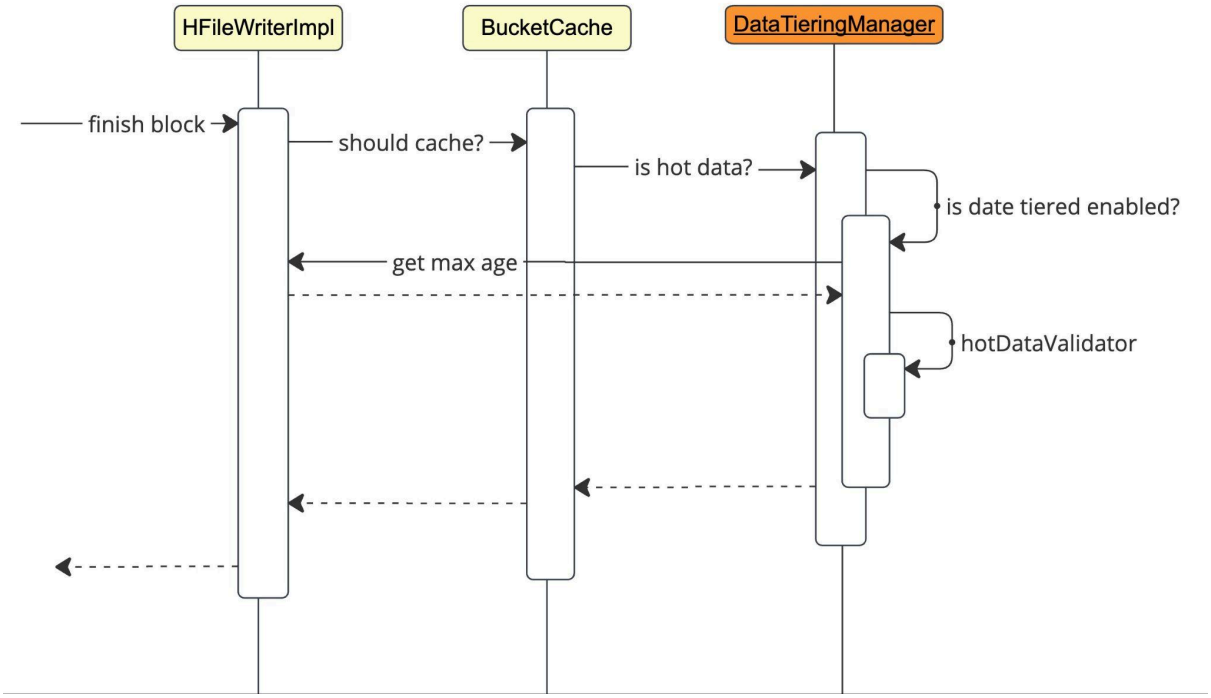
The feature would be enabled at column family level, by setting the "**CustomTieredStoreEngine**" as the value for the "**hbase.hstore.engine.class**".
The age for splitting the custom values between cacheable/non-cacheable is defined by a new configuration property:
"**hbase.hstore.compaction.date.tiered.custom.age.limit.millis**". It should also be set at CF configuration.
Since now there are more than one data tiering type, the **hbase.hstore.datatiering.type** property must also be set to **CUSTOM** in the CF configuration.
With the CUSTOM tiering type, the **TieringValueProvider** custom implementation should be defined via the **hbase.hstore.custom-tiering-value.provider.class** property also in the CF configuration. Custom tiering defines a reference implementation that uses a qualifier based value for the comparison, **CustomCellTieringValueProvider,** which is used by default if CUSTOM tiering type is enabled.
Any implementation of TieringValueProvider is free to have its own additional properties in the CF configuration, and those should be documented accordingly.
The **CustomCellTieringValueProvider** requires a "TIERING_CELL_QUALIFIER" property in the CF configuration, which specifies the column qualifier whose value should be used for the data tiering comparison.

The box below shows a sample command for enabling the CUSTOM tiering using cell qualifier provider on a given table:

```
None
hbase:016:0> alter 'test-val-1', {NAME => 'cf',CONFIGURATION => {
'hbase.hstore.engine.class' =>
'org.apache.hadoop.hbase.regionserver.S3ACustomTieredStoreEngine',
'TIERING_CELL_QUALIFIER' => 'dt',
'hbase.hstore.datatiering.type' => 'CUSTOM',
'hbase.hstore.datatiering.hot.age.millis' => '315576000000'
}
}
```

Cache Priority Tiering Modifications

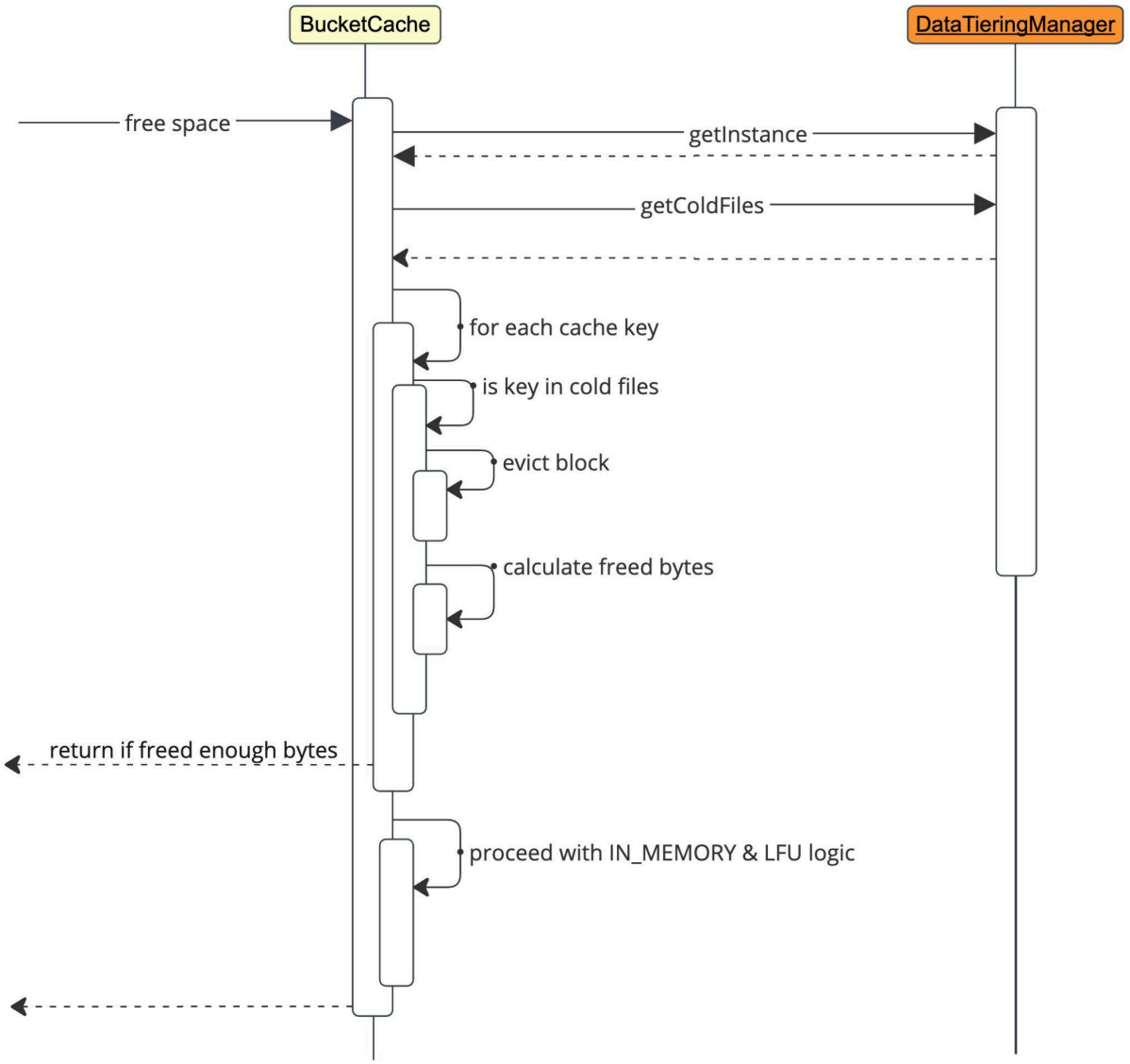
Within HBASE-28466, HBASE-28468 and HBASE-28469, current implementation for time based cache priority defines a singleton DataTieringManager class that is referred from BucketCache and HFileWriterImpl internals to delegate caching tiering logic:



Data Tiered Caching during writes

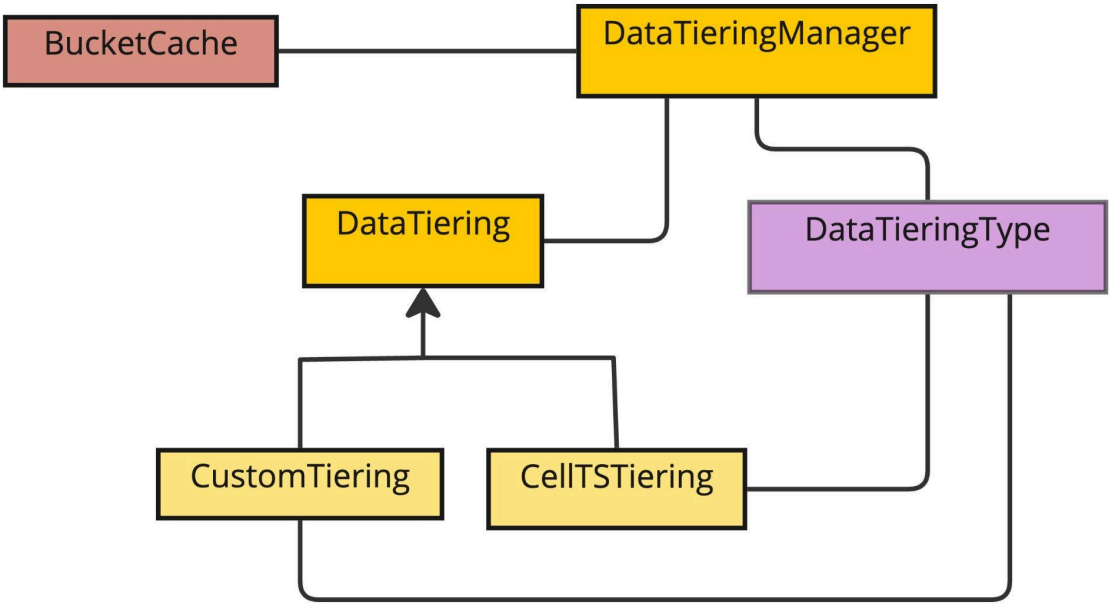
As both cache writing and mass eviction are relying on the DataTieredManager for validating when a block should be considered hot or cold, DataTieringManager allows for configurable implementations of the tiering value, defining two internal strategies:

1. CellTSTiering - Considers the time range defined in the TIMERANGE file info section as the block age.
2. CustomTiering - Considers the time range defined in the CUSTOM_TIERING_TIME_RANGE file info section as the block age.

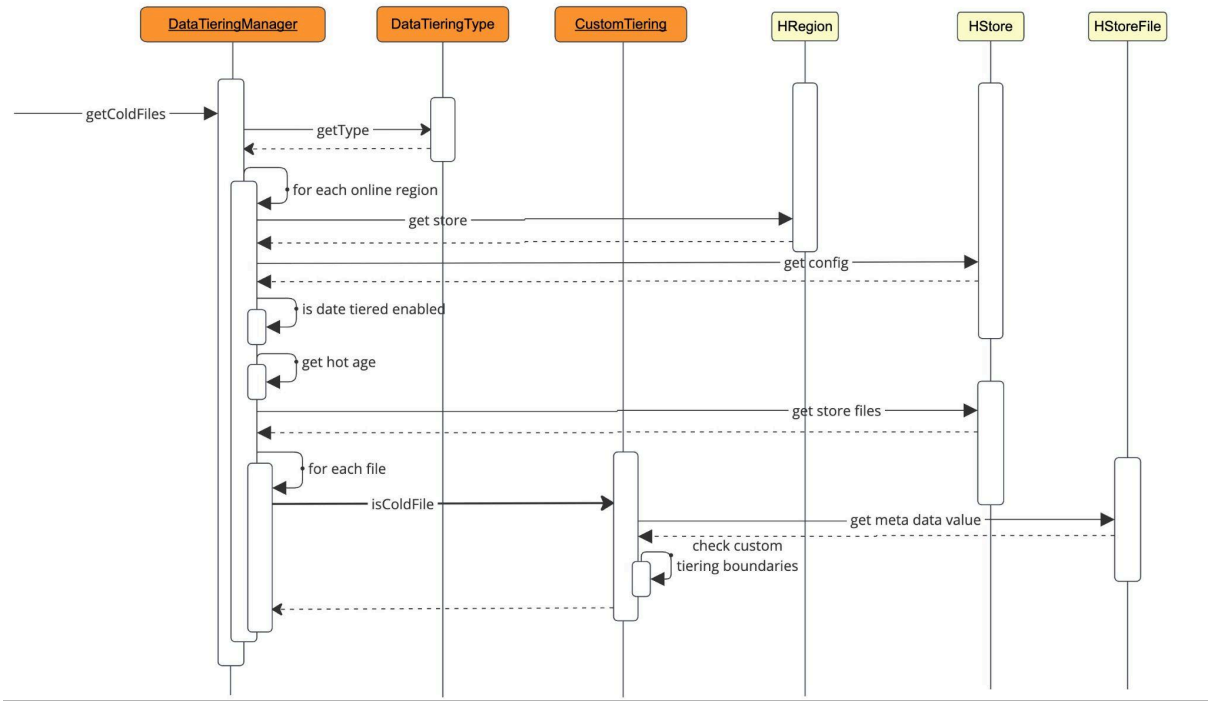


Data Tiered Caching mass evictions flow

For example, the BucketCache free space wouldn't require any code changes, the custom tiering values would be encapsulated within the DataTieringManager.getColdFiles method. Internally, DataTieringManager finds out which is the tiering type implementation and delegates the logic for identifying hot/cold files.



The diagram below pictures the flow when using CustomValueTiering for distinguishing hot/cold files:

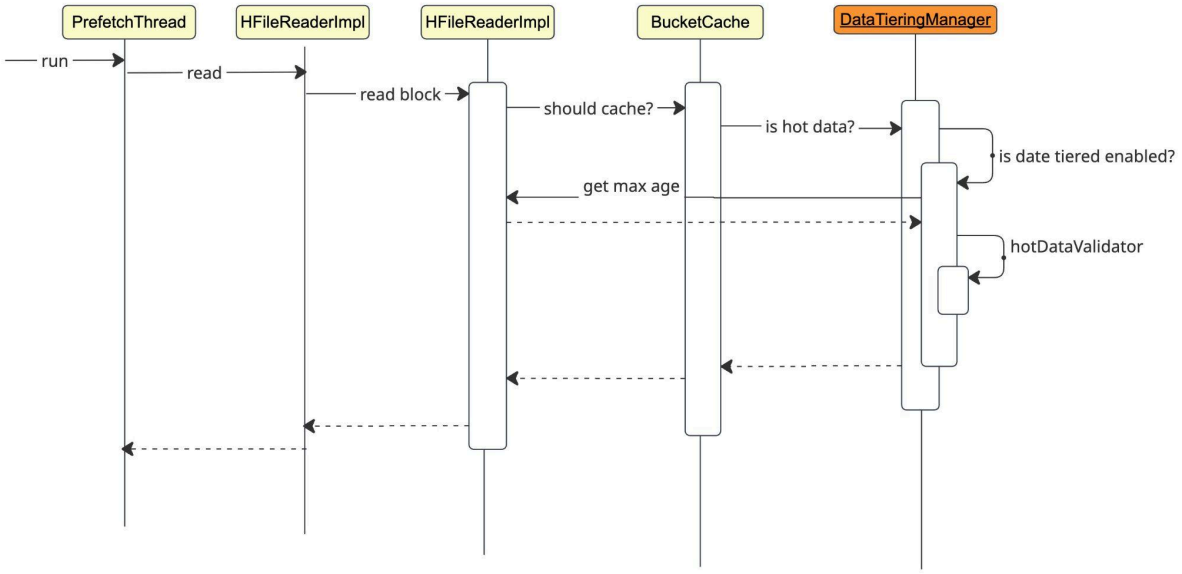


DataTieringManager with Custom Cell Value Tiering

Note that the specific tiering type implementation is defined at the column family configuration level.

Prefetch

Like the eviction process, prefetching utilizes the DataTieringManager to ascertain block age priority, thereby determining which blocks must be cached and which must be bypassed.



Cache on Write

If custom tiering should also be scoped for cache on write, additional changes are required in write path, probably in *HFileWriterImpl* for finding the custom cell value for each row and setting it as tags in each cell of the given row.

High Level Implementation Requirements

1. The additional logic for finding each row's custom cell tiering value shouldn't cause performance impact.
2. Some rows may not contain the custom cell for data tiering. In those cases, current time should be assigned to the existing row's cells.
3. This would require multiple writers, in similar fashion currently implemented by Date Tiered Compaction