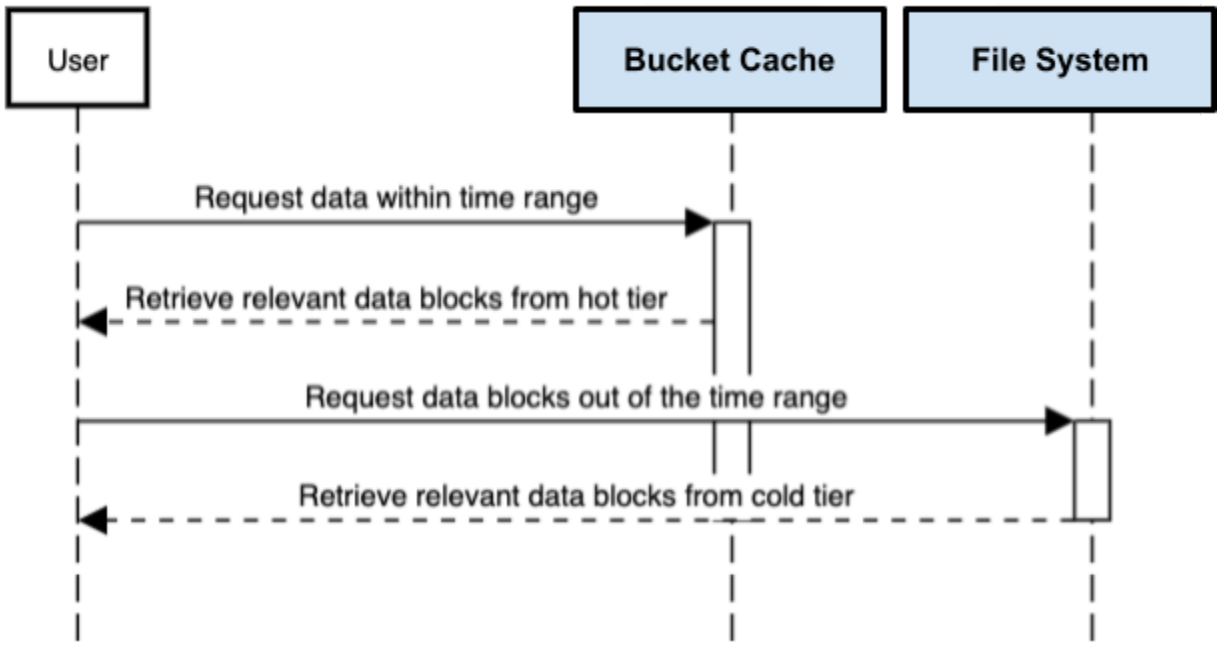
HBase: Time Based Priority for BucketCache

1. Introduction

In this document, we will outline the design for a time-based priority caching, where data within a specified time range will be given higher priority to remain in the cache, while older data will be more likely to get evicted.

1.1 Purpose

The purpose of this feature is to optimize storage efficiency and access performance by segregating data based on its recency. By keeping recent data in the cache and evicting older data, the system aims to provide a more flexible control over the cache allocation and eviction logic via configuration, allowing for defining time priorities for cached data. The need for a more extensive cache allocation mechanism becomes even more critical on HBase deployments where cache access reflects on significant performance gains, such as when using cloud storage as the underlying file system.



1.2. Input Section

Users should be able to set the following configuration parameters to enable the feature and set data age priorities:

- **Enabling the feature:** Configuration to enable the time-based data tiering: The user needs to set the following parameter **DATATIERING** to **"TIME_RANGE"** (default is **NONE**) to enable the data-tiering at the table level or at the column family.

- **Hot Data Age:** Specify the time duration for which the data is considered as hot and should have highest priority to be kept in the cache. This parameter "**hbase.hstore.datatiering.hot.age.millis**" needs to be set at the table level to implement the time-based data-tiering for all the column families of the table or this parameter can also be set at the column-family level to make the feature effective for the column family.

2. System Architecture

2.1 Components Impacted

- **Compaction:** Date tiered compaction that helps in organizing data into files based on the provided user time range and compaction strategy.
- **Eviction Policy:** Add new implementation/logic that considers data age according to the specified configuration when deciding data to be kept/evicted.
- Prefetch Logic: Should also be modified to consider this "age priority" configuration when considering blocks to be cached.

2.2 Workflow

- Data Ingestion:
    - Data is automatically flushed from the memstore to the cache based on a predefined configuration (cacheOnWrite). Hence the recent data will always stay in the cache.
- Compaction:

    - When the date-tiered compaction is enabled, the recent data within the specified time range is compacted into smaller files, while older data is grouped into larger files.
- Eviction:
    - The eviction policy determines which data blocks in the cache do not belong to the specified time range.
    - Blocks containing data outside the user-provided time range are evicted from the cache.
- Prefetching:
    - When accessing data within the specified time range, the prefetch logic ensures that only relevant files (based on the time range specified in the header of HFile , split by the compaction) are fetched from the cloud storage for caching
    - Files outside the specified time range are not prefetched, reducing unnecessary data transfer and cache usage.

3. Detailed Design

3.1 Overall Approach of Time-Based Data Tiering

The user defines a hot data age limit at either the table level or column family level or at the system-level configuration parameter. This age defines the duration for which the data is considered hot data and is given preference to be stored in the cache. We use the same configuration parameter as defined for date-tiered-compaction for this purpose.

The HStoreFile(the Store data file) already maintains the time-range information in the metadata block of the file. This time range represents the cell with the minimum timestamp and the cell with maximum timestamp that are stored in the file.
`org.apache.hadoop.hbase.regionserver.HStoreFile#getMinimumTimestamp`
`org.apache.hadoop.hbase.regionserver.HStoreFile#getMaximumTimestamp`

We can use the maxTimestamp, to achieve data tiering within the bucket cache. We compare this information with the hot-data age and the current timestamp to determine if the blocks of the file need to be evicted.

The data-hotness decision can be based on the following formula.

```
if ((currentTime - maxTimestamp) < hotDataAge)
      FileDataIsHot = true
```

To decide if the block is hot or cold, we need the following information:
1. Hot Data Age: This is defined per column family or per table.
2. Maximum Timestamp: The maximum timestamp associated with the file.

We maintain the reference to the map containing the online regions in the region servers. We traverse this list of online regions to reach the particular region and its associated files. We then traverse through these files to get to the required file and its metadata. This metadata is subsequently used for making the eviction decisions.

3.3 Caching and Eviction Logic

Caching of the block

CacheOnWrite: If cacheOnWrite is enabled, blocks are cached into the bucket cache when they are first written since the recently written data can be termed as hot data. Once the block is in cache, the hotness of the data would reduce with time.
CacheOnRead: If cacheOnRead is enabled, files accessed during scans need to be validated for hotness of data to determine if its blocks need to be cached.

Time Based Eviction Logic

The block eviction decisions are made within the bucket cache when the bucket cache runs out of the available space. Hence, the eviction logic needs access to the file timestamp metadata associated with the blocks.
We plan to use the reference to online regions (map of HRegions) to obtain the required metadata and perform the evictions. If there is still not enough space after the eviction of cold blocks, the existing mechanism of the LFU algorithm is used to evict the hot blocks from the cache.

3.4 Prefetch Logic

During the prefetch, during the creation of the prefetch reader for the file, the metadata block is first read from the file. This metadata block contains the minimum and maximum timestamps. We also detect the hot data age from the configuration associated with the file. We, then, decide whether or not to prefetch the file based on its hotness, allowing for the prefetch to skip entire files without reading its data when the minimum timestamps falls out of the configured age window.

3.5 Compaction

Hbase implements a compaction strategy named DateTieredCompaction strategy that structures store files in date-based tiered layout. This strategy aids in the data tiering since only the files with hot data will need to be cached.

3.6 Feature Enablement

The feature is enabled with the following steps:
Requires global config switch:

```
<property>

<name>hbase.regionserver.datatiering.enable</name>
  <value>true</value>
</property>
```

This feature requires a date-tiered compaction policy for the table/column family to support the time-based priority caching. Hence, we enable the date-tiered compaction along with the time-based priority caching configuration parameters.

```
alter 'usertable', CONFIGURATION =>
{'hbase.hstore.datatiering.type' => 'TIME_RANGE',
'hbase.hstore.datatiering.hot.age.millis' =>
'315360000000', 'hbase.hstore.engine.class' =>
'org.apache.hadoop.hbase.regionserver.S3ADateTieredStor
eEngine', 'hbase.hstore.blockingStoreFiles' => '60',
'hbase.hstore.compaction.min'=>'2',
'hbase.hstore.compaction.max'=>'60'}
```

4. Conclusion

. By adding an optional, configurable data age "awareness" in the compaction, eviction and prefetch logic, we can extend HBase caching capabilities for use cases where most recent data are the most frequent accessed and have the most critical SLA requirements, to increase the likelihood of these most relevant data to be kept in the cache and maintain consistent client read performance.