Mirroring HBase root dir on Ephemeral local SSD Disks when using cloud storage

Overview

For HBase deployments using cloud storage such as S3, we have been relying on file based BucketCache over ephemeral SSD disks provided within the instance types used for our RegionServers (we currently use 1.6TB SSD disk for BucketCache). This allows for similar performance of an equivalent HBase cluster using HDFS over block storage, however at a lower TCO. This approach has some limitations, though:

1. Rolling OS upgrade support - As SSD disks used by BucketCache are ephemeral, OS upgrades that restart the instances will reset the disks and all data on the disks is lost. In our bucket cache configuration, the whole 1.6TB of data needs to be reloaded from S3, causing temporary performance impact on workloads reading this data.
2. Performance impacts on region moves - moving regions to new servers, for example, when adding new nodes, would cause temporary performance impacts on workloads reading these regions data.
3. Additional heap usage by bucket cache - With a fully used 1.6TB ephemeral bucket cache and average hfile block size of 64KB, bucket cache index map would account for around 9GB of permanent heap usage. This limits the ability to use more SSD disk space currently available in new generations of instances.

One possible solution to overcome the limitations highlighted above is to use these SSD disks for the HDFS deployed within, using HDFS storage types and policies features, so that the root dir content could be mirrored at an HDFS location configured with the ALL_SSD policy. The “mirroring” would work as follows:

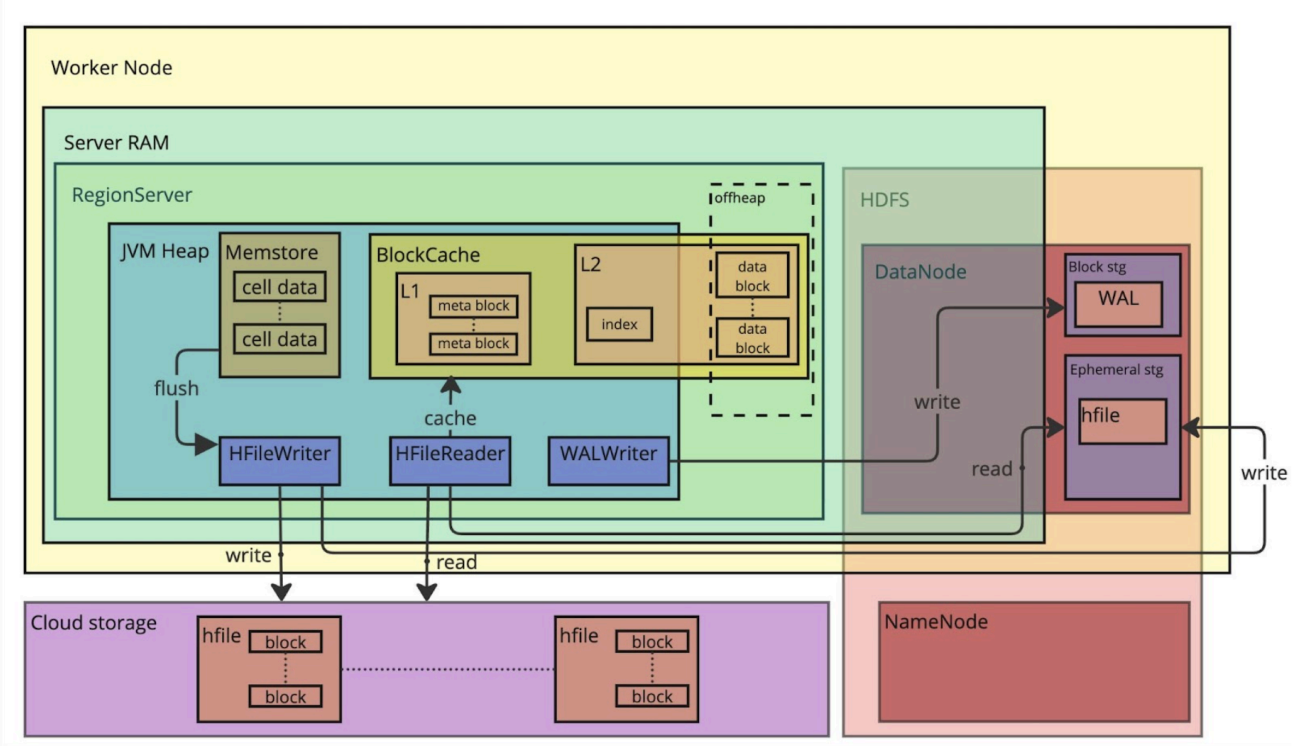
1) At write time, data files are written to both file systems.
2) At reader opening time, if the given file exists in the HDFS SSD directory, the reader should open an input stream on that location. Otherwise, it would open an input stream from the cloud storage location.

Such approach can have the additional benefits over the ephemeral “BucketCache”:

1) Release BucketCache to use offheap memory, providing another layer of caching.
2) Can use larger SSD capacity available with new generation instances without further impacts on RSes’ heap.

Technical Details

An overview of the architecture on the RS deployment can be seen as below:



Additionally to the creation and reading of duplicated files in the mirrored location, the following extra functionality must be provided:

Error Handling and Recovery

When defining whether to create an input stream from HDFS SSD or S3, extra checks need to be done to verify if the file in the HDFS SSD directory doesn't have missing blocks. If the file in HDFS SSD has missing blocks, it should be deleted and the input stream should be open from S3. The deletion of corrupt files during reader opening may still not be sufficient to completely cleanup the SSD directory, as files already compacted or regions split would not be referred anymore. An additional background chore needs to be developed in order to check for corrupt files and delete those accordingly.

During writes, any failure to write a block to the secondary file system should cause the whole file write on this secondary file system to be aborted. The client operation may fail or not, according to configuration.

HDFS SSD files cleanup

There are two types of cleanup that must be done in the HDFS SSD dir:

1. File deletion by HBase internal operations, such as compaction, split, truncate, table deletion;
2. Corrupt files due to missing blocks, as described in the previous section.

For the former, extending the archive cleaner to look for the files selected for deletion also in the additional file system might be sufficient. For the latter, we need to define a new background chore that checks on the hdfs state of the files in the SSD dir, deleting files identified as having missing blocks.

Duplication of pre-existing data (prefetch)

Pre-existing data can be manually copied into the SSD directory with the hdfs distcp tool. We could also implement an hbase builtin tool for that, or maybe a background process for copying files between the root dir and the mirrored location.

How to handle SSD storage at capacity (eviction logic)

An additional chore running on the active Master (or even as a complete separate process) must be implemented to keep monitoring the additional FS usage and potential corrupt files, performing a bulk deletion once a configurable threshold usage is reached. It should use pluggable priority policies for selecting the files to be deleted.

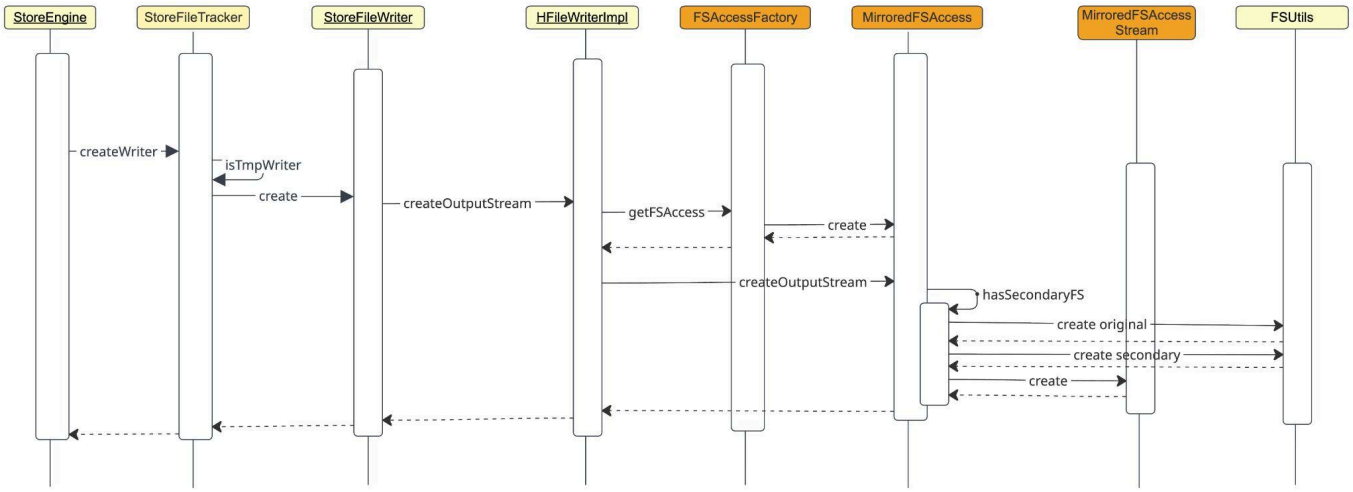
Balancer for SSD storage locality

In order to achieve optimal HDFS data locality (thus, better performance), a new region balancer that accounts for the secondary FS locality should be implemented.

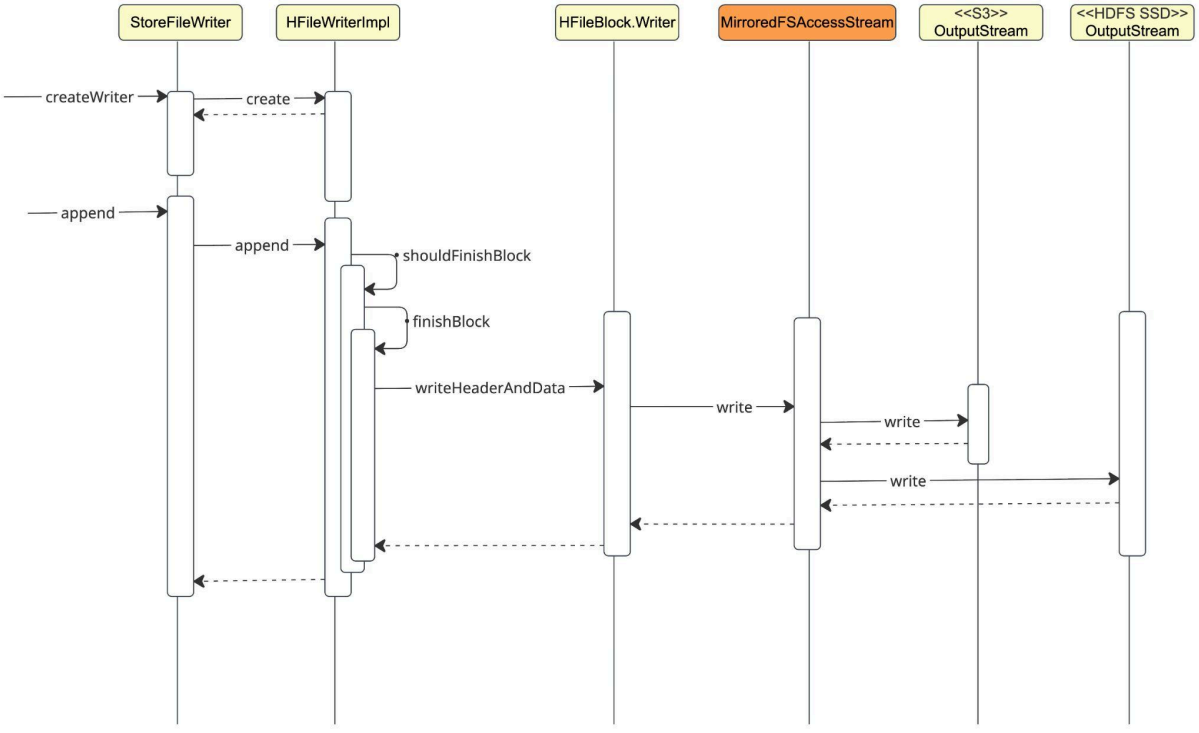
Reading/Writing From/To Mirrored Location

At the file access layer, a new abstraction could be defined in the current logic that creates file system output/input streams, allowing for pluggable behaviour in accessing underlying store files. Rather than creating streams directly, components such as HFileWriterImpl/StoreFileImpl would defer this task to implementations of the FSAccessStrategy interface. The specific strategy to be used should be defined via configuration, and the FSAccessFactory should be used to retrieve the related strategy implementation.
Within this work, we would be providing two implementation strategies: DefaultFSAccess, which simply uses the single file system as defined in the root dir configuration, and the MirroredFSAccess that “mirrors” contents from the root dir in an additional file system.

For the mirrored FS use case, the MirroredFSAccess strategy encapsulates all the knowledge related to create streams to/from the primary and secondary file systems. When creating output streams for writes, the MirroredFSAccess creates the output streams for the two file systems and instantiate a MirroredFSAccessStream, which is an extension of d the current FSDataOutputStream type used, so that it wraps the two actual output streams and delegates all related write calls to both streams.

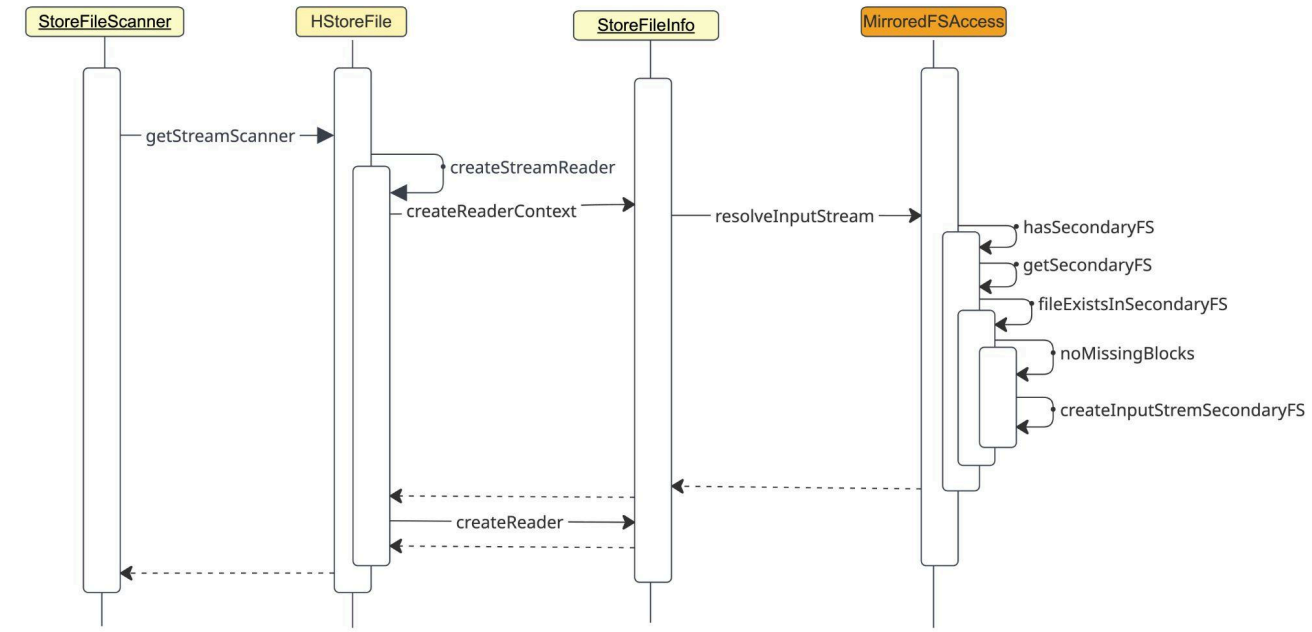


File OutputStream creation sequence



Block write with the proposed MirroredFSAccessStream writing to both cloud storage and HDFS SSD output streams

The same MirroredFSAccess component would also be used at the file read layer, to “proxy” the required input stream accordingly.



File Input Stream creation on the secondary file system when the file is available and has no missing blocks

Key Components

The following sub-sections describe the key components that would need to be modified or created to complete this mirroring root dir feature:

HFileWriterImpl

HFileWriterImpl implements the main logic of writing block data into the output stream (FSDataOutputStream) for the hfile being created.
It defines a static, createOutputStream method that could be modified to delegate the creation of the output stream to the new MirroredFSAccess component. The stream created here will be used later by the HFileWriterImpl instance, as it is passed in this class constructor parameter.
It doesn't create the output stream itself, this stream is passed from upper layers.

MirroredFSAccess (new)

The MirroredFSAccess should implement all logic to create and return streams to callers. It would validate if the additional file system is configured and the directory location, in order to create the related streams. For writes, in the case when an additional file system is to be used, it would create the new, MirroredFSAccessStream that duplicates writes to both file systems. For reads with an additional file system, the

MirroredFSAccess should validate if the file is available in the mirror file system, create an input stream to that file system and return this stream to the caller.
There are different read paths that create input streams:

- HFile.createReader
- HFilePreadReader constructor
- StoreFileInfo.createReaderContext
- ReaderContextBuilder.withFileSystemAndPath

The input stream creation in all four places listed above must be delegated to the MirroredFSAccess, which should implement following logic in the stream creation:

1. If secondary file system is to be used:
    a. Find out the related path in the secondary file system for the file being read;
    b. Create the FileSystem instance for the secondary file system;
    c. If the file exists in the secondary file system:
        i. Validate there's no missing blocks for the file in the secondary file system:
        ii. If there's no missing blocks for the file:
            1. Create an input stream for the file in the secondary file system and return this stream;
        iii. Otherwise:
            1. Delete the file in the secondary file system;
            2. Create the input stream for the file in the original file system and return this stream;
    d. Otherwise:
        i. Create the input stream for the file in the original file system and return this stream;
2. Create the input stream for the file in the original file system and return this stream;

MirroredFSAccessStream (new)

An extension to the FSDataOutputStream currently used for writing blocks, it should be used only when an additional file system is configured. Wraps two output streams, one to the original file system where hbase root dir is configured, the other to the additional configured file system. It overrides the write method, delegating write calls to both streams in order to duplicate the file being written in both file systems.

HFile

HFile currently defines a createReader method that is used by the mapreduce HFileInputFormat for reading hbase files in mapreduce jobs. This method should be modified to delegate the input stream creation to the MirroredFSAccess.

HFilePReadReader

The HFilePReadReader constructor creates an input stream when prefetch on open is enabled, and it should be modified to use the MirroredFSAccess. This is not a critical code path, as the stream would be used solely for caching blocks in the BucketCache, which will be mounted on OFFHEAP when a secondary file system is mounted on the SSD disks. However, it might reduce S3 access contention if we can point this to the HDFS SSD location (provided the file is available there).

StoreFileInfo

The createReaderContext method defined within the StoreFileInfo class is where the input stream is created for file readers required by scanner. It should be modified to use the MirroredFSAccess for getting the input stream.

AdditionalFSMetrics (new)

This should track metrics such as hit/miss of files in the additional FS, as well as number of files purged and number of purges due to FS reaching capacity, number of files, FS usage, FS remaining space.

AdditionalFSFileLoader (new)

A new component that copies individual files from the original to the secondary FS. Once a reader is opened for a file that doesn't have a copy in the HDFS SSD dir, the background copy should be triggered. Could use a threadpool with a configurable number of threads.

BalancerWithAdditionalFileSystem (new)

The current implementation of StochasticLoadBalancer already defines cost functions to account for HDFS data locality when calculating new assignments. A new extension of StochasticLoadBalancer that prioritises block locality of files in the secondary file system should be implemented and made the default balancer when a secondary file system is configured.

AdditionalFSAllocationManager (new)

A housekeeping component that monitors the additional FS usage and implements priority logic for deleting files, given a configurable threshold usage has been reached. It should also take care of deleting files with missing blocks.

Eviction policy to be made pluggable, but as a minimal baseline, the following should be considered:

- Snapshot referenced files: Files being referenced only by snapshots and no open region should be deleted from the secondary FS;
- Files age: group by age according to creation date and delete files older than a configurable age;
- Table/Region state: delete files for non open tables/regions first;
- LFU/LRU

Could be run as a background chore in the Master, or even a complete separate process.

HFileCleaner

This is an existing component that currently performs deletion of hfiles present in the archive folder and not being referred to by links. We may be able to extend it to also delete files in the additional FS.