CloudFS

Cundao Yu <cundaoy>

Overview

CloudFS is a cloud-backed local file system, user's file system calls are interposed using FUSE. Users operate files under the mount point like a regular local file system, but the actual file data is stored on SSD which is another mount point or on cloud object store depending on the files' size. When storing large files on cloud object store, CloudFS provides two modes: no-dedup mode and dedup mode. Under no-dedup mode, CloudFS will handle each file as a whole. Under dedup mode, CloudFS will split large files' content into chunks using Rabin fingerprint algorithm, and generate chunk keys using MD5 hash. CloudFS provides snapshot functionality including snapshot create, restore, install, uninstall, delete. CloudFS uses a LRU cache to accelerate file operations and reduce cloud cost.
CloudFS contains the following layers or components:

1. **FUSE interface layer**: Implemented in cloudfs.cc. Implemented FUSE callbacks that are required to build a file system. Some of these callbacks are handled directly, and others are passed to CloudFS controller to be handled.
2. **CloudFS Controller**: Defined in cloudfs_controller.h, implemented in cloudfs_controller.cc. Handles FUSE callbacks related to file content manipulation like open, read, write, close, truncate. It also provides functions for accessing file metadata.
3. **Buffer File Controller**: Defined in buffer_file.h, implemented in buffer_file.cc. This component provides a set of buffer file operations and cloud operations. The concept of "buffer file" will be discussed later. This component also uses a cache to accelerate operations and reduce cloud cost. But the cache replacement is done by the cache replacer, which is another component.
4. **Chunk Splitter**: Defined in chunk_splitter.h, implemented in chunk_splitter.cc. This component is responsible for consuming contents and outputting split chunks information.
5. **Chunk Table**: Defined in chunk_table.h, implemented in chunk_table.cc. This component is responsible for recording reference counts of chunks, and persisting this information across mounts.
6. **Cache Replacer**: Defined in cache_replacer.h, implemented in cache_replacer.cc. This component provides functions for recording access, cache deletion, cache eviction. The cache replacement policy is LRU. This component is also responsible for maintaining LRU list information across mounts.
7. **Snapshot Controller:** Defined in snapshot.h, implemented in snapshot.cc. This component provides a set of interfaces for snapshot create, restore, install, uninstall, delete and list. This is implemented by interposing ioctl system call and the users can use snapshot functionality by passing corresponding command parameters into ioctl.
8. **Utility:** Defined in util.h, implemented in util.cc. Provides functionality like debug logging, path checking, object key generator and file compression.

The structure of CloudFS is shown as Figure 1:

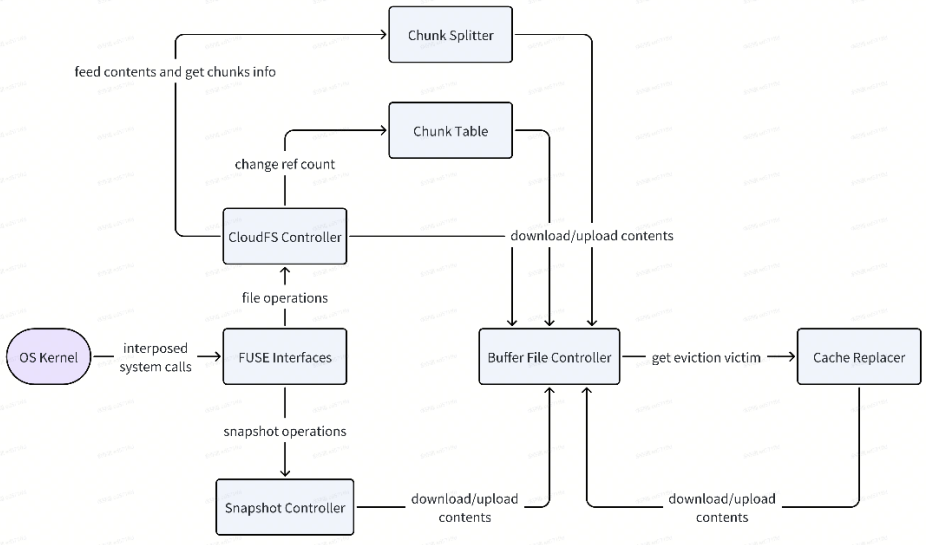

Figure 1 CloudFS Structure

File Management

File Representation

CloudFS stores one file as one "main file" and one "buffer file".
The main file's name is the file's original name. It doesn't store actual file content, it only store metadata including:

1. Buffer path: the path of the buffer file. Stored as extended attribute of the main file.
2. Chunk list: If under dedup mode, chunk list of the file will be stored in the main file.
3. File mode: stored in the main file's inode, using the mechanism of the file system on SSD (ext4).
4. Timestamps: stored in the main file's inode, using the mechanism of the file system on SSD (ext4).

The buffer file stores contents of small files and is used as a cloud content buffer for large files. Actual file content operations are taken on the buffer file. The buffer file also stores the size of the file as an extended attribute of the buffer file.

Why using two files to represent one file?
CloudFS wants to reuse ext4's mechanism like access mode management, access protection, directory creation, timestamp management, hard link and symbol link. But if only using one file, we cannot fetch cloud data for a read-only large file since we cannot write data into it. So, we need another file for buffering cloud data. By using two files, metadata and data are also separated, making system more modular. When adding cache, cache replacer just needs to communicate with buffer file controller.

File Metadata Management

Under no-dedup mode, file metadata only contains buffer path, file mode, timestamps and size. The object key of large file is generated by replacing '/' in the absolute path to '-', so CloudFS do not need to store it. CloudFS distinguishes whether a file is small (size <= threshold) or large (size > threshold) by comparing file size with the threshold. The file size is stored as an extended attribute of the buffer file. When the file size is updated, CloudFS will update this extended attribute. And FUSE callback 'getattr' will first call ext4's 'stat' to get the stat struct of the main file, then replace the size field with the value got from the 'user.cloudfs.size' extended attribute of the buffer file.
Under dedup mode, chunk list, indicating which chunks make up a large file, is stored as content of the main file. It is a list of entries, each entry contains start offset, chunk length and object key. The start offset and chunk length is determined using Rabin fingerprint algorithm and the object key is generated using MD5 hash. Every time one operation that would modify the file content happens, such as write and truncate, the chunk splitter will be used to update the chunk list.

File Content Manipulation

The content of a small file is stored in the buffer file directly. So read, write and truncate is implemented as performing ext4's read, write and truncate on the buffer file.
For large files, the content is stored on cloud and the buffer file is just used for buffering cloud data. No-dedup mode and dedup mode have different behaviors for large files.
Under no-dedup mode, the whole large file is stored as an object on cloud. On open, CloudFS downloads the object into the buffer file, then manipulates the buffer file. On close, CloudFS uploads the buffer file content to the cloud using the same object key, if the buffer is dirty.
Under dedup mode, one large file is split into multiple chunks using chunk splitter. When read, write or truncate, CloudFS first calculates the chunks needed for the current operation, then download from cloud to the buffer file. At the end of the operation, CloudFS may just clear the buffer file (for read), or rechunk the file content and upload new chunks to the cloud (for write and truncate). When doing rechunking, CloudFS may need to download more following chunks to rechunk until find a same boundary. This will be discussed later in detail.
When one small file' size is increased to reach the threshold, CloudFS may upload its content to cloud as one object or do chunking on it and upload a list of chunks, depending on the deduplication mode.
When one large file is truncated and file size drops under the threshold, CloudFS will download all content to the buffer file and now the file becomes a small file.
A Diagram showing the file management of CloudFS is shown as Figure 2 below:

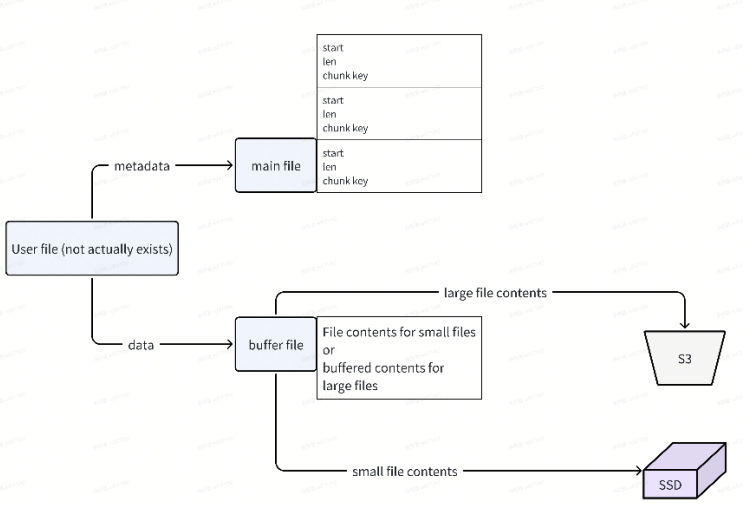

Figure 2 File Management

Deduplication

Deduplication consists of two parts, boundary searching and object key generation. Boundary searching is done by using Rabin fingerprint algorithm and object key generation is done by using MD5 hash. By doing so, different large files can share the same duplicated chunks, and these chunks are only stored on cloud once, reducing the cloud cost a lot.

Boundary Searching

Rabin fingerprint consumes a sequence of bytes and indicates whether there's a new boundary in this sequence. CloudFS will feed Rabin with the content in the buffer file iteratively when doing chunking or rechunking and Rabin will tell CloudFS the start and length of chunks that newly occurs, this process will be done until no remaining file content or an identical boundary has been found. "identical" means the offset in the complete file of a chunk end is the same as a previous one's.
How to choose average segment size?
The minimum segment size, average segment size, max segment size and Rabin window size are set by the user when initializing CloudFS. And since all tests give reasonable sizes, there's no need for CloudFS to determine these parameters by itself.

Chunk Key Generation

After getting the start and length info of each chunk, CloudFS calculates the chunk key for each chunk by doing MD5 hashing on each chunk's content.

Chunk Splitter

In CloudFS, these two steps above are actually performed simultaneously, and are encapsulated as a chunk splitter. Chunk splitter provides interfaces for initialization, feeding contents and get chunks info back, getting last (tail) chunk info. The Rabin fingerprint algorithm and MD5 will consume the content at the same time, and when there's a new boundary found by Rabin, MD5 will output the key for the new chunk, then chunk splitter adds a new entry containing start offset, chunk length and object key to the return list.
Also, encapsulate Rabin and MD5 as a component can make codes logic clearer. Chunk splitter doesn't interact with file, it only consumes memory buffer passed in as parameter, and return a chunk info list. The bugs in the chunk splitter can be restricted only in this component and after sure there's no bugs in chunk splitter, outside components or layer can simply use chunks splitter's APIs to do chunking.

Chunk Table

CloudFS uses a chunk table to record reference count and snapshot reference count for each cloud chunk. Reference count means how many duplicated chunks in the current file system state are using this cloud chunk, snapshot reference count means how many snapshots are using (at least 1 reference count in the snapshot) this cloud chunk. If either reference count or snapshot reference count of one cloud chunk is larger than 0, this cloud chunk should be kept on cloud. When a reference count of one chunk drops to 0 and snapshot reference count is also 0, CloudFS deletes this chunk from the cloud. Chunk table is persisted across mounts. During mount initialization, chunk table is reconstructed using persistent file and the persistent file is generated when unmounting.

Why using two types of reference count?

This makes snapshot create, restore and delete easier. When create a snapshot, this snapshot only captures reference count but does not capture snapshot reference count. It only increases snapshot reference count. When restore a snapshot, the reference count of each chunk is restored directly, but the snapshot reference count remains the same. Using such design, no matter how we change the reference count, create snapshots, restore snapshots and delete snapshots, as long as the snapshot reference count is larger than 0, the chunk won't be deleted, which protects the data integrity for snapshots that are using this chunk.

Cache

CloudFS has a cache layer to accelerate file operations and reduce cloud cost. The cache contents are managed by the buffer file controller, and eviction decision is made by cache replacer. All cached chunks are stored as files under ".cache" directory.
This structure is shown as Figure 3 below:

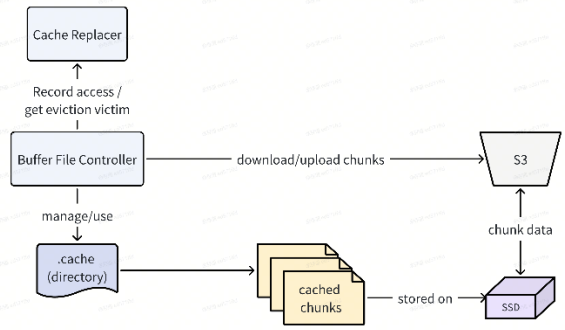

Figure 3 CloudFS Cache

Write-back Cache

The cache used by CloudFS is a write-back cache. Chunk downloading and uploading function is implemented in buffer file controller, and the cache is implemented under the buffer file controller and other components don't know there's a cache layer.

Chunk Downloading

When other components request buffer file controller to download a chunk to a buffer file, buffer file controller first check if this chunk is in cache. If in cache, buffer file controller copies the content from cache to the buffer file locally. If not in cache, buffer file controller first does eviction to make space for the chunk to be downloaded. After eviction, buffer file controller downloads the chunk under the cache directory (.cache) as a file. Then buffer file controller copies the cache content to the buffer file.

Chunk Uploading

When other components request buffer file controller to upload a chunk, buffer file controller does not upload it immediately, but first check if the chunk is already in the cache. If yes, just return, or else do eviction and then just put the chunk in the cache. Buffer file controller needs to mark these chunks to be uploaded as dirty. When these chunks are evicted, buffer file controller uploads them to the cloud.
CloudFS also makes sure the cache is persistent across mounts. Since cache are already stored as files, CloudFS only needs to let cache replacer to save access records when unmounting and restore these records when mounting.

Cache Replacer

CloudFS uses a LRU cache replacement policy. A doubly linked list is used to record access and choose victim. The head of this linked list is the most recently used chunk and the tail is the least recently used chunk.

Why extract cache replacement policy out as an independent component?

This makes it easier to switch policies. Cache replacer is defined as a base class with a set of virtual methods for accessing, evicting. LRU is just an implementation for cache replacer. If LRU works bad on this project, we can implement the cache replacer using another algorithm and just simply switch to it. Actually, I tried LRU-K algorithm when doing this project, but find it work bad due to the timestamp precision issue. I changed to LRU and find LRU works well.

Cache Performance

I use the test on Autolab and some tests written by myself to test the performance of the cache. After using a LRU cache, CloudFS only uses 37% cost compared to without cache.

Snapshot

Users use snapshot functionality by calling ioctl with corresponding command parameters. CloudFS provides snapshot functionality including create, restore, delete, install, uninstall and list.

Snapshot Content

Each snapshot contains all metadata and small file data that are in CloudFS when it was created. When creating a snapshot, CloudFS traverse the directory tree. If one file is a directory, write the directory path and directory's stat structure (containing file type, access mode and timestamps) to the snapshot file. If the file is a regular file, CloudFS also write the path and stat structure, but will check the file size and save the file size. If the file is small file, CloudFS will save the buffer file contents directly into the snapshot file, after the file's metadata. If the file is large file, CloudFS will just save the chunk list into the snapshot file.

Cost Reduction

To reduce cloud cost, CloudFS will archive the snapshot file and use zip algorithm to compress the snapshot, then upload it to the cloud. When restoring or installing the snapshot, CloudFS will download the snapshot object first, uncompress and unarchive the snapshot, then retrieve metadata and data from it to reconstruct the file system.

Some Workflow Charts

Now all main components have been introduced, to give a better understanding of the whole system workflow, a flowchart for read (dedup mode) and a flowchart for write (dedup mode) are shown below as Figure 4 and Figure 5.

Figure 4 read (dedup mode)

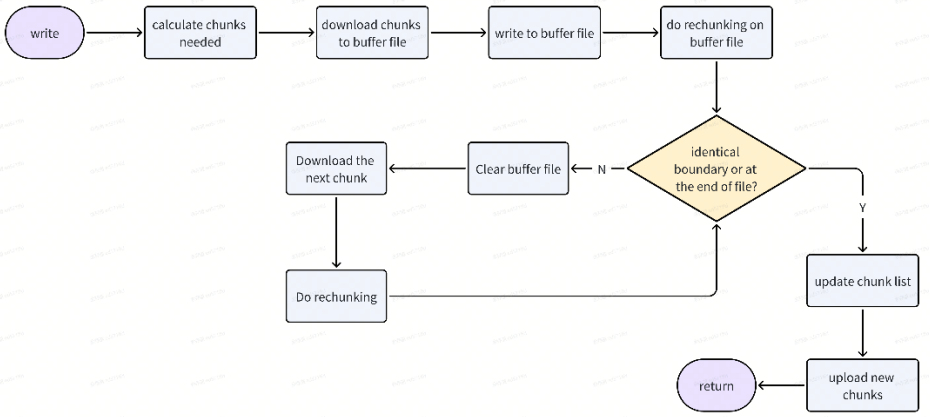

Figure 5 write (dedup mode)

Some Cost/Performance Trade-offs

Download one/multiple chunks when rechunking following chunks?

When a write changes a large file's content, CloudFS need not only to rechunk the contents in the buffer file, but also to download following chunks to do rechunking until find an identical boundary. So how many chunks does CloudFS download each time? If download multiple chunks, Rabin may find an identical boundary before the last downloaded chunk, so some downloading is wasted. If download only one chunk, frequent file clear and then write may have bad performance effect. After testing, CloudFS chooses to only download one chunk each time. Because compared to unnecessary cloud cost, local file system performance cost can be ignored. And actually, cloud transferring can incur more bad performance effect than frequent local file cleaning.

Compress file content chunks or not?

Snapshot files are compressed to reduce cloud cost, then can we also compress file content chunks? After testing, this gives little benefit, compared to extra compression/decompression performance cost. This is because chunk sizes are usually small, so the reduced space is small compared to extra metadata stored for decompression. Under no-dedup mode or when chunk sizes are large, compression may work better.