SSD & Cloud Hybrid File System

HingOn Miu
Carnegie Mellon University

Introduction

My CloudFS implementation is a file system that integrates solid-state devices and cloud storage systems. For **Part 1**, I implemented a hybrid file system that stores small files in a solid-state device and large files in a cloud storage system. For **Part 2**, I implemented block level duplication that identifies the duplicate content in large files and only stores unique segments to reduce storage cost. For **Part 3**, I implemented taking and restoring snapshots of my file system such that my file system can undo mistakes and jump to a previous state.

The significance of CloudFS lies in the combination of two different storage components. The cloud storage system is scalable such that user can easily request a larger capacity service. Also, the data stored in cloud is always available such that storage does not wear out. On the other end, solid-state devices are very fast, while they do have limited capacity and have to handle wear-leveling to improve the lifespan of the solid-state devices. Therefore, a great CloudFS design should carry the best features of both storage components: fast speed in smaller random accesses and high scalability of storage management.

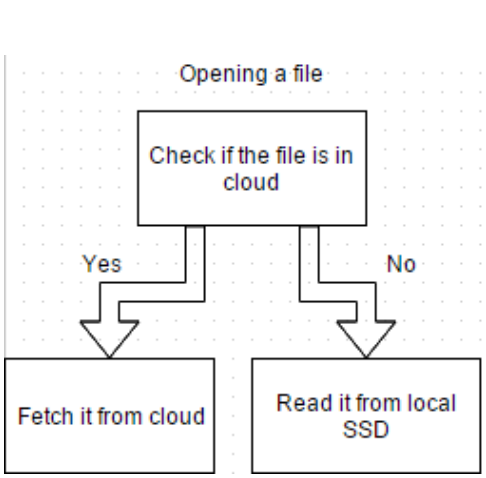
Part 1: Hybrid File System

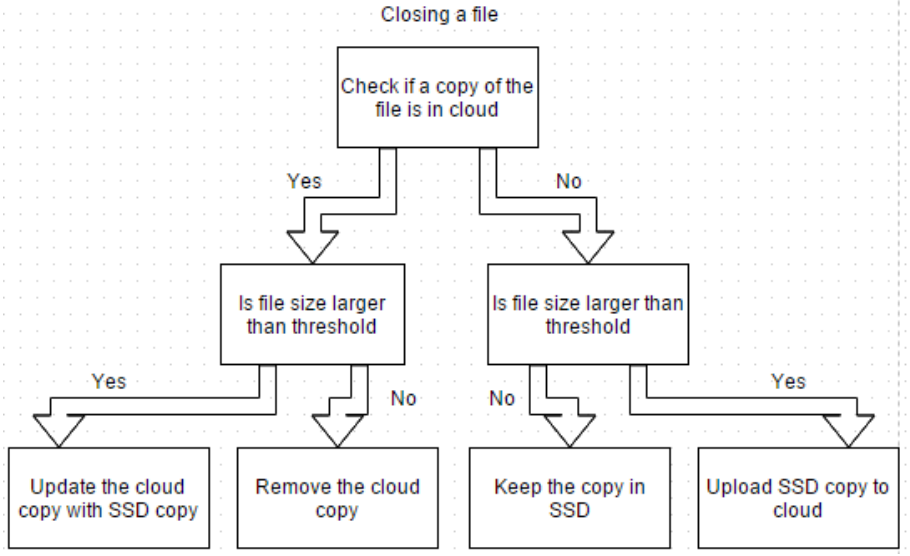
Goals

My top goal here is to correctly separate large files from small files such that large files are stored in cloud and small files are stored in solid-state device. Moreover, a very essential goal here is to record and reflect status modifications to client after client updates a file's status. Last but not least, my implementation should incur as little cloud storage cost as possible to reduce the runtime cost of CloudFS.

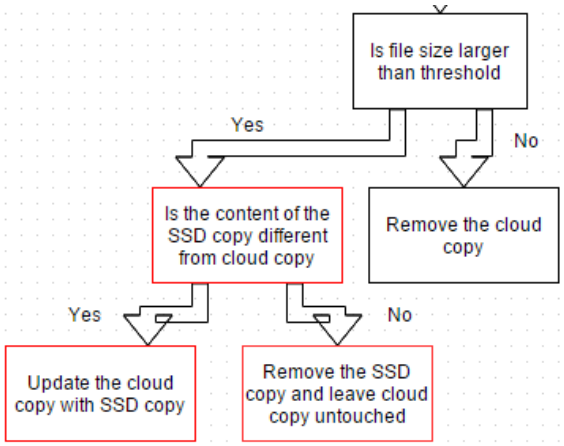
Design

To correctly separate large files from small files (1st goal), I designed to choose whether upload the file to cloud or not when client closes the file. So in closing a file, I would always first check if a copy of the file is currently in solid-state device or not. If the file is only in solid-state device and it exceeds the file size threshold, I then upload the file to cloud. If the file does not exceed the threshold, it always stays in solid-state device. If a copy of the file is already in cloud, this implies that the file was previously closed and uploaded to cloud. In this case, I check if the solid-state device copy of the file gets smaller than the file size threshold. If it does, the cloud copy is deleted and solid-state device copy becomes the sole copy of the file. Otherwise, the cloud copy is updated with the solid-state device copy. At the same time, whenever a client opens a file, the file is fetched from cloud if the sole copy of the file is in the cloud. Below diagrams capture the simple control flow of opening and closing a file.





To record and reflect file status modifications to client (2nd goal), I decided to use extended attribute of a file to keep track of status changes if the file content is uploaded to cloud. The solid-state device copy of the file is then truncated to file size of 0, while the entire struct stat is stored in the extended attribute such that all status changes made the file can be recorded locally in solid-state device copy. This design also reduces cloud cost as status modifications done to files uploaded to cloud are not actually flushed to cloud so there is less cloud communication cost.



To further reduce cloud cost (3rd goal), I keep track of whether the file content of local solid-state device copy is changed such that this avoids unnecessary cloud storage cost (uploading same file with same content to cloud). Left diagram reflects the changes in red of closing a file.

Evaluation

Tests	PASS or FAIL	Details
1	PASS	Check if small files are stored in SSD properly
2	PASS	Check if large files are stored in cloud properly
3	PASS	Check if small and large files are stored in SSD or cloud properly
4	PASS	Check if small and large files are stored in SSD or cloud properly
5	PASS	Check if changing file modification time is recorded and reflected properly
6	PASS	Check if changing file permission is recorded and reflected properly
7	PASS	Check if changing file extended attribute is recorded and reflected properly

Test 1 to Test 4 verifies whether small files are correctly stored separately from large files (1st goal). This proves that my above diagrams capture the correct control logic of storing the files. Test 5 verifies file modification time. Test 6 verifies file permission. Test 7 verifies file extended attribute. So, this proves that the status modification of files is properly recorded and reflected in my CloudFS implementation (2nd goal). Reducing cloud cost (3rd goal) is tested in a later section.

Part 2: Deduplication

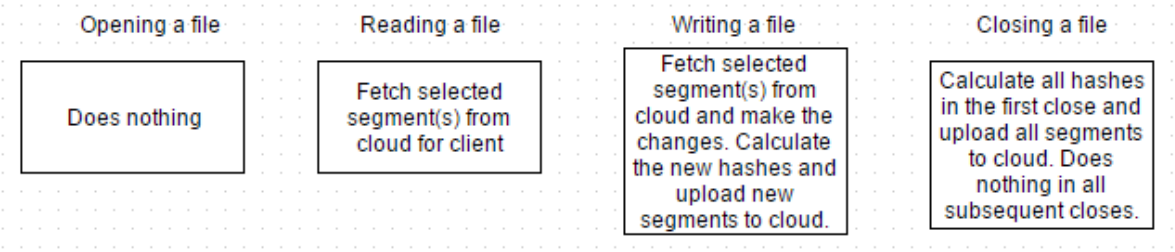
Goals

The top goal is to implement deduplication correctly such that duplicate data segments are not uploaded to cloud repeatedly. Moreover, my implementation should incur as little cloud storage cost as possible to reduce the runtime cost of CloudFS.

Design

When deduplication is enabled, a file is broken into segments and uploaded to cloud separately when closing a file. To correctly implement deduplication (1st goal), I designed a hash table in solid-state device to keep track of the reference count of each file segment. I created a directory and named it my hash table. For each

segment, I have a file with its hash as its filename and its reference count as its file content. The most essential of my deduplication design is that the file segments of a file are uploaded to cloud after first closing the file, then all subsequent opening and closing the file basically does nothing to communicate with the cloud. This is made possible because all cloud uploading and downloading are moved to read/write, where the actual file access happens. Below diagram captures the simple control flow for deduplication.



To reduce runtime cost (2nd goal), only the file segment(s) client requests to read/write is fetched from the cloud. Therefore, reading/writing a few bytes of file does not need to fetch all file segments of a file. Moreover, when a selected segment is modified, only this selected segment's hash is recalculated. The old segment reference count is decremented and the new segment is inserted to hash table and uploaded to cloud. Hence, it could have been a O(n) cloud cost operation of fetching and uploading all file segments of a file, while my design implement a O(1) cloud cost operation in which only the selected file segment is fetched and uploaded back to cloud.

Evaluation

Tests	PASS or FAIL	Details
1	PASS	Check if small and large files are stored in SSD or cloud properly
2	PASS	Check if small and large files are stored in SSD or cloud properly
3	PASS	Check cloud storage cost
4	PASS	Check cloud storage cost
5	PASS	Check if small and large files are stored in SSD or cloud properly
6	PASS	Check if small and large files are stored in SSD or cloud properly
7	PASS	Check if small and large files are stored in SSD or cloud properly
8	PASS	Check if small and large files are stored in SSD or cloud properly
9	PASS	Check if small and large files are stored in SSD or cloud properly
10	PASS	Check if small and large files are stored in SSD or cloud properly
11	PASS	Check if changing file modification time is recorded and reflected properly
12	PASS	Check if changing file permission is recorded and reflected properly
13	PASS	Check if changing file extended attribute is recorded and reflected properly
14	PASS	Check cloud storage cost

Test 1, Test 2, Test 5 to Test 10 verifies whether small files are correctly stored separately from large files with deduplication enabled (1st goal). This proves that my above diagrams capture the correct control logic of storing the files. Test 11 verifies file modification time. Test 12 verifies file permission. Test 13 verifies file extended attribute. So, this proves that the status modification of files is properly recorded and reflected in my CloudFS implementation. Reducing cloud cost (2nd goal) is mainly tested in Test 14. It checks whether entire file is downloaded from the cloud to SSD when only reading/writing a selected segment. This proves that my CloudFS cloud usage is indeed minimized.

Part 3: Snapshots

Goals

The top goal is to implement snapshots correctly such that snapshots can be created, restores, deleted, listed, installed and uninstalled. Moreover, my implementation should incur as little cloud storage cost as possible to reduce the runtime cost of CloudFS.

Design

To implement snapshots correctly (1st goal), my code creates snapshots by first checking if there is already maximum number of snapshots allowed created. If not, my code uploads all files and directories to cloud. The snapshot id is also properly stored. To restore snapshots, my code first checks if the snapshot id exists. If it does, my code fetches the files and directories of the snapshot from cloud to the root directory of SSD and remove all old files and directories. Installing and uninstalling is done similarly in a snapshot folder.

To reduce cloud storage cost (2nd goal), instead of uploading all original small files to cloud, my code tar them first before upload. Although there is a tar runtime overhead, the overall cloud storage cost must decrease. Since I previously used extended attribute to record file status and libtar does not support extended attributes, my code generates a proxy file for each extended attribute during tar and recovers all extended attributes during untar.

Evaluation

Tests	PASS or FAIL	Details
1	PASS	Check snapshot creation and restore
2	PASS	Check snapshot creation and restore
3	PASS	Check snapshot creation and restore
4	PASS	Check snapshot install and uninstall
5	PASS	Check snapshot creation and restore
6	PASS	Check snapshot creation and restore
7	PASS	Check snapshot creation and restore
8	PASS	Check snapshot creation and restore
9	FAIL	Check snapshot creation and restore
10	FAIL	Check file copy from snapshot install folder
11	PASS	Check snapshot install and uninstall
12	PASS	Check snapshot install and uninstall
13	PASS	Check snapshot creation and restore
14	PASS	Check snapshot creation and restore
15	PASS	Check cloud storage cost

Test 1 to Test 14 checks the correctness of my snapshots implementation (1st goal). My implementation fails at Test 9 and Test 10 because of an edge case in installing snapshot to a folder. Nevertheless, my implementation passes most correctness tests. Test 15 checks the storage cost of my snapshots implementation. My score is 77.25 and I am ranked 11th of all students with non-zero scores. This proves that I did a relatively good job in reducing storage cost. The tar file size of the small files must be smaller than those small files' combined size.

Conclusion

If the average segment size is too large, it may become too difficult to for two files to share same file segment because this implies most of the two files would be identical. This case is rather rare in real-life, and so breaking files into segments is pointless when no duplication can be identified. If the average segment size is too small (say 1 byte), it may become too easy for two files to share many same file segments. However, this implies the reference count of each file segment would be very large, and so the runtime of hash table operations and traversing the tiny segments become the bottleneck. Therefore, depending on the data type, there should be optimal average segment size that minimize storage cost and runtime cost at the same time.