System description

1. Introduction

This is a prototype for data deduplication system implemented using C++. It combines ideas from [1] [2] [3] [4]. This system breaks up an incoming file into chunks using Karp-rabin algorithm and store only unique chunks.

In order to identify unique chunks, SHA1 is used for generating a chunk's hash value. To compare two chunks, comparison between their hash values is used instead of byte-to-byte comparison between their contents because the former method requires only comparison between 20 bytes when SHA1 is used.

2. Data Structures

2.1. Containers

A container is the basic unit for storing chunks and each container can contain up to 1024 chunks and their metadatas.

2.2. File Recipe

A file recipe records which chunks a file consist of. According to a file recipe's content, the data deduplication system retrieves the corresponding chunks from containers and use them to restore the file.

2.3. Hash Table Index for key-value pairs

Let us assume that average chunk size is 8KB and 20 bytes for the hash of a chunk. For 1TB unique chunks, there will be 2560MB for all chunk hashes. Therefore, it is not always possible to store all the chunk hashes in memory. As chunk hashes are stored on disk, a data structure for accelerating the lookup speed is needed. So I implement a key-value system similar to the Log Store in SILT [2]. The key is the hash of a chunk and the value is the id of the container where the chunk is stored. All key value pairs are stored on disk and a cuckoo hash in memory works as filter and index so that a key-value pair can be localized within O(1) as shown in figure 1.

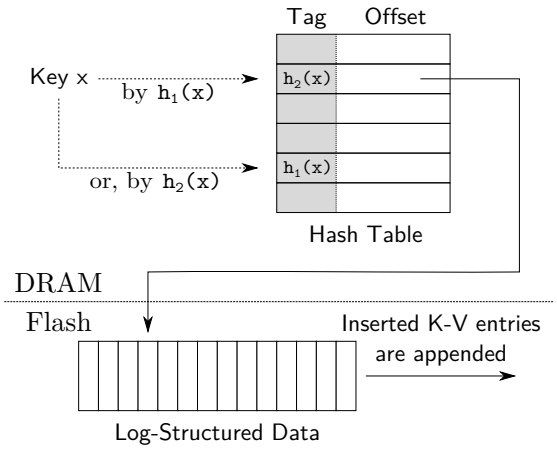

Figure 1: Implementation of Log Store

3. Procedures of This System

3.1. Insertion procedure

The insertion procedure is similar to that in [4] as shown in figure 2. Details will not be introduced here.

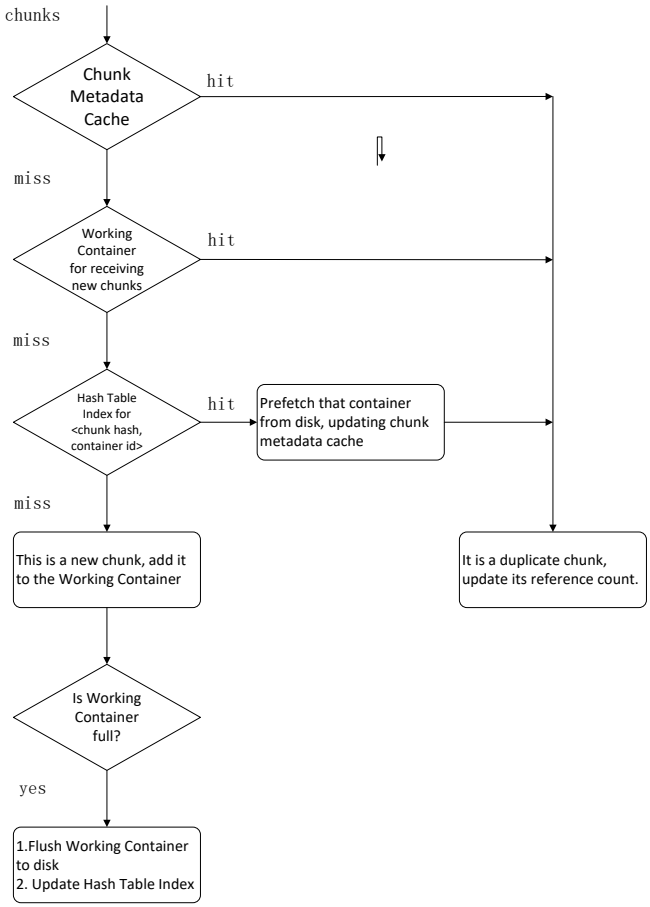

Figure 2: Insertion Procedure

3.2. File restore procedure

For file restore, I implement the Forward Assembly Area in [3] to accelerate file restore because we know which chunks will be used in the future when restoring a file according to its file recipe.

4. future work

I do not implement the Hash Store and Sorted Store in SILT [2] for I find that SILT spends a lot time doing internal works. Furthermore, to generate a new Sort Store, the old Sorted Store needs to be read from disk, combined with Hash Stores, then written back to disk. As a result, write amplification might be a huge problem. Maybe a Log-Structured Merge tree is better suited for this system.

Deletion and update are not yet implemented because such operations happen rarely for data deduplication systems and they are very complex. I leave them for future work. But fields related to deletion and update are kept within corresponding data structures(e.g. reference count for each chunk).

Concurrency will be considered.

References

[1] Zhu, Benjamin, Kai Li, and R. Hugo Patterson. "Avoiding the disk bottleneck in the data domain deduplication file system." In Fast, vol. 8, pp. 269-282. 2008.

[2] Lim, Hyeontaek, Bin Fan, David G. Andersen, and Michael Kaminsky. "SILT: A memory-efficient, high-performance key-value store." In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 1-13. 2011.

[3] Lillibridge, Mark, Kave Eshghi, and Deepavali Bhagwat. "Improving restore speed for backup systems that use inline chunk-based deduplication." In 11th USENIX Conference on File and Storage Technologies (FAST 13), pp. 183-197. 2013.

[4] Debnath, Biplob K., Sudipta Sengupta, and Jin Li. "ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory." In USENIX annual technical conference, pp. 1-16. 2010.