Dropbox-Like Storage

Yanqiu Chen Jie Lu Yangruirui Zhou
chen1997@bu.edu jielu666@bu.edu yrrzhou@bu.edu
Department of Electrical and Computer Engineering, Boston University
Dec. 13, 2020

1. Introduction

Storing a piece of data in a PC hard drive is straightforward. However, in other practical situations with a large scaling (e.g. RAID, clustered file system, network-attached/grid-oriented storage) it is usually needed to store files in a partitioned state, and put each of these data chunks into different disks/servers/etc. (This is called data striping.)

Storing in the form of chunks has the extra advantage of not needing to store the same chunk twice on disk, thus saving disk space. The savings can be very impressive in cases of multiple highly-similar files.

As the guides and requirements, the proposed data storage needs to fulfill these four functionalities:

- loading of text files: When a file is loaded, it is divided into fixed sized chunks (e.g. 1K), the last chunk is padded such that it has the same size with the other chunks, the chunks are stored into your master storage unless they have not been stored before, and the filename to chunk relation is marked
- listing of loaded files: Your system should be able to list the names of the existing files;
- retrieving back of text files: When a request to download a file is done, it should be generated back from its chunks and returned back to the user as it was uploaded;
- deleting of text files: when a file delete is requested, the marks associating the file with the chunks and the chunks associated only with this file must be removed.

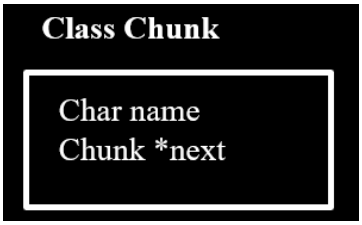
2. Data Structure

We totally have five very important data structures. They are three structs and two dictionaries built by hash.

2.1. Three structs

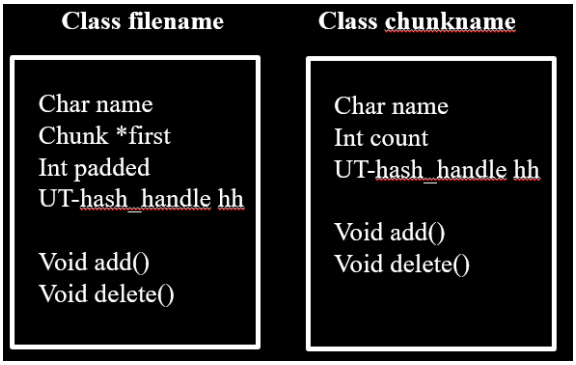
We use structs to save different info types. In the following three structs:

2.1.1. Class chunk is one element in a linked list, which help us to save the sequence of one upload file. It saves its **chunk name** (a value given during we address the uploaded files, we can use this value to find the divided chunk file) and the **next chunk** it should point to, which will help to recover the original uploaded file.



2.1.2. Class filename: variable **name** saves the uploaded file's name, the chunk ***first** is a linked list which is a sequence of word chunks (we can get the same content when we go through the whole linked list compared with the original content in the uploaded file), **padded** means whether we pad the rail chunk file, if there is no padded file in the linked list, its value is 0. The UT-hash_handle is a value given by hash function. The two functions in this class will help to add new elements and delete elements in the class filename.

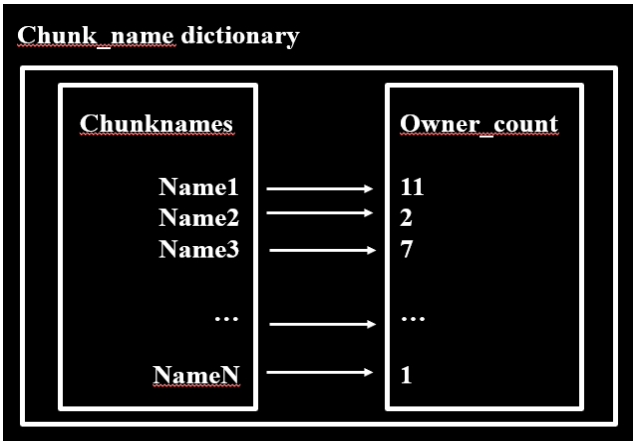
2.1.3. Class chunkname: variable **name** is the chunk's name, and we use int **count** to represent how many uploaded files own this chunk (has a part of content same as this chunk's content). The UT-hash_handle is a value given by hash function.



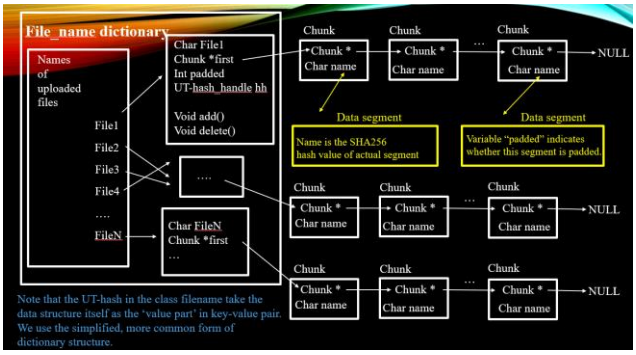
2.2. Two dictionaries

We use dictionary-like data structures to represent the relationships between different classes:

2.2.1. Dictionary chunk-name saves the relationships between **chunks and owner_count**, which can help to keep track of the numbers of files that contain a specific chunk. If the count is 0, the relevant data segment needs to be removed. The dictionary entry also needs to be deleted.



2.2.2. Dictionary file_name saves the relationships between names of uploaded files and class filenames. We can use the file's name to find the specific class which save the info of this file. And our algorithm can be implemented by this data structure.



3. Algorithm

After we have the above data structures, we can use them to implement our algorithm by some important functions. In a high level, we can describe our algorithm as a few steps:

3.1. Step 1

Load the first file we want to store, after that, save the file's name into file_name dictionary and use hash function to map it to a new class filename. Save the file's name into the char name[] of the class filename, and divide the file's content into fixed-size parts and save these chunks into new files one by one.

Save the name of these chunks into different new-built class chunks, find the order of these chunks and build a linked list to save these chunks as their original order. Pad the last part if the last part needs to be padded, and update int padded. Add all these chunks into chunk_name dictionary, and set the relative owner_count be 1.

3.2. Step 2

Load one more file after the last relationship is built and all information has been saved in file_name dictionary. Save the file's name into file_name dictionary and use hash function to map it to a new class filename. Save the file's name into the char name[] of the class filename.

Divide the current uploaded file into fixed-size parts as step1 did. Then, compare the split parts with our previously saved chunk files one by one.

If one part has the same content with previously saved chunk files, we can update the relative owner_count of the currently compared chunk file by adding 1. Link the last chunk *next or *first in class filename to the same-content class chunk.

If we cannot find an existing chunk file has the same content with this part, save the part's content into a new file, and create a new class chunk to save the new file's name. Link the last chunk's *next or *first in class filename to the new class chunk. Add the new chunk into chunk_name dictionary, and set the relative owner_count be 1.

Pad the last part if the last part needs to be padded, and update int padded.

3.3. Step 3

Repeat step2 until we have loaded all files we want to upload.

3.4. Step 4

Test two or more existing stored files, output their contents consist of different chunk elements. If they are same and the whole storage size is smaller than before, that means our algorithm is effective.

4. Function outlines

4.1. Split and Merge

Split one chunk:

Read a fixed size frame in the input file, then pass the frame to the Hash() function, which return a hash value as the name of the chunk file. In the split(), we choose a fixed size frame in the input file as our chunk file . The rest of the input file, we save as the output file.

Merge one chunk:

Read the chunk file into a buffer, then open our input file in write mode and add the chunk to the end of input file. Save this new input file into the output file.

4.2. Pad and depad

Pad one chunk:

If we detect there is a file need to pad, we put the text name into function pad(char* c). In this function, we firstly detect the whole text and judge the length of this file.

If the length is equal to our set size, return back. Otherwise, we need to detect the last character of this file.

If the last one is the same as our first-order padding-label, we will choose the second-order label instead. Otherwise, we will use the first one and plus multiple labels after the original last character until the size of file can equal to our set size.

Depad one chunk:

If we detect there is a file need to depad, we put the text name into function depad(char* c). In this function, we will delete the added labels in the file rail.

At first, we need to judge if the last one is the same as our first-order padding-label, we will choose the second-order label instead.

After we know which label we used here, we can delete the rail labels until we detect the first no-chosen character.

4.3. File loading

Generate a new class filename, initialize its name and insert it into file_name dictionary. Then, split the loaded file into chunks.

For each chunk, do a SHA256 hash and search for same entry in chunk_name dictionary. If found, simply add its owner_count by 1; otherwise, save the actual data segment using the hash value as name.

Build a new class Chunk for the class filename and link the previous Chunk to the current Chunk. Judge if the last segment needs to pad. If it needs to be padded, set padded=1. Then hash and build Chunk same as above.

4.4. File listing

List all file names in file_name dictionary in alphabetical order.

4.5. File Retrieval

Search by input name in the file_name dictionary. If the input name is found in the dictionary, find the corresponding class filename.

Go through the entire linked list, and copy the chunk for each node in the linked list, then merge it to the end of the output file.

Judge if the last chunk needs depadding by using the value of int padded and depad it if it equals to 1 before merging.

Return the output file.

4.6. File deleting

Search by input name in the filename dictionary. If input name is found in dictionary, find the corresponding filename structure.

Go through the entire linked list, and find the entry for every node in the linked list and reduce the relative owner_count in chunk_name dictionary by 1. If now owner_count is 0, remove the actual data segment and chunk_name dictionary entry.

Remove the class filename, and its entry in the file_name dictionary.

5. Datasets and time complexity analysis

Use ten 10M files as our test data. And our target is to keep its storage size under 15M totally.

For evaluation, Because dictionary lookup is O(1) assuming no collisions and traversing through the linked list is O(number of chunks) which equates to O(size of the file), we have:

5.1. Load

Loading a text file: O(size of the file).

5.2. List

Listing of loaded files: O(nlog(n)), where n is the number of files in the storage. Listing itself is only O(n) but in our case an alphabetical order comparison sort is done before.

5.3. Retrieve

Retrieving back a file: O(size of the file).

5.4. Delete

Deleting a file: O(size of the file).

Obviously, the above analysis is blind to actual interactions with Linux file systems. Considering now we are putting all of the chunks in the same directory, and if the disk is under ext4 file system (or similar ones that use H-trees for file lookup), then the lookup time of the file system will be O(1), trivial compared to the functionalities. We also assume writing/deleting data on the disk is O(size of the file).

6. Program Implement

The program consists of some interface functions: split/merge chunk, pad/depad chunk, file load, retrieve

and delete functions. These different functions are divided into different source file. The Storage.cpp is the main function to run the program. In the command line, use the ‘make’ command to run the program. The input file from user’s input is suggested use the full path of the file. Or it might have errors if the input file doesn’t in the local directory. The program also needs to use some unique symbol to represent what operation it will go over. ‘- l’ and ‘- d’ means loading the file and deleting files. For example, when user choose a file to upload, the result will return the number of chunk files created. After files successfully loaded, if users want to keep deleting files, they still need to run the program again to support this operation. Every time if user want to do a new operation such as operation delete after load the files. It requires to re-run the program and input ‘input_file path -l’.

7. Afterthoughts

After we implemented our project, we think there are still somewhere to do more works.

7.1. The number of the buckets

It is easy to see that there will be thousands of, if not more, chunk names in the chunk_name dictionary. There are some solutions to the heightened chance of collisions, such as increasing the number of buckets, or adopting a dynamic resizing and rehashing scheme. Due to the uncertain nature of how many entries will actually be in the dictionary, maybe resizing/rehashing is the more appropriate solution. It will inevitably introduce time overheads, but estimating at O(times that the number of entries exceed upper limit) it is trivial.

If a better estimation of the usage scenario can be done (or an absolute limit of chunks in one directory), then a good size of the hash table can be picked. The problem and proposed solutions are applicable to the filename dictionary as well.

7.2. Size of the chunk, granularity

The smaller the size of chunk, the more savings we can extract from saving the same chunk only once. However, not only it will make the dictionary size problem mentioned worse, but also increase the storage overhead for storing chunk information. And because the hash function is O(1) regardless of the size of input, the constants in O(size of file) will increase as well.

Because the actual duplicate rate of files will vary greatly for different types of file and application, and because our data structures and algorithms are not sensitive to chunk size, it is possible to deploy a variable chunk size scheme for different files, based on usage analysis and/or possibly pattern recognition algorithms.

Author Contributions Jie Lu implemented the function of split and merge, loading, and build connections between different functions, wrote the **“implement”** part in this report. Yangruirui Zhou implemented the function of pad and depad, GUI, and wrote the **“algorithm”** and **“data structure”** parts in this report. Yanqiu Chen implemented the function of retrieval, list and delete, and wrote the **“introduction”** and **“time complexity”**, **“afterthoughts”** parts in this report.