Deduplicator Final Report

Team member: Zulin Liu, Jiaxin Tang,
Ganquan Wen, Xiang Zheng, Zisen Zhou

Abstract

File systems often contain redundant copies of information: similar files, files only edit few characters or even identical files. To improve the file systems and handle the duplication, deduplication calls attention. Data deduplication is an emerging technology that introduces reduction of storage utilization and an efficient way of handling data replication in the backup environment.

In our project, we will design and implement an efficient data storage locker that utilizes deduplication. Users are able to store file in our locker, and to retrieve and delete files from the locker. Our locker will save the common data block in the file only once to fully utilize the storage.

1. Introduction

The project mainly makes a data storage locker that utilizes deduplication. The locker is able to receive files and store them (for later retrieval) with a minimum storage by storing some common data blocks only once. We have implemented minimum requirements in project suggestion, and for extension part, we have implemented binary file storage, GUI that displays storage process in real time, and file deletion from lockers.

This report will introduce our project mainly in three parts: firstly, we will introduce our method to realize de-duplicator (including algorithm we used to partition original files), how to retrieve or delete files from the locker, and GUI implementation. Secondly, we will show some test results for ASCII and binary file. At last, we will make some conclusions on our speed of these operations(store, retrieve and delete), and compression rate of storing several files.

Implemented features

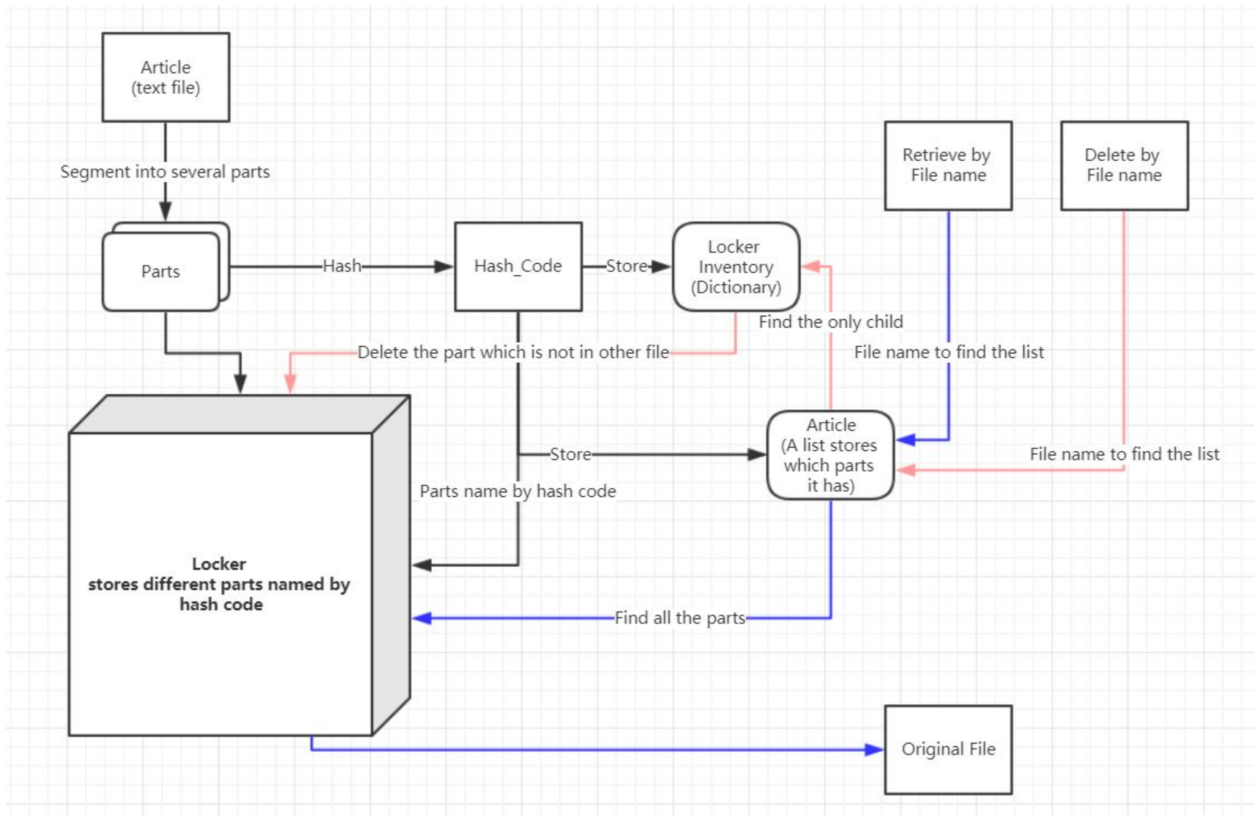
1. Ability to store ten 10MB ASCII files, any two of which differ in at most 5 character edits (character edits are insertions of a character, deletion of a character, or modification of a character), using at most 20MB of storage.
2. Ability to retrieve all files stored in the locker in any order and at any time.
3. The program does not require any live state (i.e. it should be possible to stop and restart the program, even on a different computer, with the storage locker contents in order to reproduce the stored files).
4. Command-line User Interface that allows users to insert files into the locker and displays current

storage usage. Format: store -file [file] -locker [locker location].

5. Ability to store binary files.
6. Graphical User Interface that storage progress (and file allocation) in real time.
7. File deletion from the locker.

2. Method Description

We generally construct a high level diagram of our locker system, as depicted in figure 1, and implement every part of the diagram. We elaborate all of the parts in the diagram in detail below.



Article (text file)
Segment into several parts
Parts
Hash
Hash_Code
Store
Locker Inventory (Dictionary)
Retrieve by File name
Delete by File name
Find the only child
File name to find the list
Delete the part which is not in other file
Store
Article (A list stores which parts it has)
File name to find the list
Parts name by hash code
Locker stores different parts named by hash code
Find all the parts
Original File

Fig 1. High level diagram of system

2.1 Partition

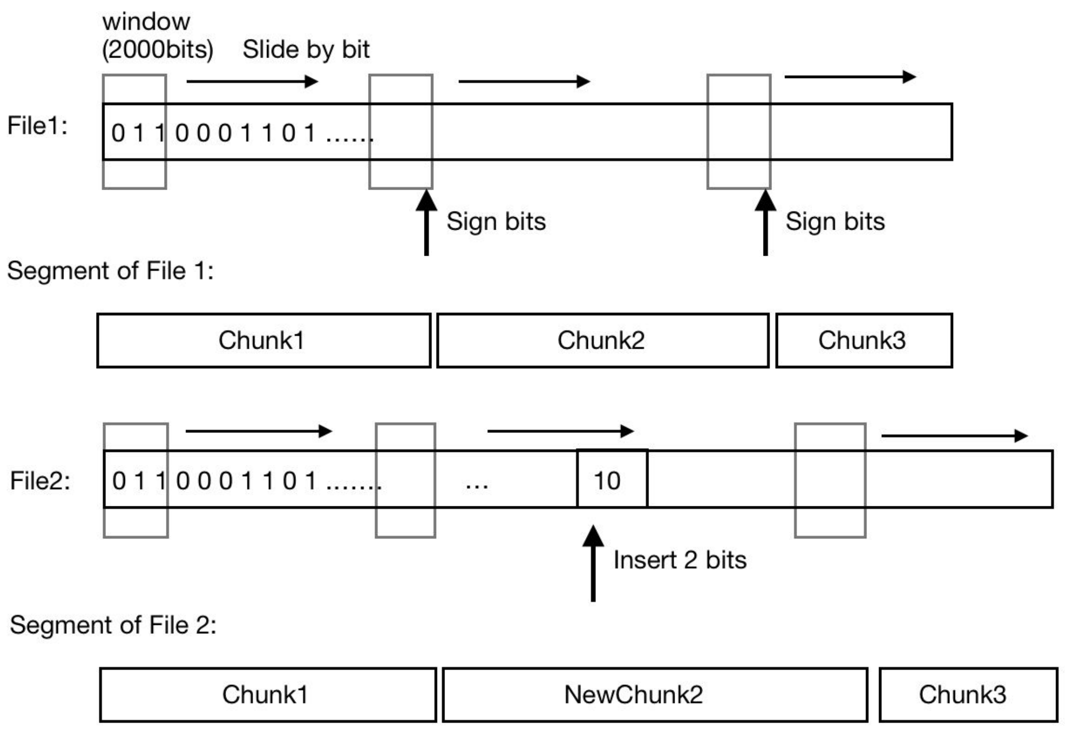
2.2.1 ASCII file

For ASCII files, we can easily track the characters, words, sentences, and paragraphs.

As mentioned in requirement, there are at most 5 characters different from each 2 files and the size of each file is 10MB. Thus, it is ineffective to segment an ASCII file by detecting word by word, or sentence by sentence.
What we do is first detecting how many paragraphs in this file by counting the ‘/n/n’ numbers. After that, if the total paragraphs we have are less than 1000, we divide the file just by ‘/n/n’. Otherwise, we set the step to be number of paragraph divided by 1000, then we divide the file by step multiplies ‘/n/n’ number. In this way, we can limit the chunks number between 1000 and 2000.
Because most of paragraphs of duplicate files are exactly the same, we don’t need much extra space to store them. In worst case, variation in one file can cost at most five extra paragraphs to store.

2.2.2 Binary file

While we cut a ASCII file into pieces by its paragraph, binary files usually have no meaning and paragraphs, which means we cannot segment binary files by new line breaks. However, based on patterns of 0 and 1 in binary file, we can use sliding window techniques to do partition works.



Sliding window is a technique that different from fixed-size chunking, which generate the blocks by predetermined size. Sliding window technique solves the problem arising from fix-sized chunking that inserting bits will permanently change the chucking blocks after the insert position. In sliding window, the chunking block where the insert bits are located in will change, while other chunks maintain. As we can see in Segment of File 2 by sliding window, only chunk2 changes.

For example in figure above, we insert only 2 bits in File1 to get File2. We predinfine some sign bits(or we called pattern in code), which indicates where to cut a file into a chunk. Every time our last few bits of window match this pattern, we cut here and hash.

To construct the sliding window structure, we need 2 parameters to be set ahead: window length and pattern, which are determined by the size of the file. If the size of the binary file is less than 100000, we set the window length to be size divide by 100,and the pattern is ‘101’. Otherwise, window length is the size of file divided by 1000 and pattern is ‘010101’. In these combination of pattern and and window length, the number of chunks can be restricted between hundreds and thousands .Pattern is set empirically, that is based on the fact that arbitrary binary file has uniform distribution of 0 and 1. It is possible that given a particular unique binary file that no bits combination matches our pattern, the performance of deduplication is unpromising. But in most cases, it gets a good performance in practice.

2.2 Hash

In order to hash a string into hashcode, we use sha256 function in hashlib library in python. In general, given a paragraph of article or a chunk of binary code, hash function sha256 generates a unique 256 bits long output. Usually, we use an hexadecimal representation, which uses 64 digits (from 0 to 9, A to F) to represent 256 bits. Because hash function has a property that hash value are the same if inputs are exactly the same, and different from each other when inputs are slightly different, we apply it as a way to find the duplicate chunks. Hashed value is functioned as index while storing, retrieving, and deleting files.

2.3 Formation of list file and inventory

Every file stored in the locker has a list associated with this file, which is called list_filename.txt. However, only one inventory associated with this locker stored as inventory.txt in our program.

When we store our first file, we also create a inventory.txt file as a “dictionary” type of our locker. The key of this dictionary is hash value we compute for each segment, and the value is the file name which also contains this segment.

After the segmentation of the file, besides every segment named by its hash value is stored in the locker, the list file associated with this file is stored in the locker too.

Then we store this segment in the pointed path(lockers folder), hashcode as filename. This function will also create a list file that contains all the hashcode of paragraphs in the current file which will be used in the retrieve and delete of the files.

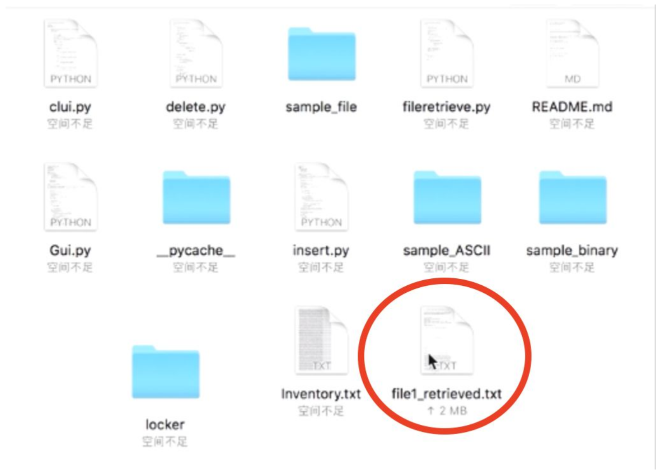
In follow-up files, we do hash function and renew the inventory. If the hash code is in the inventory then we add another value with respect to the key ,otherwise we create a new hashcode-filename pair and store the segment contents in a new file named with its hash code in the lockers.

2.4 Retrieval

The retrieve function takes two variables: the list of the file you want to retrieve and the path to the locker. The type of file and path are string.

First, the program goes to the locker and find the list using the first variable file. Then the program picks parts from the locker according to the list and connects all parts together to recover the original article.

The retrieved file will be named by filename_retrieved and stored as text file. And the retrieved file will will be stored at the folder where the program in.



2.5 Deletion

The deletion function takes two variables: the list of the file you want to retrieve and the path to the locker. This is same as retrieve.

First, the program goes to the locker and find the list. Then the program will transfer the ‘inventory.txt’ to a dictionary. The program will check every part on the list, to see if this file is this part’s only related file, according to the inventory. If this part is only related to the file to be deleted, then this part should be deleted. Also, it should be removed from the Inventory too. Else, this part can’t be deleted because it is also a part of other files. But the file name should be removed from the list of related files of the corresponding part on the Inventory.

After the deletion, the Locker and the Inventory will be updated, and the list of that deleted file will be removed.

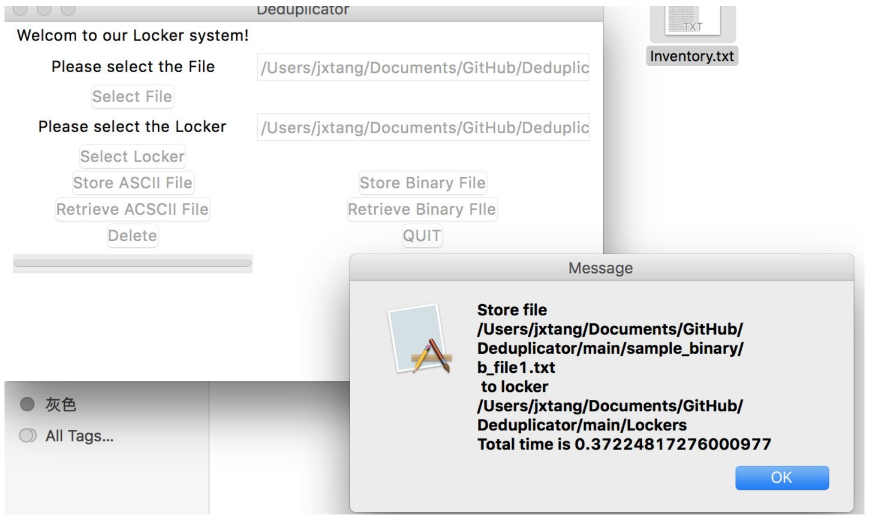
```
delete -file file1.txt -path locker/

remove locker/6392d3f942369d33d666433d8
cb73883e42ab93c96efa7ad5f319da4aee3c603
.txt
remove locker/c8162ba044bfa7a3c74189960
f0b6e6a6cfa10258796ad4bbee1af6361ffb67f
.txt
file1.txt deleted
file1.txt is deleted successfully.
```

The two lines of ‘remove…...’ denotes that the two parts that only related to file1 has been removed. ‘File1.txt deleted’ denotes the list_file1.txt has been removed. The last line denotes the functions

2.6 GUI

The GUI is Implemented by python package tkinter. We use file dialog to choose file and the location of the locker. Different buttons are linked to different functions to process files. For the progressbar, we use canvas to display the progress in percentage and the progressbar in ttk to show the progress. Besides, the speed of our process can be really fast, normally less than 0.3 second, so we decide to update the progress bar based on the length of the file. If the file is small, which means the process time is short, the progress bar will be updated three to ten times. Also, we create a seperate thread to make sure the update won’t affect much running time.



Welcom to our Locker system!

Please select the File /Users/jxtang/Documents/GitHub/Deduplic

Please select the Locker /Users/jxtang/Documents/GitHub/Deduplic

Store file
/Users/jxtang/Documents/GitHub/
Deduplicator/main/sample_binary/
b_file1.txt
to locker
/Users/jxtang/Documents/GitHub/
Deduplicator/main/Lockers
Total time is 0.37224817276000977

3. Result

3.1 Storing

We used 10 2MB articles as our ASCII test samples and (binary).

In ASCII testing, after the first file is stored, the storage usage is a little more than 2MB, because the articles we are using have no repeat content in a single file. So we need to store the entire file. We also created the inventory and the list of the file, it also take some storage. After we stored all 10 files, any two of which differ in at most 5 character edits (character edits are insertions of a character, deletion of a character, or modification of a character), the total storage usage is 3MB. We managed to use 3MB to store 20MB files.

In binary files testing, we test binary files with 10MB. After storing, we successfully store all of them in a locker with only 4.7 M. The reason why the storage is even smaller than one binary file is that we copy and paste the combination of 0 and 1 in a same file in order to create a 10M binary file quickly, so every same segment in one binary file won't store again. Thus, we have significantly compression rate if there are many replica of some snippets in one file. (Details in Test & Result part in presentation PPT)

3.2 No live state required

Stopping and quitting our program would not affect its functions. Because, we store the lists and the inventory after every functions is run. Every time before any function is ran, the lists and the inventory will be restored.

3.3 Retrieving

The retrieved files are identical to the original ones. Detail compare are discussed in unit test. The retrieved file is exactly the same with the original file using our own test file.

3.4 Deletion

Any part that related to other files would not be deleted by mistake. The parts that only related to the file to be deleted will be deleted. The inventory is updated correctly.

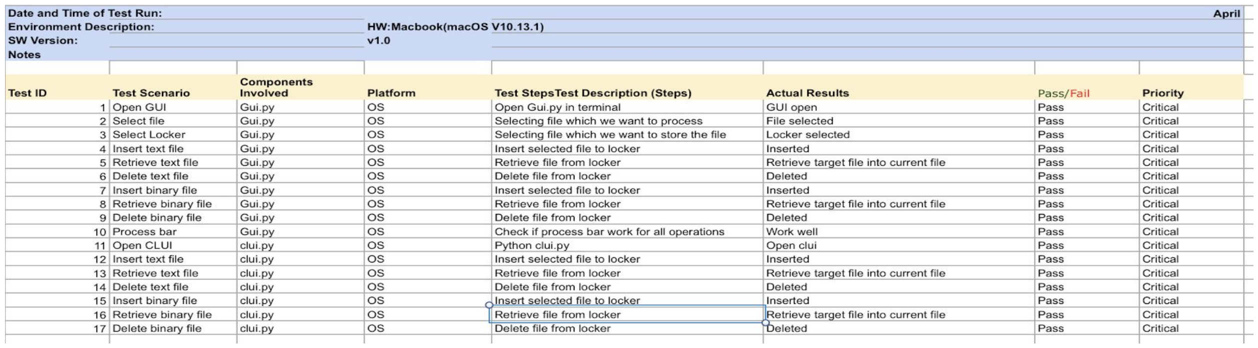
3.5 Command line user interface

All functions can be called in the UI. Incorrect inputs, including illegal command, incorrect format and nonexistent file and path, will not be accepted and notification will be printed on the terminal.

3.6 Unit test result

In the project, we also applied unittest to test all function of our deduplicator, including file insertion, file retrieval and file deletion. Result indicates that our system can pass these tests.

3.7 System test result



Date and Time of Test Run:							**April**
Environment Description:			**HW:Macbook(macOS V10.13.1)**				
SW Version:			**v1.0**				
Notes							
Test ID	**Test Scenario**	**Components Involved**	**Platform**	**Test StepsTest Description (Steps)**	**Actual Results**	Pass/Fail	**Priority**
1	Open GUI	Gui.py	OS	Open Gui.py in terminal	GUI open	Pass	Critical
2	Select file	Gui.py	OS	Selecting file which we want to process	File selected	Pass	Critical
3	Select Locker	Gui.py	OS	Selecting file which we want to store the file	Locker selected	Pass	Critical
4	Insert text file	Gui.py	OS	Insert selected file to locker	Inserted	Pass	Critical
5	Retrieve text file	Gui.py	OS	Retrieve file from locker	Retrieve target file into current file	Pass	Critical
6	Delete text file	Gui.py	OS	Delete file from locker	Deleted	Pass	Critical
7	Insert binary file	Gui.py	OS	Insert selected file to locker	Inserted	Pass	Critical
8	Retrieve binary file	Gui.py	OS	Retrieve file from locker	Retrieve target file into current file	Pass	Critical
9	Delete binary file	Gui.py	OS	Delete file from locker	Deleted	Pass	Critical
10	Process bar	Gui.py	OS	Check if process bar work for all operations	Work well	Pass	Critical
11	Open CLUI	clui.py	OS	Python clui.py	Open clui	Pass	Critical
12	Insert text file	clui.py	OS	Insert selected file to locker	Inserted	Pass	Critical
13	Retrieve text file	clui.py	OS	Retrieve file from locker	Retrieve target file into current file	Pass	Critical
14	Delete text file	clui.py	OS	Delete file from locker	Deleted	Pass	Critical
15	Insert binary file	clui.py	OS	Insert selected file to locker	Inserted	Pass	Critical
16	Retrieve binary file	clui.py	OS	Retrieve file from locker	Retrieve target file into current file	Pass	Critical
17	Delete binary file	clui.py	OS	Delete file from locker	Deleted	Pass	Critical

4. Reference

1. Data Deduplication for Data Optimization for Storage and Network Systems. Daehee Kim • Sejun Song • Baek-Young Choi
2. *Study of Chunking Algorithm in Data Deduplication*. A. Venish and K. Siva Sankar https://www.researchgate.net/publication/287251603_Study_of_Chunking_Algorithm_in_Data_Deduplication
3. *A Study of Practical Deduplication* Dutch T. Meyer and William J. Bolosky
4. Tkinter example http://python-textbok.readthedocs.io/en/1.0/Introduction_to_GUI_Programming.html
5. Description of SHA-256, SHA-384,SHA-562

5. Acknowledge

Thanks professor kulis for giving advice on our project, and thanks all people who helped us in the project.

Appendix

GitHub link: https://github.com/GanquanWen/Deduplicator
For the supporting files include instruction on how to use our program, the example of using our program, unit tests and system test, please check our GitHub. We have detail instructions and videos on how to use and examples. The singed work breakdown is in our GitHub as well.