Deduplicated file system on object store using public cloud

COEN 241, Fall 2016

Johnny Bai, Lakshmi Manasa Velaga and Satya Keerthi Chand Kudupudi
Santa Clara University
Email: {jbai1, lvelaga, skudupudi}@scu.edu

Preface

As students pursuing a Master of Science degree in computer engineering, we are expected to be able to not only find and understand relevant research papers, but also to contribute insights of our own. However, under current time constraints, it is unlikely that we will be able to contribute original findings. In this paper, we will first describe a data deduplication scheme which will effectively reduce the amount of storage required to store data, and then provide an implementation of the scheme using Java and Amazon Web Services (AWS). By investigating data deduplication techniques in the cloud, we hope to gain a better understanding of how cloud computing and services work at a more detailed, complex and un-abstracted level.

Abstract

Cloud storage is one of the major services provided in cloud computing which has been increasing in popularity. The main advantage of using cloud storage from the customer's point of view is that he/she can reduce their expenditure in purchasing and maintaining storage infrastructure while only paying for the amount of storage requested, which can be scaled-up and down upon demand. With the growing data size of cloud computing, a reduction in data volumes could help providers reducing the costs of running large storage system and saving energy consumption. So data deduplication techniques have been brought to improve storage efficiency in cloud storages. In this paper, we are using deduplication technique which utilizes file segmentation, file compression, and hashing to achieve a dynamic deduplication scheme for cloud storage, using Rabin fingerprinting for dynamic chunking and uploading files to Amazon S3.

Acknowledgements

We would like to take this opportunity to express our profound gratitude to Professor Ming Hwa Wang, for the valuable feedback and constant encouragement throughout the duration of the project.

List of Figures:

List of Tables:

Table of Contents

1. Introduction

Cloud computing describes a computing paradigm in which a set of computing resources are made available via the internet (Hwang, et al.). Furthermore, these computing resources are available on-demand, scales rapidly to meet consumer demands, making them virtually unlimited, and costs depend on usage. The appeal of cloud services stem from its low entry cost for a reliable set of resources and services that support a wide range of applications.

One of the most popular cloud services is the data storage service (e.g., Amazon S3, Google Drive, and Dropbox). As aforementioned, cloud services follow a utility-based pricing model in which consumers pay for what they use. Although storage is relatively cheap and virtually unlimited, data duplication "wastes network resources, consumes a lot of energy, and complicates data management" (Yan, et al.). Ultimately, data deduplication aims to reduce cost by minimizing the amount of duplicate data that exists in the cloud.

Another consideration to make when implementing a deduplication scheme is computing cost. In general, deduplication involves two steps, each of which can be expanded upon. First, the data must be analyzed to generate a fingerprint that represents the file's contents. Second, the file's fingerprint must be compared to other fingerprints. If a match is found, the file is deemed to be duplicate, thus requiring no upload. Conversely, if no match is found, the file proceeds to be uploaded. Regardless of whether or not the consumer chooses to perform these actions in the cloud or on a local machine, the computations should be performed in a timely manner to prevent delays.

The data deduplication scheme we propose to implement is based off the one presented by Upadhyay, et al. and utilizes file segmentation, file compression, and hashing to achieve deduplication. While Upadhyay, et al. proposed a viable architecture for deduplication, they

failed to meet our standards in two crucial aspects: (1) the method to fragmentize a file; and (2) a redundancy system to prevent or minimize data downtime. It is our goal to implement a deduplication protocol that addresses these concerns. We will explore the advantages and disadvantages between fixed-length and variable-length data fragmentation, and the effects a redundancy system has on cloud resource usage.

2. The Problem

Data duplication refers to the presence of multiple copies of identical data. For the purposes of our investigation, the term data strictly refers to the contents of the files (i.e. metadata, file attributes, and file names will be ignored). If data is excessively duplicated beyond reasonable standards (e.g. more copies than needed to create a redundancy system), it is a blatant waste of storage space, and possibly network bandwidth if data is not stored locally.

Data deduplication is the process of discovering and preventing the existence of excessively duplicated data. In the simplest case, a deduplication scheme simply prevents two identical files from existing in the storage device. However, a more sophisticated scheme is needed to have make more efficient use of storage. Instead of addressing the issue at the file level, the file can be partitioned into segments.

There exists an inherent issue when implementing segment-level deduplication: the size of the segments. There is a trade-off between segment length and performance. Having shorter segment lengths give us higher resolution when performing pattern matching, but also increase the number of segment files needed to be transferred. Depending on the file system and the storage hardware, the overhead of transferring multiple small files may incur a significant performance penalty as opposed to a single larger file. One of the challenges will be in using segment lengths

that are within a range that will minimize duplicates without sacrificing performance or being computationally expensive.

3. Related Work

The framework for our deduplication protocol is based on the architecture presented by Upadhyay, et al. Their solution is comprised of three components: (1) segmentation; (2) deduplication and compression; and (3) file retrieval. Our proposed solution will include these components, a redundancy system, and incorporate Amazon's Simple Storage Service (S3) and Elastic Compute Cloud (EC2) to fulfill our goals.

The first step in the segmentation process is to split file into segments. The issue with fixed-length segmentation is in its inability to detect a duplicated segment if the segment in question spans two segments in another file. To address this concern, variable-length segmentation techniques will be explored.

Rabin fingerprinting:

The rabin fingerprinting algorithm calculates a rolling checksum over data (a file to store in S3). The window of the data to look at is configurable, but typically a few dozen bytes long. The rabin module will read through a file, and let the window "slide" over the data, recalculating the fingerprint each time when advancing a byte. When the fingerprint assumes a special value, the rabin module considers the corresponding window position to be a boundary. All data preceding this window position is taken to be a "block" of the file.

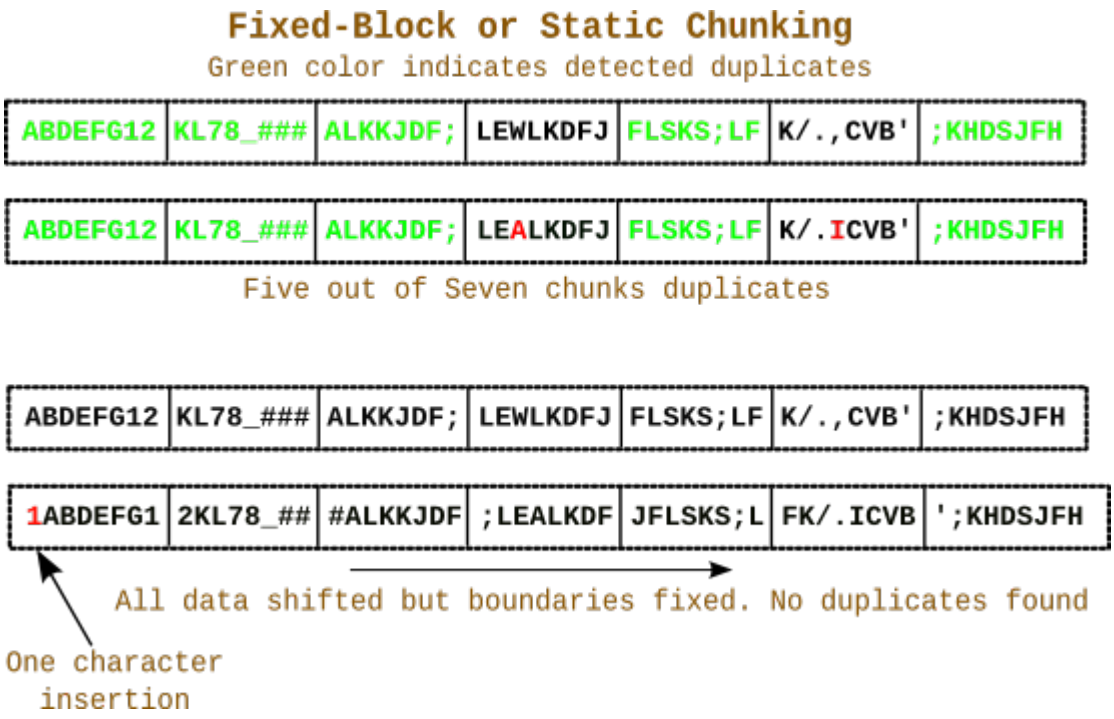


Figure 1: Example of pattern matching using fixed length segments(Ghosh)

Calculating a checksum relative to a previous one is not very expensive cpu-wise. The operations involved are:

1. Multiplying the previous checksum by a prime (the prime is a parameter of the algorithm).
2. Adding the value of the byte that has just come into the sliding window.
3. Subtracting the value of the byte that has just slid out of the window times the nth power of the prime (where n is the width of the sliding window; the 256 possible values are precalculated during module initialization).
4. Taking the modulo of the value by the average desired block size (for now, the average desired block size is required to be a power of two).
5. Check whether the new checksum has the special value that makes it a boundary.

This means the algorithm has three parameters: prime, width and modulo. The properties that make this algorithm useful for our purpose is that: 1. they are cheap to calculate; 2. they find the same boundaries, no matter where in the file they occur. Thus, when a byte has been prepended to or inserted into a file, this has no influence on how later blocks are formed, as opposed to the case where all block boundaries are at fixed file offsets.

We can specify a minimum and maximum block size. If a block boundary occurs before the minimum block size is encountered, it is ignored. If no block boundary occurs before the maximum block size, the window is treated as a block boundary anyway and a new block emitted.

Our parameters:

As Amazon allows limited free blocks to be uploaded, we had to assign higher values Taking cost into consideration; we have set the minimum and maximum block size to 4096 bytes and 8192 bytes respectively in our implementation.

In order to achieve variable-length segmentation, the Rabin fingerprinting method will be employed (Karp and Rabin). The Rabin fingerprinting algorithm will generate fingerprints of a file over a sliding window that progresses one byte at a time (Sun, et al.). Of all the fingerprints generated for a given file, only a fraction of them are saved to represent the file. Since a rolling hash is performed, duplicate segments of various lengths can be discovered (Ghosh).

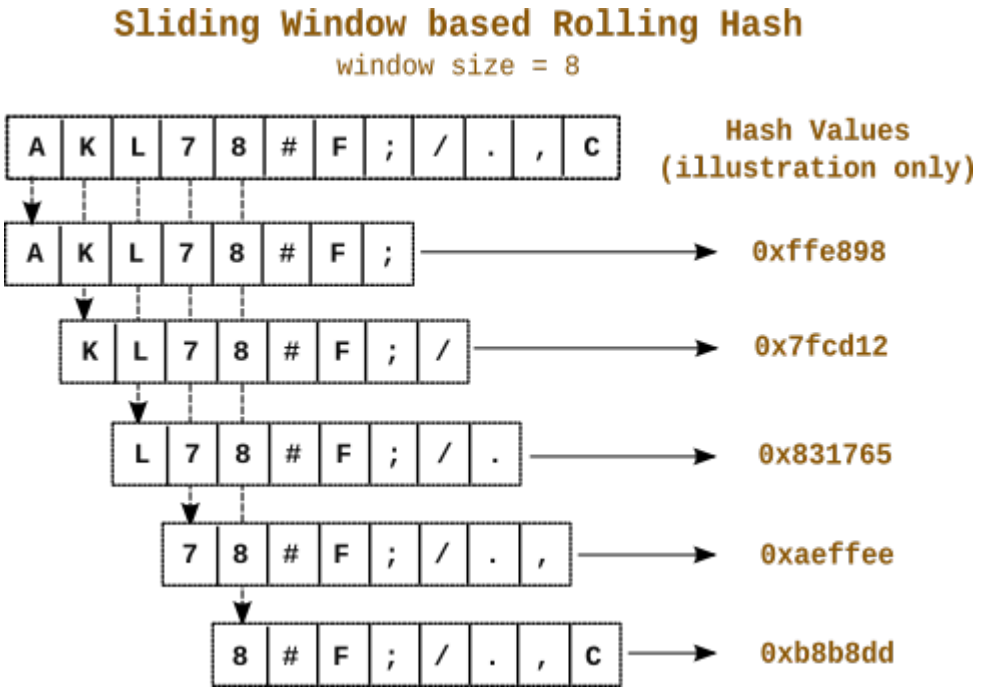


Figure 2: Example of rolling hash with a window size of 8(Ghosh)

After the files are segmented, each segment has its fingerprint computed, via a hash function. The hash value serves as the segment's unique identifier. When a new file is uploaded, each of its segments' hash values are compared to existing ones. If there is a match, the already uploaded segment will be linked to this new file and no upload will occur. If there is no match, the segment is compressed and uploaded. When a client requests a file, the mapping of the file to its list of segments is accessed. The compressed segments are sent to the client who then decompresses them and reconstructs the file.

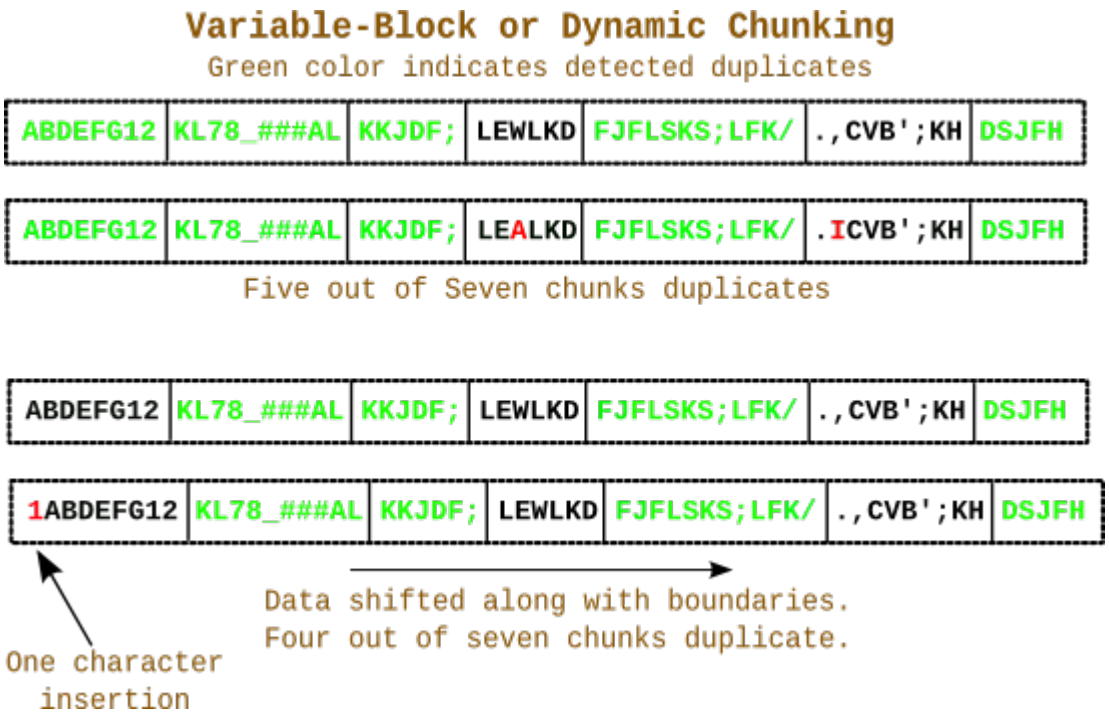


Figure 3: Example of pattern matching using variable length segments(Ghosh)

One concern when using hashing functions to generate fingerprints is collisions. A collision occurs when there are two distinct objects with different contents hash to the same value. Depending on the hash function used, the collision rate varies greatly. However, for a well-established and appropriately designed hash function, such as Secure Hash Algorithm (SHA), the probability of a collision occurring is low enough that we do not have to worry about it (van Bergen).

Our proposed solution is a modification of Upadhyay, et al.'s solution that uses Rabin fingerprinting to generate variable-length segments. Variable-length segments will detect duplicate segments at a higher rate than the restrictive fixed-length segments. Furthermore, our deduplication solution will incorporate Amazon's EC2 and S3 cloud services. Through Amazon's services, we will also be able to implement a redundancy system across multiple S3

instances. By monitoring usages before and after deduplication, we will be able to provide numerical data to show the effects deduplication has on usage.

4. Hypothesis

In this paper, our main goal is to build and test a complete deduplicated file system from scratch and implement a variable-length segmentation method for splitting the files into multiple segments. All the segments will be compressed before sending to cloud. Our performance goals are to achieve storage efficiency, decrease the data transfer costs, increase bandwidth utilization.

Storage efficiency is achieved by fine grain data deduplication and compression. Data transfer costs will be reduced due to deduplication being done on the server which in turn avoids sending duplicate segments to cloud by utilizing network bandwidth efficiently.

Variable-length deduplication is a method for breaking up a data stream via context-aware anchor points. This segmentation method provides greater storage efficiency for redundant data regardless of where new data has been inserted. In
variable-length deduplication method, the length of segments vary, thus achieving higher deduplication ratios.

Variable length deduplication gives better deduplication over fixed sized segmentation. For example, if small data is prepended to large file, fixed size segmentation will result in all the segments of a file to be unique whereas in variable size segmentation only first few segments will be different that results in high deduplication ratios.

5. Methodology

We have implemented a prototype of the deduplication system aiming to validate the effectiveness of the proposed mechanisms. Our implementation uses Java. Development and deployment is done on Elastic Compute Cloud (EC2) instances in AWS. We used Elastic Block Storage (EBS) to store all the segments associated with each file.

Our code takes files as input, which will be segmented using variable-length segmenting techniques. The Rabin fingerprinting algorithm will allow us to split the file at context-aware anchor points. After we split the file into variable length segments, we store the segments associated with each file and their lengths in mapping file. The mapping file is stored in EBS. Each line in mapping file stores fingerprint of segment and its length. Each segment will be stored as an object in Simple Storage Service (S3) provided by AWS. The unique fingerprint will be used as an object name when we store in S3.

Hash value (fingerprint) is calculated for each segment using the SHA-1 algorithm. The SHA-1 algorithm creates a 160 bit hash value for each segment. Hash values created are stored in an in-memory hash map.

This hash map is used to identify duplicated segments. In the hash map, the fingerprint is stored as key and value represents number of references that are pointing to this segment. Whenever a duplicate segment is encountered, we increment the value in the hash map with the corresponding matching fingerprint key. We are using the reference count method to keep track of number of pointers to each segment. When a reference count goes to zero, we will delete those objects from S3 to reduce storage space on unwanted segments.

File upload: When user wants to upload a file, file is split into segments and fingerprint is computed for each segment using above-mentioned process. We compare fingerprint of each and every segment in hashmap. If the same fingerprint exists, the value corresponding to matched fingerprint (key) is incremented by one. If fingerprint is not found in hashmap, we insert new fingerprint as key and value as one. All the segments associated with each file and with their lengths is stored in the mapping file on EBS. The deduplicated segments are then compressed using gzip and are transferred over the network to store as objects in the cloud.

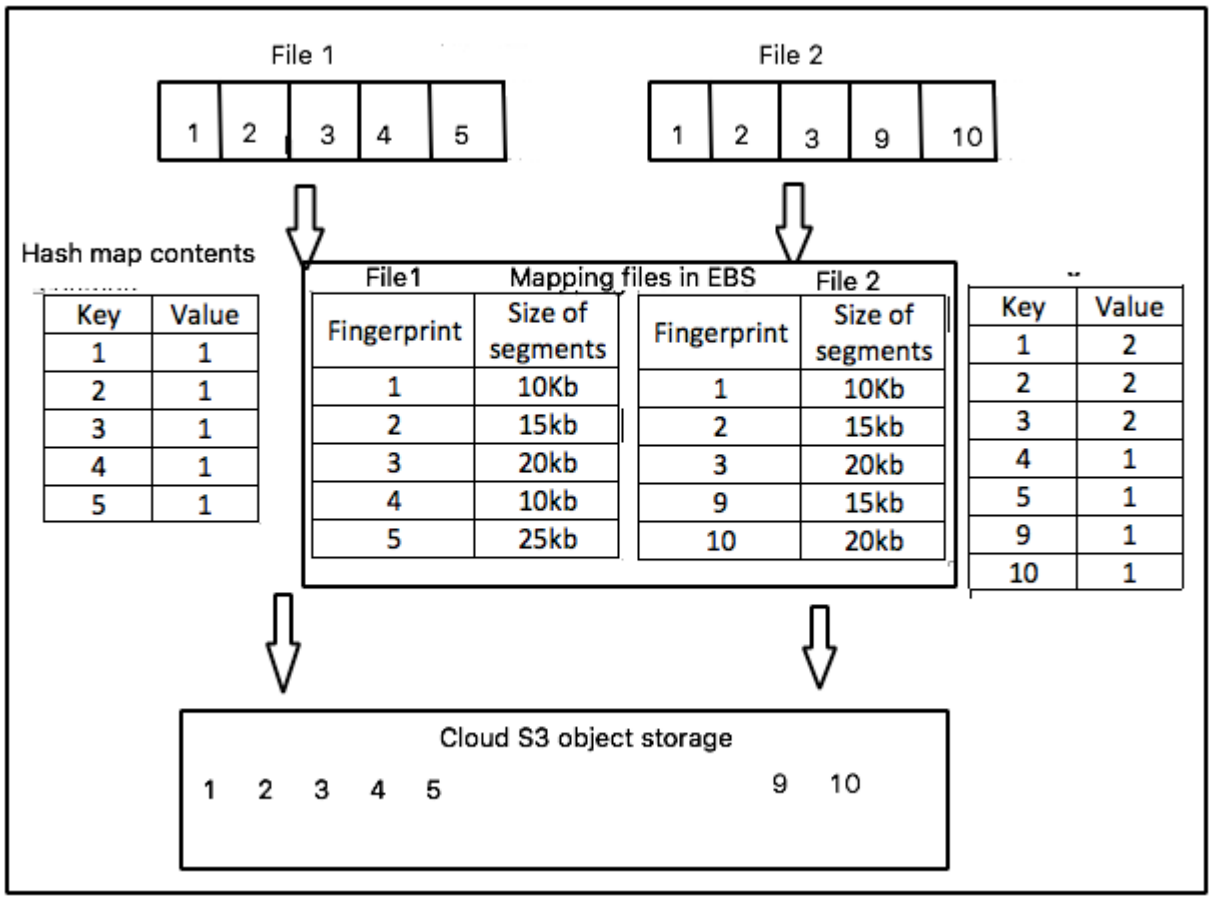


Figure 3: Process of file upload

File download: When user requests for a file, we will look for the mapping file on EBS to identify all the associated segments and their lengths. Each line in mapping file contains

fingerprint of segment and its length. We traverse through each line in mapping file and retrieve corresponding object from S3. After we get object over the network from cloud, we will decompress the segment. After we retrieve and decompress all the segments from mapping file, segments are put together to form the original file.

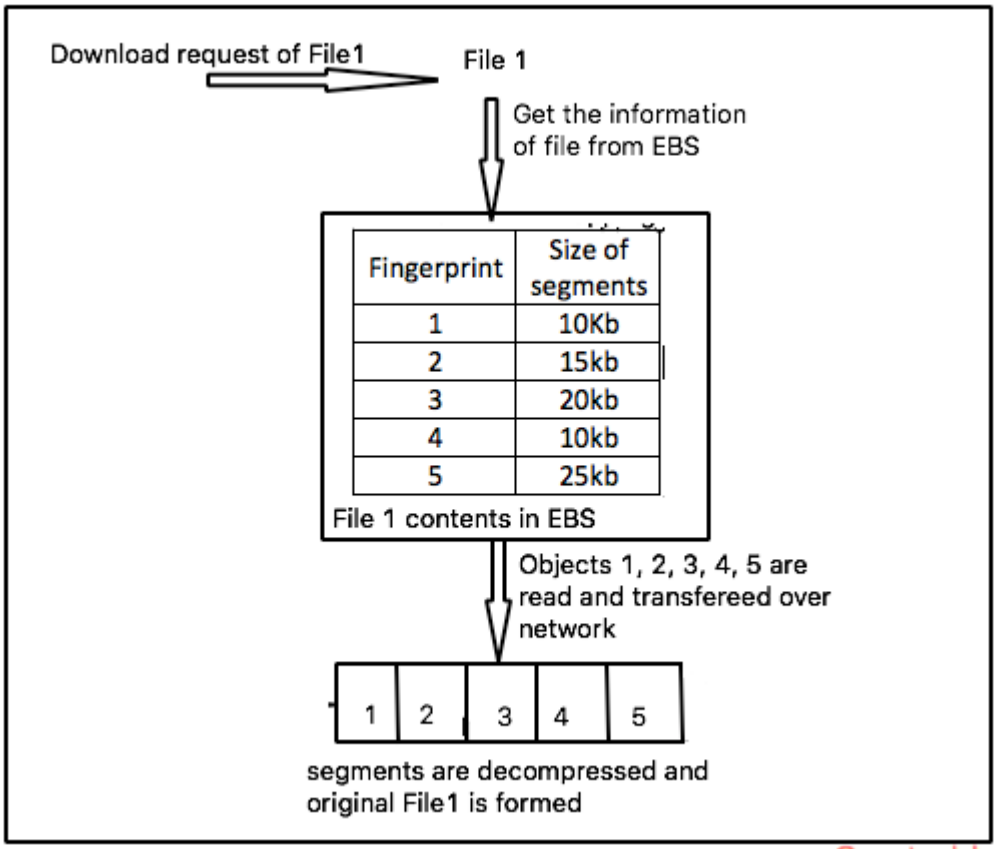


Figure 4: Process of file download

File deletion: When user wants to delete a file, we will look for the mapping file on EBS to identify all the associated segments. We will traverse through each line in mapping file and decrement value in in-memory hash map associated with each fingerprint. If for any fingerprint, reference count (value) reaches zero, we will delete that segment from cloud. Once all the segments associated with that file are adjudicated, we are deleting mapping file.

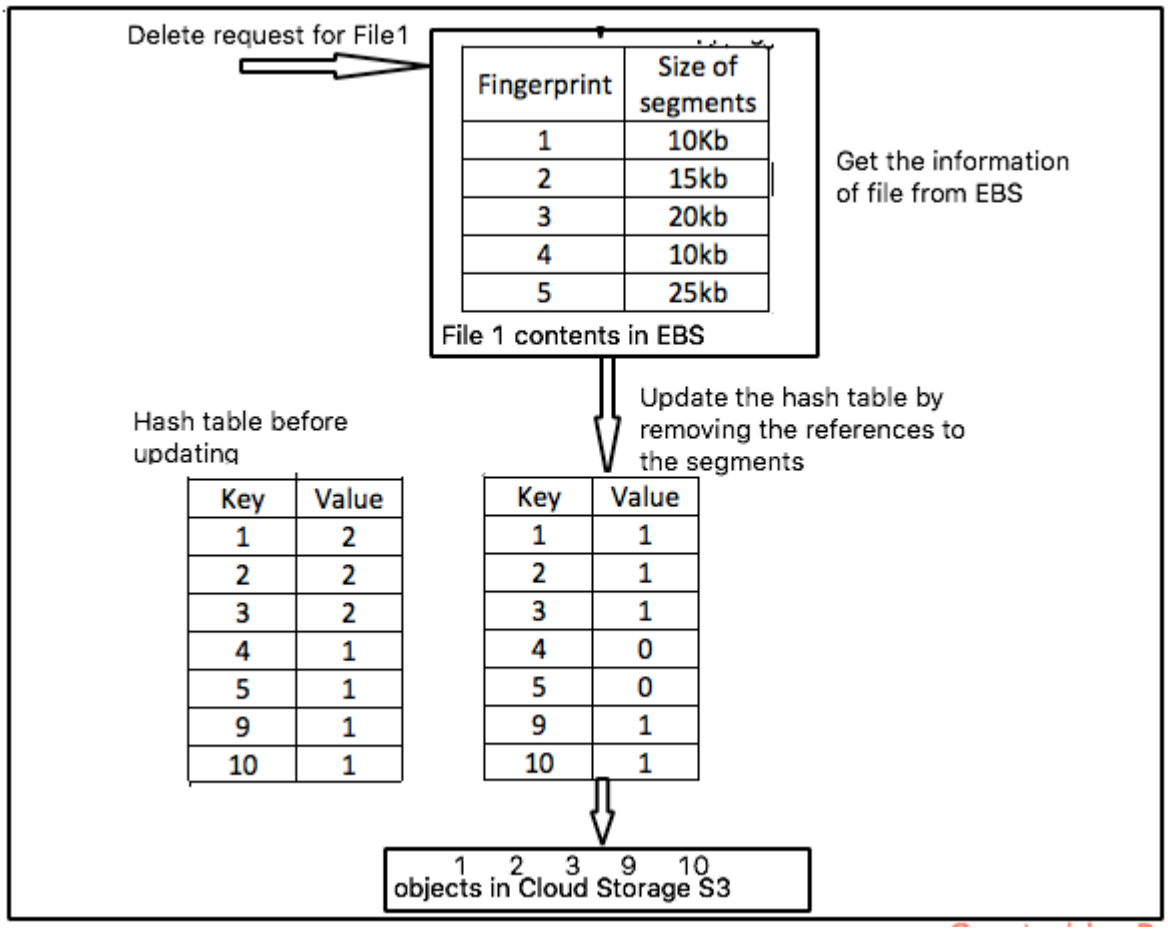


Figure 5: Process of file deletion

We can test storage efficiency by measuring the storage space of original file and number of segments stored in cloud after deduplication. If the storage space after using deduplication technique is less compared to the original file size, we can prove that our method improves storage efficiency.

6. Implementation:

Uploading a file: The flow chart for updating a file is shown in figure 7

Client:

Check for put argument and filename/path. If file exists, Create Chunks using the fingerprinting explained above with filenames as their Hashed value. Also, during this process we are creating

a .data file which contains the Hashed value of the chunks being created in a particular order. After chunking the contents of the file, we verify each chunk is present in the server or not. The functionality of server is discussed in the next subsection. If the server response is upload, it records the chunk's name to be uploaded, else if it is skip it ignores the chunk. The server then returns the set of all the Hashed values that are not present in the server. The .data file is uploaded to the server and the segments are then uploaded to S3.

Server:

When the server gets the request, it forwards the request to thread pool. The Server threads then take up the request from the client to process it. The server thread reads for the type of the request. If the request type is upload, it will check the hashed value of the segment. If the server doesn't have the segment, it will request for the client to transfer the segment with upload command to save it in S3 instance and increments the segment count in hash table. Else, it sends a skip command and increments the segment count in hash table. After the file is uploaded, the mapping file is requested and is saved.

Downloading a file: The flow chart for deleting a file is shown in figure 8.

Client:

Check for get argument and filename/path. If the command is a get, a request is sent to the server to check if the file is present in S3 or not. If the server responds "File does not exist!", it informs the user that file doesn't exist and client terminates. Otherwise, the chunk names of files are received from the server into a list (to keep the order in-tact). After receiving the chunk names from the server, the chunks are downloaded from Amazon S3. The file is then reconstructed.

Server:

When the server gets the request, it forwards the request to thread pool. The Server threads then take up the request from the client to process it. The server thread reads for the type of the request. If the request type is get, it will check if the file (containing the chunks' details) exists in the Server repository or not. If the file is not present, it will inform the user "File does not exist!", else the server returns that the file is found and sends the chunk names of the file to be downloaded from S3.

Deleting a file: The flow chart for deleting a file is shown in figure 9.

Client:

Check for get argument and filename/path. If the command is a delete, a request is sent to the server to check if the file is present in S3 or not. If the server responds "File does not exist!", it informs the user that file doesn't exist and client terminates.

Server:

When the server gets the request, it forwards the request to thread pool. The Server threads then take up the request from the client to process it. The server thread reads for the type of the request. If the request type is delete, it will check if the file (containing the chunks' details) exists in the Server repository or not. If the file is not present, it will inform the user "File does not exist!", else it gets the file in the server. For each segment of the file, the count is decremented in the hashtable and if the value is zero, the segment is deleted from the S3.

Server-Side

EC2Server

This class is intended to be instantiated on the server machine, which is an AWS EBS-backed EC2 instance in our case. This class maintains a hash map with segment names as the

keys and reference count as values. The hash map object is saved in the form of a serialization file in order to survive server restarts. The bulk of the work is performed by EC2ServerThread; a new EC2ServerThread is created for each client request.

Method	**Behavior**
saveMap()	Saves the HashMap<String, Integer> object, which maps segment name to reference count as the serialized file "segmentMapping.ser". Called whenever a EC2ServerThread instance finishes executing.
loadMap()	Loads the HashMap<String, Integer> object from the "segmentMapping.ser" file. Called whenever the EC2Server starts, e.g. when we start the server.
putSegment()	Puts a segment into our mapping. If the segment exists, we increment the reference count.
removeSegment()	If the segment (key) maps to a value greater than 1, this method decrements the value, otherwise, deleteSegment() is called.
deleteSegment()	Removes the segment (key) from the hash map and deletes the corresponding file from S3 buckets.
hasSegment()	Returns true if the specified segment is present in the hash map, otherwise returns false.

Table 1: Methods in EC2Server class

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.util.HashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * Intended to be run on the EC2 instance
 */
public class EC2Server {
```

```
	static final String saveDest = "./SERVER/";

	private ServerSocket serverSocket;
	private String address;
	private int port;

	private volatile static HashMap<String, Integer> segmentToCount = null;
	private ExecutorService threadPool;

	public EC2Server() {
		try {
			serverSocket = new ServerSocket(0); // automatically find a port
			port = serverSocket.getLocalPort();
			address = serverSocket.getInetAddress().getHostAddress();
		} catch (IOException e) {
			System.out.println(e);
		}
		System.out.println("Server is up: " + address + " : " + port);
		segmentToCount = loadMap(); // Load up the mappings when this runs
		printSegments(); // Uncomment this if you wish to see what segments get
								// loaded when server starts
		threadPool = Executors.newCachedThreadPool();
	}

	/**
	 * Debug purposes, prints the set of keys after loading serialized
	 * hash file.
	 */
	private static void printSegments() {
		for (String key : segmentToCount.keySet()) {
			System.out.println(key + ": " + segmentToCount.get(key));
		}
	}

	// Save our mappings after each thread halts
	public static void saveMap() {
		if (segmentToCount != null) {
			File mapFile = new File(saveDest + "segmentMapping.ser");
			try (FileOutputStream fileOut = new FileOutputStream(mapFile);
					ObjectOutputStream objectOut = new
ObjectOutputStream(fileOut)) {
				objectOut.writeObject(segmentToCount);
			} catch (IOException e) {
				System.out.println(e);
			}
```

```
		}
	}

	// Loads our mappings
	private synchronized static HashMap<String, Integer> loadMap() {
		HashMap<String, Integer> map = null;
		File mapFile = new File(saveDest + "segmentMapping.ser");
		if (mapFile.exists()) {
			try (FileInputStream fileIn = new FileInputStream(mapFile);
					ObjectInputStream objectIn = new
ObjectInputStream(fileIn)) {
				map = (HashMap<String, Integer>) objectIn.readObject();
				return map;
			} catch (IOException e) {
				System.out.println(e);
			} catch (ClassNotFoundException e) {
				System.out.println(e);
			}
		}
		return new HashMap<String, Integer>();
	}

	// String segment is the segment file
	public synchronized void putSegment(String segment) {
		if (segmentToCount.containsKey(segment)) {
			int count = segmentToCount.get(segment);
			segmentToCount.replace(segment, count + 1);
		} else
			segmentToCount.put(segment, 1);
	}

	public synchronized void removeSegment(String segment) {
		if (segmentToCount.containsKey(segment)) {
			int count = segmentToCount.get(segment);
			if (count == 1) {
				// delete this segment
				deleteSegment(segment);
			} else
				segmentToCount.replace(segment, count - 1);
		}
	}

	private synchronized void deleteSegment(String segment) {
		segmentToCount.remove(segment); // remove the segment
		MyS3Client client = new MyS3Client();
		// Delete objects from S3
```

```
            client.deleteSegment(segment + ".zip");
        }

        public synchronized boolean hasSegment(String segment) {
            return segmentToCount.containsKey(segment);
        }

        public static void main(String[] args) {
            File file = new File(saveDest);
            if (!file.exists()) {
                file.mkdir();
            }
            EC2Server ec2Server = new EC2Server();
            try (ServerSocket serverSocket = ec2Server.serverSocket) {
                while (true) {
                    ec2Server.threadPool.submit(new EC2ServerThread(ec2Server,
serverSocket.accept()));
                }
            } catch (IOException e) {
                System.out.println(e);
            }
        }
}
```

EC2ServerThread

This class handles the interactions between the server and the client. Specifically, this class is a thread that is created every time a client connects to the server via its IP address and port number. The thread will handle the necessary actions to satisfy put, get, and delete requests from clients.

Method	**Behavior**
receiveFile()	Reads a stream of bytes from the client and saves it into a file.
run()	Standard method for a thread; handles client requests.
run(): input = "duplicate check"	When a client sends a set of segment names to the server, the server tells the client which segments do not already exist in S3.

run(): input = "upload"	When a client sends the file metadata over, the server downloads the file from the client and saves it to the specified server directory.
run(): input = "get"	When the client sends a request to get a file, the server opens the file's metadata and sends the client an ordered list of segments that the file is comprised of.
run(): input = "delete"	When the client sends a request to delete a file, the server opens the file's metadata and calls EC2Server.removeSegment() on each segment.

Table 2: Methods in EC2ServerThread class

```
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class EC2ServerThread extends Thread {

	private EC2Server server;
	private Socket socket;

	public EC2ServerThread(EC2Server server, Socket socket) {
		this.server = server;
		this.socket = socket;
	}

	/**
	 * The method that downloads the file the client is uploading.
	 */
	public static void receiveFile(String savePath, int fileLen, InputStream inputStream) {
		try (
			ByteArrayOutputStream baos = new ByteArrayOutputStream(fileLen);
			FileOutputStream fos = new FileOutputStream(savePath);
			BufferedOutputStream bos = new BufferedOutputStream(fos)
			) {
			byte[] aByte = new byte[1];
			int bytesRead = inputStream.read(aByte, 0, aByte.length);
```

```
                do {
                        baos.write(aByte);
                        bytesRead = inputStream.read(aByte);
                } while (bytesRead != -1);
                bos.write(baos.toByteArray());
                bos.flush();
        }
        catch (IOException e) {
                System.out.println(e);
        }
}

public void run() {
        // Runs when a connection is made
        try (
                PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
                BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                        socket.getInputStream()));
                ) {
                String input, output;
                input = in.readLine();
                // If client needs to determine which segments to upload
                if (input.equals("duplicate check")) {
                        // We are expecting a series of "check segment" lines
                        while ((input = in.readLine()) != null) {
                                if (!server.hasSegment(input)) {
                                        out.println("upload");
                                        server.putSegment(input);
                                }
                                else out.println("skip");
                        }
                }
                // If client is uploading a file
                if (input.startsWith("upload")) {
                        String[] split = input.split(" ");
                        String fileName = split[1];
                        System.out.println("FILE NAME: " + fileName + ", length: " +
split[2]);
                        receiveFile(EC2Server.saveDest + fileName,
Integer.parseInt(split[2]), socket.getInputStream());
                        // ./SERVER/fileName
                }
                // If client wants to get a file
                if (input.startsWith("get")) {
                        String[] split = input.split(" ");
```

```
                        String fileName = split[1];
                        File file = new File(EC2Server.saveDest + fileName + ".data");
                        if (!file.exists()) {
                                out.println("file: " + file.getName() + " does not exist!");
                        }
                        else {
                                out.println("File exists");
                                try (
                                        BufferedReader br = new BufferedReader(new
FileReader(file))
                                        ) {
                                        String line;
                                        while ((line = br.readLine()) != null) {
                                                out.println(line);
                                        }
                                }
                                catch (IOException e) {
                                        System.out.println(e);
                                        e.printStackTrace();
                                }
                        }
                }
                // If clients wants to delete a file
                if (input.startsWith("delete")) {
                        String[] split = input.split(" ");
                        String fileName = split[1];
                        File file = new File(EC2Server.saveDest + fileName + ".data");
                        if (!file.exists()) {
                                out.println("file does not exist!");
                        }
                        else {
                                System.out.println("Starting to delete: " + fileName);
                                out.println("Deleting file");
                                try (
                                        BufferedReader br = new BufferedReader(new
FileReader(file));
                                        ) {
                                        String line;
                                        while ((line = br.readLine()) != null) {
                                                String segName = line.split(",")[0];
                                                server.removeSegment(segName);
                                        }
                                        if (file.delete()) {
                                                System.out.println("Deleted metadata file: "
+ file.getName());
                                        }
```

```
                                        }
                                        catch (IOException e) {
                                                System.out.println(e);
                                                e.printStackTrace();
                                        }
                                }
                        }
                }
                catch (IOException e) {
                        System.out.println(e);
                        e.printStackTrace();
                }
                EC2Server.saveMap();
        }

}
```

Client Side

DedupClient

This class is intended to run on any local machine. The client is anyone who is attempting to either upload a file to S3, obtain a file previously stored in S3, or delete a file that is currently stored in S3. The client is responsible for segmenting files and creating the metadata file corresponding to the file and its segments. The metadata file is to be stored on the EC2 server, and unique segments will be stored on the S3 buckets. To get a file from storage, the client first obtains a list of segments that comprise the file. From the list, a set of unique segments is obtained to be used to download the necessary segments from S3. Once those segments are obtained, the client reconstructs the file. An instance of MyS3Client is used to facilitate interactions between the client and S3 buckets.

Method	**Behavior**
prepareSegments()	Reads the file's metadata and returns a set of unique segments.

getFilesToUpload()	Communicates with the server to determine which segments in a list of segments need to be uploaded. Returns a set of segments that need to be uploaded.
sendFile()	Sends the specified file to the server. Although this method can send any file, we use it only to send files' metadata.
getFileSegments()	Communicates with the server to determine the list of segments that comprise of the specified file. Returns a list of segments that correspond to the file.
deleteFile()	Sends a request to delete a file to the server.
compressFile()	Uses Java's native zip package to compress a file.
decompressFile()	Decompresses the file specified by zipFile to SAVE_DIR/fileName/saveName
reconstructFile()	Rebuilds the file after obtaining segments from server/S3. The segments are deleted after the reconstruction.

Table 3: Methods in DedupClient class

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;
```

```
import java.util.zip.ZipOutputStream;

public class DedupClient {

	static final String SAVE_DIR = "./FROM_SERVER/";

	/**
	 * Segmentation and finger-printing of files should return a list of segment
	 * names (e.g. finger prints). A new file should be created that contains a
	 * list of segments to reconstruct the file.
	 */

	/**
	 * The metadata file contains information on the number of segments a file
	 * is split into and the segments that make up the file.
	 */
	public static HashSet<String> prepareSegments(File metadata) {
		HashSet<String> segments = new HashSet<>();
		try (BufferedReader br = new BufferedReader(new FileReader(metadata))) {
			String line;
			while ((line = br.readLine()) != null) {
				List<String> arr = Arrays.asList(line.split(","));
				segments.add(arr.get(0));
			}
		} catch (IOException e) {
			System.out.println(e);
		}
		return segments;
	}

	/**
	 * Input is a set of segments/finger prints. It communicates with EC2 to
	 * determine which need to be uploaded. Returns a set of segments that are
	 * unique and thus need to be uploaded.
	 */
	public static HashSet<String> getFilesToUpload(HashSet<String> segments, String
address, int port) {
		HashSet<String> toUpload = new HashSet<String>();
		try (Socket socket = new Socket(address, port);
				PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
				BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()))) {
			out.println("duplicate check");
			System.out.println("Connected!");
			String input;
			for (String segment : segments) {
```

```
                        out.println(segment);
                        if ((input = in.readLine()) != null && input.equals("upload")) {
                                toUpload.add(segment);
                        }
                }
        } catch (IOException e) {
                System.out.println(e);
                e.printStackTrace();
        }
        return toUpload;
}

/**
 * This method sends the file to the server at the specified address and
 * port.
 *
 * @param file
 * @param address
 * @param port
 */
public static void sendFile(File file, String address, int port) {
        try (Socket socket = new Socket(address, port);
                        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
                        FileInputStream fis = new FileInputStream(file);
                        BufferedInputStream bis = new BufferedInputStream(fis);
                        BufferedOutputStream toServer = new
BufferedOutputStream(socket.getOutputStream());) {
                // upload filename length
                out.println("upload " + file.getName() + " " + file.length());
                byte[] fileByteArray = new byte[(int) file.length()];
                bis.read(fileByteArray, 0, fileByteArray.length);
                toServer.write(fileByteArray, 0, fileByteArray.length);
                pause(1); // Note: need to sleep for byte array to populate
                toServer.flush();
        } catch (IOException e) {
                System.out.println(e);
                e.printStackTrace();
        }
}

/**
 * Returns an array of segments to reconstruct our file
 */
public static ArrayList<String> getFileSegments(String fileName, String address, int
port) {
        try (Socket socket = new Socket(address, port);
```

```
                        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
                        BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()))) {
                out.println("get " + fileName); // Tell server which file we want
                String input;
                input = in.readLine();
                if (input.endsWith(" does not exist!")) {
                        System.err.println(fileName + " does not exist!");
                        System.exit(1);
                } else {
                        // Build our array of segments
                        ArrayList<String> arr = new ArrayList<String>();
                        String line;
                        while ((line = in.readLine()) != null) {
                                List<String> list = Arrays.asList(line.split(","));
                                arr.add(list.get(0));
                        }

                        return arr;
                }
        } catch (IOException e) {
                System.out.println(e);
                e.printStackTrace();
        }
        return null;
    }

    /**
     * Deletes the specified file from the server and S3 buckets
     */
    public static void deleteFile(String fileName, String address, int port) {
        System.out.println("Deleting: " + fileName);
        try (Socket socket = new Socket(address, port);
                        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
                        BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()))) {
                out.println("delete " + fileName);
                String input = in.readLine();
                if (input.equals("File does not exist!")) {
                        System.out.println(fileName + " does not exist!");
                }
        } catch (IOException e) {
                System.out.println(e);
                e.printStackTrace();
        }
    }
```

```
	/**
	 * Compresses the specified file into fileName.zip
	 */
	public static void compressFile(String fileName) {
//		System.out.println("Compressing " + fileName);
		File unzippedFile = new File(fileName);
		try (FileOutputStream fos = new FileOutputStream(fileName + ".zip");
				ZipOutputStream zos = new ZipOutputStream(fos);
				FileInputStream fis = new FileInputStream(new File(fileName))) {
			byte[] buffer = new byte[1];
			zos.putNextEntry(new ZipEntry(fileName));
			int length;
			while ((length = fis.read(buffer)) != -1) {
				zos.write(buffer, 0, length);
			}
			zos.closeEntry();
		} catch (IOException e) {
			System.out.println(e);
			e.printStackTrace();
		}
		File zippedFile = new File(fileName + ".zip");
		long unzippedLength = unzippedFile.length();
		long zippedLength = zippedFile.length();
		DecimalFormat format = new DecimalFormat("#.00");
//		System.out.println("Unzipped file: " + unzippedLength + ". Zipped file: " +
zippedLength);
//		System.out.println("% of unzipped file: " + format.format(100 * zippedLength /
unzippedLength));
	}

	/**
	 * Decompresses the file specified by zipFile to SAVE_DIR/fileName/saveName
	 *
	 */
	public static void decompressFile(String fileName, String zipFile, String saveName) {
		// Save to SAVE_DIR + fileName + / + saveName
		// e.g. unzip file1's segments to ./FROM_SERVER/file1/saveName
		System.out.println("Decompressing: " + zipFile);
		try (FileInputStream fis = new FileInputStream(zipFile); ZipInputStream zis =
new ZipInputStream(fis);) {
			ZipEntry entry = zis.getNextEntry();
			byte[] buffer = new byte[1];
			File file = new File(SAVE_DIR + fileName + "/" + saveName);
			System.out.println("Unzipping to " + file.getPath());
			FileOutputStream fos = new FileOutputStream(file);
```

```
                int length;
                while ((length = zis.read(buffer)) != -1) {
                        fos.write(buffer, 0, length);
                }
                fos.close();
                zis.closeEntry();
            } catch (IOException e) {
                System.out.println(e);
                e.printStackTrace();
            }
        }

        /**
         * Rebuilds our file after obtaining segments from server/S3. The
         * segments are deleted after the reconstruction.
         */
        private static void reconstructFile(String fileName, ArrayList<String> segments) {
            try (PrintWriter pw = new PrintWriter(SAVE_DIR + fileName + "_deduped");) {
                System.out.println("File deduped to: " + SAVE_DIR + fileName +
"_deduped");
                for (String segment : segments) {
                    File seg = new File(SAVE_DIR + fileName + "/" + segment);
                    FileInputStream fis = new FileInputStream(seg);
                    byte[] fileContent = new byte[(int) seg.length()];
                    fis.read(fileContent);
                    fis.close();
                    for (int i = 0; i < fileContent.length; i++) {
                        pw.append((char) fileContent[i]);
                    }
                    seg.delete(); // delete the unzipped segments
                }
                File saveDir = new File(SAVE_DIR + fileName);
                if (saveDir.exists() && saveDir.isDirectory()) {
                    saveDir.delete();
                }
            } catch (IOException e) {
                System.out.println(e);
                e.printStackTrace();
            }
        }

        private static void pause(int ms) {
            try {
                TimeUnit.MILLISECONDS.sleep(ms);
            } catch (InterruptedException e) {
                System.out.println(e);
```

```
                e.printStackTrace();
            }
        }

        public static void main(String[] args) {
            if (args.length != 4) {
                System.err.println("Usage: java DedupClient [put | get] [file] [address]
[port]");
                System.exit(1);
            }
            // Collection information
            String command = args[0]; // put, get, or delete
            String myFile = args[1]; // file path or file name
            String address = args[2]; // server's address
            int port = Integer.parseInt(args[3]); // server's port
            ContentBasedChunking cbc = new ContentBasedChunking();

            if (command.equalsIgnoreCase("put")) {
                /*
                 * Step 1: Segment and finger print the files (output is metadata
                 * file)
                 */

                // File we want to upload
                File file = new File(myFile);

                // Directory containing our file, its metadata, and its segments
                String fileDir = file.getParent();
                System.out.println("fileDir: " + fileDir);

                try {
                    cbc.digest(myFile);
                    // myFile = /path/to/myFile
                    // cbc.digest(myFile) creates /path/to/myFile.data
                    // and /path/to/segments
                } catch (IOException e) {
                    e.printStackTrace();
                    System.out.println(e);
                }
                System.out.println("Digest finished: " + myFile);
                // Metadata of file to be sent to server
                File metadata = new File(myFile + ".data"); // /path/to/myFile.data

                // Step 2: Determine segments of our file
                HashSet<String> segments = prepareSegments(metadata);
```

```
                        // Step 3: Determine segments that need to be uploaded
                        HashSet<String> toUpload = getFilesToUpload(segments, address, port);

                        // Step 4: Send metadata file to the server
                        System.out.println("Sending meta data");
                        sendFile(metadata, address, port);

                        // Step 5: Initialize S3 client
                        MyS3Client client = new MyS3Client();

                        // Step 6: Send segments to buckets
                        for (String upload : toUpload) {
                                String segPath = fileDir + "/" + upload;
                                File segFile = new File(segPath);
                                // Compress the seg file to seg.zip
                                compressFile(segPath);
                                segFile.delete(); // Delete original file
                                File segZip = new File(segPath + ".zip");
                                client.uploadFile(segZip); // Upload zipped file
                                segZip.delete(); // Delete zipped file
                        }
                        metadata.delete(); // Delete file metadata
                }
                if (command.equalsIgnoreCase("get")) {
                        File saveDir = new File(SAVE_DIR);
                        if (!saveDir.exists()) {
                                saveDir.mkdir();
                        }
                        // Step 1: Get the segments needed to reconstruct our file
                        ArrayList<String> fileSegments = getFileSegments(myFile, address,
port);
                        HashSet<String> uniqueFiles = new HashSet<>(fileSegments);
                        // Step 2: Initialize S3 client
                        MyS3Client client = new MyS3Client();
                        // Step 3: Download objects from S3
                        client.downloadZippedSegments(myFile, uniqueFiles); // saves to

                        // ./FROM_SERVER/filename/
                        // Now we have a directory of .zip files: unzip and reconstruct the
                        // file
                        for (String zippedSegment : uniqueFiles) {
                                String pathToZipSegment = SAVE_DIR + myFile + "/" +
zippedSegment + ".zip";
                                decompressFile(myFile, pathToZipSegment, zippedSegment);
                                // Delete zip file
                                new File(pathToZipSegment).delete();
```

```
                }
                reconstructFile(myFile, fileSegments);
            }
            if (command.equalsIgnoreCase("delete")) {
                deleteFile(myFile, address, port);
            }
        }
}
```

MyS3Client

This class is intended to be used by the client to interact with S3 buckets. An instance of MyS3Client obtains the necessary authorization to access S3 buckets via a private key file. All actions that involve S3, such as upload, download, and deleting segments, must issue requests through this class.

Method	Behaviour
uploadFile()	Uploads a file to all S3 buckets.
downloadZippedSegments()	Downloads a file from the S3 bucket.
saveS3Object()	Used in *downloadZippedSegments()* to convert an S3Object into a file.
deleteSegment()	Deletes the specified segment from all S3 buckets.

Table 4: Methods in the MyS3Client class

```
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;
import java.util.HashSet;

import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.services.s3.model.DeleteObjectRequest;
```

```
import com.amazonaws.services.s3.model.GetObjectRequest;
import com.amazonaws.services.s3.model.PutObjectRequest;
import com.amazonaws.services.s3.model.S3Object;

public class MyS3Client {

	static final String[] BUCKETS = new String[] { "dedupbucket1", "dedupbucket2",
"dedupbucket3" };
	static AmazonS3 client;

	public MyS3Client() {
		AWSCredentials credentials = null;
		try {
			credentials = new ProfileCredentialsProvider("default").getCredentials();
		} catch (Exception e) {
			throw new AmazonClientException("Cannot load the credentials from the
credential profiles file. "
					+ "Please make sure that your credentials file is at the
correct "
					+ "location (/Users/<username>/.aws/credentials), and is in
valid format.", e);
		}
		client = new AmazonS3Client(credentials);
	}

	/**
	 * Uploads the specified file to all of our S3 buckets
	 */
	public void uploadFile(File file) {
		try {
			System.out.println("Uploading: " + file.getName());
			for (String bucket: BUCKETS) {
				client.putObject(new PutObjectRequest(bucket, file.getName(),
file));
			}

		} catch (AmazonServiceException ase) {
			System.out.println("Caught an AmazonServiceException, which " +
"means your request made it "
					+ "to Amazon S3, but was rejected with an error response"
+ " for some reason.");
			System.out.println("Error Message:    " + ase.getMessage());
			System.out.println("HTTP Status Code: " + ase.getStatusCode());
			System.out.println("AWS Error Code:   " + ase.getErrorCode());
			System.out.println("Error Type:       " + ase.getErrorType());
			System.out.println("Request ID:       " + ase.getRequestId());
```

```
                } catch (AmazonClientException ace) {
                        System.out.println("Caught an AmazonClientException, which " + "means
the client encountered "
                                        + "an internal error while trying to " + "communicate with
S3, "
                                        + "such as not being able to access the network.");
                        System.out.println("Error Message: " + ace.getMessage());
                }
        }

        /**
         * Download the zipped segment files.
         */
        public void downloadZippedSegments(String fileName, HashSet<String> segments) {
                int attempt = 0;
                boolean done = false;
                while (!done) {
                        String s3Bucket = BUCKETS[attempt];
                        String[] segmentArray = new String[segments.size()];
                        segments.toArray(segmentArray);
                        try {
                                for (int i = 0; i < segmentArray.length; i++) {
                                        String segment = segmentArray[i];
                                        System.out.println("Downloading: " + segment + ".zip");
                                        S3Object s3Object = client.getObject(new
GetObjectRequest(s3Bucket, segment + ".zip"));
                                        saveS3Object(fileName, s3Object);
                                        s3Object.close();
                                }
                                done = true;
                        }
                        catch (AmazonServiceException ase) {
                                System.out.println("Caught an AmazonServiceException, which "
+ "means your request made it "
                                                + "to Amazon S3, but was rejected with an error
response" + " for some reason.");
                                System.out.println("Error Message:    " + ase.getMessage());
                                System.out.println("HTTP Status Code: " + ase.getStatusCode());
                                System.out.println("AWS Error Code:   " + ase.getErrorCode());
                                System.out.println("Error Type:       " + ase.getErrorType());
                                System.out.println("Request ID:       " + ase.getRequestId());
                                if (attempt == 2) {
                                        // We failed!
                                        System.err.println("Failed to get segment in 3 attempts");
                                        System.exit(1);
                                } else
```

```
                                attempt++;
                        } catch (AmazonClientException ace) {
                                System.out.println("Caught an AmazonClientException, which " +
"means the client encountered "
                                                + "an internal error while trying to " +
"communicate with S3, "
                                                + "such as not being able to access the network.");
                                System.out.println("Error Message: " + ace.getMessage());
                                if (attempt == 2) {
                                        // We failed!
                                        System.err.println("Failed to get segment in 3 attempts");
                                        System.exit(1);
                                } else
                                        attempt++;
                        } catch (IOException e) {
                                System.out.println(e);
                        }
                }
        }

        /**
         * Saves the s3Object that represents our segment file
         */
        private static void saveS3Object(String fileName, S3Object s3Object) throws
IOException {
                InputStream in = s3Object.getObjectContent();
                File fileDir = new File(DedupClient.SAVE_DIR + fileName);
                if (!fileDir.exists()) {
                        fileDir.mkdir();
                }
                File zippedFile = new File(DedupClient.SAVE_DIR + fileName + "/" +
s3Object.getKey());
                Files.copy(in, zippedFile.toPath(), StandardCopyOption.REPLACE_EXISTING);
                in.close();
        }

        /**
         * Deletes a segment from our s3 buckets
         */
        public void deleteSegment(String segment) {
                try {
                        System.out.println("Deleting: " + segment);
                        for (String bucket: BUCKETS) {
                                client.deleteObject(new DeleteObjectRequest(bucket, segment));
                        }
                } catch (AmazonServiceException ase) {
```

```
                        System.out.println("Caught an AmazonServiceException, which " +
"means your request made it "
                                        + "to Amazon S3, but was rejected with an error response"
+ " for some reason.");
                        System.out.println("Error Message:    " + ase.getMessage());
                        System.out.println("HTTP Status Code: " + ase.getStatusCode());
                        System.out.println("AWS Error Code:   " + ase.getErrorCode());
                        System.out.println("Error Type:       " + ase.getErrorType());
                        System.out.println("Request ID:       " + ase.getRequestId());
                } catch (AmazonClientException ace) {
                        System.out.println("Caught an AmazonClientException, which " + "means
the client encountered "
                                        + "an internal error while trying to " + "communicate with
S3, "
                                        + "such as not being able to access the network.");
                        System.out.println("Error Message: " + ace.getMessage());
                }
        }

}
```

ContentBasedChunking

This class is intended to be used by the client to split file into multiple segments using rabin fingerprinting algorithm. An instance of ContentBasedChunking file initializes rabinfingerprinting algorithm by entering values for static parameters like bytes_per_window, min_chunk_size, max_chunk_size. Functions in this class creates new file for each segment which can be used to upload into S3. This class also contains functions for storing fingerprint and length of each segment associated with each input file in separate metadata file.

Method	Behaviour
storeSegmentInFile()	This method stores each segment in separate file. This file will later be compressed by function in Dedupclient before uploading to S3.
digest()	This function takes filename as argument and converts whole file into byte array which will

	be used by rabin fingerprinting algorithm to calculate rolling hash for identifying boundaries for splitting file.

Table 5: Methods used in ContentBasedChunking class

```
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.nio.ByteBuffer;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

import org.rabinfingerprint.fingerprint.RabinFingerprintLong;
import org.rabinfingerprint.fingerprint.RabinFingerprintLongWindowed;
import org.rabinfingerprint.handprint.BoundaryDetectors;
import org.rabinfingerprint.handprint.FingerFactory.ChunkBoundaryDetector;
import org.rabinfingerprint.polynomial.Polynomial;

public class ContentBasedChunking {

	static long bytesPerWindow = 48;
	static Polynomial p = Polynomial.createFromLong(10923124345206883L);
	private RabinFingerprintLong fingerHash = new RabinFingerprintLong(p);
	private RabinFingerprintLongWindowed fingerWindow = new
RabinFingerprintLongWindowed(p, bytesPerWindow);
	private ChunkBoundaryDetector boundaryDetector =
BoundaryDetectors.DEFAULT_BOUNDARY_DETECTOR;
	private final static int MIN_CHUNK_SIZE = 4096;
	private final static int MAX_CHUNK_SIZE = 8192;
	private RabinFingerprintLong window = newWindowedFingerprint();

	/*
	 * This function will store each segment in new file in same directory as input file.
	 */
	public void storeSegmentInFile(String myFile, byte[] buf, int chunkLength, long
fingerPrint) {
		try {
			File file = new File(myFile);
			String fileDir = file.getParent();
			Path p2 = Paths.get(fileDir + "/" + fingerPrint);
			OutputStream out = new
BufferedOutputStream(Files.newOutputStream(p2));
```

```
                    out.write(buf, 0, chunkLength);
                    out.close();
                } catch (Exception e) {
                    e.printStackTrace();
                }
        }

        /*
         * This function will convert file to be segmented into byte array.
         * Rabinfinger printing algorithm will calculate rolling hash for each
         * window and identify boundaries between min and max length. Each fingerprint
         * and its corresponding length will be stored in .data file for keeping
         * track of offset to fingerprint mapping.
         *
         * @param myFile File name to segment
         */
        public void digest (String myFile) throws IOException {
                long chunkStart = 0;
                long chunkEnd = 0;
                int chunkLength = 0;
                Path p1 = FileSystems.getDefault().getPath("/", myFile);
                byte[] barray = null;

                barray = Files.readAllBytes(p1);
                FileWriter fw = new FileWriter(myFile + ".data", true); //the true will append the
new data

                ByteBuffer buf = ByteBuffer.allocateDirect(MAX_CHUNK_SIZE);
                buf.clear();
                /*
                 * fingerprint one byte at a time. we have to use this granularity to
                 * ensure that, for example, a one byte offset at the beginning of the
                 * file won't effect the chunk boundaries
                 */
                for (byte b : barray) {
                        // push byte into fingerprints
                        window.pushByte(b);
                        fingerHash.pushByte(b);
                        chunkEnd++;
                        chunkLength++;
                        buf.put(b);

                        /*
                         * if we've reached a boundary (which we will at some probability
                         * based on the boundary pattern and the size of the fingerprint
                         * window), we store the current chunk fingerprint and reset the
```

```
                         * chunk fingerprinter.
                         */
                        if (boundaryDetector.isBoundary(window)    && chunkLength >
MIN_CHUNK_SIZE) {
                                byte[] c = new byte[chunkLength];
                                buf.position(0);
                                buf.get(c);

                                /* Store fingerprint in metadata file */
                                String s = fingerHash.getFingerprintLong() + "," + chunkLength +
"\n";
                                fw.write(s);

                                storeSegmentInFile(myFile, c, chunkLength,
fingerHash.getFingerprintLong());

                                chunkStart = chunkEnd;
                                chunkLength = 0;
                                fingerHash.reset();
                                buf.clear();
                        } else if (chunkLength >= MAX_CHUNK_SIZE) {
                                byte[] c = new byte[chunkLength];
                                buf.position(0);
                                buf.get(c);

                                /* Store fingerprint in metadata file */
                                String s = fingerHash.getFingerprintLong() + "," + chunkLength +
"\n";
                                fw.write(s);

                                storeSegmentInFile(myFile, c, chunkLength,
fingerHash.getFingerprintLong());

                                fingerHash.reset();
                                buf.clear();
                                chunkStart = chunkEnd;
                                chunkLength = 0;
                        }
                }
                byte[] c = new byte[chunkLength];
                buf.position(0);
                buf.get(c);

                /* Store fingerprint in metadata file */
                String s = fingerHash.getFingerprintLong() + "," + chunkLength + "\n";
                fw.write(s);
```

```
                storeSegmentInFile(myFile, c, chunkLength, fingerHash.getFingerprintLong());
                fingerHash.reset();
                buf.clear();
                fw.close();
        }

        private RabinFingerprintLongWindowed newWindowedFingerprint() {
                return new RabinFingerprintLongWindowed(fingerWindow);
        }
}
```

7.Data Analysis and Discussion:

A. Storage

We conducted four different tests with two files in each test. We used randomly generated text files to ensure that deduplication is working correctly. In this test case, the original storage space, and the storage space after deduplication, compression are calculated and the results are shown in Table VI.

In the first test run, two files that are duplicates of size 52MB are used. Each file created 7393 segments of size ranging from 4KB to 8KB.

For second test run, two files that are unique are used. First file created 7008 segments. Second file created 7415 segments. Each file took 40.64MB space.

For the third test run, two files that are 50 percent similar were used. The files were created by issuing the command “base64 /dev/urandom | head -c 25000000” to create three 25MB files:

file1.txt, file2.txt, and temp.txt. To guarantee a similarity of 50%, the contents of temp.txt were appended to both file1.txt and file2.txt. The compressed size of both files were 78.59MB. Both files created 6,820 unique segments that compressed to 39.3MB. It was determined that out of the 13,640 segments that comprise of both files, only 10,299 were unique between them. Those segments compressed to a size of 58.95MB. Compared to compressed size of the non-deduplicated files, our deduplication process yielded an expected 25% savings in storage space.

For the fourth test run, two files that are 80 percent similar were used. The files were created by issuing the command “base64 /dev/urandom | head -c 10000000” to create two 10MB files: file3.txt and file4.txt. The command “base64 /dev/urandom | head -c 40000000” created a single 40MB file: temp.txt. To guarantee a similarity of 80%, the contents of temp.txt were appended to both file3.txt and file4.txt. The compressed size of both files were 78.6MB. File4.txt created 6,816 unique segments that compressed to 39.3MB. File5.txt created 6,820 unique segments that compressed to 39.3MB. It was determined that out of the 13,636 unique segments that comprise of both files, only 8,179 were unique between them. Those segments compressed to a size of 47.2MB. Compared to the compressed size of the non-deduplicated files, our deduplication process yielded an expected 40% savings in storage space.

Test Runs	Size of each file	Storage space before Deduplication	Storage space after deduplication and compression
1	52MB	104MB	40.64 MB
2	52MB	104MB	81.28 MB
3	50MB	100MB	58.95 MB

4	50MB	100MB	47.16 MB

Table 6: Space saved for different sets of data

B. Cost Analysis:

While making efficient use of space is important, another significant consideration to make is the number of segments generated from a file. The number of segments generated depends primarily on the chunk sizes we wish to obtain. Increasing the minimum chunk size reduces the number of segments, but will also reduce our resolution. By forcing chunk sizes to be too large, we may miss out on highly repeated smaller sequences. The results of testing various minima and maxima for segment sizes are in the Table VII.

When using Amazon's S3 services, what you pay depends on two criteria: the size of the data you store and the number of requests you issue. For standard use and single bucket, you get 1,000 PUT requests for $0.01, 10,000 GET requests for $0.01, and 1 GB of storage for $0.033 for the first TB.

To perform a cost analysis, suppose we uploaded file1.txt, file2.txt, file3.txt, and file4.txt using 4KB-8KB chunks. If we had uploaded these files to S3 in compressed format, we would have used 471 MB of space and 12 put requests. If we employed deduplication, we would have used 318.3 MB of space and 55,434 put requests. Without deduplication, these four files would cost $0.01 for the put requests, $0.01 for the get requests, and $0.0155 for the storage. With deduplication, these four files would cost $0.56 for the put requests, $0.02 for the get requests, and $0.0105 for the storage. However, as previously mentioned, it is possible to tweak the

segmentation process to yield lower numbers of segments being generated for a file. In this example, we saved ~68% in terms of space, but a ~5,600% increase in request cost.

The benefit of deduplication becomes more apparent and significant as more files are added to the database. Suppose instead we had thousands of files and hundreds of thousands of segments. And further suppose that we wished to modify those files slightly by inserting some chunk of text. In a non-deduplicated environment, you would have essentially doubled your needs in storage space, since you wanted to keep the originals for future reference. With deduplication, you will only have to upload the new segments, thus providing significant savings in storage space.

To test the above theory, two randomly generated 50 MB files were created, fileA.txt and fileB.txt. The contents of fileA.txt were appended to the fileB.txt to create a 100 MB file. Using chunk sizes of 4KB to 32KB, fileA.txt had 4,138 unique segments and fileB.txt had 8,230 unique segments. Between the two files, there were 8,231 unique segments. This suggests that when we append to an existing file, only the new content will need to be stored, rather than the whole file. This is very useful in the secondary storage and data backup scenarios, where only the modified data can be saved instead of saving the whole file.

Files	Similarity	4KB-8KB	4KB-32KB	8KB-32KB	1KB-32KB	512B-32KB
File1.txt and File2.txt	50%	10,299	6,120	4,609	8,142	8,651
File3.txt and File4.txt	80%	8,179	4,992	3,779	6,643	7,032

Table 7: Number of unique segments among files for varying ranges of chunk size

C. Abnormal Cases:

The success of this deduplication scheme is almost entirely dependent on the success of the chunking operation. One cause of chunking failure can be traced to the bytes per window parameter. As the name suggests, this value defines how big the sliding window is for the Rabin fingerprinting procedure.

To test the effects of the bytes per window value, a text file was created by repeatedly copying and pasting a sequence of text. If the chunking was successful, we should obtain a small set of segments with a collective size smaller than the original file. When the file was chunked using a bytes per window value of 48, 451 unique segments were created. However, when the bytes per window value was set to 24, only 3 unique segments were created. The composition of the file was essentially one segment for the beginning of the file, one segment to be repeated for the bulk of the file, and one segment for the end of the file.

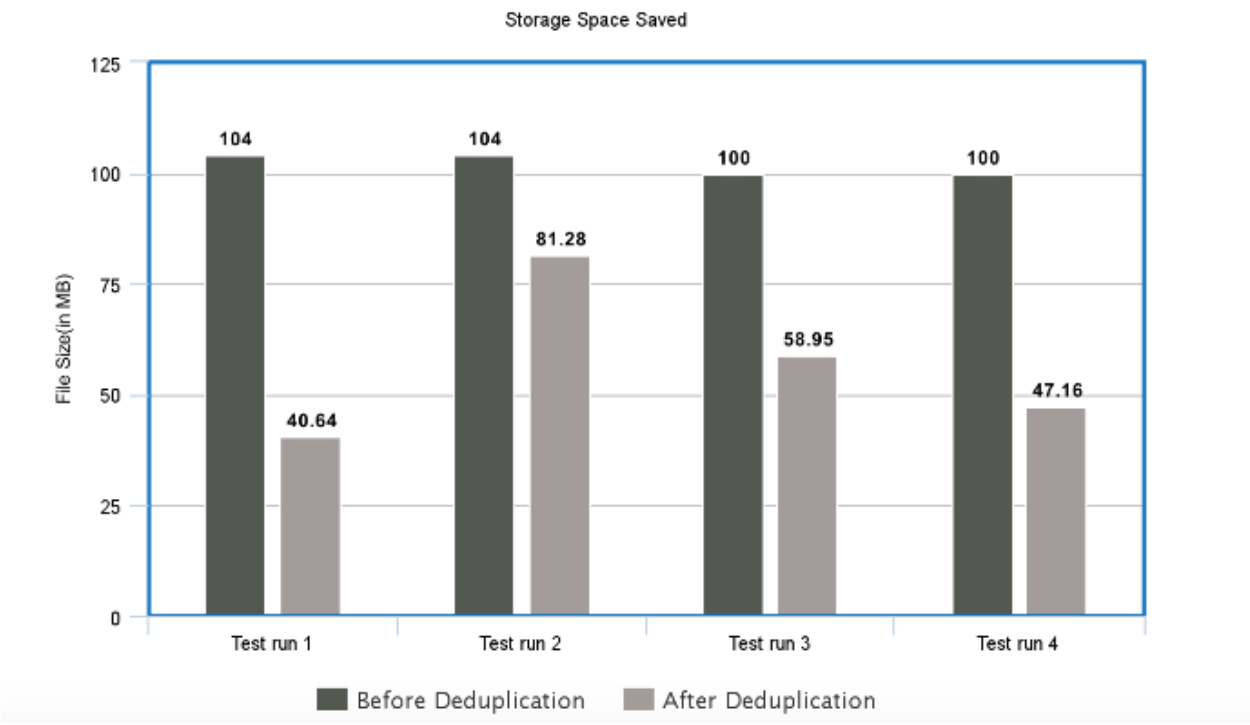


Figure 6: Storage space before and after deduplication

8. Conclusion and Recommendations:

In this paper, deduplication was implemented using a variable-length segmentation technique, which aided in saving both storage space and cost. Although initial costs are higher when compared to no deduplication scheme, it is expected that as more files are being stored and maintained, costs will decrease. Files were shown to be able to be faithfully reconstructed from their segments by comparing the md5 hashes of the original uploaded file and the file obtained from the cloud. All of the tests conducted demonstrates a considerable reduction in storage space, and eventual reduction in cost.

Future work can be focused on determining how to better generate segments for files. As

mentioned earlier, the size of segments varies and can be defined by adjusting the minimum and maximum sizes. Depending on the nature of the file, such as its size and content, different chunking parameters can yield more optimal results. Furthermore, our investigations have shown that this deduplication scheme works most effectively for files that are modified via appends. Alternative segmentation approaches can be investigated to determine how to segment two files that have similarities interspersed throughout the entire file. A caching scheme could also be implemented on the client side to store the frequently requested files

9. Bibliography

1. Hwang, Kai, Geoffrey C. Fox, and J. J. Dongarra. Distributed and Cloud Computing: From Parallel Processing to the Internet of Things. Amsterdam: Morgan Kaufmann, 2012.

2. A. Upadhyay, P. R. Balihalli, S. Ivaturi and S. Rao, "Deduplication and compression techniques in cloud design," Systems Conference (SysCon), 2012 IEEE International, Vancouver, BC, 2012, pp. 1-6. doi: 10.1109/SysCon.2012.6189472.

3. Zheng Yan, Wenxiu Ding, Xixun Yu, Haiqi Zhu, Robert H. Deng, "Deduplication on Encrypted Big Data in Cloud", IEEE Transactions on Big Data, vol.2, no. 2, pp. 138-150, June 2016, doi:10.1109/TBDATA.2016.2587659.

4. Patrick van Pergen. "Using SHA1 Hashes to Identify Files." Procurios Techblog. N.p., 2013. Web. 07 Aug. 2016. <http://tech.procurios.nl/archief/2013/06/01/Using-SHA1-hashes-to-identify-files>.

5. R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," in *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249-260, March 1987.
doi: 10.1147/rd.312.0249

6. J. Sun, H. Chen, L. He and H. Tan, "Redundant Network Traffic Elimination with GPU Accelerated Rabin Fingerprinting," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 2130-2142, July 1 2016.
doi: 10.1109/TPDS.2015.2473166

7. Ghosh, Moinak. "High Performance Content Defined Chunking." The Pseudo Random Bit Bucket. N.p., 22 June 2013. 07 Aug. 2016. <https://moinakg.wordpress.com/2013/06/22/high-performance-content-defined-chunking>

10. Appendices:

Flow Charts:

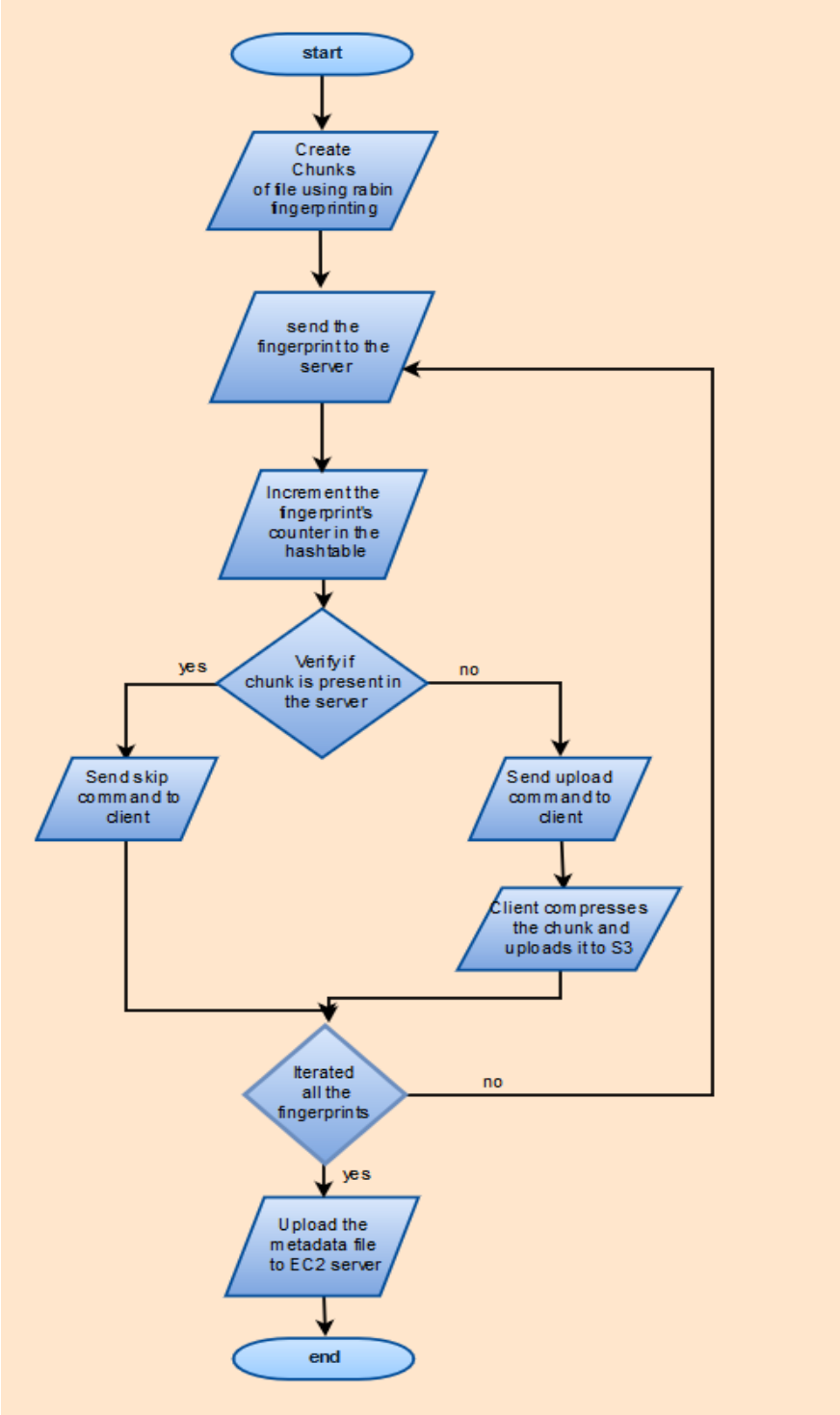


Figure 7: Flow chart for uploading file

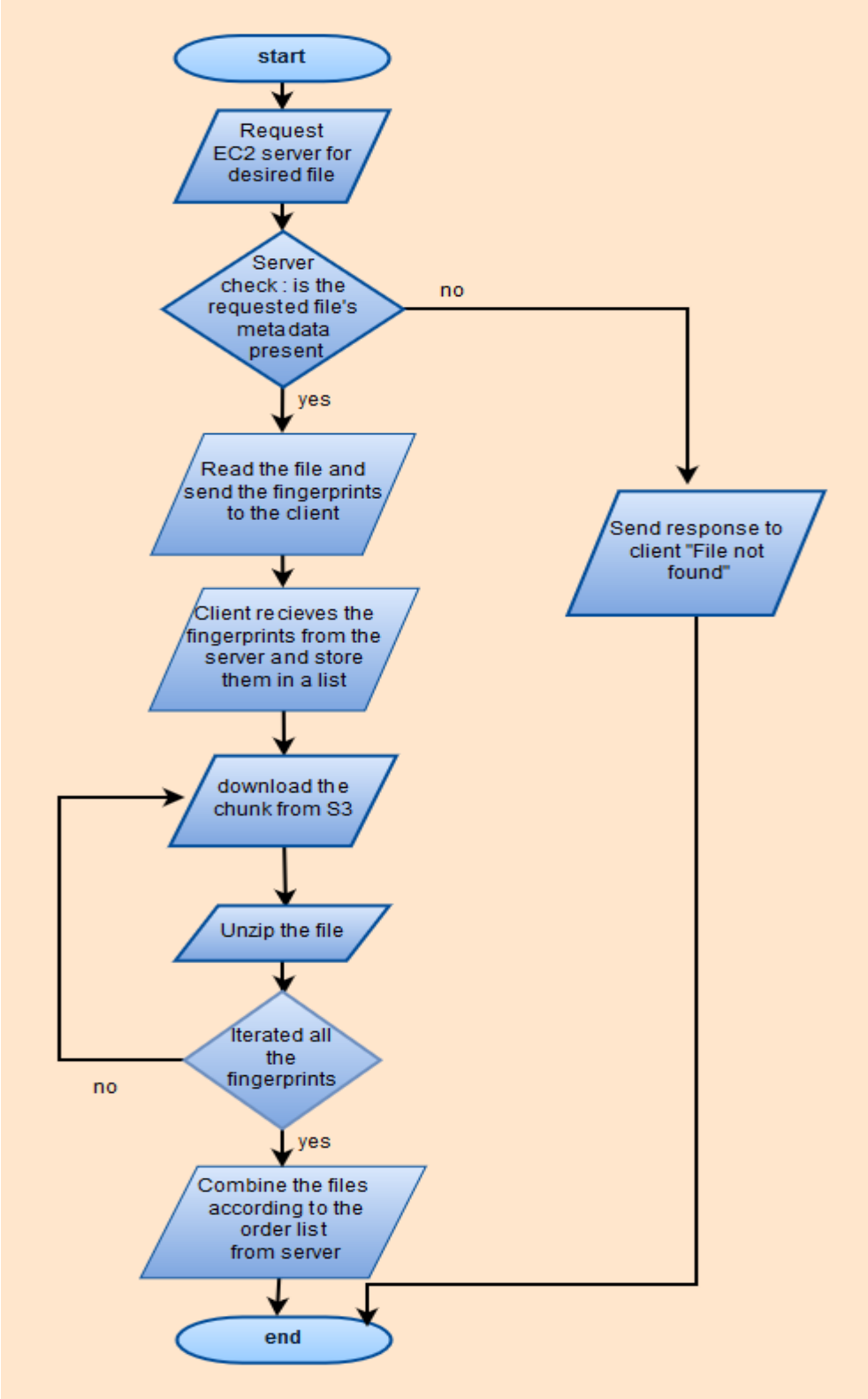


Figure 8: Flow Chart for downloading file

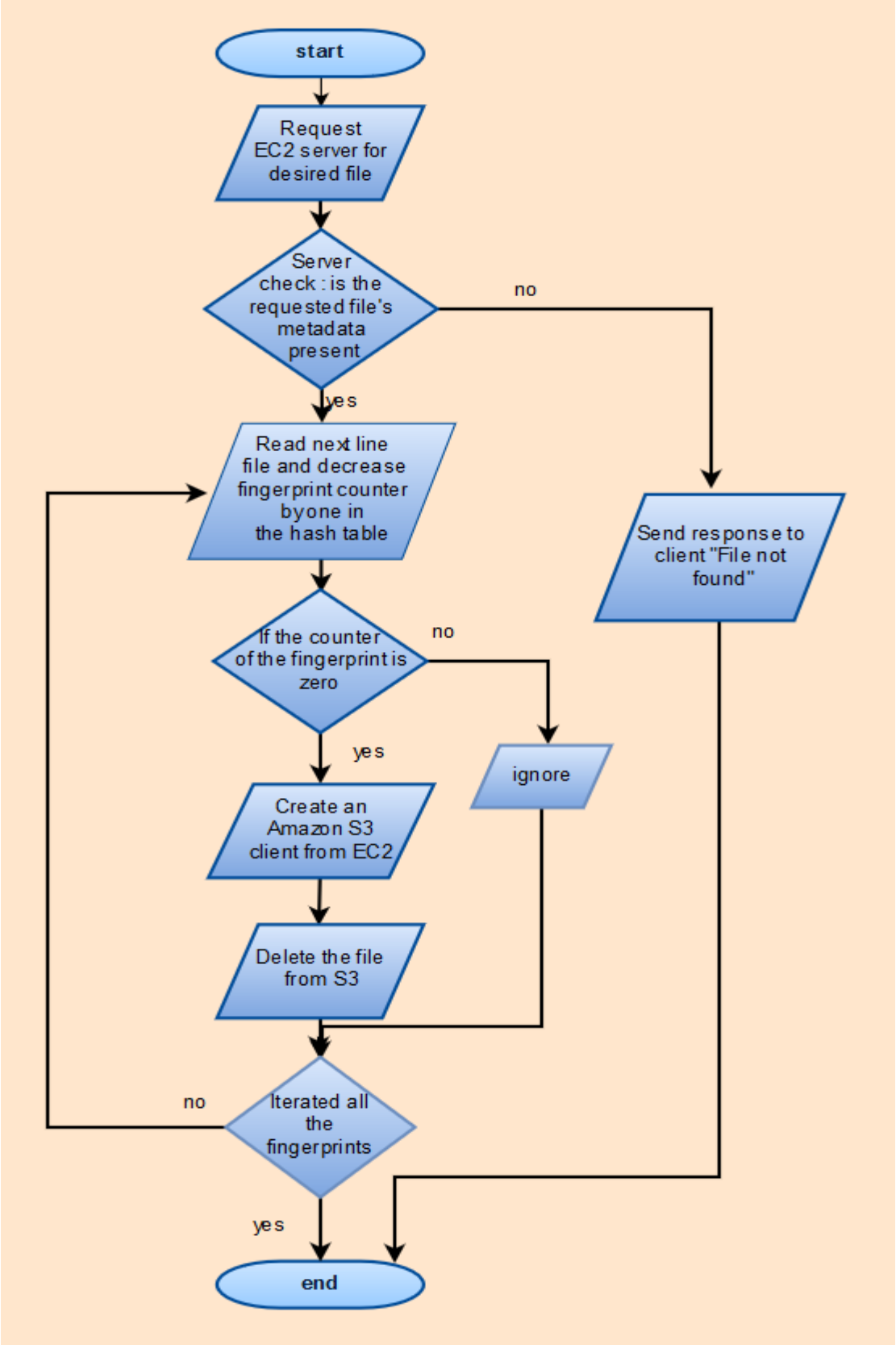
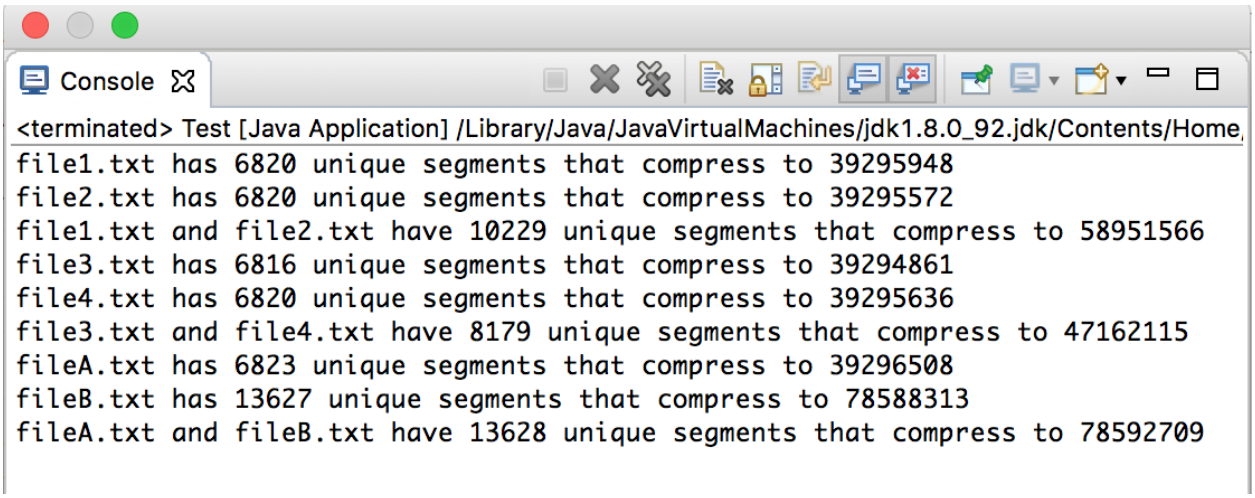


Figure 9: Flow chart for deleting files

Input Output Listing:

Input file name	Output files
file1.txt	file1.txt.data, 6,820 segments
file2.txt	file2.txt.data, 6,820 segments
file3.txt	file3.txt.data, 6,816 segments
file4.txt	file4.txt.data, 6,820 segments
fileA.txt	fileA.txt.data. 6,823 segments
fileB.txt	fileB.txt.data, 13,627 segments

The input files were randomly generated using "base64 /dev/urandom | head -c *size* > *filename*". In order to ensure a certain similarity percentage, files of certain sizes were randomly generated to be appended to files. For example, the first 25 MB worth of content in file1.txt and file2.txt were randomly generated and are expected to be unique against each other. However, the remainder of their contents are the same due to the same 25 MB worth of content being appended to both of them.



The above screenshot describes the results of deduplication. Each file was individually segmented, and their segments were compressed. In order to determine the unique number of segments between two files, a HashSet object was used since it prevents duplicates from being added. The HashSet object contains all segments from the given pairs of files (e.g. file1.txt and file2.txt, file3.txt and file4.txt, fileA.txt and fileB.txt).