

Data Deduplication, a lazy approach

Dinoja Padmanabhan, Raghavendran Nedunchezian
{dpadman, rnedunc} @ ncsu.edu

1. Abstract

Data deduplication, data reduction, capacity optimized storage - is a specialized data compression technique for eliminating redundant data in the storage file system. It is often designed to make network backups to disk faster and more economical. It removes redundant data segments to compress data into a highly compact form and makes it economical to store backups on disk drive instead of tape drive. The technique identifies unique chunks of data, or byte patterns and stores them during the time of analysis. The chunks of new data are hashed and compared with the existing hashes of the other chunks to determine if there is a match. This project is aimed at creating a data deduplication file system which performs post-processing (lazy) analysis on the data stored into the file system.

2. Introduction

Data is growing at an alarmingly asymptotic rate. It is determined that the data growth rate is at the order of few peta bytes and that the total amount of data that will accumulate in the next decade would be in the order of tens of zeta bytes. Hence efficient data reduction and capacity optimized storage techniques need to be employed to manage and handle the enormous flow of data [1].

Data management industries have shown that efficient management of data by implementing data mining and redundant-data detection algorithms can effectively reduce the storage requirements of an enterprise. Dropbox, an online storage provider provides a free online storage of private data of upto 2 GB. Google, a search engine enterprise has recently added a new product called the Google Drive which provides a free online storage of upto 5 GB. These enterprises are able to achieve this by implementing an efficient data deduplication file system and by managing redundant data efficiently. Most of the storage solutions provider industries such as EMC and NetApp, employ data deduplication techniques in their backup solutions software. It is used to reduce the amount of daily backed up data sent from the client servers to the backup servers over the Internet thereby reducing bandwidth and cost.

This project is our attempt at efficiently managing the redundant data in a file system using the techniques from data deduplication. We have used FUSE abstraction to implement file I/O system calls such as open/close/read/write, etc. FUSE interacts with the libc shared library to redirect system calls to the user space where we provide methods to handle them. Implementing a data deduplication file system provides an insight over the amount of data that actually get stored in a file system with redundant data. The rest of the sections is arranged as follows. Section 3 provides a general overview of the system. Section 4 provides the technical details in building a data deduplication file system along with the implementation details. Section 5 discusses about the system interaction and the call flow. Section 6 talks about the related work and finally in section 7 we provide the conclusion along with the future work.

3. Overview

Figure 1 shows the overall sequence of operations in the data deduplication system [9]. The system will consist of a chunk divider module, hash computation engine, redundant chunk eliminator module and a repository for storing the chunks and its associated meta data. The chunk divider module will divide the file into small pieces of data called the data blocks or the file chunks. The data blocks would be variable sized chunks since it provides better resilience to file updation. These data blocks will then

reach the hash computation engine which calculates the hash value, an unique identifier of each data block. Once the hash is calculated for each data block, the hash is looked up in a database maintained by the redundant chunk eliminator module. If the data block does not exists in our block store, then the identifier is added to the table of identifiers and the block is stored in the block store. However if the data block already exists in our block store then appropriate data structures are modified to increment the reference count for this block of data. The system follows a lazy method of deduplicating data since it would be efficient in terms of speed and response time for the user.

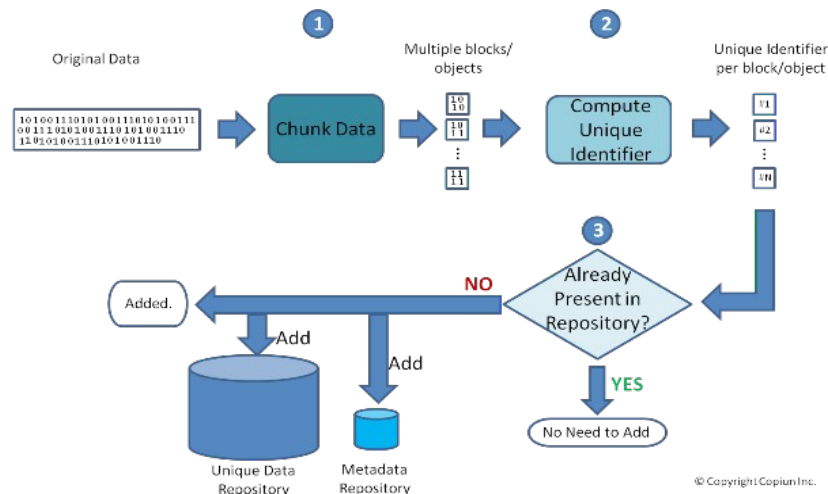


Figure : 1 – Overview of the data deduplication system

4. Technical Details

The following are some of the major components involved in performing successful data deduplication.

- I) Chunking of data
- II) Identifying duplicate chunks
- III) Elimination of redundant chunks
- IV) Metadata File
- V) Thick Write and Bitmap Files

4.1 Chunking of data

The file to be stored in the file system needs to be split into blocks of data called as chunks. The process of splitting the file into chunks of data can be done in two ways. The first method splits the file data into fixed size chunks and the second method splits the file into variable-size chunks. We have used variable-size chunks in this project as it provides better resilience during file updation.

4.1.1 Design and Implementation

We have employed Rabin-Karp Fingerprinting algorithm to split the file into variable size blocks.

Rabin-Karp pattern matching algorithm is a naive approach augmented with a powerful programming technique called the hash function. Every chunk, a sequence of characters $s[]$ has a total of 'm' bytes (m can be variable length). The hash computed from this series of bytes with length 'm' is denoted as 'H'. The algorithm requires a pre-computed pattern hash RM (read as R power 'm') where R is the radix raised to the power 'm'.

The value of RM is precomputed as

$$RM = (R * RM) \% Q,$$

where Q is a prime number

The above calculation is iterated for length 'm' and is kept constant throughout the algorithm.

The hash 'H' for each series of bytes of length 'm', with index 'i' is calculated as

$$H = \sum_{i=0}^m (R * H_i + s[i]) \% Q$$

The approach mentioned above is inefficient if a rolling hash cannot be formed during a single file read. To calculate the rolling hash we employ the following method,

$$H = (H + Q - RM * s[i-m] \% Q) \% Q;$$
$$H = (H * R + s[i]) \% Q;$$

In the above method, we readjust the hash for every byte of data read from the file by removing the leading character and adding the trailing character in forming the hash value.

By computing the hash value for the pre-computed pattern and the hash value for sequence of bytes of length m, we will have to just compare two hash values instead of comparing two strings of equal length 'm'. After computing the hash value of a data stream of 'm' bytes, we check if the last 'n' bits are set to predefined bit pattern which can have all bits set to 1 or 0 or a different pattern altogether [7]. This serves as the endpoint in dividing a file into variable sized blocks. The size of the pre-defined bit pattern is configurable and can be varied according to the requirement.

Every chunk in this approach has a minimum chunk size and a maximum chunk size. Minimum and maximum chunk sizes are required so that data chunk is neither too small nor too large. These sizes are configurable and can be varied based on the size of the file.

Based on the above design approach, our implementation of Rabin-Karp fingerprinting algorithm [8] [10] has the following tunables.

The maximum chunk value is 8192 bytes (8KB)

The minimum chunk value is 4096 bytes (4KB)

The substring length for series of bytes in the file is 48 bytes

The predefined bit pattern is series of consecutive 12 bits of 1's

4.2 Identifying duplicate chunks

Once the data/file is divided into variable sized chunks, an unique identifier for that chunk is determined. The identifier is typically a hash value for the chunk, used as a representation of the data in the chunk. The hash value computed for each chunk is compared against the list of hashes already available in the database to determine whether the chunk already exists.

4.2.1 Design and Implementation

We have used the standard hash function, SHA-1 (160 bits) for computing the hash of the data chunks. This function provides unique hash value and has a very low probability of hash collisions.

The SHA-1 function, produces a 160 bit hash value resulting in a very large address space (2^{160}). The probability of collision in SHA-1 ($1/10^{15}$) [11] is less than the probability of severe hardware failure (memory bit error rate or failure rate for an underlying enterprise RAID) ($1/10^{14}$) [12] [13]. The probability of hash collision can be further reduced by employing more stronger SHA encryption algorithms that produces a longer digest. However since SHA-1 provides a good resilience compared to hardware failures we have used it in this project.

The variable sized chunk from the chunk divider module is fed into the SHA-1 module. The module computes the SHA-1 hash for the data chunk and provides a 160 bit hash value which is stored in the metadata file. For each SHA-1 hashed chunk, a chunk file is created with SHA-1 hash as its name in the hash block store.

Figure 2 below shows the output of the implementation of SHA-1. The program takes an input file and produces a 160 bit SHA-1 digest for the variable sized data block divided using Rabin-Karp algorithm. The 160 bits value is unique for each data chunk.

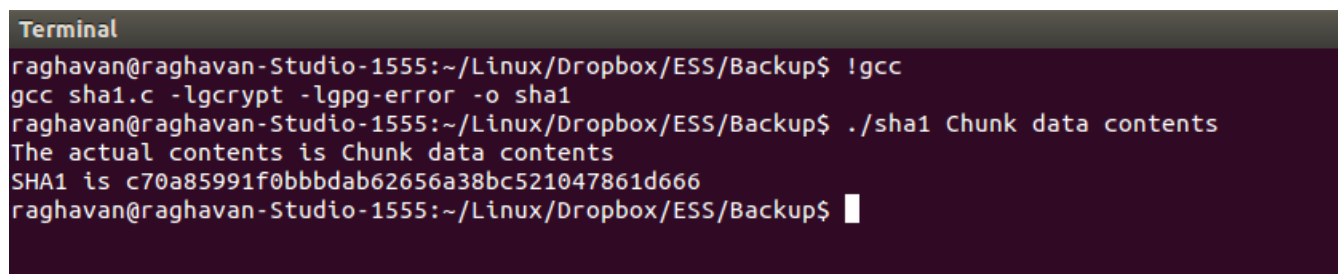
A terminal window with a dark background and light text. The prompt is 'raghavan@raghavan-Studio-1555:~/Linux/Dropbox/ESS/Backup\$'. The user enters '!gcc', and the prompt changes to 'gcc sha1.c -lgcrypt -lgpg-error -o sha1'. The user then enters './sha1 Chunk data contents', and the output is 'The actual contents is Chunk data contents' followed by 'SHA1 is c70a85991f0bbbdab62656a38bc521047861d666'. The prompt returns to 'raghavan@raghavan-Studio-1555:~/Linux/Dropbox/ESS/Backup\$'.

Figure : 2 – Implementation of SHA-1

4.3 Elimination of redundant chunks

Every file chunk in the hash block store shown in Figure 4, is associated with a reference count. It serves to keep track of the number of occurrences of the file chunk in different files stored in the file system.

4.3.1 Design and Implementation

For each redundant chunk, the reference count is updated and the corresponding information is updated in the meta data file which reference the data chunk. In our design a separate file called the nlinks.txt which resides in the same directory as the hash block is used to store the reference count of the data chunk. Certain design decisions need to be made while updating the file containing the reference count. Since we are employing a lazy method of deduplication, there can be more than one thread accessing the reference count file at the same time. To avoid possible race condition in the filesystem, we have used the flock abstraction which can be used to apply or remove advisory locks on an open file descriptor.

```

raghavan@raghavan-Studio-1555:/tmp/dedupe_hashes$ ls
00 07 0e 15 1c 23 2a 31 38 3f 46 4d 54 5b 62 69 70 77 7e 85 8c 93 9a a1 a8
01 08 0f 16 1d 24 2b 32 39 40 47 4e 55 5c 63 6a 71 78 7f 86 8d 94 9b a2 a9
02 09 10 17 1e 25 2c 33 3a 41 48 4f 56 5d 64 6b 72 79 80 87 8e 95 9c a3 aa
03 0a 11 18 1f 26 2d 34 3b 42 49 50 57 5e 65 6c 73 7a 81 88 8f 96 9d a4 ab
04 0b 12 19 20 27 2e 35 3c 43 4a 51 58 5f 66 6d 74 7b 82 89 90 97 9e a5 ac
05 0c 13 1a 21 28 2f 36 3d 44 4b 52 59 60 67 6e 75 7c 83 8a 91 98 9f a6 ad
06 0d 14 1b 22 29 30 37 3e 45 4c 53 5a 61 68 6f 76 7d 84 8b 92 99 a0 a7 ae
raghavan@raghavan-Studio-1555:/tmp/dedupe_hashes$ cd 46
raghavan@raghavan-Studio-1555:/tmp/dedupe_hashes/46$ ls
0558 0d57 3df2 4e2a 5053 6d7c 7124 7b14 7d9c 808f 93e8 b489 ca38 ceee dd7d e839
09a1 278d 4d85 4f28 64f4 6da0 7480 7c30 7ec7 812c 99c6 b9c2 ccdf d819 de1d fd6c
raghavan@raghavan-Studio-1555:/tmp/dedupe_hashes/46$ cd 7d9c/
raghavan@raghavan-Studio-1555:/tmp/dedupe_hashes/46/7d9c$ ls
2dfe1e00
raghavan@raghavan-Studio-1555:/tmp/dedupe_hashes/46/7d9c$ cd 2dfe1e00/
raghavan@raghavan-Studio-1555:/tmp/dedupe_hashes/46/7d9c/2dfe1e00$ ls
cdb3a465eb435bbbf32d837355
raghavan@raghavan-Studio-1555:/tmp/dedupe_hashes/46/7d9c/2dfe1e00$ cd cdb3a465eb435bbbf32d837355/
raghavan@raghavan-Studio-1555:/tmp/dedupe_hashes/46/7d9c/2dfe1e00/cdb3a465eb435bbbf32d837355$ ls
467d9c2dfe1e00cdb3a465eb435bbbf32d837355 nlinks.txt
raghavan@raghavan-Studio-1555:/tmp/dedupe_hashes/46/7d9c/2dfe1e00/cdb3a465eb435bbbf32d837355$

```

Figure : 4 – Hash Block Store

If the chunk is removed from the file store, then just like file references, the chunk reference count is decremented. When the reference count reaches zero, the chunk is removed from the hash block store and the storage space is reclaimed by the system.

4.4 Metadata File

For each file written by the user into our filesystem, a corresponding meta data file will be created. This meta data file has the list of hashes for each chunk of data required to build the entire file and its corresponding location in the data store. This meta data also contains a basic file stat such as the directory structure, permissions, size, etc.

4.4.1 Design and Implementation

The metadata file is the essential component of our system. It consists of the file stat information, along with the offset range and the SHA-1 hash value for the data within the offset. Though two metadata file need not contain the same content, every file that is written into the file system will have a unique metadata file associated with it. For 'n' redundant files stored into the file system, there will be an equivalent metadata file stored in the metadata store.

The metadata file consists of the following structure. The first line in every metadata file will contain the stat information of the file. From the next line, the start offset, end offset and hash ID of all the hash blocks of the file is stored.

Figure 4 shows a snapshot of the metadata file in the metadata store.


```

raghavan@raghavan-Studio-1555:/tmp/dedupe_metadata$ ls
01. Soldier Soldier .MP3                               Ennamo Aedho - TamilWire.com.mp3
06 Kaadhal Yen Kaadhal - Dhanush, Selvaraghavan.mp3    Eppadio Maatikiten.mp3
Adiyae Kolluthey.mp3                                   Folder.jpg
AlbumArtSmall.jpg                                     server.c
Azhaghai%20Pookkutte.mp3                             Tamilmp3world.Com - Iragai Pole.mp3
raghavan@raghavan-Studio-1555:/tmp/dedupe_metadata$ cat server.c
2053:1314485:33204:1:1000:1000:0:130658:4096:256:1335488035:1335488035

0:8191:be18cff7f7d9109bc711e13e779183a437dccbc1
8192:16383:d728d9cdd38080be223bfa305b9afe3d78d4ec5b
16384:22809:135ce5b3ca21927f9fb5cf9f8eadb09de0ff5c57
22810:28681:17855cba8fd82be85fbe34a17da481ab8582ebb4
28682:36873:f790b3310a7c9e0c7db0a982e8e808472a65828f
36874:45065:fe69b07988abef1fd6324bf23a7111730a58d31f
45066:51273:4a2ca917c2de2dce325ad67e31c71907ac592547
51274:59465:32d6c13229ad588c93b1b48d77cbf9b48a6cf909
59466:63966:ae73bd7844ee0c2ddb90554046c76fa0aa6dad58
63967:72158:11148480bb32d3a9717fe02b349db6868a67e3e7
72159:77405:2a3409dd9ff0935d1c4753b87ddc718c349630b3
77406:85597:32d6c13229ad588c93b1b48d77cbf9b48a6cf909
85598:90098:ae73bd7844ee0c2ddb90554046c76fa0aa6dad58
90099:98290:11148480bb32d3a9717fe02b349db6868a67e3e7
98291:103537:2a3409dd9ff0935d1c4753b87ddc718c349630b3
103538:111729:da19874cbbd12871e43d5a19d3693a5c3c67c970
111730:116229:96be1d0b0b572be49678a5349432ecaa39f2322b
116230:124421:11148480bb32d3a9717fe02b349db6868a67e3e7
124422:129668:2a3409dd9ff0935d1c4753b87ddc718c349630b3
129669:130657:d4d2a08b783000fe3e09207d0cf7edc3dfe3a5ee

```

Figure : 4 – Metadata file

4.5 Thick Write and Bitmap Files

For each file created by an user, an equivalent thick write file and a bitmap file is created in the filestore. The thick write file is a sparse file used to store only the file modifications done by the user. Initially when user requests to create a new file, a thick write file is created and the data is stored in them. Another file called the bitmap file is also created which is an array of integers (bitmap) used to mark the modification of a 4K block in the file. These files are checked periodically by the dedupe thread during every dedupe pass to see if the file has been recently updated. If a bitmap is set for a 4k block then the corresponding 4K block is read from the thick write file and sent to the chunk divider module to perform Rabin-Karp fingerprinting and SHA-1.

4.5.1 Design and Implementation

Figure 5 shows the bitmap file and the corresponding thick write file in the file system.

```

dinoj@dinoj-Satellite-L655:/tmp/dedupe_file_store$
dinoj@dinoj-Satellite-L655:/tmp/dedupe_file_store$ ls -al
total 16
drwxrwxr-x  2 dinoj dinoj 4096 2012-04-26 19:55 .
drwxrwxrwt 22 root  root  4096 2012-04-26 21:39 ..
-rw-----  1 dinoj dinoj    0 2012-04-26 19:54 viminfo
-rw-----  1 dinoj dinoj 8192 2012-04-26 19:54 viminfo._bitmask
dinoj@dinoj-Satellite-L655:/tmp/dedupe_file_store$ od -xc viminfo._bitmask
00000000  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000
          \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
00200000
dinoj@dinoj-Satellite-L655:/tmp/dedupe_file_store$ █

```

Figure : 5 – Thick Write and Bitmap files

5. System Interactions

File System in User Space (FUSE) provides mechanisms to define our own APIs for file I/O system calls. We have made use of FUSE to implement the deduplication file system. Figure 6 shows how to create a dedupe file system using our implementation. The implementation lets user to create a new file, open an existing file, close an existing file, read the file, perform random read, write the file, perform random write, create a new directory, delete an existing directory, do a stat of the file, truncate the file, etc.

```

dinoj@dinoj-Satellite-L655:/tmp/dedupe_fs$ ls
testfile.txt
dinoj@dinoj-Satellite-L655:/tmp/dedupe_fs$ cat testfile.txt
"Show matching brackets
set showmatch

"Shows what you are typing as a command
set showcmd

"Set line number
set number

"Cool tab completion stuff
set wildmenu
set wildmode=list:longest,full

"Enable mouse usage
"set mouse=a
set ignorecase
set smartcase
set incsearch

"Spaces are better than a tab character
set expandtab
set smarttab

"Who wants an 8 character tab? Not me!
set shiftwidth=2
set softtabstop=2

"Highlight things that we find with the search
set hlsearch

dinoj@dinoj-Satellite-L655:/tmp/dedupe_fs$ █

```

Figure : 6 – Working Sample of dedupe_fs

Since we perform a lazy deduplication, our implementation contains a separate thread which periodically queries the filestore and checks the bitmap file and the thick write file for any changes to the filesystem. Note that during this pass any new file will be deduped into the hash block store and changes made to the existing file will also be deduped again.

Once the user adds a new data into the file system, the dedupe thread will read the directory to obtain the list of new files that need to be deduped. It will then lock each file using flock() (LOCK_EX and LOCK_UN) and retrieve the file stat. Our implementation first tried to make use of lockf(), however we found out that lockf() does not work when threads are used, hence we chose to use the flock() utility. The thread reads the bitmap file to obtain the list of fixed sized chunks modified since the last dedupe pass. Note that for a new file created, the entire bitmap for the file would be set and the entire file is read from the thick write file. Once the file has been successfully read, a new metadata file is created in the same name of the file. The stat of the file is stored in the metadata filestore and the file contents is sent to the Rabin-Karp module for boundary detection. The Rabin-Karp module will divide the file into variable-size blocks with their offsets. Once the offsets are known, the hash of the file block is created using SHA-1 algorithm, and only the new file block is stored in the file block store along with a reference count of one. If the hash already exists in the file block store, only the reference count of the file block is incremented.

If the metadata file already exists in the metadata store, then the dedupe thread has found a file which has been updated recently. However file updates can overwrite the chunk boundaries. [7] talks about different scenarios that need to be handled during file updations and those have been handled in the implementation.

```
dinoj@dinoj-Satellite-L655:~/Projects/fuse-2.8.6/example/Lazy-Deduplication$ make
rm -f dedupe_fs
rm -fr /tmp/dedupe_*
mkdir /tmp/dedupe_file_store
mkdir /tmp/dedupe_metadata
mkdir /tmp/dedupe_hashes
mkdir /tmp/dedupe_fs
gcc -Wall `pkg-config fuse --cflags --libs` -DDEBUG -DHAVE_CONFIG_H -D_FILE_OFFSET_BITS=64 -D_REENTRANT -g -w dedupe_fs.c
debug_log.c rabin-karp.c sha1.c -o dedupe_fs -lpthread -lm -lgcrypt -lgpg-error
dinoj@dinoj-Satellite-L655:~/Projects/fuse-2.8.6/example/Lazy-Deduplication$ ./dedupe_fs /tmp/dedupe_fs -s -d
FUSE library version: 2.8.6
nullpath_ok: 0
unique: 1, opcode: INIT (26), nodeid: 0, isize: 56
INIT: 7.16
flags=0x0000007b
max_readahead=0x00020000
[dedupe_fs_init] entry
[dedupe_fs_init] exit
[lazy_worker_thread] executing..
INIT: 7.12
flags=0x00000011
[internal_opendir] entry
max_readahead=0x00020000
max_write=0x00020000
unique: 1, success, outsize: 40
[internal_opendir] exit
[internal_releasedir] entry
[internal_releasedir] exit
unique: 2, opcode: LOOKUP (1), nodeid: 1, isize: 47
LOOKUP /.Trash
getattr /.Trash
[dedupe_fs_getattr] entry
[internal_getattr] entry
[internal_getattr] lstat failed on [/tmp/dedupe_metadata/.Trash]: No such file or directory
```

Figure : 7 – Make and FUSE

We have divided the directory structure containing the file blocks (data chunks) as follows. Initially the 160 bit hash value is converted to its hexa-decimal format into a 40 digit number. The first directory inside the file block store will contain a directory named with the first two digits of the hash value of the file block. The next level of directories will contain next 4 digits of the hash value of the file block and the next level to this directory will contain the next 8 digits of the hash value of the file block. The last level will contain the last 26 digits of the hash value of the file block. Hence while adding the file block into the block store, 4 directories need to be created. However, while searching for a file block, or while retrieving a file block, a single read is sufficient. We can achieve this by dividing the portion of the hash value with '/' and searching for the file block in the block store. Consider the following hash value fe692f8b11ec8f0e153eb6e88cbd82e0f2b98919. When issuing a read operation on the directory structure we would simply include '/' at location 2,4,8 and do an opendir system call.

Example directory structure

```
opendir("/tmp/dedupe_hashes/fe/692f/8b11ec8f/0e153eb6e88cbd82e0f2b98919")
```

6. Related Work

Data deduplication started a decade ago and is growing in importance due to the massive data burst [1]. Over these years, number of approaches to data deduplication has been deduced and employed in the data storage industries. Most of the research has gone into improving the execution time of inline processing and the post-processing method of data deduplication. Also much of the work has gone into determining the best approach towards storing the identifiers and retrieving them. In addition to these, several algorithms have been proposed for computing the hash value from the data blocks. [2] and [3] proposes deduplication models to remove the redundant data in the system. [4] proposes an efficient method to index the files. [5] proposes different methods for storing and retrieving the file chunks and the associated data blocks. [6] discusses the disk bottleneck involved in data deduplication and provides mechanisms to avoid them.

6. Conclusion and Future Work

This project is our attempt at data deduplication using a lazy approach. We have employed Rabin-Karp fingerprinting algorithm for dividing a file into variable sized blocks, SHA-1 for obtaining a hash value on the data block and handled file I/O system calls using function pointers exposed by FUSE. As a future work to this project, efficient cache can be implemented to store the hash values of the data blocks in memory rather than on disk. This would involve using sophisticated tree/trie data structures in the design. At present the filesystem support a maximum file size of 256MB. To increase this limitation, bitmap file size need to be increased.

7. References

- [1] Geer, D.; , "Reducing the Storage Burden via Data Deduplication," Computer, vol.41, no.12, pp.15-17, Dec. 2008
- [2] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The design of a similarity based deduplication system," in Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, ser. SYSTOR '09. New York, NY, USA: ACM, 2009.
- [3] Jaehong Min; Daeyoung Yoon; Youjip Won; , "Efficient Deduplication Techniques for Modern Backup Operation," Computers, IEEE Transactions on, vol.60, no.6, pp.824-840, June 2011
- [4] Thwel, T.T.; Thein, N.L.; , "An Efficient Indexing Mechanism for Data Deduplication," Current Trends in Information Technology (CTIT), 2009 International Conference on the, vol., no., pp.1-5, 15-16 Dec. 2009

- [5] Quinlan, S.; Dorward, S.; , “Venti: A New Approach to Archival Storage,” Proc. Conf. File and Storage Technologies (FAST ’02), pp. 89-101, Jan. 2002.
- [6] B. Zhu, K. Li, and H. Patterson, “Avoiding the disk bottleneck in the data domain deduplication file system,” in Proceedings of the 6th USENIX Conference on File and Storage Technologies. Berkeley, CA, USA: USENIX Association, 2008.
- [7] Athicha Muthitacharoen, Benjie Chen, and David Mazieres, “A Low-bandwidth Network File System”, in Symposium on Operating Systems Principles, pages 174-187, 2001
- [8] <http://algs4.cs.princeton.edu/53substring/RabinKarp.java.html>
- [9] <http://www.copiun.com/blog/bid/45383/Corporate-PC-backup-Anatomy-of-a-Data-Deduplication-System>
- [10] <http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=stringSearching>
- [11] http://en.wikipedia.org/wiki/Cryptographic_hash_function
- [12] <http://www.kohnos.net/files/avail.pdf>
- [13] <http://info.zetta.net/raid-5-raid-6-rain-6-data-integrity-enterprise-online-backup/>