CURE CANCER, OR SOMETHING

Ayush Sengupta[1] & Ankit Arun[2] &
Manoj Alwani[3] & Nitish Gupta[4]

5th December, 2014

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 ABSTRACT

At OSDI 12, the keynote speaker was Jeff Haussler of the UCSC cancer genomics hub. He expressed a need for collaboration between systems researchers and researchers in the biological sciences. According to Dr. Haussler, *"There are two types of problems. One is I give one person's personal genome, and you're supposed to interpret it. The other is that you have a database of thousands or millions of genomes and you're supposed to search that for patterns that are common and associated with certain kinds of disease attributes or clinical response. Both of them are incredibly important types of problems that are distinct."*

Our goal is to make a prototype to solve the second problem - given a large stream of text, we must store it efficiently in file system and query it for matches. Genome data has the property that there exists duplication within a sequence as well between two sequences.

We are utilizing this property of genome data to store it in memory. We have used content aware chunking and inline deduplication of data to store the non repeating blocks of data into the file system.

2 ARCHITECTURE

The file system we have designed is of size 4 GB which is divided into 4 major blocks (as shown in Figure 1 on the following page). The 4 blocks are briefly described below:

2.1 Counter-Block Partition

The first block contains the address of the tail of super block and data block. It is of 8 bytes size. This block is useful to keep the track of the first empty location in super block and data block.

2.2 Super-Block Partition

The super block contains the file metadata. File metadata constitutes of information like file name which acts as a key in searching, size of file in bytes. The file metadata also stores pointers to a set of blocks which stores the data that each file contains.

2.3 Hash-Block Partition

The hash block serves the purpose of a hash table. It has 2^{25} entries. Each entry stores a pointer to a the base address of a block, that is the second level hash table. The size of the hash block is 32MB. The architectural view of hash block is shown in Figure 2 on page 4.

[1] *Department of Computer Science, Stony Brook University. SBU ID - 109753175, CS ID - aysengupta*
[2] *Department of Computer Science, Stony Brook University. SBU ID - 109914679, CS ID - aarun*
[3] *Department of Computer Science, Stony Brook University. SBU ID - 109335757, CS ID - malwani*
[4] *Department of Computer Science, Stony Brook University. SBU ID - 109904575, CS ID - nitigupta*

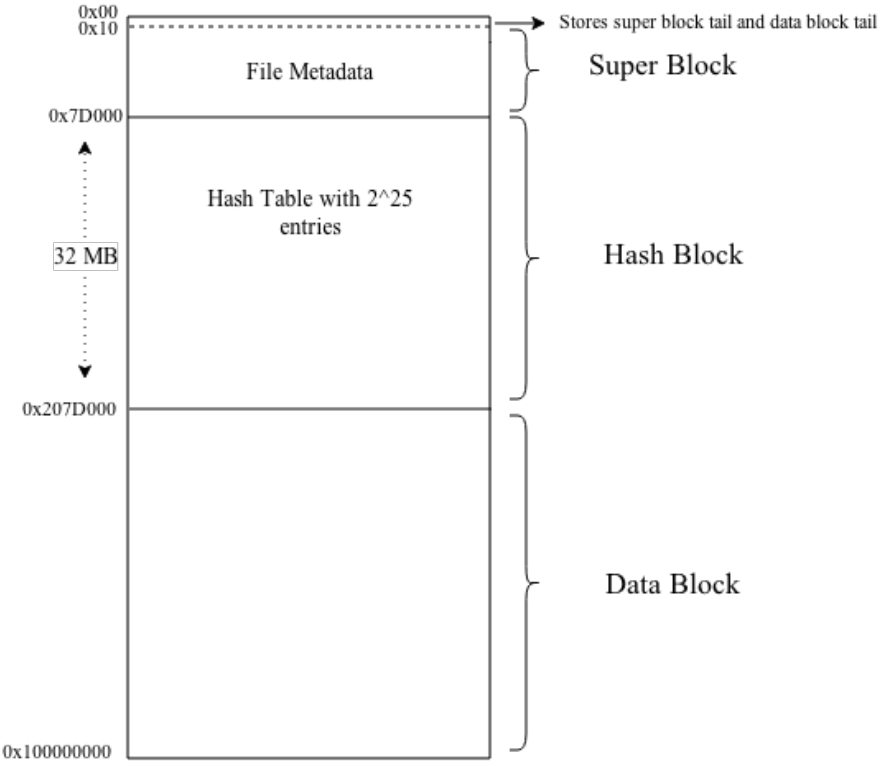

Figure 1: A structural view of the file system.

2.4 Data-Block Partition

The data block partition which comes after 0x207D000 contains the main data. This partition has two kind of blocks, **data block** and **meta-data block**. The data block contains actual genome sequence data(i.e. a string of characters A,C, G and T). The metadata block on the other hand stores a set of 128 pointers(each 4 byte in size) that points to data block addresses. The metadata block essentially stores the file metadata. This is useful because our files are large and hence instead of storing the entire metadata in our file structure, we only store pointers to metadata blocks only. This keeps our file structure small and gives us the freedom to have lots of files in our system. The size of each block is 512 bytes.

3 TECHNIQUES AND ALGORITHMS

First, given a large fasta file where each base pair is represented using one byte(UTF-8 encoding), we first divide the file into chunks based on rabin fingerprinting. We use the algorithm which was first introduced in A Low-bandwidth Network File System by Muthitacharoen et. al.[1].

The algorithm works as follows. First we take a window of size w characters from the beginning of the string. We then compute its Rabin fingerprint in GF(4)(with respect to a large prime M) and we define the last byte of the window as a chunk boundary if the t lower order bits is equal to zero. After that we shift the window by w bytes and start the search for another chunk

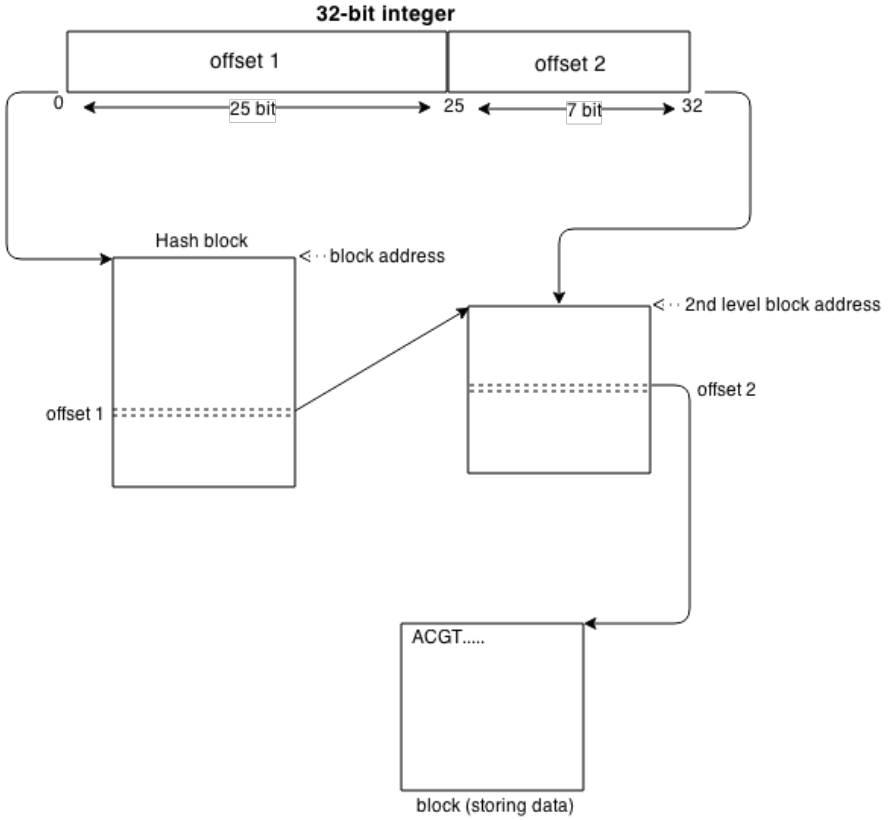

Figure 2: Architecture of Hash-block

boundary. On the other hand, if the t lower order bits is not equal to zero, we shift the window by a byte. We can get the new fingerprint for the next window in $O(1)$ time. We keep on iterating through the file until we recognize all the chunks.

Figure 3 on the following page shows the three cases of data insertion into content aware chunks.

- data is inserted into the middle of a chunk
- data is inserted that includes or produces a new chunk boundary
- data is inserted into a window that contains a chunk boundary

In first case, only the single block containing the new data must be rechunked. In both second and third cases, the containing block up to any new boundary is fingerprinted, followed by the successor, until an existing chunk boundary is encountered.

We have used $t = 12$ and $t = 10$ for our purpose. Assuming that w is large enough(48 in this case) and each of the 2^t different lower order bits combination are equally likely(this is true for an appropiate choice of M). Hence, we expect to encounter a chunk boundary after 2^t characters. This is nearly equal to 4k characters. Obviously, it is possible that our input defines no chunks or defines multiple very small chunks. To overcome this problem we have used a minimum chunk size of 2000 base pairs and a maximum chunk

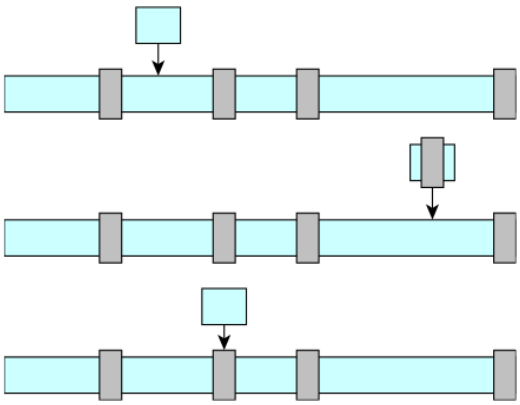

Figure 3: Three cases of insertion in content aware chunking

size of 16000 base pairs. Our content aware chunking is now immune to arbitary insertions and deletions. This is because any insertion in one place will not affect the boundaries of other chunks.

Once the files are chunked, we compress it for decreasing the storage space. Each base pair that is represented using 1 bytes can actually be represented using 2 bits(00 for A, 01 for C, 10 for T and 11 for G). This compression in itself reduces the space used by a factor of four. Now the compressed files are inserted into our file system.

We have designed our file system for efficient inline deduplication. To do inline deduplication, we first divide our chunks into blocks of size 512bytes. Now, we read each block, calculate its SHA-1 hash, and take the last 32 bits of the hash.

To store our block with respect to the 32bit hash that we get, we use a two level hash table. This is essential since a single level page table would require 2^{32} entries(each storing a 4byte address). Hence our hash table would have required 4GB of storage space! To solve this problem we use the first 25 bits of the hash to traverse a hash table and get the address which is the base of a second level hash table(Figure 2 on the previous page). Now the corresponding hash entry in the second-level hash table is recognized using the last 7 bits of the 32 bit hash. If the address in the corresponding hash address equals 0, it implies that we have encountered a new block. We then store the block in the file system and write the pointer to the block in the second-level hash entry. On the other hand, if the address is not NULL, we assume that the data in the block has already been written(i.e. this is a duplicate block) and we do not store the block(**deduplication**). Here we assume that there will not be any collision, which is a valid enough assumption since we are using the last 32bits of SHA-1.

Our file system has three logical partitions. The first partition(the **superblock partition**) essentially stores a part of the file's metadata. A file's metadata is represented using a structure which stores the size of the file, the name of the file. The file name is also an unique key to represent the file. The structure also stores an array of pointers to a set of metadata-blocks. These meta-data blocks in turn stores pointers to blocks that actually stores the

files data.

The second partition(the **hash block partition**) stores the first level hash table. The third partition(the **data-block partition**) stores each block of data(which is equal to 512bytes). We chose the block size as 512 bytes because we perform each read and write operation in multiples of 512 bytes. This is done primarily to make sure that we maintain the disk read and write granularity. The second-level hash table, that also requires 512 bytes of storage space, is stored as a block in the third partition itself. This saves us a lot of space because the second-level hash table block is created on demand.

4 DATA SET

We have downloaded the genome of E.Coli, a bacteria found in the lower intestine of humans. The genome is publicly available on the Ecogene website (`http://www.ecogene.org/`). We have downloaded the following three sequence, as a fasta file for comaprison and analysis:

- E. coli K-12 MG1655 U00096.1 (1 to 4639211 = 4639211 bp)
- E. coli K-12 MG1655 U00096.2 (1 to 4639675 = 4639675 bp)
- E. coli K-12 MG1655 U00096.3 (1 to 4641652 = 4641652 bp)

We have parsed the indivisual equences to remove white spaces newlines, and header information.

5 RESULTS AND OBSERVATIONS

5.1 Results

The files used to calculate the reult are:

- File 1 - E.coli U00096.1 (F1)
- File 2 - E.coli U00096.2 (F2)
- File 3 - E.coli U00096.3 (F3)

We have checked deduplication ratio for 1, 2, 3 and 6 files. The deduplication ratio is defined as:

$$\text{Deduplication Ratio} = \frac{\text{total \#blocks in the file}}{\text{total \# of blocks actually written}} \quad (1)$$

5.2 Fixed size chunking

The table 1 on the following page shows deduplication ratio for E.coli sequences using fixed size chunking.

Table 1: Deduplication Ratio for Fixed Size Chunking

Files Used	# blocks	# blocks duplicated	Deduplication Ratio
F1	9061	3	1.00033
F2	9062	0	1
F3	9066	1	1.00011
F1, F2	18123	4090	1.29145
F2, F3	18128	510	1.02894
F1, F3	18123	515	1.02924
F1, F2, F3	27189	4610	1.20417
F1, F2, F3, F1, F2, F3	54378	31795	2.40791

5.3 Content Aware chunking

5.3.1 *The table 2 shows deduplication ratio for E.coli sequences using content aware chunking when t=10 and w=48.*

Table 2: Deduplication Ratio for Content Aware Chunking(t=10 and w=48)

Files Used	# blocks	# blocks duplicated	Deduplication Ratio
F1	9061	12	1.00121
F2	9062	15	1.00152
F3	9066	15	1.00152
F1, F2	18123	9410	1.90994
F2, F3	18128	9856	1.99605
F1, F3	18123	9378	1.90416
F1, F2, F3	27189	19248	2.85522
F1, F2, F3, F1, F2, F3	54378	48871	5.71045

5.3.2 *The table 3 shows deduplication ratio for E.coli sequences using content aware chunking when t=12 and w=48.*

Table 3: Deduplication Ratio for Content Aware Chunking(t=12 and w=48)

Files Used	# blocks	# blocks duplicated	Deduplication Ratio
F1	9061	11	1.00116
F2	9062	11	1.00116
F3	9066	11	1.00116
F1, F2	18123	8751	1.86174
F2, F3	18128	9428	1.99399
F1, F3	18123	8717	1.85511
F1, F2, F3	27189	18168	2.78170
F1, F2, F3, F1, F2, F3	54378	46537	5.56334

5.4 Observations

For our experiments we take the window size as $w = 48$, which seems like a good choice. We try our algorithm for t = 10 and t = 12. We find better results for t = 10, primarily because that divides our file into smaller

chunks. Another reason for relatively worse result for $t = 12$ is probably because we observe a lot of forced chunks in that case(i.e the chunk size exceeds 16k base pairs and thus chunking happens forcefully). Another interesting factor in our system is that we always make sure that we read and write in multiples of 512 bytes to maintain the disk reading and writing granularity.

Our experiments also shows the efficiency of content aware chunking. Whenever we insert or delete a single base pair in the same file, the deduplication ratio remains 1 in case of fixed block chunking. In case of content aware chunking we find that the deduplication ratio almost becomes 2.(as expected)

ACKNOWLEDGEMENTS

We are thankful to William Jannen for his help and guidance. Few discriptions of content aware chunking algorithm in this report has been taken from his research proficiency paper.

REFERENCES

1. Muthitacharoen, Athicha, Benjie Chen, and David Mazieres. "A low-bandwidth network file system." ACM SIGOPS Operating Systems Review. Vol. 35. No. 5. ACM, 2001.
2. Rabin, Michael O. Fingerprinting by random polynomials. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.
3. C code to generate SHA-1 values is taken from http://www.packetizer.com/security/sha1/