File system for efficiently storing large genome sequences using content aware chunking and inline deduplication.

Architecture: We have designed and implemented a file system for efficiently storing large genome sequences. This is how our architecture works.
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 Lowbandwidth 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 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.

We have used $t = 10$ 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 $t2^t$ characters. This is nearly equal to 10k 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 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. 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**) 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 partition**) stores the first level hash table. The third partition(the **data-block**) stores each block of data(which is equal to 512bytes). 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.
Our file system is designed to efficiently store large genome sequences that have large regions of similarities among themselves.