Deduplication in Cloud Storages

Author: Gantavya Bhatt, Saurabh Kumar and Mihir Kumar.

Introduction:

Data deduplication is an essential and critical component of backup systems. Essential, because it reduces storage space requirements, and critical, because the performance of the entire backup operation depends on its throughput. Traditional backup workloads consist of large data streams with high locality, which existing deduplication techniques require to provide reasonable throughput.
Here our team did the study of methods of deduplication through which we can do deduplication.
In this project, we did 4 methods of deduplication - Fixed Length Chunking , Variable length Chunking , Sliding gate chunking and Extreme Binning. Before starting off with the main techniques, we will like to explain the subsidiary functions that we studied and implemented.

Rabin Karp's Rolling Hash:

This is a hash function that returns the integer hash key as the output. This involves conversion of the string into a number by the use of the ASCII values of the character and using a prime number for the modulus purpose. The general formula for the calculation of the hash value is

t = (k*t + str.charAt(i))%prime

Where the k is the base, which we consider for creating a polynomial. Much details and Math behind the algorithm is present in the codes attached along with.

Cryptographic Hash Functions:

In our project we used the cryptographic hash functions – MD5(Message digest).Like most hash functions, MD5 is neither encryption nor encoding. MD5 processes a variable-length message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit words); the message is padded so that its length is divisible by 512. The padding works as follows: first a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits fewer than a multiple of 512. The remaining bits are filled up with 64 bits representing the length of the original message, modulo 2^{64}. The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted *A*, *B*, *C*, and *D*. These are initialized to certain fixed constants. The main algorithm then uses each 512-bit message block in turn to modify the state. The processing of a message block consists of four similar stages, termed *rounds*; each round is composed of 16 similar operations based on a non-linear function *F*, modular addition, and left rotation. There are four possible functions. More implementation details are present in the attached JAVA code of the MD5 along with this report.

Fixed Length Chunking:

As the name suggests, this method involves division of the complete file into the chunks of fixed lengths. In the program , an iterator iterates over the complete folder and then for each file it will make constant length piece of strings, called as chunks. These chunks are stored in a form of array list. After their creation and the storage in the list, these chunks are then passed through the MD5 hash function. This will return the 128bit hash value of these chunks. Simultaneously, a hash map will also store the hash value of these chunks and it will map to the string values of these chunks.

By the property of hash map, that is it cannot contain the chunks of the same hash value, it will store only the unique chunks. Thus it provided a way to store the files in a easy way. Further the hash map can be written on the hard drive and the memory can be freed.

Variable Length Chunking:

Fixed length chunking creates a severe issue when any block of string is added prefix. This will cause the shifting of the complete string and thus can even change the MD5 hash value completely for every chunk, even by adding a single character. To overcome this we came up with an idea of variable length chunking. A variable length chunk can be created throughout the file by the following:-

Sliding Gate Algorithm:

This is an easy algorithm. Consider a String "abcdefgh". In this algorithm we will create a variable length gate. Consider the gate pointer(not to be confused with the address and memory pointer) pointing to the 1st character of the string. We will choose 2 positive numbers d and r with a special relation that r<d. Now we will consider the string starting from the 0th position to the position of the pointer in the string. For substring so formed its "Rabin Karp's Rolling hash" is evaluated. After the evaluation of the hash value, the following equation is queried - is (hash)mod d = r ? If this relation holds good here, then the pointer index is considered as the chunk boundary, and then the whole process is repeated for the rest of the file.

Sliding window Algorithm:

This algorithm is similar to the previous algorithm. Here we will consider a window that is the group of the characters in the main text we are considering for the chunking and we will slide the window through the text. Here we consider the length "n" of the window. First, we will evaluate the hash value of the first "n" characters of the text. Then it will be compared with the above given mathematical condition i.e. its modulo with d will be compared with r. If the equality holds, then we will consider the last point of the window as the boundary of the chunk, otherwise we will move the window and recalculate the hash but dynamically i.e. by considering the previous hash value, just subtracting the previous first character of the window, and adding the next character of the window in the hash accordingly. Now its modulo with d will again be compared and the whole process repeats

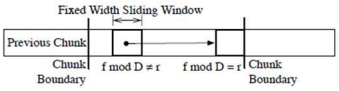


Fig. 1. Sliding Window Technique

Here in the fig. f denotes the hash value of the window.

So far, these 2 methods were the methods of creating the chunks, the later part remain the same as of the fixed chunking algorithm. We will calculate the MD5 hash of the chunks and simultaneously insert them in a hashmap mapping to the string value. Thus by this process, no similar chunk can be there in the map. Thus, we ended up reducing the stored chunks.

After this we will store these in a bipartite graph whose one side will be the files and the other side will be the chunks. Clearly if the file have the similarity in the chunks, then there will be a matching in the graph and leading to the cycle in the bipartite graph. This can be further detected by performing the union find operation and Ford Fulkerson maximum matching algorithm (beyond our scope of knowledge, but is useful in these). Later these can be written in the hard drive for the permanent storage.

Extreme Binning:

This is entirely different technique and an effective deduplication technique. This technique use the address of the storage in the hard drive and thus tries to reduce the disk IO calls throughout the storage. In this technique, first of all we will divide the files into the chunks by one of the above the above methods, and then we will handle a table involving the following fields – Minimum Chunk ID , File MD5 hash, Pointer/URL of the storage of the chunks. For the implementation point of view, we had added more fields and that could be viewed in the programme that we had attached with.

Minimum chunk ID is the minimum hash value among the hash value of the chunks of the particular file. Broder's theorem state for 2 sets S1, S2 So, if S1 and S2 are highly similar then the minimum element of H(S1) and H(S2) is the same with high probability. Extreme Binning exploits this fact and so uses the minimum chunk ID.
When a file will be provided to our program, it will first of all calculate the minimum chunk ID, file hash and will store the respective unique chunks in the file known as "bin" . These bins are located in the hard disk and the bin is allotted the name by the chunk ID for that particular file. There is a pointer from the RAM to the bin, so that these bins can be loaded any time when required.
When another file comes in the program , its chunk ID and file hash is calculated. Now if the minimum chunk id doesn't exists, then a new bin is created and the unique chunks are stored in it.
If the ID is present in the table in the RAM, then its file hash is compared. If the file hash is same then it means that the file is duplicate and so none of the chunk is stored and no disk calling happens.
If the file hash doesn't matches, then the bin corresponding to that chunk ID is loaded into the hard drive(IO call happens here) and then only unique chunk is then stored into this old bin. The complete process goes throughout similarly.

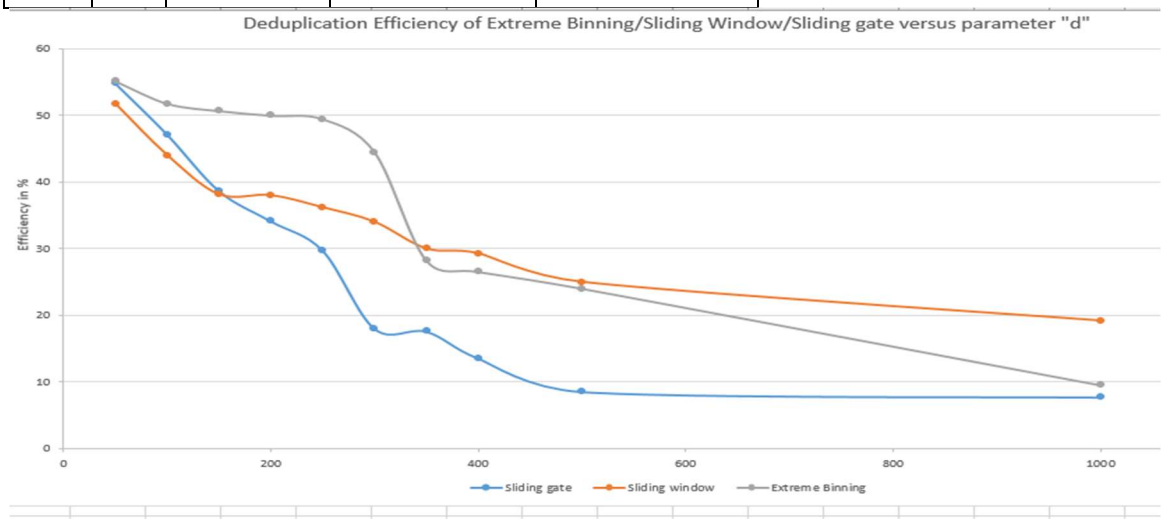
Testing:

All these 4 algorithms are now tested on the data set which includes 9 files, comprising of 3 different file sets and which also have its own versions.

Inference:

1)Following is the table of maximum deduplication obtained with the change in parameters "d" and "r"

d	r	Sliding gate	Sliding window	Extreme Binning
50	35	54.82	51.72	55.14
100	80	47.14	44	51.77
150	135	38.67	38.12	50.67
200	180	34.14	38	49.68
250	235	29.72	36.17	49.45
300	280	18	34	44.48
350	335	17.58	30	22.22
400	380	17.256	30	28.53
450	435	13.56	29.29	26.53
500	480	8.54	25	24.11
1000	750	7.71	19.2	9.51

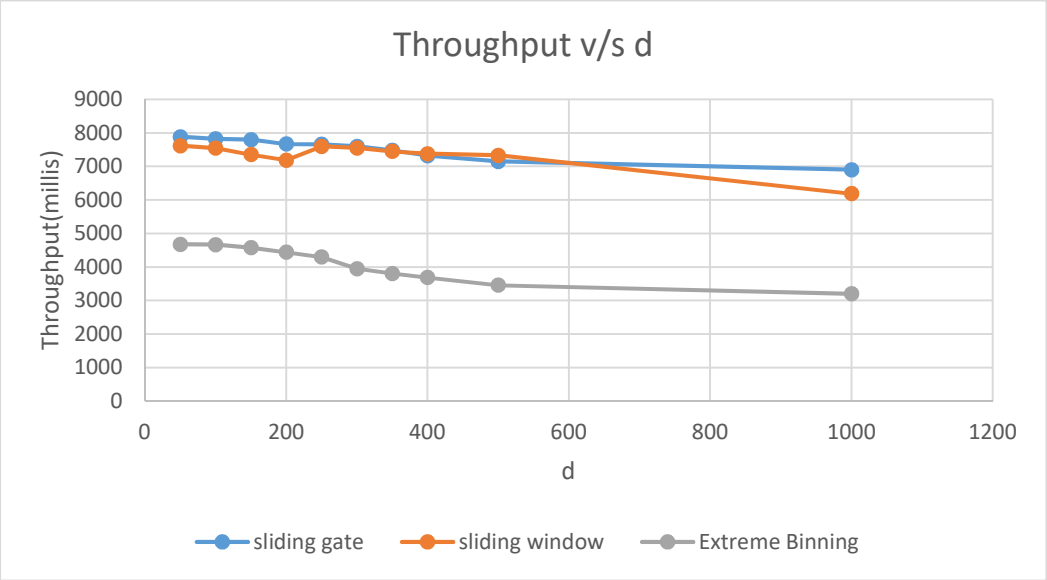


Here we can see that as the value of “d ” increases, then the percentage deduplication decreases rapidly . This is because the value of d sets the bar of length of chunk because if d is less then its highly probable that the modulo turn out to be equal to r , while the large value of d cannot guarantee the small values of chunk.
Small values of chunk will imply less character, thus we can find out more similar content in the complete file while the large chunks will imply large characters, thus we cannot say that the their MD5 hash will come out same or not. Thus the deduplication efficiency will decrease.

2) Another inference we can make regarding the running times. The first three algorithms involve multiple times hashmap/ graph construction. Note that we cannot do all the storages in the memory thus we will require on an average one IO calling per file in the storage process. Since these are tested on core i3 processor, and according to that, average time went due to the IO calls only came out to be near to 4170 millis.
When tested on a data set consisting of all the same files, all the other algorithms took much larger time with respect to the extreme binning because extreme binning does IO call only when the files are different.

3) Following is the table of the throughput (millis) with the variation of the parameters:-

d	r	Sliding gate	Sliding window	Extreme binning
50	35	7885	7616	4677
100	80	7821	7545	4666
150	135	7800	7355	4575
200	180	7668	7188	4444
250	235	7659	7599	4300
300	280	7599	7551	3951
350	335	7482	7447	3809
400	380	7322	7376	3688
500	480	7147	7330	3458
1000	750	6900	6190	3200



Note that we can see that the throughput of extreme binning is minimum as explained above in point #2. Here we can also see that as the in all the techniques, as the value of the d is increasing, the

throughput is decreasing. This can be explained as – decreased d value implies that the number of chunks will be much larger and as the number of chunks increases, the computation time will automatically increase.
4) The fixed chunking algorithm was affected adversely. Its running time was coming less than that of the variable length chunking, because no Rabin Karp's Hash computation required in the process of chunking, but the efficiency was very poor with respect to the others coming around 2 percent only. This happened because of the avalanche effects of prefix addition and thus leading to shifting of texts.

Conclusion:

From the results so obtained from the project, we can clearly see the trade-off between the deduplication efficiency and the computation time. In practice as a client point of view we want that our computation time should be small, but we also want that our storage should be used efficiently. So we can conclude for the company point of view where the efficiency and time both matters, we can use the ***extreme binning algorithm.*** This algorithm performs well even if we have prefixes as this uses the variable length chunking.

Future Scope and Next Targets:

We can see that in case of encrypted data, we cannot see the user's information. Thus for encrypted data we need a different algorithm, which can efficiently work both in terms of deduplication as well as computation time.

Acknowledgement:

We pay our gratitude to Prof J. Harshan sincerely for giving his precious time and guiding us to this new field related to distributed systems.

References:

1) Wikipedia pages for the data set.

2) Bhagawat, D., Extreme Binning: Scalable, parallel deduplication for chunk-based file backup, *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*

3)Finding similarities in the large repositories HP research paper.