Research On Modification Operations of Google File System

Yi Qi yq97

Yuhang Zhou yz853

Abstraction

This report is research about which will influence random write in the google file system and how to optimize it.

Introduction

This project is research on modification or writing operations based on the Google File System(GFS). Almost all writing operations of the traditional GFS are sequential write and most files are mutated by appending new data. And it was said that random writes within a file practically non-existent. So we implement a basic GFS architecture with random write operations and made some researches on that.

The Google File System is one of the most popular distributed file systems in the world. It helps users to expand their storage with their data servers. Users can also have access to their data with any devices. The authority management and backup servers will make sure data security.

Our research on writing is mainly about the modification operation of GFS. The modification operations are mainly designed for modifiable files like text files. In the tradition GFS, the whole file should be appended with sequential write, while in our design, only the modification part and the modification positions should be uploaded, then the chunk server will finish modifications. The data transmission time will reduce while the data or file processing time may increase and more hardware resources are needed.

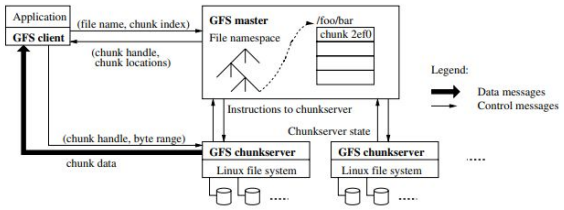
Analysis

Background

The Google File System(GFS) is a scalable distributed file system for large distributed data-intensive applications. In the design of the GFS, component failures are thought of as the norm, it’s designed for large files, most files are

mutated by appending data instead of overwriting it, and APIs are designed to be flexible.

The main architecture of GFS mainly contains a master server, multiple chunk servers, and it can be accessed by multiple clients. The chunk server is the server to save file data. Files are divided into fixed-size blocks and be saved in the chunk server. And every block will have some replications in other chunk servers for backup. The master server is designed to manage all the file system metadata like the namespace, access control information, mapping from file to blocks and location of blocks. The client contains the APIs and it's designed to communicate with the master server and chunk servers to read and write file data.



There are some other features of GFS to improve its performance. The block size is designed as 64MB, which can reduce the number of requesting block position times. The master server is very fast by saving metadata in the memory and it also updates the information of chunk servers periodically. Operation logs are also saved so that the recovery process can take a shorter time if some damages happened.

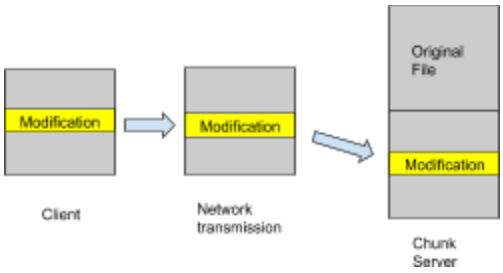
To improve the fault tolerance of GFS, chunk servers can be replicated easily. Although there is only one master server in the system, every master server still has a shadow master server which will take over if the main master server is down. The checksum is used as the method to check data integrity.

Motivation

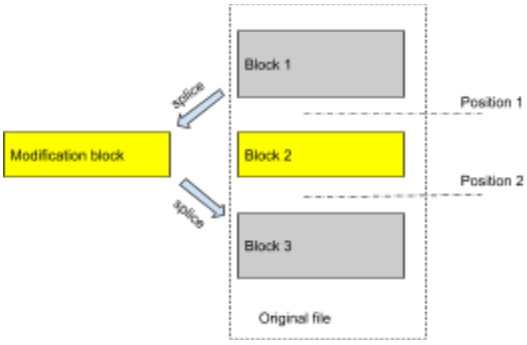
The Google File System is a pretty mature system but we still have some questions with it's writing operation.

It was said in the GFS paper that "most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially.". So if the

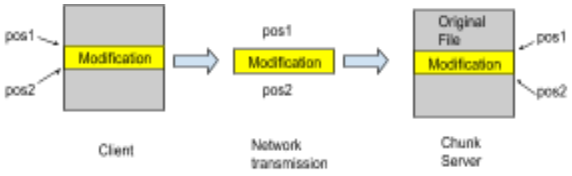
user would like to make some modifications to the existed file, the whole file should be uploaded to the chunk server after modifications and the chunk server will sequentially write the whole file after the old file instead of rewriting it.



In our design, we take random write operations into consideration. The advantage of random write is that it can write data on any position as we want so that we can transfer just the modification data to the chunk server if the modification positions also can be recorded.



There also some disadvantages of random write



s. Random writes are always slower than sequential writes, especially on the hard disk drive. Also, it requires much more CPU resources than sequential write. In our implementation, firstly the program will divide the original file into three blocks with two input positions. These two positions record the date range to replace with the modification data. Then the program will splice the first block, modification block, and the last block into a new file.

These file parting and splicing operations should be done by chunk servers. It's quite important to mention that in the design concept of GFS, the chunk server architecture is designed to be deployed on hundreds or even thousands of storage machines built from inexpensive commodity parts. So the performance of chunk servers should be thought to be bad, so it will be hard to decide whether it's appropriate to implement random write on these

machines for its high requirements for hardware performance although it can save lots of data transmission time. So we made some researches and experiments on comparing sequential write and random write based on different conditions like different modification sizes, file sizes or others. And find the balancing point or specific conditions that random writes can perform better.

Analysis & Evaluation

The whole experiment is based on 2 laptops and 2 iLab servers. In the experiment, we use the two laptops as two clients and one iLab machine as a master server and the other iLab machine as a chunk server. We write a simple GFS system with python and use rpyc protocol as the communication protocol to connect the distributed system.

The basic operation is that one client uploads the origin file, then the other client does the modification operation use different methods. We record the modification time under different scenarios and analysis what causes these results to finally find a good strategy to choose the modification method. As we talked about above, one method is sequential write, the other is random write.

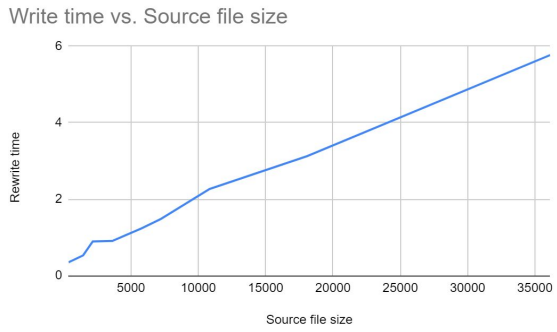
To test the performance between different scenarios, we create 10 different sized original files and 20 modification files based on different percentages of the original files,10 and 50 percent, and after each turn of an experiment, we will renew the origin file.

So first, we do experiments compares the file size's influence when the modification is 10% of the original file size. The experiment is that after uploading the origin file. We will modify 10% of each file. For example, if we have a txt file with 100 words, we will change 10 words of it

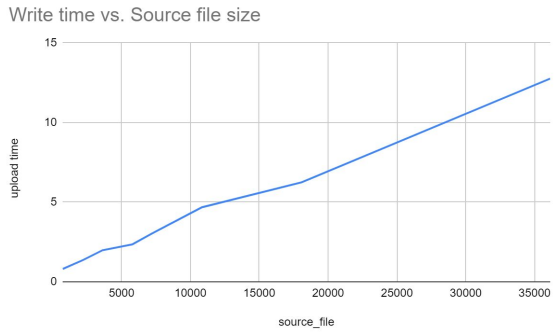
The next experiment is to test the modification time of modifying 50% of the origin file. And we also have these 10 different sized origin file and corresponding modification files.

Observation

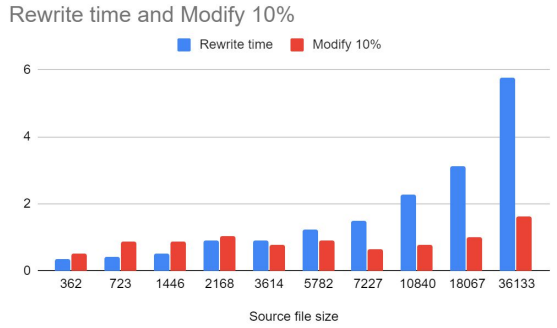
Firstly we do the sequential write experiment and get the write time vs file size figure.



It is obvious that the write time is linear to the file size.

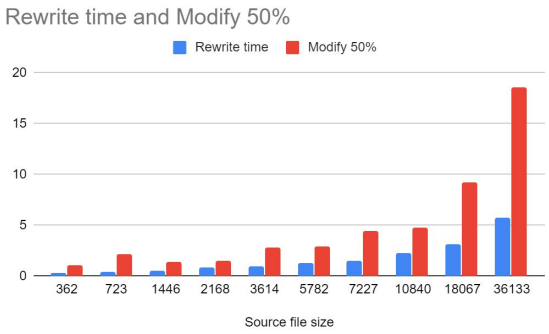


Then we get the data from the 10% modification experiment.And all of them are doing the modification from the beginning..

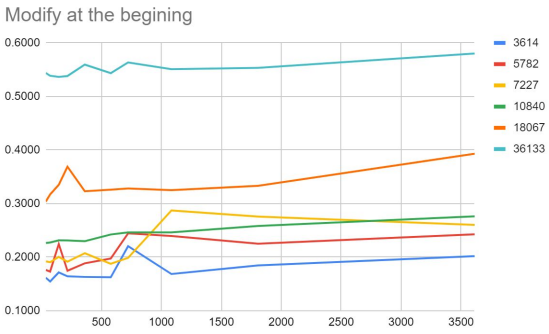


It is interesting that the 10% method uses a much lower time when the origin file size is big. And in small files, it doesn't perform well since the time saved from uploading fewer files doesn't cover the time chunk server takes to do the write operation.

Then we test 50% modification



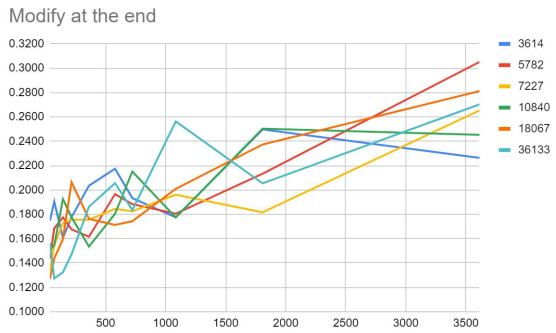
And it is obvious that when we want to use the chunk server to deal with a lot of data, its penalty is much higher than the time saved from smaller files uploaded.



Then we tested the time cost under different source file size and and modification size. This time we choose to modify it from the beginning. The x

axis explains the modification size and the y axis shows run time, different curve means different source file size. So we can obviously see that modification size doesn't influence runtime much in this scenario. And it is the source file size which becomes the issue since each source file size has its exclusive run-time curve.

Then it comes to another scenario where we do the modification at the end of the file.

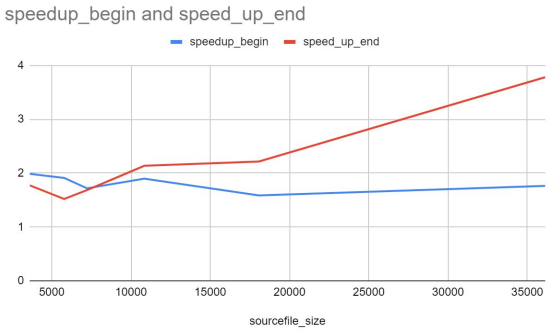


The x axis explains the modification size and the y axis shows run time, different curve means different source file size.

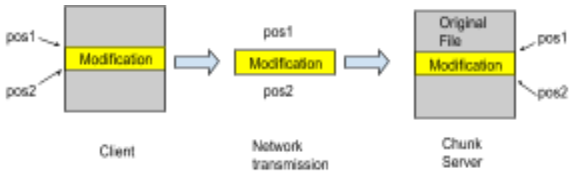
We find that in this scenario, modification size dominate the influence of runtime. Different source file size curve seems very similar.

This next figure shows the speed up between modification at the end and at the beginning.

The x axis shows the source file size and the y axis shows speed up.



When the source file size is small, there is no big difference between the two curves. But when the source file size become larger, modify at the end has a far more higher speed up with same modification size.



So why the run-time is big when we want to do a modification at the beginning. We can see in figure above, in our method we will rewrite the file after pos2 which is we read it to memory then write it to disk again. And it do two I/O operations which is expensive if this part is large. So when we modify a large chunk at the beginning we will read a big file and then write it back which cost a lot of time compared to other parts.

Conclusion:

So we have done some experiments about GFS file modifications. We find that if the modification size is small compared to original size. We have have a good performance.

And when we do the modification at the end of the chunk, we can have a better performance than write at the beginning.

And both of them are because of the modification operation need to read the end of file and write it to disk again, which doubles the I/O load. It is expensive compared to other operations.

Then if we want to modify a big file with small modification or modify at the end of a chunk, we could use the modification method to get a better performance.