**Big Data Platforms**

**Small files and MapReduce**

**Benjamin Scholom and Dror Meirovich**

**01/03/2022**

**github: https://github.com/drormeir/Reichman\_BigData\_Project**

**Abstract**

Small files are a problem in Mapreduce run both over HDFS and over Object Storage – albeit for different reasons. For HDFS, small files take up their own blocks – causing an increase in metadata that puts stress on the Namenode and also increasing the processing time for Mapreduce. In Object Storage, the inability to rename/edit objects means that changing files becomes very I/O intensive – which can cost more money in a public cloud environment at the worst or just slow down processing speed. A number of solutions have come up to address these issues. In HDFS, solutions revolve around aggregating smaller files, increasing the number of Namenodes, or altering Mapreduce to deal with small files more appropriately. For Object Storage, solutions also revolve around aggregating files, but also changing Object Storage to be more amenable to operations required in Big Data processing. In this paper we decide to focus on merging files using a virtual file layer and a small files aggregator rather than changing object storage itself, as we felt many users would not have control over their object storage environments and thus would need an easy to implement solution they could use.

**Motivation and Background**

Since the dawn of the information era in the late 20th century, the amount of data in existence is increasing exponentially every year. As a result, there is inherent demand for systems and frameworks that can process this data efficiently. In an effort to tackle the issues around large amounts of data or “Big Data”, Google developed GFS (Google File System) in 2003 order to store the large amount of information involved in running its search engine. A year later, the company developed Mapreduce so that the aforementioned data could be processed[1].

Mapreduce is a paradigm that leverages “map” and “reduce” functions to do parallel computations on a large amount of data. The “map” function selects specific data based on criteria determined by the user while the “reduce” function aggregates the results of the map phase. The paradigm uses key/value pairs in both the map and reduce phases – with the map phase producing intermediate key/value pairs based on those from the input data. The reduce phase will then take those intermediate key/value pairs and group or consolidate them into the final output of the processing[2].

A few years after Mapreduce’s development, Yahoo created the Hadoop; which integrated Mapreduce along with HDFS – a distributed file system inspired by Google’s GFS – into a framework for Big Data processing. At its most basic form, HDFS consists of a Namenode, multiple Datanodes, and a client. The Namenode contains a directory listing where different cluster files are kept, while the Datanodes are responsible for storing the data – which can be located on different hardware. The client talks to the appropriate nodes in order to run map and reduce tasks[1]. In HDFS, data is stored in blocks of 128MB. All in all, these innovations allowed for parallel processing of distributed data in an efficient manner.

Object storage, in contrast to HDFS, is not block based and contains no directories. It has a flat organizational structure in which data values are represented as key/value pairs for easy retrieval (querying the key will return the value). Unlike other file storage systems, Object storage uses hashing to identify and so editing or renaming files is not an option – such an operation requires the system to copy the current file and perform any requisite changes[4]. Like HDFS, data stored using object storage is hardware agnostic, with each object containing metadata that allows the server to retrieve it from any physical location[3].

When it comes to running Mapreduce, there are some key differences between HDFS and Object storage:

1. HDFS is tied to Hadoop and requires Hadoop clusters to run while Object Storage is independent
2. In HDFS, compute and storage are tied to together and must be scaled together, while in Object Storage both can be scaled separately.
3. In HDFS the Namenode is a single point of failure, while Object Storage is by nature decentalized
4. Unlike HDFS, Object storage does not chunk data during uploading (it lacks read locality) and cannot rename files (it lacks write locality). This means that HDFS, which can dynamically assign data blocks to different locations, has an easier time spreading out and managing I/O tasks[4].

As can be seen, both HDFS and Object Storage have advantages and disadvantages. However, due to the explosion in public cloud usage, Object Storage is becoming more ubiquitous and it becomes important to further examine how to combine it with Mapreduce.

**Small Files – A Key Problem With HDFS**

As described above, HDFS uses block storage with each block containing 128mb of data. That is, data from larger files is split into chunks for processing. Though this is quite efficient in most cases, a problem arises when trying to perform Mapreduce on smaller files. For starters, when HDFS writes a file it creates at least a block to store it (depending on the file size). If the file is smaller than a block one whole block is still created. Consequently, writing lots of small files to HDFS increases the meta that needs to be stored by the Namenode – potentially overloading it despite storing relatively little information. Additionally, the time it would take for the Namenode to coordinate file information across all Datanodes would increase. Processing small files is also costly, as each file only has 1 map task. Efficiency is lost in the reduce phase as only 1 map output file is processed rather than an aggregation of numerous map output files as per usual[5].

To date, there are quite a few solutions to the small files problem. For starters, there are techniques that revolve around file merging such as Hadoop Archive Files – which were created by Hadoop in order to solve the Namenode memory issue by consolidating metadata from small files into a single archive file. Also along the lines of consolidation – is simply to chunk small files into larger ones without any indexing mechanism if the original files do not need to be preserved or by using some method to preserve the indexes of of the original files if needed. Lastly, one can use sequence files – which are large enough to fill out blocks and contain key/value pairs that hold metadata for lots of small files[6]. These solutions may increase preprocessing time as new files/indexing needs to be created but they improve system throughput[7]. The map process will create output files on large files that can be reduced efficiently.

Another category of solutions revolves around Namenode scalability – reducing the memory burden on on the Namenode by using more than one such node (federated name nodes). This is especially ideal if Hadoop is being used for different applications; thus each application can have a designated Namenode[6]. Though this solves the issue of a the Namenode being a single point of failure and improves cluster scalability, it also creates new problems around load balancing coordination amongst the various Namenodes[7].

The last set of solutions involves data management optimization and seeks to correct the problems caused by holding many under-filled blocks caused by small files. The first of this category, prefetching, involves cacheing either files or metadata that are frequently accessed in order to improve disk I/O (access efficiency) and response time. Another involves altering the Mapreduce process slightly to combine multiple InputSplits (the smallest unit of Mapreduce processing; created for each file) so that there are fewer mapping processes called. Though these solutions improve access efficiency, they can be hard to implement[7].

As can be seen, there are a variety of potential solutions to address the issues caused by small files in HDFS. Much like with HDFS, small files also cause issues when running Mapreduce over Object Storage. Especially in public cloud environments, Writing multiple smaller objects is more time consuming and more costly than doing so for a smaller number of larger objects[8]. In addition, as alluded to previously, Object Storage uses hashing to write objects to a server and thus does not allow renames. Objects are written to the server with their own metadata and without any chunking – unlike how distributed file systems store files in blocks in different locations. Due to a combination of the above, Object Storage systems are more sensitive to I/O issues, but scale more easily and are unaffected by metadata storage issues that HDFS faces[4]. Consequently, solutions to running Mapreduce over object storage center around reducing the burden of I/O. We will examine a couple of them more closely in the next section.

**Our Solution to Mapreduce in Object Storage**

There are two directions for solving the small files issue in Mapreduce over Object Storage: the first option involves creating a layer or layers over object storage that consolidate smaller files into a larger file and avoid renaming to minimize the burdens stemming from I/O. The advantage of this path is that it is relatively easy to implement and can effective solve the issue. This type of solution can also be wrapped around any public cloud environment making it useful in a variety of situations. However, due to the need to avoid file renames and other costly writing operations, many more calculations need to be done in memory during runtime, and these processes could serve as a point of failure in Mapreduce. Additionally, there is a limit to the amount of improvements that can be made with this method[4] – you can optimize the the file merging layer but you still do not control the underlying storage system.

This brings us to the second option; creating a new Object Storage System that optimizes Mapreduce and other analytics functions. Such systems like Swift Analytics add additional location metadata to enable placement control and object renaming and performing chunking in a similar manner to HDFS[4]. However, this sort of a solution takes more effort to implement and it assumes that the user has complete control over their own environment. Companies that are using public cloud storage may not have the ability to alter how their object storage system works. Ultimately, this is the factor that is going to be important from a business perspective. For companies that are too small to invest in their own big data infrastructure or who do not want to, this sort of solution is out of their reach and thus it behooves us to create something that is easy to utilize for all.

Our solution is geared toward the first aforementioned alternative and entails 2 components: a virtual file layer that can mimic some of the functionality of HDFS in Object Storage, and a Small Files Container. The virtual file layer breaks a file into chunks and stores it in partitions in memory (storing location data in the last chunk), so as to avoid I/O while allowing efficient processing of the file. The partition size can be controlled by the user. Each block is meant is meant to be processed by one thread. The Small Files Container will create an instance of the virtual file where multiple small files can be aggregated while preserving their location information and then processed efficiently. By operating on a virtual layer in memory, our minimizes costly writes to the disk while while retaining the ability to operate in different Object Storage Systems.

**Conclusion and Next Steps**

In order to see the efficacy of our solution, we tried running our solution on roughly 100,000 small files (3 row csv files) with and without using our Small Files Container. We found that the Mapreduce process took 1/3 less time to complete (51 seconds vs 81 seconds) – a solid performance. However, there is still additional testing in different environment that can be done. We had access only around 8 threads at maximum to the hardware restrictions, and while this helps demonstrate the usefulness of the solution in a hardware limited environment, it will be important to test it when there is more computing power available. Furthermore, the preprocessing with the Small Files Container takes time to run, and optimizing this process will be important moving forward.

**Bibliography**

1. Chethan, R. Et al. “A Selective Approach for Storing Small Files in Respective Blocks in Hadoop”, *International Journal of Advanced Networking & Applications*
2. Maria Moise, Diana. “Optimizing Data Management for Mapreduce on Large Scale Distributed Infrastructures”, *ENS Cachan*.
3. Sharma, Sunita. “How Object Storage Can Improve Hadoop”, *womeninbigdata.org*
4. Rupprecht, Lukas, Et al. “SwiftAnalytics: Optimizing Object Storage for Big Data Analytics”*, IBM Research and Imperial College London*.
5. Zhou, Fang, Et al. “SFMapReduce: An Optimized MapReduce for Small Files.” *Auburn University*.
6. El-Sayed, Tharwat, Et al. “Impact of Small Files on Hadoop Performance: Literature Survey and Open Points”. *Menoufia Journal of Electronic Engineering Research.*
7. Aggarwal, Ravenna, Et al. “Small Files’ Problem in Hadoop: a Systemic Literature Review.” *Journal of King Saud University – Computer and Information Sciences.*
8. Fan, Bin, Et al. “The Case for Packing and Indexing in Cloud File Systems” *Carnegie Mellon University.*