Big Data Platforms - Small files and MapReduce

Roei Arpaly,
Ron Darmon

12/02/2022

Abstract

This paper presents the small files problem, which can interfere with the MapReduce process. The MapReduce is a model composed of a map, shuffle and a reduce phase to process large datasets. We discuss how to handle large inputs of small files and suggest two possible solutions to deal with it. The first approach recommended using SequenceFiles so the data will be streamed into a container, while the second approach was Merging Small Files. We implemented a prototype code for the second approach, which deals with small files of the same type and structure (dataframes), packing them together to evenly sized blocks by using a bin-packing approximation algorithm. The solution goal was to improve the performance of the MapReduce process, and we solved it by decreasing the number of blocks the mappers receive.

Motivation And Background

MapReduce is the form of the combination of two operations Map and Reduce, and it is considered as a framework of processing large datasets in parallel. MapReduce engine is used to submit the different tasks through different computers (nodes). The collective of all nodes is defined as a cluster, and it has a main node which is responsible on collecting and controlling the other nodes (this relation is often described as master-slaves).

Map function takes the input of (key, value) pairs, processing it and producing a different set of (key, value) pairs as the output (Dean et al., 2008).

The **shuffle** step involves transferring data from the mappers to the reducers. Worker nodes redistribute data based on the output keys (produced by the map function), such that all data belonging to one key is located on the same worker node. This step is actually the most important part of MapReduce. This is the part where all the heavy lifting occurs; decreasing the running time dramatically hence contributes to the performance of the MapReduce (Aggarwal et al., 2021).

The **Reduce** operation takes place once the shuffling and sorting step is done. It combines the obtained results and produces a new set of values that share a key to a smaller set of values (Dean et al., 2008).

MapReduces tasks can take place against datasets which are stored in a database (relational) or in a filesystem which supports unstructured data (non-relational).

HDFS (Hadoop Distributed File System) is a distributed file system built to handle large datasets running on commodity hardware (computer hardware which is relatively inexpensive). HDFS enables the storage of data while MapReduce is used to distribute and process tasks from the HDFS (El-Sayed et al., 2019). HDFS is commonly utilized to scale an Apache Hadoop cluster to multiple nodes (up to thousands of nodes). The HDFS filesystem is considered as immediately consistent, an object in the file system can be immediately visible to the user, or other clients. The architecture is also fault-tolerant and resilient, it allows the overall system to recover from a failure in a specific node by automatically replicating data across different nodes. Therefore, HDFS can recover quickly from hardware failures, access to streaming data, accommodate large datasets and it is highly portable (transferable between multiple hardware platforms). HDFS is also scalable, with a relatively cost-effective solution for businesses that need to handle and preprocess large amounts of data (Zhou et al., 2015).

Object storage is an architecture of storing data as objects, rather than in a block that makes up a file. Each object contains the data itself and a metadata, which includes a unique identifier for the object (Leavitt, 2013). Objects that are stored in an object storage must be manipulated as a whole (unlike the block storage which allows you to incrementally edit one part of a file). Object storage systems are widely used to store unstructured data (such as parquet files) and other data types (images, videos, songs, documents, etc).

HDFS can operate on commodity hardware, which is flexible for scaling horizontally (adding more machines), and it has both compute and storage components. On the other hand, Object storage is pure storage without computational power. It is vertically scalable (adding more storage), but it is not easy to add additional servers (Ghazi et al., 2015).

HDFS is used when the data is entirely structured and is relatively cheap. Both HDFS and Object storage take measures to protect the data, by using data replicas and erasure coding accordingly. Furthermore, object storage systems are less dependent on the master node than HDFS, and their storage service is usually cheaper than the cost of block storage (for HDFS).

HDFS allows faster processing of data compared to object storage, especially large volumes of unstructured data (the benefit of a distributed filesystem). This is utilized by the mapping

system which locates the data within clusters, and by the fact that the tools used to process the data (powered by the engine), are also located on the same servers.

Hence, using MapReduce, HDFS is able to distribute tasks (execution in parallel) on petabytes of data across a large number of nodes. Moreover, HDFS uses a simple programming model under the hood, which allows programmers to develop MapReduce programs easily. Finally, one of the most important advantages of HDFS is the fault tolerance, by using its commodity hardware (that can be crashed and replaced at any moment), the data can be accessed via replications of other nodes in the cluster.

It is possible to utilize object storage for scaling the MapReduce jobs (the servers will write the mapper tasks to the object storage) without the need to run, operate or maintain additional block storage. The serverless (object) storage can take multiple measures to ensure fault tolerance (for example writing temporary files). Using object storage as part of the MapReduce procedure can be difficult, as all of the data is constantly recycled, which makes it sometimes not possible to read.

Small Files Problem

To understand the small files problem, we can imagine a large chuck of images where each image is taken as a unique file, and there is no simple method to combine the images into one large file.

In the HDFS Ecosystem, small files are not aligned with the HDFS block size (standard of 64MB). Usually, when you want to use HDFS to handle small files is when you have a large amount of them (otherwise, why use a distributed framework?). HDFS struggles to handle large numbers of files since each block uses around 150 bytes (El-Sayed et al., 2019). If we use (for example) 30 million small files where each one uses 1 block of the HDFS, we would end up using around 10 gigabytes worth of memory, which ends up being unproportionate to the scale of the task.

On top of struggling with handling the small files, HDFS is not built well enough to access small files as well. It was mainly created to stream files of large scale rather than small ones. If every file is very small, each map task will process very small input, and the HDFS will have to

distribute a lot of those small tasks to different working nodes, which can result in a lot of overhead. Hence, reading small size files will cause an inefficient operation such as jumping from node to node searching to read all the files. This causes jobs of lower input size of many files to run much slower than a single input job of very large size.

In order to solve the problem, some sort of a container is needed to merge the files together. The most common solution is to use a SequenceFile. By using the filename as a key and the file content as data, forming a new larger file consisting of a bunch of small files. SequenceFiles can be splitted, hence MapReduce can break them into blocks and then operate on each block as in a normal process of MapReduce. It is also possible to compress the SequenceFiles, reducing its total size.

One major drawback is the intermediate step of storing existing data inside the SequenceFiles. Yet, it is possible to run it in parallel (by packing small SequenceFiles together to create a large SequenceFile). Furthermore, an additional solution for saving the conversion time is to write the data directly to a SequenceFile, by modifying the data pipeline, thus avoiding the intermediate step.

Our Approach

There are several approaches to deal with the small files problem when running MapReduce (El-Sayed et al., 2019).

The first approach is using SequenceFiles. As mentioned in the previous section, SequenceFiles work well in practice to deal with the small files problem. We can define a program which inserts the files into a single SequenceFile. This way we will be able to process and operate on the Sequence file in a streaming fashion. We already know that MapReduce is capable of breaking the SequenceFile into several blocks (because SequenceFile is splittable) and operate on each block separately. Finally, we should define an interval of time / space which serves as the criteria of deploying the MapReduce operation on the SequenceFile. E.g. when the size of the SequenceFile reaches to 1GB (or after a certain period of time) we will run MapReduce (splitting to approximately 64MB blocks, as we don't want to split a small file).

The advantages of using SequenceFiles are the computational power saving on merging the small files by utilizing the data pipeline, as the files will be streamed directly into a SequenceFile. The approach is simple and maximizes the MapReduce ability to split data into

blocks. Although SequenceFiles has some advantages, MapReduce can still struggle with splitting the SequenceFile into evenly sized blocks because small file sizes can vary. One more potential issue is that the streaming may never reach its space criteria, and in that case the MapReduce phase won't commence.

A second approach is using a merge method of small files.

If we assume for example that we want to process 32,000 small sized files (of around 2MB), then if we just transfer these files to the mappers we will have 32,000 mappers running simultaneously. As discussed, it is not an efficient approach. Hence, if we merge these files (each file is around 2MB so 32 files can form a block of 64MB) we will end up having only 1,000 mappers to run the same operation in a more efficient manner. It would be even dramatically better when the file's size is much smaller.

The algorithm is taking n amount of files into a bucket, merging them into a single file so the size is closer to 64MB, then run the mapper. This approach is similar to the known bin-packing problem, and we can use a similar approach to deal with allocating the small files into evenly sized blocks. Once the block size reaches the size of approximated 64MB (user defined) by packing a batch of small files together, it will send it as a batch.

The advantages of using a merge method of small files are the ability to deal with large quantities of files in a simple approach. The merge method of small files also makes sure that small files won't be sent to nodes if the block size is less than 32MB (or user defined). Additional advantage of this method is the fact that a large block size can't be formed (because of the 64MB user defined limit).

However, this approach presents some challenges. The algorithm of combining small files is not implemented with threads, which means in a scenario of many small files the algorithm does not utilize multiple nodes as it can't run in parallel. Also, it will be hard to pack tremendous amounts of small files together by using an approximated bin-packing algorithm as the running time is similar to sorting $\Theta(n \log n)$.

Additional challenge is to deal with the reserved small files that can't be packed into a reasonably sized block (defined by the user). They will stay at the cache and won't proceed to the mappers. We approached this issue by creating a time criteria which users can define, a time interval that will send low size blocks to the mappers after a certain threshold has been met.

In our solution we implemented the merging algorithm to deal only with data types of dataframes (which have the same properties).

Next steps

Although Hadoop supports working with large amounts of data by parallelizing the processes so it runs efficiently, dealing with small files has been a known problem and in practice the users' have a large variety of demands and requirements. Our prototype can't handle a wide variety of data types and can't run the merging algorithm in parallel.

The next steps would be to run the merge of the small files that was recommended in the section above in parallel, using the available nodes of the MapReduce engine. Supporting large amounts of data types, and bin-pack the same data types together if necessary. Although processing and transforming audio, video and image files via MapReduce is considerably a more complex problem, A more robust algorithm will be able to handle those types of files while increasing the flexibility and maintaining the efficiency.

It is possible to include data validation steps as well, such as dealing with large files (that have been inserted with the small files), and decide how to deal with it. E.g. process one large file as a block or split the large file if possible. Furthermore, if the files have approximately similar memory size, using the bin-packing algorithm can be redundant.

Conclusion

Hadoop framework (HDFS and MapReduce) was developed for storing and processing big data, and it is commonly used due to its simplicity and scalability which is boosted by the availability and the surge in development of cloud computing. Although it is widely known in handling large scales of data, its ability to handle large amounts of small files is a known issue. Dealing with large inputs of small files impacts the Hadoop performance, as storing and processing small files result in a lot of "bookkeeping".

In order to solve this problem, we proposed two approaches, SequenceFiles and Merge Small Files. In our paper we showed that by monitoring files streamed to the MapReduce, it is possible to detect and apply algorithms which are "aware" of the small files problem. By using the suggested approaches, the MapReduce will receive blocks less prone to efficiency issues. Finally, we discussed several open points, advantages and challenges which should be considered for the next steps and the improvement of the suggested algorithm.

Bibliography

1. Fang Zhou, Hai Pham, Jianhui Yue, Hao Zou and Weikuan Yu, "SFMapReduce: An optimized MapReduce framework for Small Files. (2015). *IEEE International Conference on Networking, Architecture and Storage (NAS)*, 23-32.

2. El-Sayed, Tharwat & Badawy, Mohammed & El-Sayed, Ayman. (2019). Impact of Small Files on Hadoop Performance: Literature Survey and Open Points. *Menoufia Journal of Electronic Engineering Research*. 28. 109-120.

3. Raveena Aggarwal, Jyoti Verma, Manvi Siwach, Small files' problem in Hadoop: A systematic literature review. (2021). *Journal of King Saud University - Computer and Information Sciences.*

4. Leavitt, N. (2013). Storage Challenge: Where Will All That Big Data Go? *Computer*, 46, 22-25.

5. Jeffrey Dean and Sanjay Ghemawat. (2008). MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51, 107-113.

6. Ghazi, Mohd & Gangodkar, Durgaprasad. (2015). Hadoop, MapReduce and HDFS: a developers perspective. *Procedia Computer Science*. 48. 45-50.

Prototype code:
https://github.com/RoeiArpaly/BigData/blob/main/merge_small_files_mapreduce.py