Big Data Platforms

Small files and MapReduce

Eyal Michaeli - 207380528

Tzach Labroni - 302673355

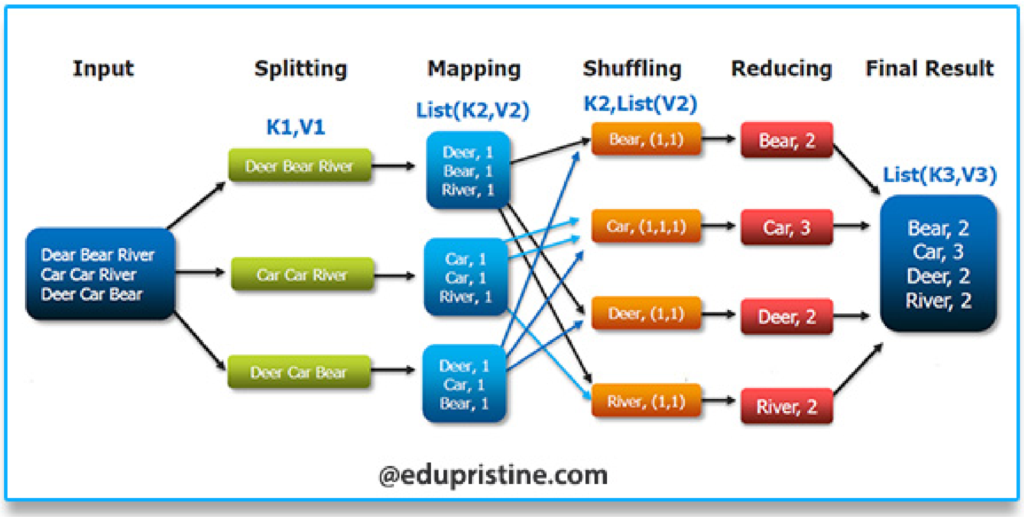
01.03.2022

Abstract

MapReduce is a powerful and widely used framework to process data. MapReduce offers a way to parallelize data processing in a scalable cloud environment and is optimized for working with large files. The demands of functionalities vary in the real world, and one of the most significant demands is to efficiently process a large amount of small files as well, which causes the process to be less efficient. Initially, MapReduce was collocated with HDFS - Hadoop's file system, but since then MapReduce was adjusted to work over files persisted in other file systems, namely, Object Storage. In the following paper we will explain the small files problem and how it affects MapReduce both for files persisted in HDFS and for files persisted in Object Storage. We will review several approaches to resolve the issue and map out their advantages and disadvantages. Finally, we will implement one of the solutions in Python, show how it effectively resolves the small files problem and suggest next steps to the solution.

Motivation and background

MapReduce is a form of computation structuring so the computation can be done on multiple machines. Under the MapReduce framework each process is constructed by three phases: map phase, shuffle-sort phase and the reduce phase. During the map phase a single job is broken down to multiple smaller jobs, so that jobs could run in parallel. Then, each of these broken-down jobs gets assigned a mapper and a simple calculation is performed; the outputs of this processing are then being forwarded to the shuffle-sort phase. The shuffle-sort phase is where the data is being grouped by the key, aggregated accordingly and then sorted. The aggregated and sorted results of the shuffle-sort phase are then moved to the reduce phase, in which the input is being combined together [1]. The result of the reduce functions is the output of MapReduce as a whole. The following diagram shows a simplified flow diagram for the MapReduce program [2]:



Object storage is an architecture that saves data as distinct units, each unit being referred to as an object. An object is constructed by the ID of the object, the data itself, the data's metadata and attributes that describe the object itself (permissions etc.). Each object is being stored in a bucket and is being replicated several times for data integrity and fault tolerance. In Object Storage, the hierarchical relationship between two objects is flat, despite the fact they might be related [3]. The most recognizable Object Storage platforms are Amazon's S3, Google cloud and IBM cloud.

HDFS (Hadoop Distributed File System) is a file system framework that manages big data storage in a distributed fashion. HDFS clusters follow a master-slave architecture, in which in every cluster there is one NameNode and several DataNodes. The NameNode is not interchangeable and the DataNodes are. While the DataNodes hold the actual data, their "mastering" NameNode holds their DataNode's metadata and manages the DataNodes. Each DataNode holds a block of the full file that is being stored, which is typically a 64MB block. Each of the NameNodes is being replicated, so in case one crashes, the DataNode can replace the node with a replica, thus ensuring fault-tolerance. Moreover, the DataNodes periodically send signals to the NameNode called HeartBeats, letting the NameNode know they haven't crashed. If one DataNode fails to send a HeatBeat to the NameNode, it will be replaced by a replica and another replica will be generated. In that sense too, HDFS is fault-tolerant [11, 12].

The difference between running MapReduce over data persisted in HDFS versus data persisted in Object Storage is rooted in the fact that HDFS, by default, uses a fixed size of memory to store each DataNode. This means that running MapReduce on HDFS will also result in processing and mapping a maximal fixed-sized chunk of the data. On the contrary, in Object Storage the object's size is not fixed, hence running MapReduce on it, specifically the Mapping phase, may not run on equally-sized chunks of data. In this sense, MapReduce on HDFS has an advantage - it is more likely to split its computational resources in an efficient manner, as the chunk sizes are usually equivalent to HDFS' block's size. In Object Storage there is a larger variance between chunk sizes, which can result in inefficient allocation of resources. However, some problems may arise in HDFS since blocks have a maximal fixed-size, E.g. if we wanted to count the number of words in a file and splitted the file by size, some words could get splitted into two different DataNodes. In turn, this can result in inaccurate results, whereas in Object Storage it would not happen. Lastly, in HDFS, compute resources and storage capacity work together on the same nodes. If we need more storage, we must add comparable compute power, and vice versa - we cannot add one without the other. While in object storage, the flat organization of the address namespace, together with extensible metadata functionality, allows us to scale storage capacity and compute power independently. As our needs change, we can add object storage nodes without having to build up compute power that we will not use. So, in that sense, object storage is much more modular and scalable [drew inspiration from 5].

Small files problem

When discussing the small files problem in MapReduce, we will first explain why the problem is important and how it is getting amplified by the rise of Big Data. Next, we will elaborate on the problem while running MapReduce on files persisted in Object Storage. Later, we will present the same issue and how it manifests when running MapReduce on HDFS, specifically the problems small files cause given the architecture of HDFS. Lastly, we will categorize the approaches of dealing with the small files problem to three, and exemplify each of these approaches with several existing solution proposals.

The two components at hand of Hadoop, HDFS and MapReduce, were both developed to tackle large files and allow the files' distributed storage and processing, respectively. However, with the rise of Big Data, alongside large files that need storage and processing, a lot of small files require the same as well. Social media and other prominent data manufacturers generate a lot of small files that are crucial for data-driven decision making and therefore cannot be ignored and need to be stored and processed in large scales. However, as mentioned, HDFS and MapReduce were not designed with these files in mind [4].

The MapReduce problem of small files stems from the abundance in mappers and reducers assigned when dealing with small files. When running a lot of mappers and reducers on small files, each mapper and reducer gets very little amount of data. Creating and closing these processes creates overhead, especially since the processed data of each mapper/reducer is very small. To conclude, the effects are: (a) Massive number of tasks created, leading to task scheduling overhead; (b) Time consumption in start and shutdown of these tasks; (c) Wastage of cluster computing resources [6].

The small files problem in HDFS is divided into two problems - (a) small files storage under HDFS' architecture and (b) small files problem in MapReduce. The first sub-problem of the HDFS side is the inefficiencies caused by having many DataNodes. Under HDFS' framework, large files and small files are treated the same. While large files will be splitted into 64MB chunks and if the last chunk is under 64MB it will still occupy a full block, each small file will be treated just like the last chunk; Meaning, each small file will also occupy a full block - a whole DataNode. Having many tiny DataNodes creates inefficiencies with how the data itself is stored. The second HDFS sub-problem is the pressure being applied on the NameNode by having so many DataNodes. As mentioned before, each DataNode sends a HeartBeat to the NameNode to ensure it still holds the data. However, when there is an abundance of DataNodes, there are a lot of HeartBeats being sent back to the NameNode, and orchestrating over so many HeartBeats wears the NameNode down and increases latencies. In addition, as mentioned before, the NameNodes keep the metadata of the DataNodes data as well. Having a lot of DataNodes means that the NameNode has to store a lot more data in it, which can cause faults, crashes and latencies. The second sub-problem is the MapReduce part of the problems and is similar to the general MapReduce problem with small files, as mentioned earlier - having a lot of map and reduce tasks on small files creates overhead and leads to waste of the cluster's resources [7].

There are several approaches to deal with the problems mentioned. The first one suggests to completely avoid the problem by converting the small files into larger files. Under this approach, falls the existing HAR solution (Hadoop Archive approach), in which an archive file is being created that contains multiple small files. Each HAR file is stored in a different NameNode, which resolves the HDFS issues that were mentioned earlier. However, these packed HAR files are being unpacked for the MapReduce processing phase, and therefore do not resolve the MapReduce problem that was mentioned, and the processing efficiency still lacks. This approach includes the SequenceFiles solution as well. According to this solution, different files are saved under the same as key:value pairs, in which the key contains the file's name and the value is the data itself. The storing and mapping are all done on the SequenceFile level, and thus the small files are enlarged to a SequenceFile. Other prominent solutions that fall under this approach are the Batch File Consolidation solution and HBase. The first aggregates multiple small files into one larger file which is held on a DataNode, but does not allow access to the original small file, and the latter stores the data in Map Files, which are indexed SequenceFiles which allow quick fetching and processing of the stored data. We will elaborate further and implement the Batch Files Consolidation later on in this paper [8, 9].

The second approach deals with the NameNodes part of the problem. The main solution that exists under this approach is splitting the load that is currently being put on the single NameNode to several NameNodes. This solution allows for multiple NameNodes to exist in a single HDFS cluster, splitting the responsibility of holding the metadata for all DataNodes [7, 8].

The third and last approach deals with the abundance in mappers. As noted, since several mappers are assigned per block and the block does not hold a large amount of data, there isn't enough data to be processed by each mapper. MapReduce had suggested using CombineFileInputFormat, which packs several blocks together and allocates mappers to the aggregated blocks. This will ensure that each mapper has a sufficient amount of data to handle, and increases the efficiency of the MapReduce process [4].

Our approach

The two approaches we chose to dig deeper in are adaptations of the Batch Files Consolidation and the SequenceFiles solutions that were presented earlier. We will adapt these solutions, which are frequently brought up with regards to implementation of MapReduce over files persisted in HDFS, to be compatible with files that persist in Object Storage.

Under Batch Files Consolidation, all small files are being sent to a temporary folder and are being consolidated periodically by a dedicated job. The consolidated files are then moved to a DataNode, which is the input for MapReduce. We would like to suggest a version of the Batch Files Consolidation solution, that will be applicable to Object Storage as well as to HDFS, as opposed to the current HDFS-based version. Under our solution, given a queue of objects that should be processed and persist in the Object Storage, a new step would be added to MapReduce. In this additional step, we will queue the small files objects that need to be processed by MapReduce, and then batch them together to larger files. Afterwards, MapReduce will operate on these larger files and effectively avoids the small files problem [7].

The biggest strong point of the Batch Files Consolidation is that it effectively resolves the small files problem as presented earlier, from both the HDFS angle (when applied to HDFS) and from the MapReduce angle. Basically, we avoid altogether having small files and only deal with large files that aggregate the small files. The system's fault-tolerance can be maintained at the same level, given the temporary small files are replicated as well, until they are consolidated to a larger file, which has the same fault-tolerance as HDFS.

However, there are three major disadvantages to this approach. The first is that it adds another step to the MapReduce process - lining up a queue, scanning it and batching together small files. Only after this step can the MapReduce process start. This addition increases both the complexity of the MapReduce process and its latency. Given a high amount of files, this process can be long and costly. As we argued before, when explaining the significance of the problem, a large amount of files is what we should typically expect and encounter when dealing with small files, making this point a major disadvantage.

The second weak point of this approach is that it is not traceable - when we are grouping the files together, we do not keep track of each file's origin. Therefore, if we had wanted to debug our MapReduce output and understand which origin file caused a certain issue in the data, we would not be able to trace the process back and find it. Furthermore, the lack of traceability makes MapReduce not applicable for certain use-cases. For example, reduce functions that rely on the input files such as Inverted Index, can produce false results when working on consolidated files. If we are trying to count the number of files that a string appears in, for example, and we aggregate 100 files to 5 and only then run the MapReduce, the results will refer to the 5 aggregated files and not to the original 100 files [10].

Lastly, the third weak point of the Consolidated Batch Files solution is the decrease in availability. With adding the consolidating phase, the small files at hand cannot be processed until the consolidating job finishes its work, which hinders the availability of the file system. Compared to HDFS or Object Storage, in which the small files are available relatively quickly, under this solution the additional phase decreases availability, which is a disadvantage of the solution.

The second solution we would like to elaborate on is the SequenceFiles solution. SequenceFiles is a solution that is currently being deployed in HDFS to deal with the small files problem [9]. SequenceFiles are flat files that contain multiple small files in them in a key:value structure, where the key is the original file's name and the value is the actual data. The files themselves can be either compressed or not, whereas the uncompressed version of it makes this solution different from the HAR solution for example. That way, HDFS can group several small files together and avoid the small files problem altogether [4].

Again, we would like to suggest an adjusted version of the solution, so it would be applicable to both HDFS and Object Storage. In our suggestion, we would add a queue of objects that should be processed, whether they are being stored in HDFS or in Object Storage, iterate over them and sequence them to larger files. In a similar fashion to the Batch Files Consolidation, MapReduce will then run on these sequenced files and we could avoid the small files problem altogether.

Since both the SequenceFiles and Consolidate Batch Files solutions are of the same approach, their advantages and disadvantages are related. SequenceFiles, as opposed to Consolidated Batch Files, is traceable, which is one of its strongest advantages. As we have mentioned when discussing the previous solution, the lack of traceability of the Consolidated Batch Files solution makes MapReduce inapplicable in some cases - an issue that can be avoided with the implementation of SequenceFiles. However, we would like to mention that to capitalize on this added traceability, some adjustments need to be made to the map function. If we return to the Inverted Index example, to be able to know from which small file did the data arrive, we will need to look at its matching key and take it into account, which is somewhat different than looking at the file name [7].

The major weak points of SequenceFiles closely resemble the ones that were brought up against the Consolidated Batch Files solution. The major drawback of this approach, and especially the implementation that supports both files that persist in HDFS and Object Storage, is that it adds complexity and latency to the MapReduce process by adding an additional step to the process. In addition, this additional step decreases the availability of the system, as the data is not available until it is completed the aggregation to the SequenceFiles.

Between the two solutions, it seems to us that SequenceFiles is the better option. SequenceFiles seem to be more complicated to create and to process, as they hold both the file name and the data itself and are probably larger compared to the Consolidated Batch Files, which makes a system that is built on SequenceFiles to have lower availability. In addition, it seems SequenceFile that holds the same amount of data as a Consolidated Batch File, will probably be larger in size, since the SequenceFile holds the files names as well. Therefore, it seems that under a SequenceFiles framework we will need more MapReduce jobs to process the same amount of data than a framework built on Consolidated Batch Files.

However, the additional traceability offered by SequenceFiles leads to a larger range of analytical operations that can be performed by MapReduce. For example, Inverted Index is not possible under the Consolidated Files Batch but is possible under SequenceFiles. We tend to think that the additional analytical flexibility offered by SequenceFiles, despite its shortcomings, makes it a preferable option compared to Consolidated Files Batch.

Prototype

Please see the prototype on our GitHub repository:

https://github.com/EyalMichaeli/BDP-FinalProject

In our implementation of the Consolidated Batch Files solution, we have implemented a modified version of the word-counter application, in which we count the number of appearances of each file in the original files. As explained in the notebook, we ran MapReduce twice: once on a 1000 small files we have generated locally and the second time on a consolidated version of the same files. We have compared running times of each process and measured the running time of the consolidation process as well. We have shown that the proposed solution is more efficient in total and that the additional step we have discussed in the weak points segment before is indeed significant and adds a lot of latency to the process.

Next steps

As the two proposed solutions are closely related, we chose to write the next steps for both together. Whenever there is a next step that is applicable just for one of the solutions, we will point it out specifically.

Increase availability: The first action item is to increase the availability of the system. As we have mentioned, in both solutions there is a set “downtime” in which the small files get aggregated to larger files, whether Consolidated Batch Files or SequenceFiles. As we have shown in our prototype, this downtime can be significant - it took 6-10 seconds to generate the aggregated files, whereas it took 1-2 seconds to process them in MapReduce. An immediate next step is to make the aggregation process more efficient and thus increase the availability of the system as a whole. Also, we will need to decide when is the optimal time to do the consolidation/sequencing - whether it is optimal to do it when the small files arrive or to schedule an aggregation job off-hours.

Ensure fault-tolerance: Currently, it is unclear whether the small files are being replicated while they “wait” to be aggregated. If they are not replicated, during the waiting time the system does not guarantee fault-tolerance. Meaning, if the intermediate small file holder crashes, we cannot recover the file.

Allow traceability to Consolidated Batch Files: As mentioned, the current implementation of Consolidated Batch File does not have full traceability and therefore does not allow for certain MapReduce use-cases. However, we can consider a version of Consolidated Batch Files that allows traceability - for example, if each of the CSVs will hold an additional column with the origin file name.

Conclusion

To sum up, in the paper above we have covered MapReduce's small files problems. We started with a high-level explanation of the MapReduce process, its relevant components and its two related file systems - HDFS and Object Storage. We have covered the architecture of the two file systems and explained the differences between running MapReduce over files that were persisted in HDFS and files that were persisted in Object Storage. We have laid out the components of the small files problems with regards to the MapReduce process itself and with regards to running it over files that persisted in HDFS. We reviewed the most updated literature and suggested three approaches to resolving the small files problems: (a) running MapReduce over an aggregation of small files, (b) adapting the HDFS NameNode-DataNode architecture to having a lot of small files and (c) to tackle the abundance in mappers and reducers. We chose the first category and elaborated on two solutions that fall into that category - Consolidated Batch Files and SequenceFiles. We laid out the two solutions, adapted them to fit runs of MapReduce over files that persist in Object Storage, and dug deep into the solutions advantages and disadvantages. We paid close attention to fault tolerance, availability and traceability in reviewing the solutions. Next, we implemented the Batch Files Consolidation solution on Python, running MapReduce on many small files and over several consolidated files. We showed that the described solution leads to a significant decrease in running time. Specifically, when we ran MapReduce multiple small files it took approximately 10X the time of running MapReduce on a consolidated version of the same files. Namely, it took approximately 15-20 seconds to process the small files and 1-2 seconds to process the consolidated files. We would like to add that the consolidation of the data took 6-10 seconds approximately. Lastly, we have laid out the next steps to the Consolidated Batch Files and SequenceFiles solutions.

Bibliography

1. MapReduce: simplified data processing on large clusters
2. Big Data & Hadoop: MapReduce Framework | EduPristine
3. Object storage: the future building block for storage systems
4. Small files' problem in Hadoop: A systematic literature review
5. https://blog.westerndigital.com/apache-hadoop-object-storage-data-lake/
6. SFMapReduce: An Optimized MapReduce Framework for Small Files
7. Impact of Small Files on Hadoop Performance: Literature Survey and Open Points
8. Improving Hadoop Performance in Handling Small Files
9. SequenceFile Formats
10. Small Files Consolidation Technique in Hadoop Cluster
11. What is HDFS? Hadoop Distributed File System overview
12. A Review of Various Optimization Schemes of Small Files Storage on Hadoop