CS 3605 Final Project

Small Files in MapReduce Jobs

Agathe Benichou and Gal Almog
agathe.benichou@post.idc.ac.il, gal.almog@post.idc.ac.il

Reichman University
Department of Computer Science

1 Abstract

Apache Hadoop is an open-source software framework designed for the storage and processing of large scale data on clusters of commodity hardware. The Hadoop ecosystem consists of Hadoop common (libraries and other modules), HDFS (the Hadoop Distributed File System), YARN (the resource negotiator), and MapReduce (the programming paradigm behind Hadoop). Hadoop is highly effiicent for processing big data, and splits files into blocks that are assigned to a datanode (which is the storage unit). This configuration is very effective for applications that use big data (large files), but runs into problems when users need to process large amounts of small files. For Hadoop to be effective, users must make sure that files are in the range of the DataNode block size, otherwise performance will be impacted. This paper examines the reasons behind this and offers two solutions to this small files problem. We provide an implementation of one approach (merging the small files into larger files), as well a prototype available on GitHub.

2 Motivation and Background

Apache Hadoop is an open-source software framework that functions to store and process large amounts of data on clusters. Hadoop consists of a great number of utilities, including Hadoop Common, HDFS (the Hadoop Distributed File System), YARN (Hadoop resource manager), and MapReduce (the Hadoop programming paradigm). The foundation of Hadoop lies in HDFS, which is made for handling large files that can be distributed across a cluster. This is done by breaking up the data files into blocks (typically of 128mb) – this way, the user is not limited by the storage limitations of a single hard drive [1]. This also enables the user to execute the processing of large files across multiple machines, and in parallel. This distribution allows HDFS to be very tolerant to failures and makes it very attractive for applications making use of extremely large data sets. However, the performance of HDFS is significantly decreased with small file sizes.

MapReduce is the programming model used by Hadoop to process and execute computations on the HDFS by distributing the processing across the cluster [2]. The term 'MapReduce' originates from the two main functions that are carried out: first Hadoop 'maps' the data (applies some function to transform it), and second it 'reduces' the data

(aggregates it by its keys) [3]. The mapper function converts each record in the data into (key, value) pairs. The key is what the reducer will later aggregate on, and the value is the data itself. This step is important because it allows significant storage optimization, as any data that is not needed can be dropped at this stage. Next, MapReduce performs a "shuffle and sort", which is essentially a merge sort that groups the data outputted from the mapper by their keys, and sorts the keys. The reducer function takes this output and applies a second function/transformation to produce the final output. This MapReduce processing is vital to Hadoop's distributed processing. Before the mapper function is executed, the data is divided into partitions – this allows each mapper function to be executed in parallel on separate commodity computers [2]. As well, reducers can also be distributed, with each reducer responsible for reducing a certain set of unique keys. In addition to significantly improving performance for big data, the combination of MapReduce and the HDFS allow for great handling of failures. Hadoop includes many built-in options for ensuring clusters are safe and data is backed-up in case of the failure a commodity machine(s).

HDFS is highly scalable when compared to a regular file system. Scalability refers to the ability of the system to adapt to increasing amounts of data and demands in processing. The two broad methods of scaling include horizontal scaling, which requires distributing data and data processing over multiple machines, and vertical scaling, which requires improving the memory capacity and processing hardware in a single machine. While a RDBMS is restricted to vertical scaling, Hadoop, through the use of the MapReduce computational model, implements horizontal scaling which allows it to store and distribute much larger data sets [4]. Additional nodes and servers can be easily added to the Hadoop cluster, allowing businesses that scale up to easily increase their processing potential. In addition to scalabiltiy, the distributed file system deployed by Hadoop also allows for significant improvements in performance and speed. The MapReduce model partitions data and each mapper function is applied to each partition separately. This allows each mapper function to be executed in parallel on separate machines [2], which can greatly improve speed. As well, this means that the data being processed and the mapper function/processing tools being used in that partition are stored on the same machine, so data processing is very fast and efficient [4]. [5]. This is an important benefit because many companies that use MapReduce have data coming in from multiple sources, some of which may be structured (such as website transaction data) and some of which may be unstructured (such as social media data). The HDFS avoids the need to convert all data into a standard format, which would be very difficult and time-consuming, and allows the direct processing of different types of data in the same place [5].

The scalability offered by MapReduce is important for more than just convenience; this also offers a very cost-effective solution for companies that need to scale up their data. Companies that use a RDBMS must invest in machines with increasingly large storage and processing capacities. The cost of this can significantly restrict how much a company can expand, with many companies in the past resulting to down-sampling data before processing to reduce processing needs [6]. As storage is also very expensive, much of the data would simply be discarded, severely restricting the potential of future analyses on the data that may have changed over time. MapReduce is much more

cost-effective as all the raw data generated by a company can be stored on commodity servers with attached storage [7]. This is generally a much less expensive architecture than a dedicated storage area network (SAN) as would be the case for a RDBMS. Finally, MapReduce is very resistant to failures. All data stored in the HDFS is automatically duplicated to ensure multiple copies are available in the case of failure [4]. This happens when data is sent to a node – the data is also replicated to additional nodes in the cluster (on a different computer), so there is guaranteed to be another copy available, even if an entire machine goes down. When dealing with large data sets, failures can often occur during computations. In a RDBMS, computation would often need to restart each time there is a failure somewhere. Since Hadoop uses replicas, they each reside in a different computer, allowing for efficient recovery from failures. There are various design considerations regarding how many replicas to create and where to store them; for example, one may want to place 2 replicas in the same rack and a third replica in a different rack. This decision will depend on the data and the expected failures that may occur.

Hadoop splits each file into blocks (shards). The default block size is 64MB, up to a maximum of 128MB. Blocks are split across many machines at load time, and different blocks from the same file will be stored on different machines [1]. Blocks are also replicated across multiple machines - so Hadoop not only splits the files into blocks, but also replicates the blocks and spreads them across different machines. This organization allows single files to be larger than any single disk in the network - because Hadoop cuts them up. Each block is assigned to a datanode, which is the storage unit of the block. The HDFS cluster also consists of a single NameNode, which is a master server that manages the file system namespace and regulates access to files by clients. This organization, when used with large files, allows for very efficient processing using Hadoop MapReduce, as well as very efficient recovery from NameNode failure / DataNode failure / network failure. Clearly this organization is very robust and has many advantages, but these fall through when Hadoop is used with many small files rather than large files.

3 Small Files Problem

While there are clearly many advantages of running Hadoop on a cluster, there are also scenarios where it is more optimal to use a simple RDBMS. The most striking one is the problem of small files with MapReduce. While some use cases require the processing of very large files, many other require large numbers of small files. For example, when dealing with biological data for research, one might need to store and process many small (200 Kb) protein sequence files, rather than fewer larger files. The HDFS is built of many DataNodes that actually store the data, and a single Namenode that is used to keep track of where each chunk is assigned - it knows where each shard is and where each of its replicas are. This is crucial for Hadoop's failure handling. Ideally, the large files handled by Hadoop should be in the range of the DataNode blocks size (128 Mb), and as we will see, small files that are less than this will greatly reduce Hadoop's performance [8]. With small files, the NameNode may not have enough memory to store all of the metadata, resulting in very high memory consumption [9]. The NameNode is also responsible for maintaining a block report, which holds the status of each block in the DataNodes. Heartbeats are used to update

this block report, as well as detect DataNode failures. A DataNode sends a heartbeat message to the NameNode every set number of seconds, and for each DataNode, the Namenode maintains information about the last time it received a hearbeat. If the NameNode does not receive a heartbeat for a specified amount of time since the last one, it can determine that the DataNode has failed. The block reports can only be updated every 21600 seconds [10], and while the block report is updating, no other operations can be performed on the DataNode. Therefore, another problem with small files in HDFS is that they result in a large number of blocks, so updating the block report can take a significant amount of time and reduce overall performance [8]. Overall, the NameNode's memory utilization is a bottleneck for the performance of Hadoop that is severely jeopardized when using many small files [11].

It is worth noting that this small files problem differs from the broader problem of using Hadoop when it is simply not required. While many companies require the processing of very large (multiple terabytes, petabytes, or exabytes) data, most companies simply do not have such large data demands. If there is not enough data to justify its use, a Hadoop cluster may end up slowing down processing and being less cost-effective than an RDBMS. It can be argued that dynamic random access memory (DRAM) is getting significantly cheaper over time, and increasing the server memory means fewer servers will be required [12]. As well, Moore's law states that the number of transistors in a dense integrated circuit (IC) doubles roughly every 2 years. In simple terms, this means that the speed and capability of modern computers increases every two years, and that they become cheaper. Therefore, every year increasing memory on a single machine becomes cheaper – and most companies may be able to achieve enough storage by hosting a RDBMS without Hadoop.

4 Our Approach

4.1 Approach 1: Merge small files into big files

This approach takes all of the small files (less than the NameNode block size) and merges them into a single large file on the client side before submitting them to MapReduce. This allows the files to be sent as a single mapper. The strong points of this approach are that it allows for configuring a prefixed threshold size and allows for choosing the output format. This approach also offers great flexibility, as it allows for independent and secure implementation, and can be made to be fault tolerant by keeping the files stored in a cloud storage solution. The weak points of this approach are that it assumes that each small file has the same data type, and it does not maintain the order of the files being processed.

4.2 Approach 2: Using Sequence Files

A second approach to the small files problem is to use sequence files. Here, the file name is used as the key and the file contents are used as the values in order to process them in a streaming fashion. A sequence file keeps the

storage and classification in contiguous blocks, thus maintaining the order of the records in which they were entered. A strong point of this approach is that sequence Files can be split, so MapReduce can break them into chunks. This allows MapReduce to operate on each chunk separately in order to maintain the logical divide. Sequence files also support compression for blocks of several records at once to enhance the workload. Weak points of this approach include that sequence files can be heavy to process data, but this can be worked around by creating a collection of sequence files in parallel. The biggest drawback of using sequence files is that they are slow, and work mostly with cheaper storage mechanisms such as magnetic tapes.

4.3 Comparing Approaches

Approach 1, merging small files, has the possibility to be very adaptable to a wide range of use cases. For example, it can be configured to merge files that are below a certain threshold and stop the merge if the large file reaches a certain size. Approach 2, using sequence files, is effective, with the ability to reduce configuration time due to the little software support required. However, sequence files are difficult to maintain and modify, and they mostly behave as append-only. Additionally, any adjustment to the file will require the whole process to be re-run and the sequential nature could cause throttling issues. We believe that the versatility of merging small files makes it the better option. It can be used locally or on the cloud, can be configured to specific use cases or made vague enough to work for many, and it can handle parallel, streaming and batch workloads.

5 Prototype

We chose to create a prototype of the first proposed solution: merging small files into large files. Our prototype code can be seen in a Jupyter notebook, which we have included in our submission along with this report. The notebook is also published on GitHub here Note that the AWS access key and secret access key have been removed before publishing to GitHub. They are available via the Moodle submission.
The notebook has the following sections:

- Lithops Setup: Complete the installations and define the configuration blocks necessary for Lithops usage.
- AWS Cloud Setup: Declare the configuration block necessary for access to an AWS account and an image of the S3 bucket we will be accessing.
- Creation and Upload of files: Before jumping into the prototype, we wanted to run an experiment where the small files issue could be seen. We generated 100MB of data that is split differently, such that: 1000 small files of size 100KB were created, then stored to a folder in the S3 bucket and 1 large file of size 100MB was created, then stored to a different folder in the S3 bucket. The creation and upload process itself was done on a local Python script (also included in submission) but this section will give a general description of the S3 bucket to display the data.

- Display MapReduce on both sets: Modify the Lithops usage of MapReduce class and functions from homework assignment 3, such that it does not rely on an SQL database. Execute MapReduce on the small files directory, then execute MapReduce on the large file directory.
- Compare execution on both sets: Review the time it took to execute MapReduce on the small files and large file directory, to highlight the problem.
- Solution Proposal and Implementation: Summarize our prototype, before implementing and testing it.

6 Next Steps

The solution we proposed is versatile and can be built to handle specific MapReduce use cases. It can be optimized for processing any type of file, but easiest with CSV or Parquet files. An important consideration is that the results from the MapReduce execution might need to be returned to their original small files. This is something that wasn't implemented in our prototype, because we were mainly concerned with the execution time but it is something that will need additional configuration.

7 Conclusion

We have shown that when Hadoop is used to process many small files, the NameNode may not have enough memory to store all of the metadata, which leads to high memory consumption and slow processing times. For optimal usage, users must make sure that files are in the range of the DataNode block size, otherwise performance will be impacted. In our project submission, we were able to explain and elaborate on Hadoop and the small files issue in MapReduce. We were able to simulate the issue itself in the Jupyter Notebook and prove that time-wise, the issue could be reproduced. We proposed several approaches to solve the issue and implemented a prototype which was tested with the same generated data.

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

[3] ASF Infrabot. Apache hadoop documentation. `https://cwiki.apache.org/confluence/display/HADOOP2/Home`, 2019.

[4] Ahmed Hussein Ali. A survey on vertical and horizontal scaling platforms for big data analytics. *International Journal of Integrated Engineering*, 11(6):138–150, 2019.

[5] Rakesh Kumar, Bhanu Bhushan Parashar, Sakshi Gupta, Yougeshwary Sharma, and Neha Gupta. Apache hadoop, nosql and newsql solutions of big data. *International Journal of Advance Foundation and Research in Science & Engineering (IJAFRSE)*, 1(6):28–36, 2014.

[6] Jacob Leverich and Christos Kozyrakis. On the energy (in) efficiency of hadoop clusters. *ACM SIGOPS Operating Systems Review*, 44(1):61–65, 2010.

[7] Wei Kuang Lai, Yi-Uan Chen, Tin-Yu Wu, and Mohammad S Obaidat. Towards a framework for large-scale multimedia data storage and processing on hadoop platform. *The Journal of Supercomputing*, 68(1):488–507, 2014.

[8] Tharwat EL-SAYED, Mohamed Badawy, and Ayman El-Sayed. Impact of small files on hadoop performance: Literature survey and open points. *Menoufia Journal of Electronic Engineering Research*, 28(1):109–120, 2019.

[9] Sachin Bende and Rajashree Shedge. Dealing with small files problem in hadoop distributed file system. *Procedia Computer Science*, 79:1001–1012, 2016.

[10] Fang Zhou, Hai Pham, Jianhui Yue, Hao Zou, and Weikuan Yu. Sfmapreduce: An optimized mapreduce framework for small files. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 23–32. IEEE, 2015.

[11] Raveena Aggarwal, Jyoti Verma, and Manvi Siwach. Small files' problem in hadoop: A systematic literature review. *Journal of King Saud University-Computer and Information Sciences*, 2021.

[12] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. Nobody ever got fired for using hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, pages 1–5, 2012.