Big Data Platforms

Small files and MapReduce

Mordechai Ben-Yaacov
Mor Rotshtein

February 2022

1. Abstract

Big data is usually thought of as a collection of large files, overlooking the fact that a large number of small files also accounts to big data, Small files are essentially the files that are significantly smaller in size when compared to the default block size of HDFS.

A lot of applications generate small files in the form of log files, medical images, satellite images, meteorological data, PowerPoint presentations, etc.

The HDFS framework doesn't perform well when it comes to storing and processing a huge number of small files. This is because each small file consumes a block individually leading to excessive memory requirement, access time and processing time. Scaling the memory, allowing access latencies and processing delays.

In this paper we will explain the background, motivation and description of small files regarding MapReduce process and present approaches to overcome the problem, focusing on a one approach with suggestions of expanding it.

2. Motivation and background

2.1. Background

In order to start exploring the different approaches of the small files problem, we first need to define the process of Map Reduce itself and to understand what object storage is and to understand the difference between HDFS and object storage when running map reduce with the understanding that each one has its own issues.

2.1.1. MapReduce

Map Reduce is a data processing method designed for Big Data files which are so common nowadays and operates parallel. Before we describe the process, let us use a simple example that will demonstrate the need of Map Reduce.Imagine you have a few files, the text inside of them is a few stories of some well known author. You have a simple request, to count the number of times the word “water” appears in them. The regular way to do so would be looking at all of the files as one gigantic single file, and going over all the words and increasing the counter whenever the word “water” appears. in this way, if it takes for 1 file X seconds to be processed, then n file would take n*X seconds up to the final result. Sure, this example is very easy and not so breathtaking but it is the tip of the iceberg for all the numerous requests that happen every second from big data files. Now we are ready to present the Map Reduce Process which is combined with several steps:

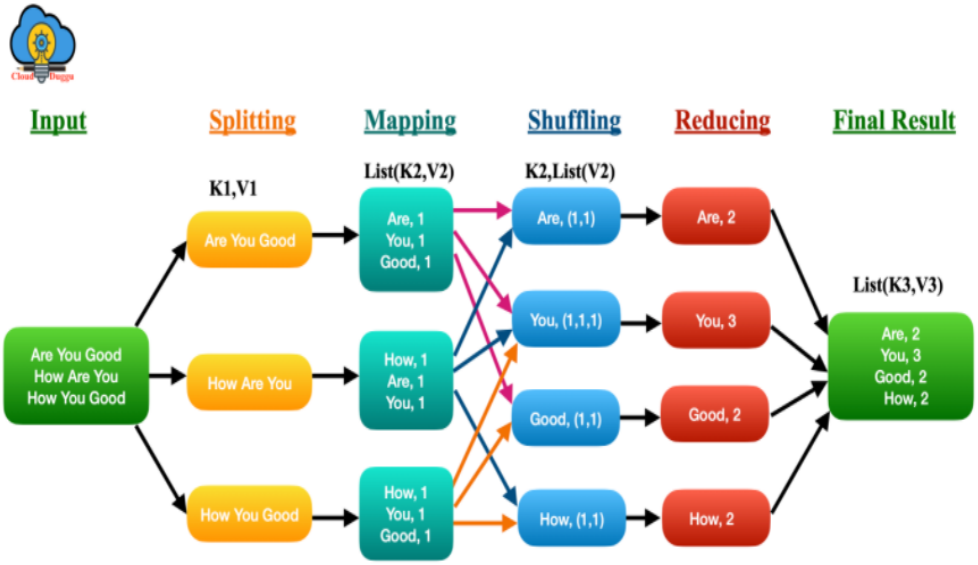
- **Preparing the data (Chunks)/ Splitting**
  This process is designed for big files. In order to shorten time and increase the parallel jobs, first each file is divided into chunks where chunk size is usually 64/ 128 Mb which is a part of HDFS. Inside each block, the file is also divided into lines/ smaller parts.
- **Map**
  In this phase, the mapper writes key value pairs from the data in its block. There are many mappers operating at the same time.
- **Shuffle**

After all mappers finished, there is a simple sort and shuffle command on all key value pairs from all mappers meaning- the key will appear once (water), and value will include all the times it appeared (1,1,1,1,1,1,1,1……,1)

- **Reduce**

  In this phase, also in a parallel way, each reducer will process the key value- in the example shown before, it will sum all "1" in value of that key.

- This is a Map Reduce scheme for a very short text:



2.1.2. Object Storage

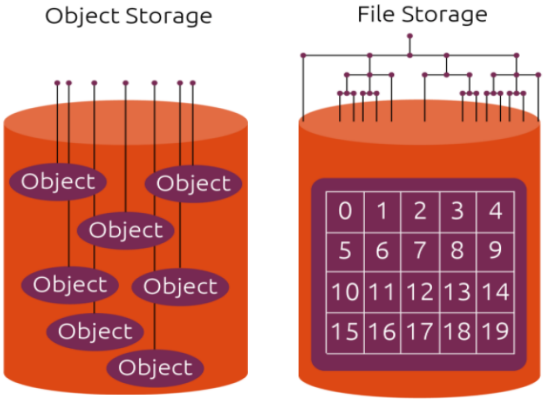
Object storage or object-oriented storage, as the name implies, is a form of data storage in separate units (objects). Objects are stored inside a single repository, and will not be aggregated into files within a directory located within other directories. Object-oriented storage uses logical frames to contain Objects in a peer-to-peer fashion. Each object will include its own data, metadata and identifier id. Cloud Storage is an object store for

all data, also unstructured data. Each object storage has a unique ID, the data itself, metadata and other attributes. The Object storage takes each piece of the data and designates it as an object that has a unique ID and is kept in separate storehouses and are not kept in the form of files in folders. There are all kinds of actions that can be performed on the data stored: retrieve, add, delete at any time. Since object storage is based on cloud storage, budget is a main issue and therefore the frequency of the use of the data stored highly influences on the plan you choose from high performance buckets from one hand to archived data for long term storage. Great advantage of object storage is its immediate time of recovery after a disaster which is highly important.

2.1.3. The difference between object storage and HDFS

Structure

- Object storage doesn't require a hierarchical structure such as HDFS. All of the objects are stored in a flat address space which makes object storage with **high scalability** compared to HDFS. Since the metadata in the object storage is placed outside the storage (linking to the object with its unique ID) it is possible to work in this flat structure.



- Object storage supports 2 actionS: PUT and GET, unlike HDFS which supports POSIX I/O calls such as: read, write, open, close, search a file.
- In the object store, data blocks are intended to be written once. This allows you not to lock the object before reading it because it is not possible that another node will attempt writing to the object while reading.
- Calling the object requires you only its unique ID which makes it very easy (using a simple hash function). It allows the computed node not to be familiar with the metadata server itself in order to know which server is the actual host.

  Consistency
- In HDFS you see the file at any given time identical to other users. All changes in data are immediate, creating **high consistency**. In object storage, obviously it will be consistent- but it will take more time, and won't be immediate.

  Atomicity
- In object storage there is a clear reflection of transactions that partly succeeded, unlike Hadoop **highly atomicity** that implements no changes when a job fails. If you think in db terms there is a low risk from this angle of corruption and it allows to keep high consistency.

  Elasticity
- Object storage allows you to add nodes without adding compute power unlike Hadoop where the compute power and the storage capacity both work together on the same node. This shows a **high elasticity** in object storage towards the change of needs that a client has.

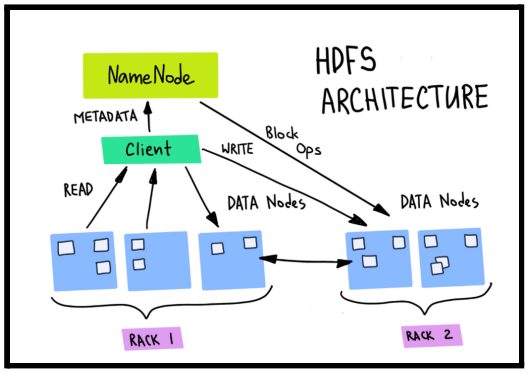
  Latency
- HDFS is usually on a private network shared with the clients who receive proper permissions and therefore there is less latency. Object storage is usually located in a far away data center which allows a higher I/O than HDFS.
- **Analytics**

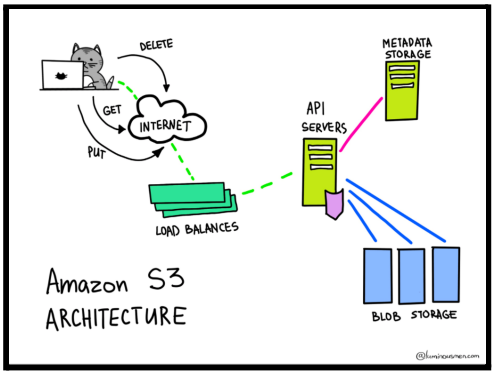
  Object storage shows more analytical power for the client since the metadata is customizable and has an unlimited number of tags. In general, you are not blocked with the amount of data you can handle, unlike HDFS.

- Table of comparison:

	OBJECT STORAGE	HDFS
Performance	Performs best for big content and high stream	Performs best for smaller files
Structure	flat	hierarchical
Consistency	not immediate, but will be eventually	immediate
Geography	data center, multiple regions	local network
Atomicity	implies the change even if a job fails	updates won't be seen if the job fails
Analytics	prefered platform	more limited
Latency		higher
Elasticity	higher	

In the following sketches, we can see the structure of HDFS and Object storage (in this example, Amazon S3) in order to start understanding the differences of running MapReduce in HDFS vs. Object storage.





in the following table we will present a few differences regarding process and the advantage as we observed:

Criterion	HDFS	Object Storage	Advantage
Infrastructure	Highly flexible for horizontal scaling, with clusters running from few nodes to thousands of nodes	more vertically scalable. We can add storage to the existing servers, but it is not easy to add a Server, limiting horizontal scalability.	HDFS
Architecture	HDFS has both compute and storage components, its clusters will consist of a minimum of one Name node with storage distributed across multiple data nodes	Pure storage with practically no compute	Object Storage
Data Types	HDFS handles structured data such as sales figures for a region, linked to Product, Salesperson. Folder hierarchies are similar to file systems, making them very transparent. It also handles 'Append,' requests.	Object storage works very well with unstructured and semi-structured data such as images, music, videos, web content where the entire 'document' is created, consumed, and deleted through a focus on ‘Create’ and ‘Delete’	HDFS

Storage Utilization	HDFS relies on replicating data to avoid data loss, which results in it using 2x or more storage space due to duplication of data	Erasure coding, which obviates the need to replicate data and yet have a fallback in case of failures, this is a significant	Object storage

3. Small files problem

3.1. Summary

Map reduce process can be held using HDFS or Object storage. Either way, running map reduce over a great amount of data which is combined of many small files will cause a major performance problem and a memory overdose. Small files problem address files size which is significantly lower than the acceptable chunk size (64/128 MB). We are living in a world with a gigantic amount of data which is increasing and respectively will increase the small files problem with the growth of the acceptable chunk size processed (128/256 MB). This situation is obligatory to come up with practical methods that will give a proper solution and won't degrade the map reduce process.

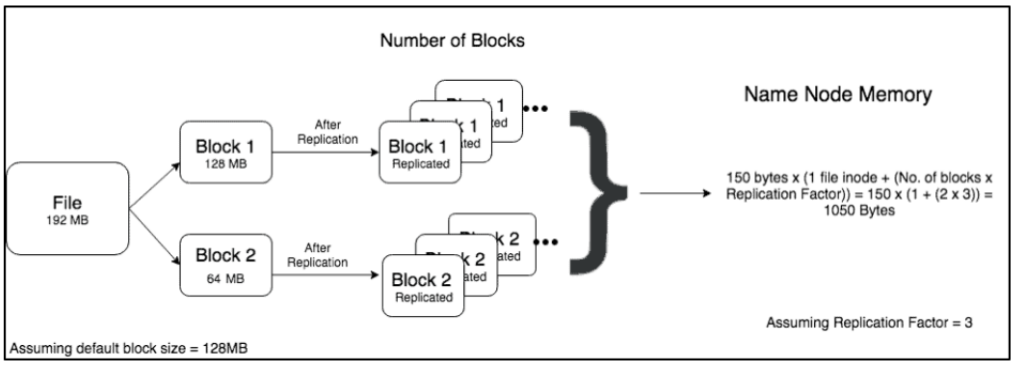
3.2. MapReduce over small files persisted in the object storage problem:

Its working principle comprises three primitive units – tables, partitions and buckets. The number of tables depends on the number of HDFS directories, the tables are partitioned corresponding to the sub-directories within each directory, and the buckets represent files in each directory (table) or sub-directory (partition) (Costa et al., 2017, Thusoo et al., 2010). Hive is an excellent option for the applications that are working on query-based languages and looking for portability to Hadoop, however, the presence of a large number of small files will result in excessive partitioning and a large number of buckets, degrading the system's performance.

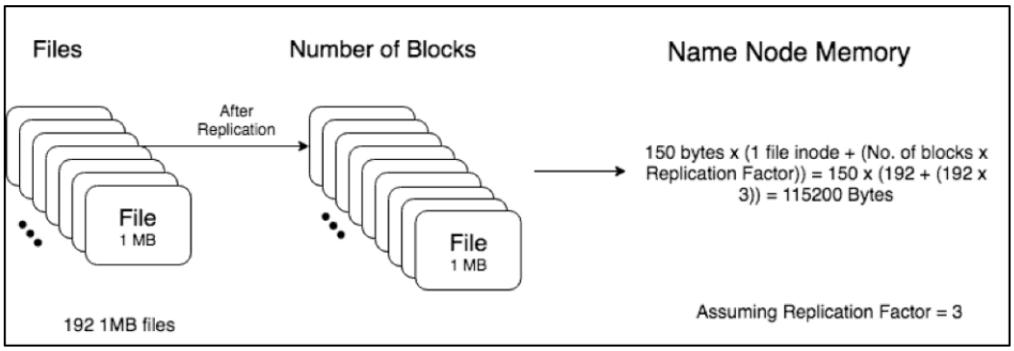
3.3. MapReduce over small files persisted in HDFS problem:

The following diagram will present a simple example facing data total size of 192 MB:

1st scenario presents a single file size 192 MB and 2nd scenario presents 192 files size 1 MB- the 192. The bottom line here is that the 2nd scenario used 110 more memory in comparison to the single file with the same total file. This situation is due to several actions that are memory consuming regardless of the file size such as their process and replicas creation. These operations when occurring on numerous small files are overloading the system and interrupting to achieve the main goal of running map reduce easily on big data. is caused due to the many operations that are memory consuming and results a massive use of memory as we can see in the next figure that takes a simple example of or needs 110 more memory than the single file:



Scenario 2 (192 small files, 1MiB each):



Hadoop was created to process large files, where files are being processed in blocks of 64/128 MB, namenode and the metadata(around 150 bytes per each). Main 2 issues in HDFS regarding map reduce is the name nodes and replicas created which are very important and fundamental in its structure but only for large files and not for a large

number of small files.

3.4. Possible approaches suggested to overcome the problem

There are several approaches and a few implementations to overcome the small files problem.

3.4.1. **First approach** which has a few branches inside of it is simple **Batching**. This approach is basically performing extra batch operations that are preliminary to the map reduce process, and after it occurs, the Map reduce is running on block size files which is how it was designed in the first place. This approach as mentioned has few ways of executing that depend on the need of the process: type of data (text etc.), origin of the data- meaning is it necessary to know what was the original file, format of the file… For simple text/ csv files it is good to use a simple batch that copy and paste all of that data under a cumulative file size block defined. simple example will be 1000 files with a list of 30 names which will turn into 1 file with 30*1000 names (under the assumption these are very small files). If the process demands the origin of the file, for example - how many times the name David appeared and in which files- it will be necessary to create **sequence files** which hold the origin file name (f.e. names_1.txt) as the key and the content as the value. after creating the sequence files it would be possible to execute the well known map reduce process on several large files instead of many small ones.

3.4.2. **HAR (archive)**- all formats supported, This is a solution given by HDFS to solve small files problems. Leaning on the archive architecture, HDFS archives many small files together as an .har file which is later on processed (using index) and saves all of the name node memory consumption that happens when performing it on many small files.

3.4.3. **Small File Layout (SFLayout) and customized MapReduce (CMR)** In order to decrease the memory/ performance problem on the name node, a unique layout for small files was created that

integrates with a customized map reduce process to this certain layout. This approach achieves more significant memory improvement than other methods shown, such as HAR. The layout part leans on batching small files where 2 files are created: an index file and a data file. —some details about what is saved. the SFlayout is combined with SFIndex, SFHeader and SFData that are not suitable for the traditional map reduce process and that is why in this approach there was a need not only in the preliminary layout but also make adjustments in the map reduce process. The customization here is to use only the map part without the reduce to achieve the answer.

3.4.4. **S3DistCp**- a unique solution for Amazon users

Amazon has developed a unique tool for amazon users that is concatenating all files by its size and making them appear 15 times faster than HDFS can do. It is using group by and target size options in order to do so. This is a private case of batch as mentioned in 3.4.1 with access to Amazon EMR users only. This approach hasn't solved the performance problem necessarily due to many map tasks that are still happening in this case.

3.4.5. Hive Compaction

Apache Hive is a distributed, fault-tolerant data warehouse system that enables analytics at a massive scale, and is built on top of Apache Hadoop so it has the ability to query large datasets, leveraging Apache Tez or MapReduce, with a SQL-like interface.

In Hive, changes in the data are saved in the partition and the longer time it's more likely that frequent changes are made and as a result there are a large number of small files. In this approach you define the list of partitions that holds more than a certain number of files and then set the reducer itself to define the approximate file size. When you execute the process, the number you have determined will be the threshold count (every # files will be operated together). Hive compaction method can ensure operating without small files.

3.4.6. **Filecrush** is a Hadoop tool which turns many small files into a few large ones (as mentioned in a few other approaches) with the same data. It gives control in several aspects: it gives the ability to define the max size of the new files created and to determine their name, knows how to 'skip' files which aren't small, controls the output compression codec and doesn't have a long running task problem. This tool allows you to use it combined in the map reduce job running (as a starter job). It knows how to scan the whole file tree and to operate only on the files which are small by definition (as it decides the threshold). Filecrush supports text files and sequence files (with any key value data type).

4. Our approach

4.1. Approach#1- Batch/ File consolidation:

4.1.1. Description

This approach adds a preliminary action which results in big files (where their origin is many small files). Those big files are stored in a specific location which from there will be the simple start point of map and reduce tasks. This approach will need to set a size which is around the acceptable block size (64/128 Mb) and will address 3 situations: first 1, when the file size is bigger than the block size, when the file size is smaller- and will distinguish if the accumulative file size is bigger or smaller then the block size defined so it will know if to start a new file. by writing small number of big files. This will reduce the DataNode memory consumption;

4.1.2. pros

This is a very simple method that is performed with a small memory performance and keeps the whole map reduce abilities and advantages since it is a preliminary step.

This will of course improve performance significantly in comparison to a map task for each small file.

4.1.3. cons

This method creates large files but doesn't keep any data/ index that refers to the original file that it was added from. This

situation limits the analysis you can perform- you won't be able to provide answers that demand the use of the index of the original file.

In addition, this method fits structured data and it doesn't necessarily describe most data files we have today.

4.2. Approach#2- Sequence Files:

4.2.1. Description

This approach is an extension of the approach mentioned in 4.1.1. but it also provides a solution for saving data of the origin of the small files. In this approach there will be a sequence file created with the exact number of small files where each key is the file name and each value will hold the data in the file. Each one is a sequence. operating map reduce on this is very easy since it knows how to split it to chunks and perform the whole process.

4.2.2. Pros

Great improvement in performance in comparison to the great memory usage without this step where collecting all the Sequence files in parallel.

This is a preliminary step that doesn't require a change in the map reduce process

Provides an answer to analysis that requires the origin of the file in its answers.

4.2.3. Cons

There may be an improvement when the sequence files exist but to create them may take a lot of time and therefore it is not suitable for any number of large files as a solution.

there is no option to list all keys that are in the sequence file and that will require going through all keys in the file.

This type of files have a certain structure that isn't necessarily supported, for example- Hive tables which are frequently used, do not support sequence files structure and that means it will need a more robust solution.

4.3. Comparison between 2 approaches

4.3.1. Performance

Both approaches consume memory in the preliminary step but keeps the map reduce process from operating many map tasks and reduce tasks (as its original design). The sequence files are much more challenging in their creation performance wise.

4.3.2. Ananlisys

Sequence files approach allows you to analyze the data including its file source, which isn't possible in the simple batch consolidation, making the sequence file approach better looking at this parameter.

4.3.3. Implementation

Both approaches aren’t complicated to implement but the first approach is leaned on very basic existing abilities which makes it easier.

4.3.4. Conclusion from Comparison

Even though the sequence files approach holds the source of all files and allows you to explore the data more widely, its cons are greater than its pros regarding the massive investment in creating the files and the very specific structure that isn't supported by basic tools. This is a call for expanding the batch file / consolidation in a more memory and structure friendly way so it will provide the answer for the cons. Until then, the simple batch file approach is the one preferred by us since it gives a suitable and simple answer to the small files problem.

5. Prototype

5.1. Description

Our prototype will lean on adding a batch phase to the local MapReduce class written where files created are approximately 200-250 Kb and we will set the size wanted as 500 Kb. The prototype takes into consideration that there can be a file that is bigger (than 500Kb) and therefore does not require handling. all files that are smaller then defined are written to a new file (one of the large files created) and after all small files have turned to larger files with all of the data, the MapReduce process begins.

5.2. Code

Link to our code (uploaded to GitHub)-
https://github.com/MordechaiBenYaacov/MordechaiBenYaacov/tree/main/final%20project

Notes to code:

- **Based on the implementation of MapReduce locally using threads.**
- **Added the batch function which gets the small files array and the threshold set or uses default of 64Mb:**
  This Function returns an array of all new large files created upon which map reduce will create its mappers and reducers.
- **In the class itself- the size of the threshold has been added as a parameter given and the batches function is operated from the map thread function.**

6. Next steps

We need to take the prototype out to a real test drive, connect to a file system or an object storage, create a significant number of small files (around 1000) and test performance before and after using our code (that will also be adjusted) in order to achieve max improvement. After that we will need to expand our code to other types of data rather than text to give a wider solution to all.

7. Conclusion

Small files problem is a high scope problem that requires more and more solutions via all MapReduce channels that it can work from. MapReduce is already a fundamental process and can't be interrupted by this small files issue. This problem is also becoming bigger as the acceptable block size is expected to increase to 256Mb. It appears there are quite a few working approaches but still no robust solution that fits all needs and data types. This requires us to keep on focusing on the approaches that can be combined with the Map Reduce process instead of changing it as some are trying to suggest. Having a built-in tool that operates as part of the process and will be a one tool fits all (data types) will give the most high quality solution, with an ability to give a proper answer to the upcoming change in block size.

8. Bibliography

1. https://www.cs.fsu.edu/~yuw/pubs/2015-NAS-Yu.pdf
2. https://mjeer.journals.ekb.eg/article_62728_c818f3f951476c6005647f9ba7364efd.pdf
3. https://luminousmen.com/post/hdfs-vs-cloud-based-object-storage-s3/
4. https://www.sciencedirect.com/science/article/pii/S1319157821002585
5. https://blog.cloudera.com/small-files-big-foils-addressing-the-associated-metadata-and-application-challenges/
6. https://medium.com/datakaresolutions/compaction-in-hive-97a1d07240
7. sciencedirect.com/science/article/pii/S1319157821002585
8. https://www.integrate.io/blog/storing-apache-hadoop-data-cloud-hdfs-vs-s3/
9. https://cloudian.com/blog/object-storage-vs-file-storage/
10. https://www.cloudduggu.com/hadoop/ecosystem/