# Big Data Platforms
# Small files and MapReduce

10.02.2022

## Contents

# 1 Abstract

Hadoop is an open-source framework used for big data processing, where data is usually persisted in large files. It is made of two main components: Hadoop Distributed File System (HDFS) and MapReduce.

Initially HDFS began with the thought of moving the compute to the data. Hadoop was designed in a way that the data was placed locally on attached storage on the compute nodes themselves. This allowed locally running tasks to directly read the data, without a network hop. While this is beneficial for many types of workloads, HDFS can’t take advantage of erasure coding to save costs and when we require data reliability, replication is not cost-effective. In addition, HDFS can not be independently scaled from compute, causing management and cost issues which is a major concern, as for many jobs the bottleneck is not network[34].

Separating compute from storage benefits both sides: compute nodes are cheaper due to not needing large amounts of storage and can be scaled quickly without high data migration costs, and jobs can run in isolation if necessary. With storage spreading out the load is wanted, onto as many spindles as possible, thus, it is beneficial on a large data pool to use a smaller active data set.

A useful way to backup your databases is object storage. It could be used to offload data from HDFS to a lower-cost tier as well. Object storage has always provided a way to store and protect data in a highly scalable, secure and cost-effective manner. We can directly access object storage when using Hadoop, as we can also leverage object storage to directly perform SQL queries[12].

We can say from existing research that Hadoop performs well with large files. However, files that are stored and processed in HDFS are not limited to very large files anymore. There exists an issue executing MapReduce on input files that are small, below the chunk size in HDFS, or the “virtual” block size in Object Storage. Map tasks process a block of input at a time, hence, if the input is very small and we have a lot of them, we get to process very little input for each map task. This also imposes extra bookkeeping overhead. As a result, we suffer a hit in the performance and effectiveness of the execution of MapReduce.

## 2 Motivation and background

### 2.1 Map Reduce

A MapReduce is a data processing tool which is used to process the data parallelly in a distributed form. This paradigm has two phases, the mapper phase, and the reducer phase. In the Mapper, the input is given in the form of a key-value pair. The output of the Mapper is fed to the reducer as input. The reducer runs only after the Mapper is over. The reducer too takes input in key-value format, and the output of reducer is the final output (Figure 1 - MapReduce architecture)[1].

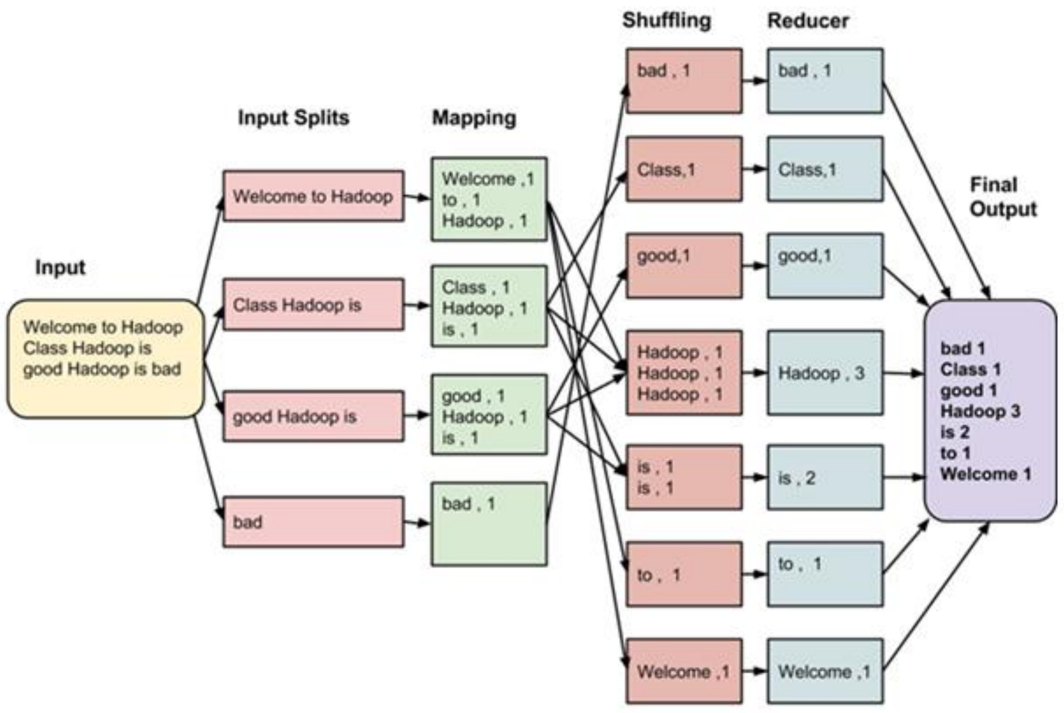
#### 2.1.1 Steps in Map Reduce

- The map takes data in the form of pairs and returns a list of <key, value> pairs. The keys will not be unique in this case.
- Using the output of Map, sort and shuffle are applied by the Hadoop architecture. This sort and shuffle acts on this list of <key, value> pairs and sends out unique keys and a list of values associated with this unique key <key, list(values)>.
- An output of sort and shuffle sent to the reducer phase. The reducer performs a defined function on a list of values for unique keys, and Final output <key, value> will be stored/displayed

### 2.1.2 Sort and Shuffle

- The sort and shuffle occur on the output of Mapper and before the reducer. When the Mapper task is complete, the results are sorted by key, partitioned if there are multiple reducers, and then written to disk. Using the input from each Mapper <k2,v2>, we collect all the values for each unique key k2. This output from the shuffle phase in the form of <k2, list(v2)> is sent as input to the reducer phase.

**Figure 1 - MapReduce architecture**



Input
Welcome to Hadoop Class Hadoop is good Hadoop is bad
Input Splits
Welcome to Hadoop
Class Hadoop is
good Hadoop is
bad
Mapping
Welcome ,1 to , 1 Hadoop , 1
Class , 1 Hadoop , 1 is , 1
good , 1 Hadoop , 1 is , 1
bad , 1
Shuffling
bad , 1
Class,1
good,1
Hadoop , 1 Hadoop , 1 Hadoop , 1
is , 1 is , 1
to , 1
Welcome ,1
Reducer
bad , 1
Class,1
good,1
Hadoop , 3
is , 2
to , 1
Welcome ,1
Final Output
bad 1 Class 1 good 1 Hadoop 3 is 2 to 1 Welcome 1

## 2.2 Object storage

Object storage, also called object-based storage, is a type of storage for unstructured, non-hierarchical data (such as email messages, documents, videos, graphics, audio files and web pages) known as objects. An object is data bundled with the metadata that describes its contents. Unlike file systems, where files are stored in folders nested within other folders, objects are stored in a flat (non-hierarchical) structure or pool. Objects may be local or geographically separated. An object is not limited to any type or amount of metadata.

Object metadata can be defined by the user and may include the type of application the object is associated with, the priority of its application, the level of data protection to assign to the object, if the object should be replicated to another site or sites and when to delete the object[2].

The main benefit of object storage is flexibility, not performance. Most object storage is based on clusters of servers. Scaling is simply a matter of adding nodes to the cluster. Data protection may be accomplished by replicating objects to one or more nodes in the cluster. Custom analytics on data use are simple to perform. Object stores are generally easy to manage, can scale almost infinitely, and can carry large amounts of metadata. Although file and block storage provide better performance, the metadata and near-infinite scalability make object storage beneficial to organizations that use big data analysis, cloud-based storage or distributed data. Object storage is a poor choice for business applications that require rapid and frequent access to data, such as financial systems.

It is easier to understand object-based storage when you compare it to more traditional forms of storage – file and block storage (Figure 2 – different data storage types)[3].

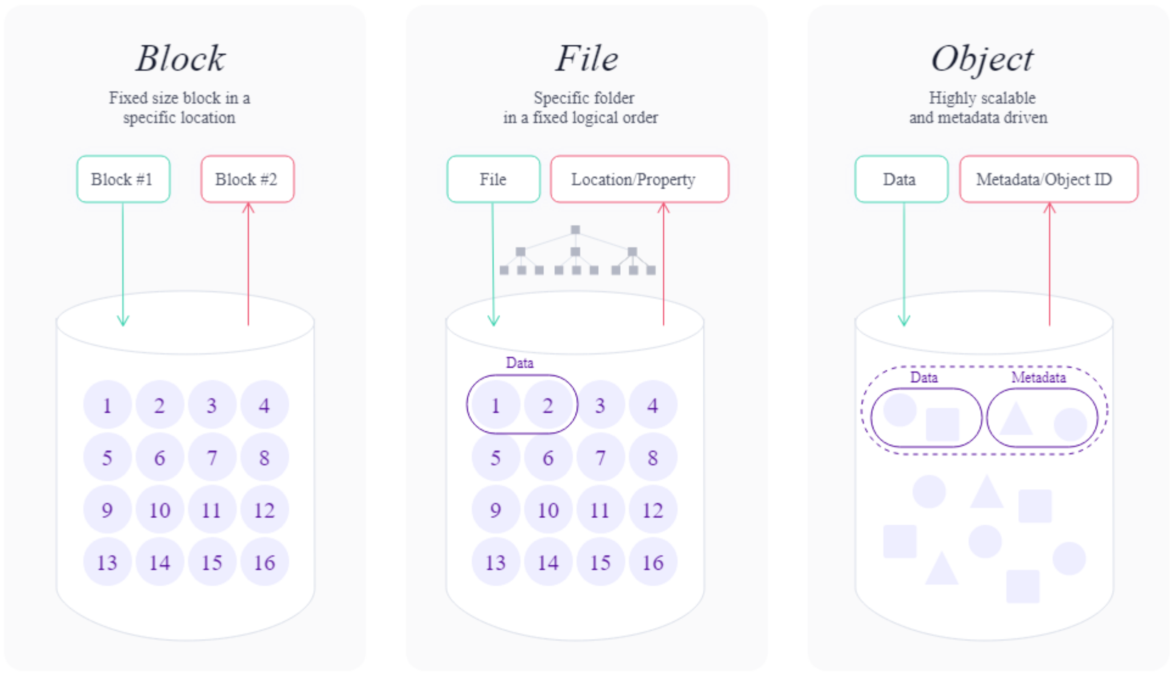
File storage stores data in folders. This method, also known as hierarchical storage, simulates how paper documents are stored. When data needs to be accessed, a computer system must look for it, using its path in the folder structure.

Block storage splits a file into separate data blocks, and stores each of these blocks as a separate data unit. Each block has an address, and so the storage system can find data without needing a path to a folder. This also allows data to be split into smaller pieces and stored in a distributed manner. Whenever a file is accessed, the storage system software assembles the file from the required blocks.

In object storage systems, data blocks that make up a file or "object", together with its metadata, are all kept together. Extra metadata is added to each object, which makes it possible to access data with no hierarchy. All objects are placed in a unified address space. In order to find an object, users provide a unique ID.

The main advantage of object storage is that you can group devices into large storage pools and distribute those pools across multiple locations. This not only allows unlimited scale, but also improves resilience and high availability of the data.

**Figure 2 – different data storage types**



## 2.3 HDFS

HDFS is the storage system of Hadoop framework. It is a distributed file system that can conveniently run on commodity hardware for processing unstructured data. Due to this functionality of HDFS, it is capable of being highly fault-tolerant. Here, data is stored in multiple locations, and in the event of one storage location failing to provide the required data, the same data can be easily fetched from another location.

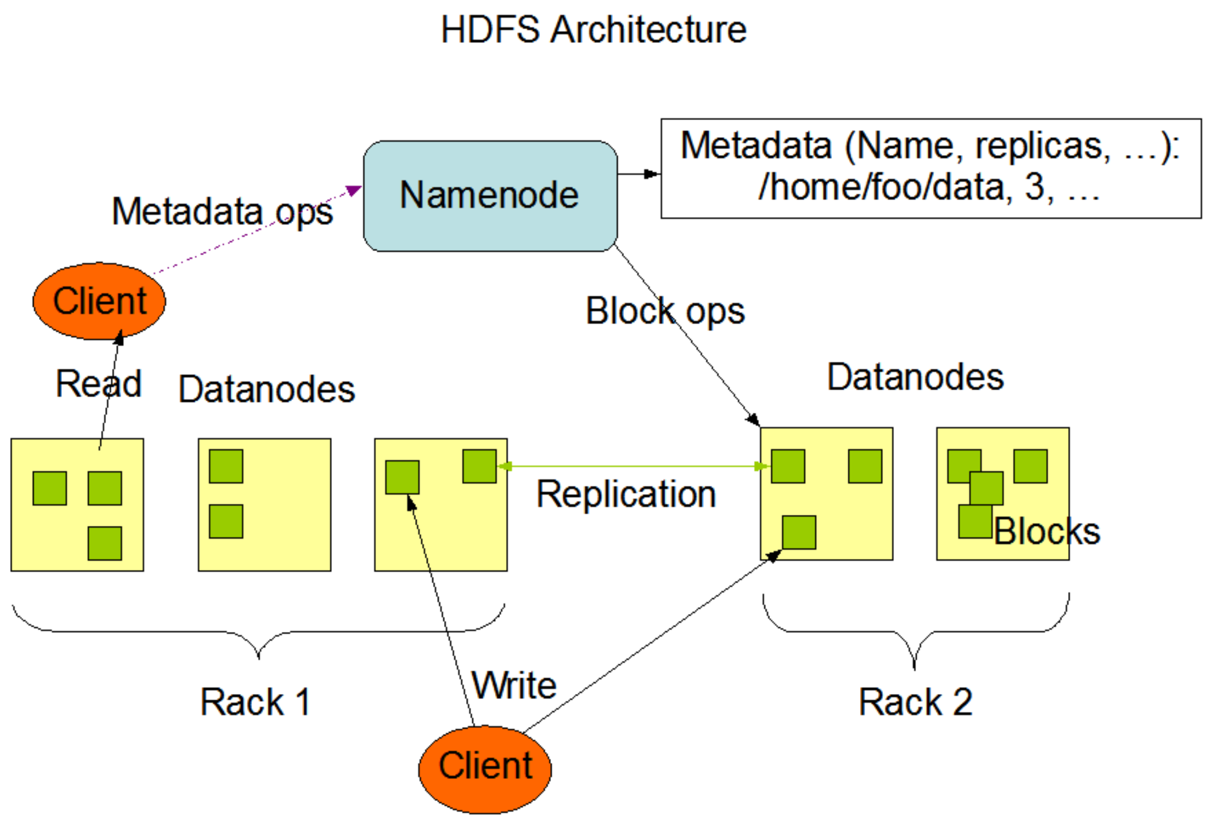
Data in a Hadoop cluster is broken down into smaller units (called blocks) and distributed throughout the cluster. Each block is duplicated twice (for a total of three copies), with the two replicas stored on two nodes in a rack somewhere else in the cluster. Since the data has a default replication factor of three, it is highly available and fault tolerant. If a copy is lost (because of machine failure, for example), HDFS will automatically re-replicate it elsewhere in the cluster, ensuring that the threefold replication factor is maintained[4].

Several attributes set HDFS apart from other distributed file systems. Among them, some of the key differentiators are that HDFS is:

- designed with hardware failure in mind
- built for large datasets, with a default block size of 128 MB
- optimized for sequential operations
- cross-platform and supports heterogeneous cluster

Hadoop Distributed File System follows the master–slave data architecture. Each cluster comprises a single Namenode that acts as the master server in order to manage the file system namespace and provide the right access to clients. The next terminology in the HDFS cluster is the Datanode that is usually one per node in the HDFS cluster. The Datanode is assigned with the task of managing the storage attached to the node that it runs on. HDFS also includes a file system namespace that is being executed by the Namenode for general HDFS Operations like file opening, closing, and renaming, and even for directories. The Namenode also maps the blocks to Datanodes (Figure 3 – HDFS Architecture)[5].

**Figure 3 - HDFS Architecture**



## 2.4 Object storage Vs HDFS

### 2.4.1 Storage architecture

HDFS supports the traditional hierarchical file organization. The user or application can create directories and store files within them. The file system namespace hierarchy is similar to that of most other existing file systems in that you can create and delete files, move them from one directory to another, and rename them.

In object storage, however, data is saved in a "object" rather than a block that makes up a file, which differentiates from file and block storage. Object storage has no directory structure; everything is stored in a flat address space. Object storage's simplicity makes it scalable, but it also restricts its utility. Once an object is created it can not be overwritten, thus all objects in the repository are immutable. To edit an object in the repository, you need to create a copy of it and make changes to it.

While Object stores are great for objects, e.g., photos, Word documents, and videos, HDFS is ideal for interactive, sophisticated analysis of very large data sets. Constant inspection and transformation of massive data sets spanning several files is a requirement of data science. It's crucial to be able to modify directories as well as files and in this aspect, object stores do not truly support directories.

### 2.4.2 Cost

To ensure information integrity, HDFS makes 3 duplicates of each dataset, so that in case a node falls flat, information is still open on other nodes. Since these are exceptionally expansive datasets, the costs of putting away all those duplicates are significant. The cost of including more information capacity must be weighed against the cost of saving all those duplicates when as it were 1/3 of the information is really utilized.

For data protection, object storage, on the other hand, employs erasure coding. Erasure is a method of data protection in which data is broken into fragments, expanded and encoded with redundant data pieces and stored across a set of different locations or storage media. It provides a comparable or better level of data redundancy while incurring far less costs than the HDFS three-way replication standard[6].

### 2.4.3 Failure handling

Hadoop consists of a master node and a number of slave nodes. The master node contains all of the Hadoop cluster's metadata. The slave nodes process the data and send the results to the master, ensuring that the data replication policies are followed. If the master node fails, the rest of the cluster is inaccessible. The loss of the master node essentially results in the loss of the cluster. Organizations need to implement their own high availability methods on the master node to avoid this single point of failure.

Object storage systems are less reliant on the master node than HDFS is. The master node's metadata can be moved and stored anywhere on the object storage system. As a result, if the master node fails, a slave node can instantly become a master node. Object stores can create backup drives to handle unexpected drive failures, and erasure encoding can be used to automatically rebuild data volumes after failure[7].

### 2.4.4 Atomicity

HDFS considers renaming and deleting actions to be atomic. This means that job output is reviewed on as an all-or-nothing basis by readers. This is critical for data reliability because when a job fails, partial data should not be written, as this could contaminate the dataset.

Object storage file system clients implement these as operations on the individual objects whose names match the directory prefix. Consequently, the modifications are not atomic and occur one file at a time. The state of the object storage represents the partially finished operation if an operation fails part way through the procedure.

### 2.4.5 Elasticity

In HDFS When you need more storage, you must also increase your processing power, and vice versa as compute power and storage capacity coexist on the same node, one cannot be added without the other.

Object-based storage architectures can be easily scaled and managed by simply adding more nodes. The data's flat name space organization, combined with its expandable metadata functionality, enables capacity and compute to be scaled separately. You can add object storage nodes as your needs change, without having to add computing power you won't use[9].

### 2.4.6 Consistency

All clients see the file's contents in the same way, as if there were only one copy of the file. The directory displaying results are current with respect to the files within that directory, and creation, updates, and deletions are seen immediately.

Object storages are, for the most part, Eventually Consistent — object creation, deletion, and updating may take some time to become visible to all calling users. For an indefinite period of time, old copies of a file may exist. Recursive file operations are used to perform the delete and rename directory operations. As a result, the time it takes to perform these operations is proportional to the number of files and in which partial updates can be observed[8].

## 2.5 MapReduce using Object storage Vs HDFS

Combining object storage in the cloud with Spark is more elastic than a typical Hadoop/MapReduce configuration. Now these applications can run with a data infrastructure that is based on object storage, which provides both performance and scaling advantages for AI workloads.

Adding and subtracting nodes to a Hadoop cluster can be done but it's not straightforward, while that same task is trivial in the cloud. Furthermore, with Hadoop if more storage is needed, it is done by adding more nodes, thus if more storage is needed, more compute is added as well whether it is required or not.

With the object storage architecture, it's different. If more compute is needed, it can be done by spinning up a new cluster without dealing with storage. Should many terabytes of new data require storage, all is needed is to expand the object storage. In the cloud, compute and storage aren't just elastic. They're independently elastic and that's good, because in practice the need for compute and storage are also independently elastic. By separating compute from the storage layer, we're no longer constrained by an infrastructure that binds both together[10].

Most HDFS clusters handle very large data sets, a situation that amplifies the 'overhead' problem created by this 3x redundancy. The obvious impact is on storage costs, which can get enormous at the petabyte scale common in many Hadoop environments. As this is the case, Users don't have unlimited resources and this cost of storage can limit how large their Hadoop clusters can be and ultimately the sizes of datasets they can handle. Due to this inefficiency users often have to ‘pick and choose’ which data they process, a limitation that can impact the accuracy of their analyses since in many HDFS use cases they’re trying to process more data, not less. The more data that can be processed the more accurate the decisions made from analysis can be. Object storage systems that make use of different data protection methods such as “Erasure coding” can offer a solution to this data redundancy problem caused by HDFS[11].

HDFS cannot save costs by using Erasure coding, as object storage does, since readable copies of data are stored directly on compute nodes. This inflexibility generated management and expense difficulties as workloads varied. For many operations, network wasn’t actually the bottleneck. Computation bottlenecks are typically CPU or memory issues, and storage bottlenecks are typically hard drive related, either disk throughput or seeking constrained[12].

Hadoop-integrated object storage systems can provide another benefit as well, this time in reliability. HDFS consolidates the metadata handling for its distributed file system on the Name Node, creating a single point of failure. Companies can address this to some extent by duplicating the Name Node but this adds cost and complexity to the cluster. Object-based architectures store the HDFS metadata in the same manner as the data objects. This creates the same highly reliable storage area with no single points of failure, providing a more reliable file system without the need for redundant Name Nodes[7].

An object storage system can be used to store data and run Hadoop on that data within the storage system itself. However, this requires more than just ‘Hadoop compatible’ storage. The object storage nodes need to be integrated with Hadoop, provide HDFS compatible storage and the CPU power to run the MapReduce process as well. This means the object storage nodes actually comprise the Hadoop cluster itself. Connectors such as IBM Stocator developed to deal with those issues, creating a seamless integration between Spark and object storage. Stocator doesn’t need local HDFS, doesn't use Hadoop modules and interacts with object storage directly[13].

# 3 Small files problems

## 3.1 Background

Hadoop has become one of the most popular high performance distributed computing standard for large scale data analytics. Well-known internet companies in the world are using Hadoop to do some relevant big data processing, such as Yahoo, Baidu, Alibaba, Facebook, Twitter. HDFS, provides the underlying data storage service for Hadoop and has good performance for large file storage and high throughput data access. However, the data which grows fastest in internet is the huge quantity of small files such as web pages, small texts and images. Therefore, storing and accessing a huge quantity of small files in the Hadoop is the most common problem. But the performance of HDFS drops significantly when storing and reading small files in Hadoop. MapReduce is a data processing tool which is used to process the data parallely in a distributed form. A MapReduce program is composed of a map procedure, which performs filtering and sorting and a reduce method. As map procedure deal with one block of data at a time, dealing with large amount of small file input will lead to low efficiency of the map and waste of the cluster computing resources

## 3.2 Problems in HDFS

### 3.2.1 Namenode Memory Bottleneck

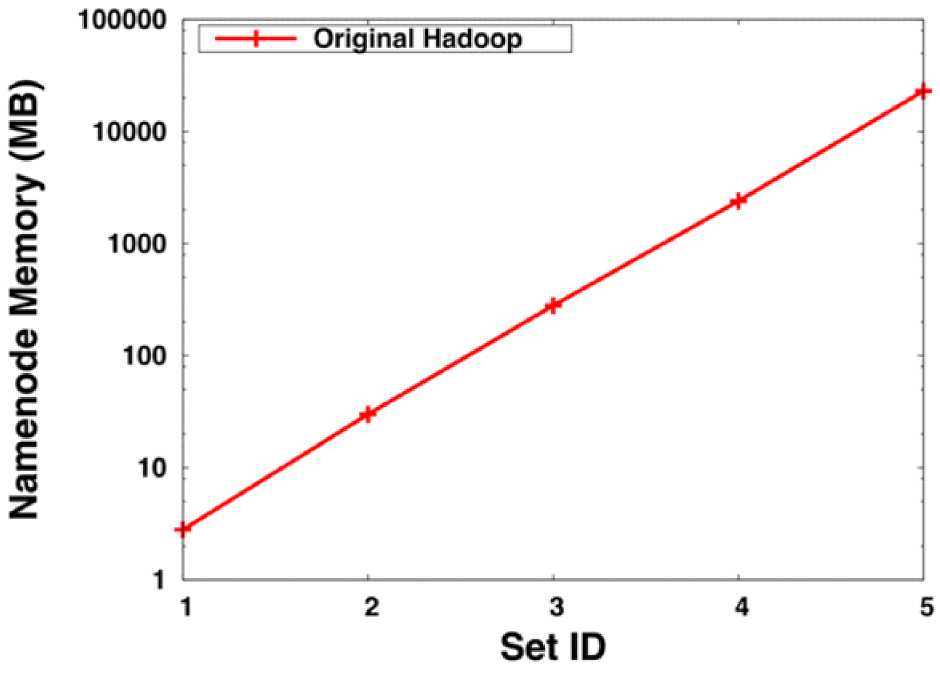
Storing huge quantities of small files in HDFS can severely consume the memory of NameNode, which can severely limit the extensibility of the Hadoop cluster and reduce the efficiency of file access and response latency to client requests.

The NameNode holds the file system namespace and is responsible for managing the metadata of the distributed file system. Typically, in NameNode memory, a single small file or directory requires 150 bytes of memory for storing its metadata, in addition to 368 bytes of memory for a block with default three replicas of each small file. If 100 million small files are stored in the system, the NameNode requires at least 30GB of memory for storing the metadata (Figure4 - Namenode memory usage). Therefore, an application consisting of a huge number of small files will eventually consume all memory spaces of NameNode. And the memory capacity of the NameNode can restrict the scalability of Hadoop cluster[14].

**Example:**
**Case 1:**
A file of 128 mb
# of blocks created –1 data block
Total metadata bytes – (1 file + 1 block) *150 – 300 mb
**Case 2:**
128 files of 1 mb
# of blocks created – 128 data block
Total metadata bytes – (128 files + 128 blocks)*150 - 38400 mb

For a file of the same size, the metadata consumption has become multi-folds because of small files.

**Figure 4 - Namenode memory usage**
5 data sets for NameNode memory tests, which include 10K, 100K, 1M, 10M and 100M files. When storing 3 TB small files into HDFS, the memory used by NameNode is more than 22GB[19].



### 3.2.2 Unacceptable storing time

HDFS is originally designed to manage large files, and not optimized for small files. Therefore, it suffers performance penalty when managing a large number of small files. It takes about 7.7 hours to store 550,000 small files, ranging from 1KB to 10KB, into HDFS. On the contrary, storing those files into a local file system takes about 660 seconds[15].

### 3.2.3 Cross-node transmission of files

In an effort to retrieve a large number of small files from the system, a lot of seeks occur, leading to a lot of hopping from one DataNode to another DataNode for each small file. This leads to excessive congestion on the network and reduced system throughput. The time spent in retrieving small files from the system cannot be amortized over the time spent in the actual processing of these files. In addition, HDFS does not take into account the correlation between files in the data storage, therefore further contributing to the traffic between different DataNodes, which will greatly reduce the performance of HDFS[16]

### 3.2.4 2. Low Disk Space Utilization of Datanode

The data in Hadoop uses the data block as the smallest storage unit, the data needs to be split and stored when the size of file is larger than 64MB. However, the file which is smaller than 64MB is directly stored in the HDFS as a data block. This will lead to the waste of memory space of DataNode, which makes the storage space utilization low[15].

### 3.2.5 High Access Latencies

The access mechanism of HDFS leads to high access latency issues. First of all, when HDFS clients access a small file from HDFS, it is necessary to obtain metadata from NameNode once for each file access. For small files, data transfer takes very short time while disk seek and managing metadata becomes major overheads. With lots of small files, HDFS clients must call NameNode every frequent which can impact the performance of NameNode significantly[16]. The high latencies occurring due to the access protocol of HDFS are not justified in the case of small files because the time spent in performing I/O operations exceed the time spent in the actual processing on the data and hence, there is a need to regulate the frequent communications among the clients, the metadata server and the DataNodes.

## 3.3 Problems in MapReduce

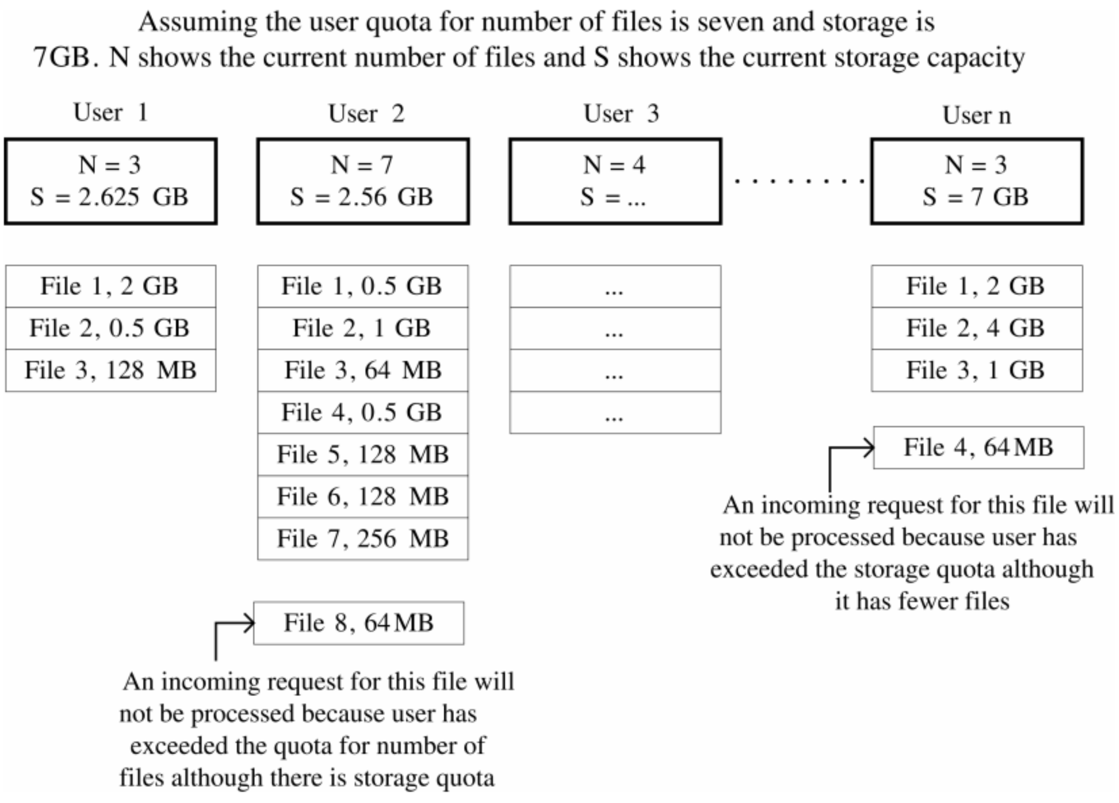
### 3.3.1 Policy restrictions limitations

MapReduce is a parallel programming model that separates Hadoop clusters into JobTracker and TaskTracker by function. JobTracker is responsible for all resource management and all task scheduling, while TaskTracker is responsible for task execution and resource management on a single node. In multi-user systems as such, users are given quotas of how much of an available file system they are allowed to access and use. The HDFS provides a scheme to put quotas on user directories. In Hadoop's implementation, administrators are given two options for applying user quotas:

1. maximum number of files per directory
2. maximum file space for a user directory

For example, in Figure 5 - policies limitations, user 2 has reached the number of files allowance, although the space quota is still not exploited. In this case the job is killed by the map/reduce scheduler, the JobTracker. The job is terminated regardless of progress, causing long running MapReduce programs to fail unnecessarily. Thus, in case of processing large number of small files MapReduce program may not complete the tasks as it will be terminated prematurely by the JobTracker due to quota policies[17].

**Figure 5 - policies limitations**



Assuming the user quota for number of files is seven and storage is 7GB. N shows the current number of files and S shows the current storage capacity
User 1
N = 3
S = 2.625 GB
File 1, 2 GB
File 2, 0.5 GB
File 3, 128 MB
User 2
N = 7
S = 2.56 GB
File 1, 0.5 GB
File 2, 1 GB
File 3, 64 MB
File 4, 0.5 GB
File 5, 128 MB
File 6, 128 MB
File 7, 256 MB
File 8, 64MB
An incoming request for this file will not be processed because user has exceeded the quota for number of files although there is storage quota
User 3
N = 4
S = ...
User n
N = 3
S = 7 GB
File 1, 2 GB
File 2, 4 GB
File 3, 1 GB
File 4, 64MB
An incoming request for this file will not be processed because user has exceeded the storage quota although it has fewer files

## 3.3.2 Low efficiency

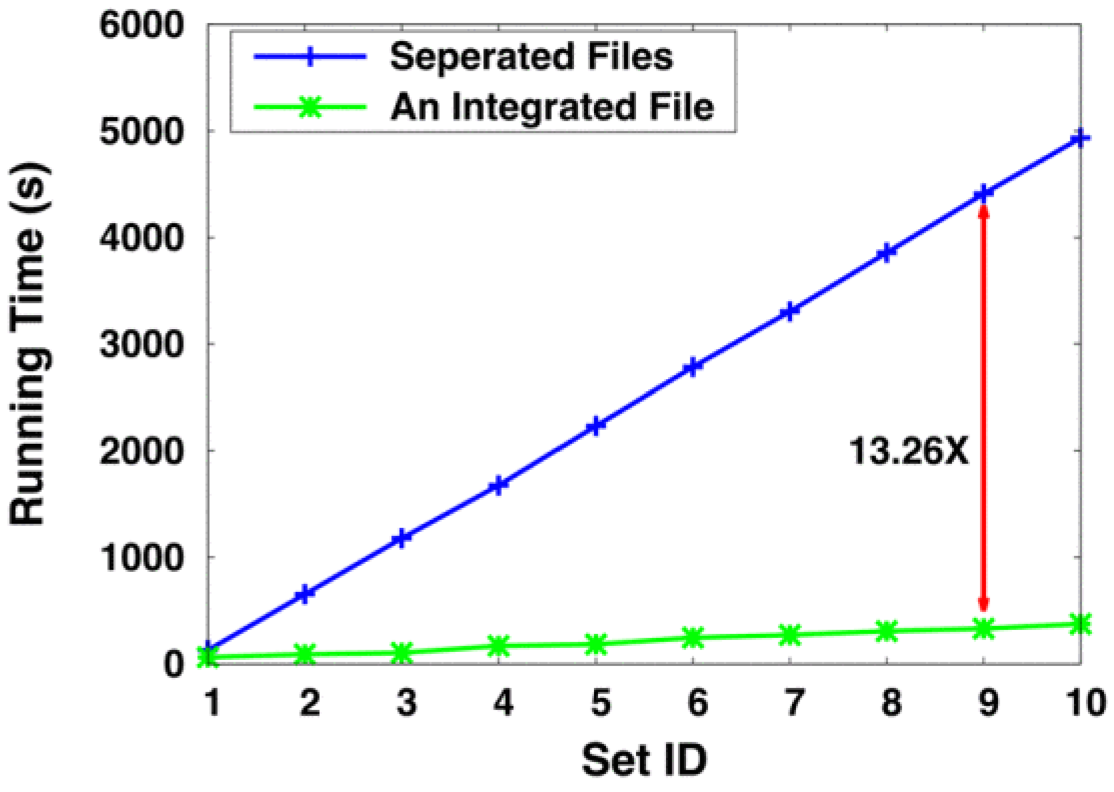
In HDFS, files are stored in blocks and the default block size is 64MB and a small file which is smaller than 64MB also occupies a block. However, a map task handles a data block at a time, in the case of massive small file input, MapReduce will open many tasks to process this data. In fact, each Map task only handles a very small amount of data. However, the start and shutdown of the map task consumes a lot of time, which leads to the low efficiency of the map and the waste of the cluster computing resources.

### 3.3.3 Processing overheads

InputSplit is the smallest unit of MapReduce processing and comprises data from a single block. FileInputFormat is a module that creates a split per file, to be processed by a map task. In the case of a large number of small files, a lot of InputSplits are generated, leading to the creation of an excessive number of map tasks[18]. The mapper will be invoked corresponding to each InputSplit generated. Between each invoking, a context switch takes place. So, creating and closing so many processes (mappers) continuously and the corresponding context switches result in a lot of overhead. This situation can seriously delay performance. Also, in case of processing a small file, the map phase has to deal with a single InputSplit. In general case, the role of a reducer is to combine the outputs obtained from multiple mappers and reduce it to a simpler output. But, in this case, we have a single mapper which means that the reducer's output is already generated in the map phase. So, to extract the output from the map phase and feed that input for the reduce phase adds another overhead. As shown in Figure 6 - Performance issues, when the number of small files increases, the gap between processing these small files and one, equal in size, integrated file grows rapidly[19].

**Figure 6 - performance issues**

The average performance of the set of small files is 10.49x lower than that of the integrated file.



## 3.4 Possible solutions for small files

### 3.4.1 Metadata management optimization

Metadata optimizations may be performed for any file sizes, but they mostly work well for small files. Various strategies have been adopted to optimize the metadata management in HDFS clusters for small files.

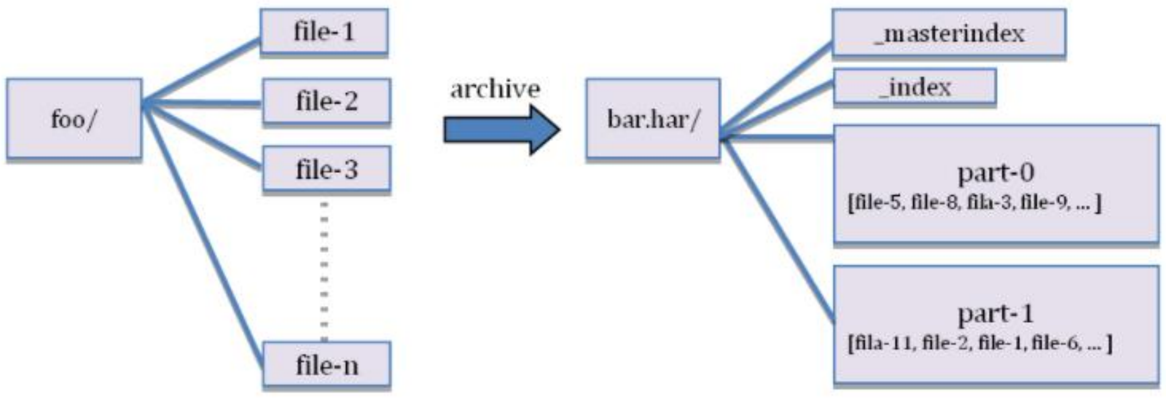
#### 3.4.1.1 File merging and indexing

The simplest choice of techniques is merging the small files into larger ones and maintaining an efficient index corresponding to this merged file. Hadoop has itself offered some solutions for the same. For example, Hadoop Archives (HAR), SequenceFiles and MapFiles. The file merging process is mostly followed by an indexing scheme so that a particular file may be retrieved back easily with the help of indices. Most of the solutions observe that the process of merging files and indexing them can be a time-consuming process. This implies that rather than improving the response time, this technique focuses on improving the system throughput.

#### 3.4.1.1.1 Hadoop Archives (HAR)

Hadoop Archive is a file archiving capability of Hadoop, which appears to relieve the problem of a large number of small files consume the memory of NameNode. HAR files are created by using the Hadoop archive command which pack numerous small files into a HAR file by MapReduce job (Figure 7 - HAR file layout). The original file, however, can also be accessed in parallel and transparently. HAR is effective in reducing the number of small files and alleviating the memory pressure of NameNode. In addition, it also supports random access to file. But there are drawbacks of HAR. Firstly, obtaining files from a HAR requires access to the two index files, which results in reading small files from the HAR even slower than in the original HDFS. Secondly, HAR file can't be changed after it has been created, and the HAR file must be re-created when it needs to add or delete file. Thirdly, the original file is not automatically deleted, which result in a waste of disk space of DataNode. Finally, the correlations between small files are not considered when creating the HAR file[20].

**Figure 7 - HAR file layout[21]**

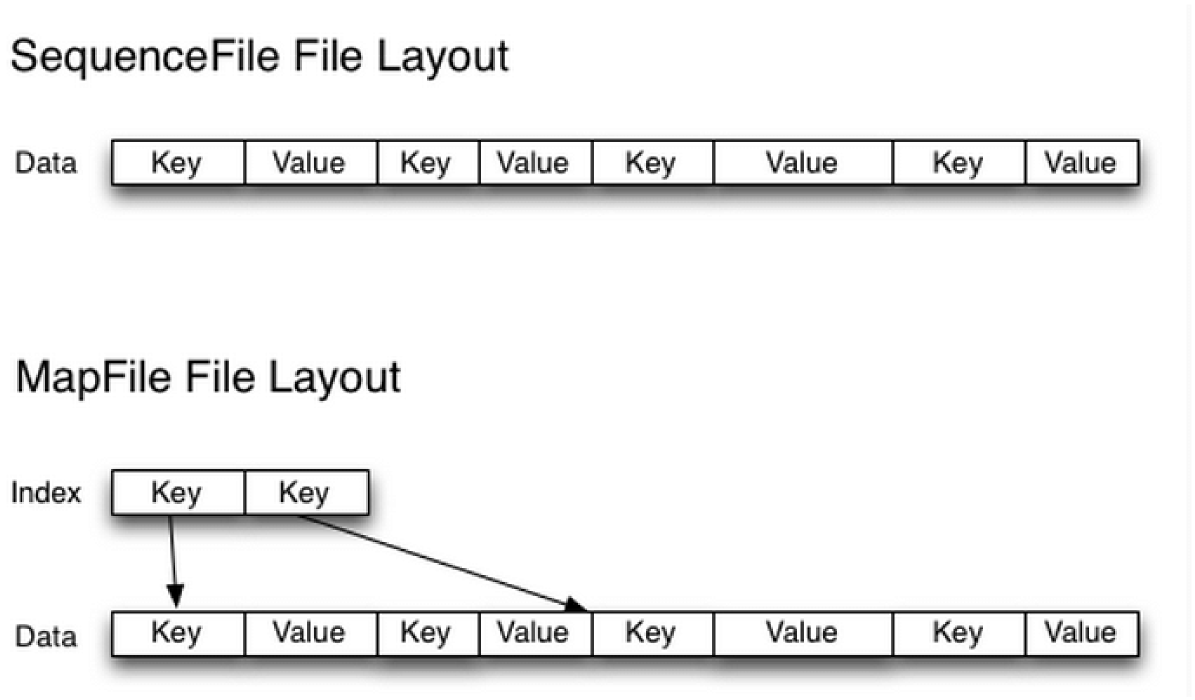

foo/
file-1
file-2
file-3
file-n
archive
bar.har/
_masterindex
_index
part-0
[file-5, file-8, fila-3, file-9, ... ]
part-1
[fila-11, file-2, file-1, file-6, ... ]

In order to overcome these difficulties a new method, based on HAR, called NHAR, was introduced. The NHAR model improves the index mechanism of HAR by changing the two-level index into a single-level index, which improves the efficiency of file access. In addition, NHAR extends the capabilities of HAR to allow additions and deletions in existing archives. The results show that NHAR not only reduces the memory footprint of NameNode but also improves the efficiency of small file access. But the correlations between small files are not taken into account when creating the NHAR file[14].

#### 3.4.1.1.2 SequenceFiles and MapFile

SequenceFile is the flat file stored in the form of key-value pairs where key holds the name of the file and value stores the data of the file which is used to complete the merge of small files. SequenceFile can significantly alleviate the memory pressure of NameNode through the merge of small files and Compression and decompression are supported at the block level, which effectively reduces the disk consumption. But there are drawbacks of SequenceFile. Firstly, SequenceFile did not establish the mapping between small files and large files so in order to find a file, we must seek the entire SequenceFile. Therefore, it is not suitable for random access with low-latency. Secondly, for a file in SequenceFile, only add operation is supported, there is no delete and update operation. Finally, the correlations between files are not taken into account. In order to solve the disadvantages of SequenceFile, MapFile was proposed and is based on SequenceFile. MapFile introduced the index mechanism and sorted the files by key. A MapFile includes a data file and an index file. Therefore, it can get a specific file by the index file. Compared to SequenceFile, MapFile not only relieves the memory pressure of NameNode, but also reduces the access delay of file (Figure 8 - SequenceFiles & MapFile layout)[22].

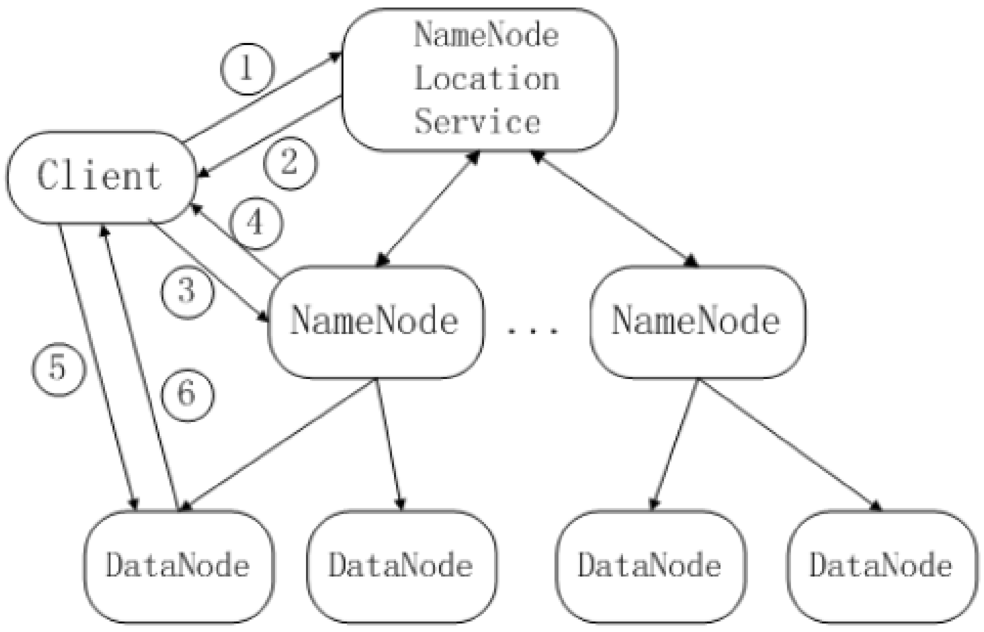
**Figure 8 - SequenceFiles & MapFile layout[23]**



#### 3.4.1.2 NameNode scalability

A single NameNode helps in keeping the architectural configurations simpler but may lead to certain performance bottlenecks. A way of dealing with the limitation of NameNode memory is to use more than one node for storing metadata. This helps in dividing the load of a single NameNode over multiple machines. This strategy further helps in overcoming the single point of failure. In the case of a single NameNode, if NameNode goes down, the entire system will go down. So, having a secondary NameNode can always avoid such situations. However, as it may solve the problem of small files' storage it gives rise to another set of research problems such as load balancing, increased complexity and synchronization among various NameNodes. In one of the suggested architectures a new module called NameNode location service (NLS) is added to the otherwise unchanged HDFS model. This component enables multiple NameNodes coexistence where each NameNode manages a separate set of DataNodes. NLS module holds all of the NameNodes' basic information and responses to any client's file location query requests (Figure 9 - federated HDFS architectures)[24].

**Figure 9 - federated HDFS architectures**



### 3.4.2 Data management optimization

The blocks in Hadoop are of fixed size. Each block stores a single file, so, if the file sizes are considerably smaller than the size of a block, it leads to inefficient disk optimization. Various strategies have been proposed to optimize the data management in Hadoop.

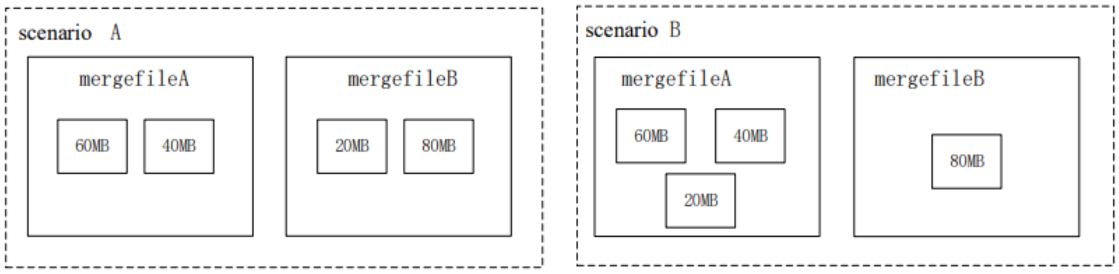
#### 3.4.2.1 Caching and prefetching

In general, Prefetching and caching schemes are widely adopted for improving the access efficiency. Prefetching can avoid disk I/O cost and reduce the response time by considering the access locality and fetching data into cache before they are requested. Three prefetching strategies are used to improve the access performance: metadata caching, index file fetching, and the merged data file fetching. Firstly, when a client requests a small file, the small file gets the metadata of the merged big file from NameNode via the metadata mapping file. If the metadata of the big file is in the local memory, the client can directly access it. Thus it can reduce the I/O cost between NameNode and the original small file. Secondly, based on the obtained metadata, the client can know the connected block and access the requested file. If the index file has been buffered from DataNode, accessing the small file belonging to the same big file can reduce the I/O operations. Thirdly, when the requested small file has been returned to the client, the related files can be cached based on their locality in a merged big file. Therefore, exploiting access locality and caching the correlated files in a merged file from DataNode can reduce the computational cost and keep high access efficiency[25].

#### 3.4.2.2 File correlations

File correlations and file merging are closely coupled together. Although file merging is a metadata optimization technique, to further improve the access efficiency of files merged together, file correlations come as an aid. There are a lot of ways to define correlation between files, correlations have been established based on their access frequency, upload time, read-write intensity of the files, file type, file extensions and other considerations. When correlations between small files and directory structure of data are considered in the process of merging small files into large files and generating the index files, it reduces the seek time and delay in reading files and improves performance. For example, consider 4 files A: 60MB, B: 40MB, C: 20MB, D: 80MB with A,B and C having higher correlations than with D. These 4 files can be merged into two equal size 100MB files, or, when correlation is considered, into two files, the first is a 120MB file containing A, B and C and the second is a 80MB file which holds only D. The latter option is preferable as it combines correlated files in the same block[26].

**Figure 10 - correlations in file merging**



#### 3.4.2.3 Reduced InputSplits

InputSplit is the smallest unit of MapReduce processing. FileInputFormat is a module that creates a split per file, to be processed by a map task. In the case of a large number of small files, a lot of InputSplits are generated, leading to the creation of an excessive number of map tasks. MapReduce has offered a tool called CombineFileInputFormat which packs several InputSplits into one, based on the nodes and rack locality. This ensures that each map task processes an adequate amount of data from multiple blocks to enhance the MapReduce execution rate. Furthermore, in case of small files, the MapReduce jobs may be replaced by Map-only applications as the job of reduce phase has already been performed in the map phase[27].

#### 3.4.2.4 Architectural changes

Some changes can be implemented in the architecture of HDFS to deal with the inefficient handling of small files. Typically, a data block can contain a single file and the metadata is only stored on the NameNode. In one solution, a block was allowed to store multiple small files. In consequence, the file-to-blocks mapping was replaced by block-to-files mapping. Also, the DataNodes were allowed to cache a part of metadata containing the small files stored in it. This helped in reducing the influence of a single master and reducing the read/write latencies.

Other architectural modification solutions proposed that in order to reduce the memory usage, the NameNode will store the metadata only till a certain threshold value. After the threshold is reached, any further metadata is redirected to a secondary flash drive media. an indexer was introduced at NameNode to locate the metadata placed on the flash drive. Another option relates to the need to reduce the network flooding. In this solution, each DataNode is allowed to hold a copy of the metadata of files contained in it, so that access requests for the same files can be handled without establishing connection with NameNode. Further solution is to eliminate a single point of failure, the metadata of each DataNode was linked to the metadata of next and previous DataNode so that the access requests can be completely handled by the DataNodes if NameNode faces an issue[28].

## 3.5 Small Files Problem in Object storage

The problem with MapReduce being executed over small files persisted in Object storage is the same when using HDFS in regard to executing the compute functions of the MapReduce. We'll touch upon the issues for workloads accessing large numbers of small files within Object storage. There are two reasons: per-file metadata server interactions and loss of namespace locality at the storage devices.

### 3.5.1 Per-file metadata server interactions:

Firstly, a client must have the corresponding mapping information and capabilities to access a file's data. The metadata server must be interacted with by the client to get them. Only then the storage devices can be directly communicated with by the client to access the data. For each access this metadata server interaction happens once. This interaction is usually a minor overhead paid over many data accesses for a client accessing the contents of a large file. However, for a small file there can be as few as one data access.
Two performance problems can result in increased latency for client access and heavy load on the metadata server. Client latency can be doubled and the metadata server can be requested to service as many requests as all the storage devices combined, since accessing each file's data requires first interacting with the metadata server.

### 3.5.2 Loss of storage locality:

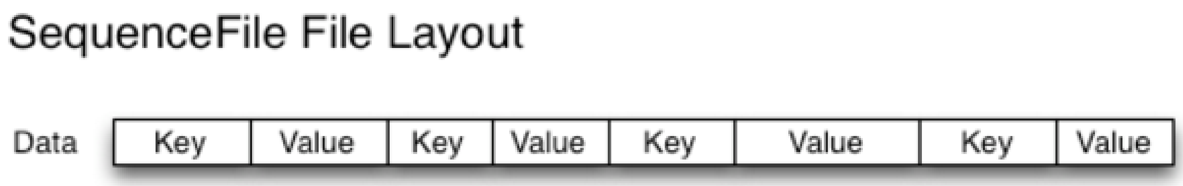
In object-based storage, for objects they store the storage devices allocate on-disk locations. The performance consequences and goals for this are essentially the same as those for local file systems. When looking at large objects, we achieve good performance by ensuring that each object's contents are placed sequentially. However, for small objects good performance requires inter-object locality, keeping objects that are likely to be accessed together close together on disk.

By exploiting the relationship hints exposed by the directory structures, most file systems achieve small file locality. Object-based storage systems make this difficult by hiding this information from the storage devices, only the metadata server knows about the directory structures. The storage devices see object IDs instead and, thus, can't effectively employ the locality-enhancing tricks common in file systems[29].

## 4 Our Approach

**First Approach:**

Let's look at the Sequence File, which tries to solve the small files problem with HDFS and adjust it slightly for our case with Object Storage. It has a file format that stores serialised key/value pairs. Sequence Files can be used as containers for a large number of small files thus solving the drawback of processing a huge number of small files. We can also use it within the MapReduce jobs as input and output formats. Internally, the temporary outputs of maps are also stored using Sequence File format.



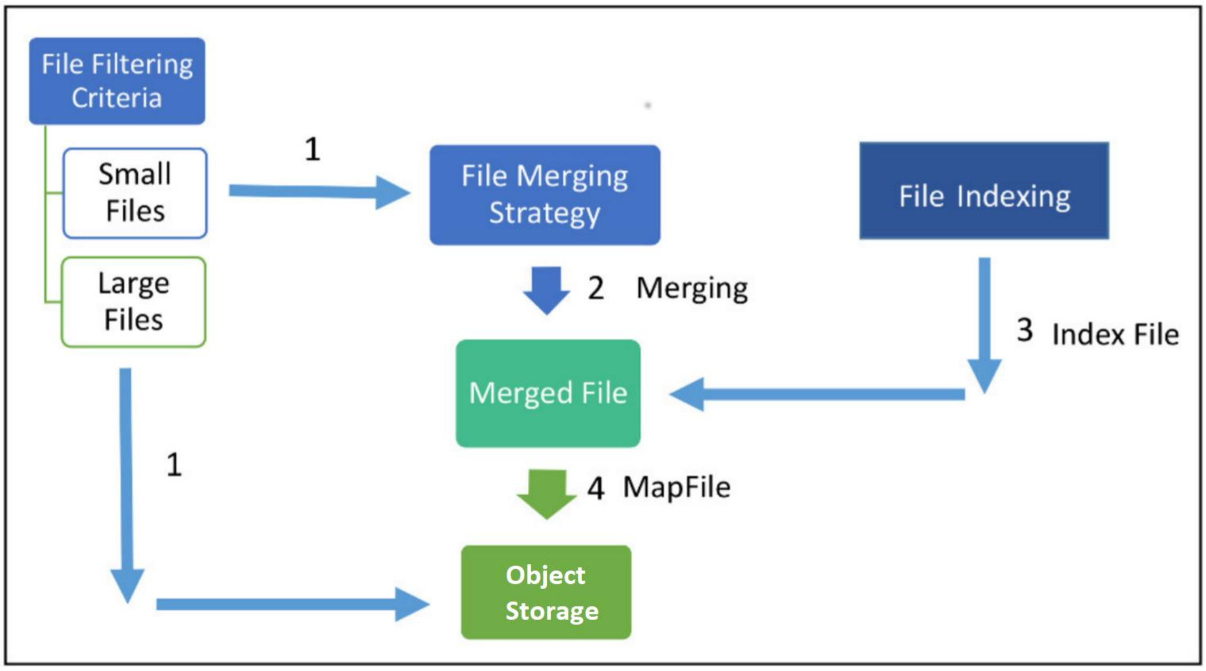
The idea behind the sequence files is that the filename is used as a key and its contents as the value. We can write a single program to put a number of files into a single Sequence file, and then you can process them in a streaming fashion by using a map reduce technique. The advantage of using sequence files is that they can be split. So MapReduce can break them into chunks and operate on each chunk independently[30].

It can be slow to convert existing data into SequenceFiles. It is perfectly possible to create a collection of SequenceFiles in parallel. As a possible solution it's best to design your data pipeline to write the data at source directly into a SequenceFile, if possible, rather than writing to small files as an intermediate step.

**Second Approach:**

To overcome problems in handling small files, let's merge small files into a larger file and store it on the object storage. Variation in the size of distribution of files is not taken into consideration while merging small files. We will use a modified MapFile technique attempting to reduce memory wastage, when considering virtual block size, while merging small files into large MapFiles, which takes into account the size distribution of files presented in a file set. As this is an extension to Sequence File, we will consider this for our prototype. In order to achieve fewer data blocks consumed for the same number of small files, we merge the small files into larger files.  Less number of data blocks means increased efficiency of data processing.

**Figure 11 - file segregation and merging**



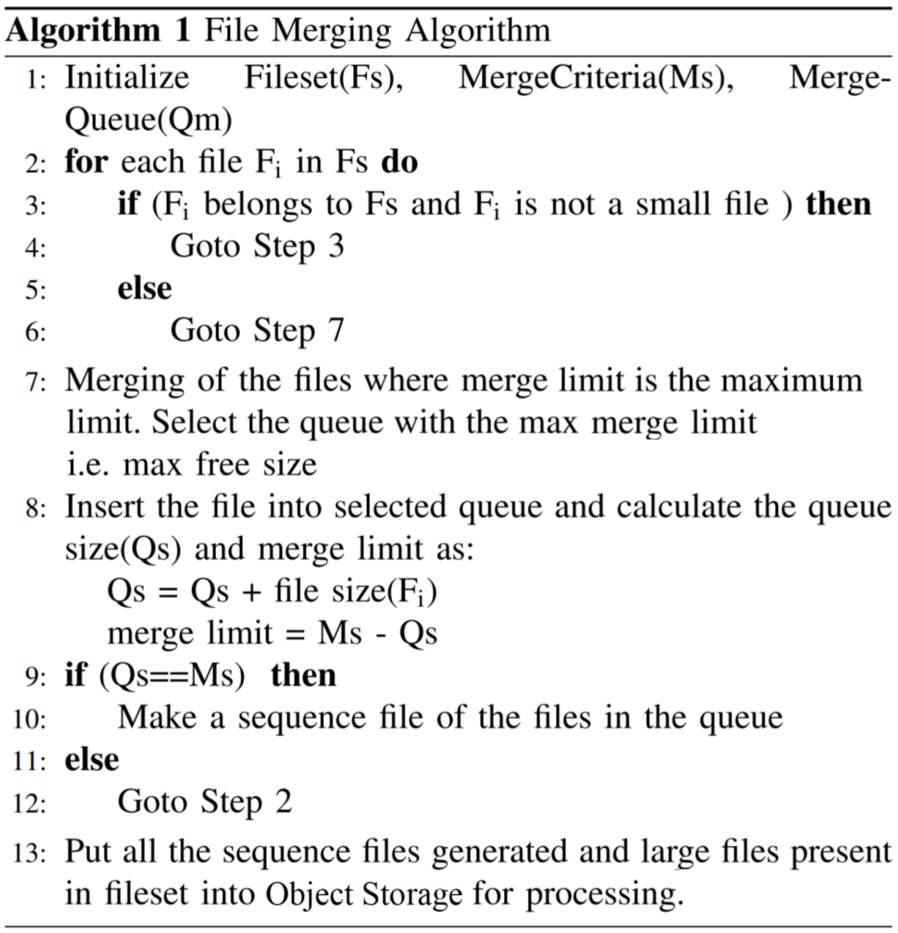
The File Set contains the set of files to be processed into the MapReduce. As shown in Figure 11**,** we first segregate large files present in the File Set from small files. Then the large files are directly put into MapReduceEngine for processing. We sort the small files in the File Set into various merging queues. Small files are put into the queue with the maximum merge limit. Files are placed into the merging queue until the queue size becomes equal to the merge criteria. After the files are sorted into various queues, the files are merged and converted into MapFiles, one per queue. MapFile is a file containing sorted key-value pairs. Here, key is the file name and value is the file contents. MapFile also maintains an index file to facilitate faster access of small files. These MapFiles are then put into MapReduceEngine for processing.

The proposed File merging strategy for MapFiles consists of two parts,

1. File filtering based on the size of small files
2. File merging using the Worst-fit strategy.

File Filtering:

For optimising performance of Object Storage for handling small files, it is crucial to decide the cut off point between small files and large files.



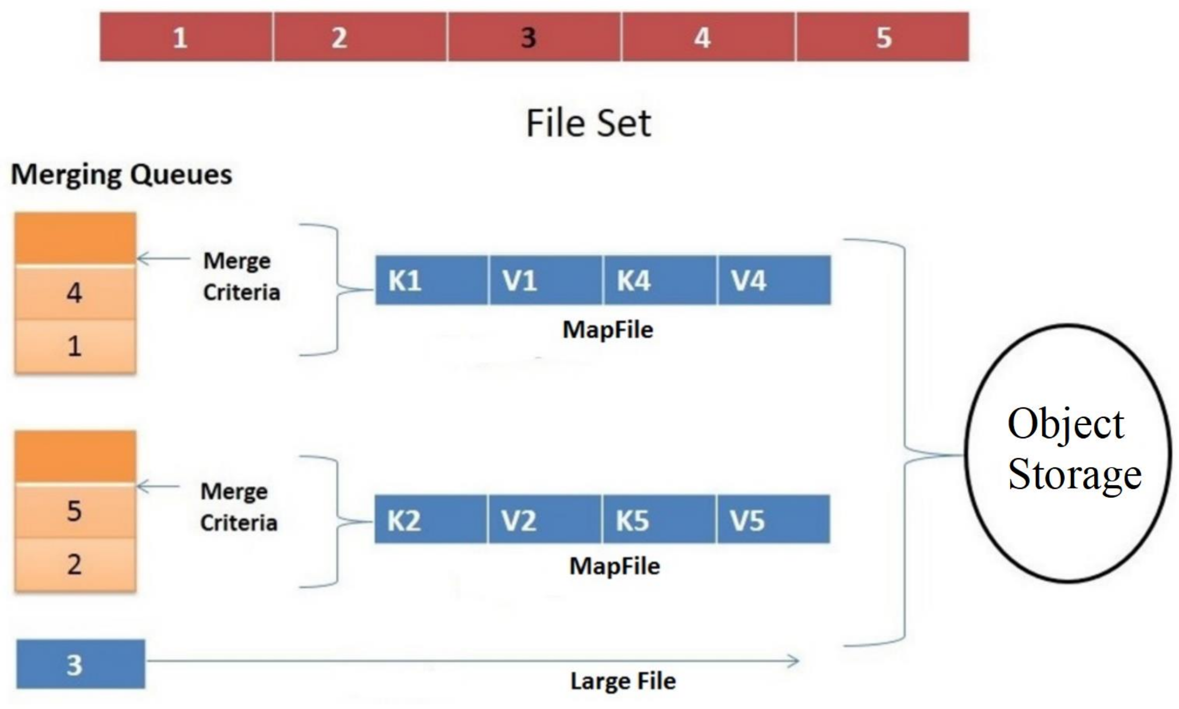
**Algorithm 1** File Merging Algorithm

```
 1: Initialize Fileset(Fs), MergeCriteria(Ms), Merge-
    Queue(Qm)
 2: for each file $F_i$ in Fs do
 3:     if ($F_i$ belongs to Fs and $F_i$ is not a small file ) then
 4:         Goto Step 3
 5:     else
 6:         Goto Step 7
 7: Merging of the files where merge limit is the maximum
    limit. Select the queue with the max merge limit
    i.e. max free size
 8: Insert the file into selected queue and calculate the queue
    size(Qs) and merge limit as:
        Qs = Qs + file size($F_i$)
        merge limit = Ms - Qs
 9: if (Qs==Ms) then
10:     Make a sequence file of the files in the queue
11: else
12:     Goto Step 2
13: Put all the sequence files generated and large files present
    in fileset into Object Storage for processing.
```

File merging:

We address the file merging process as shown in figure 12 below. There are five files namely 1,2,3,4,5 in the file list. Out of which File 3 represents a large file, while all the others are small files. Here, sizes of file 1,2,3,4,5 are taken 6,7,200,4,2 units respectively. And let's consider the threshold limit to be 10 units. Our file merging algorithm puts the file into various merging queues. Files 1 and 4 are placed into the 1st queue, while 2 and 5th into the second one. The files in the queue are then converted into MapFile (key, value pair). The MapFiles generated and the large file are then put into the Object Storage for processing.

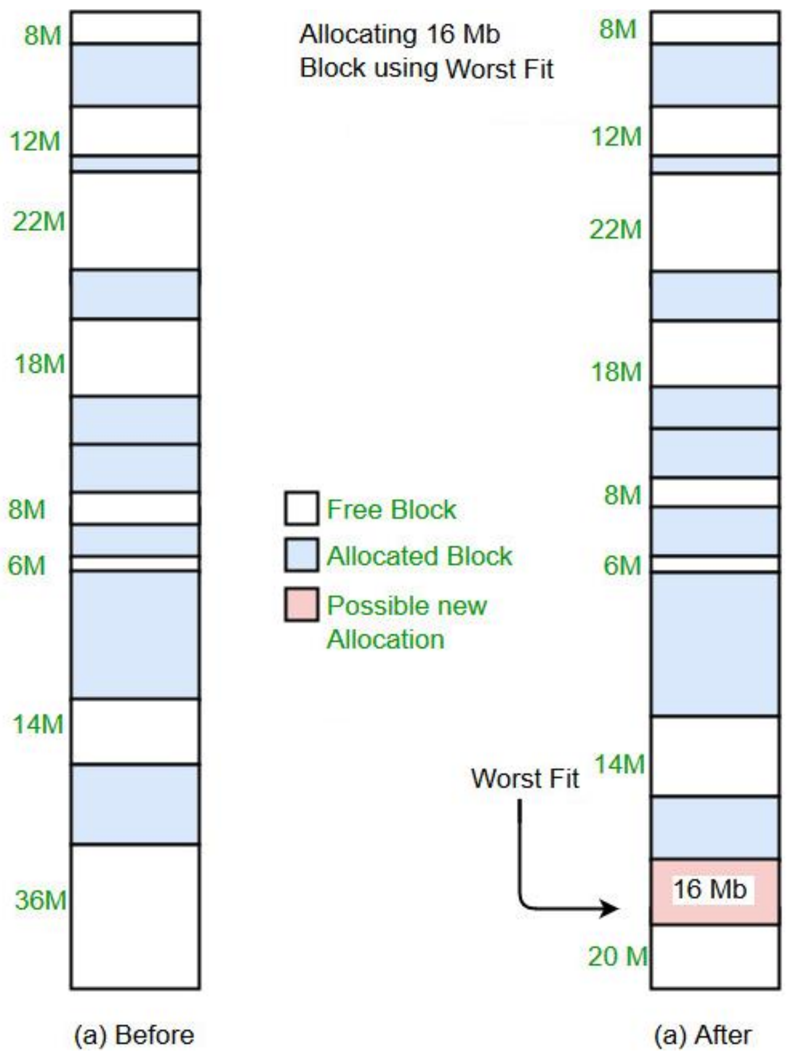
**Figure 12 - file merging process**



As described in the algorithm, We first initialise the set of all input files(Fs), the criteria for merging the files (Ms) and the merging queue (Qm). Next, we iterate for every input file of our set. If the file is a large file, we put it into the Object Storage. Else, if the file is indeed a small file, then we merge it into the Merge Queue, we put it into the queue having the maximum possible empty space. After merging the file into the merge queue, its queue size is incremented by the file size, and the merging limit is decremented by the queue size. Whenever, for a queue, the queue size becomes equal to the merge size limit, the queue is converted into a sequence file. And the sequence files are put in the Object Storage for processing. This algorithm makes use of the Worst-fit strategy as illustrated in Figure 13 below to reduce “internal fragmentation” in storing of files. As the hole (empty space) which is left, for the virtual block size, after inserting a file into a queue is the maximum, hence the chances of accommodating any new file into the left over space are maximum. Thereby wastage of space is minimised[31].

**Figure 13 - Worst-fit algorithm**[32]

To save a file of 16MB, it will be allocated largest block whose size is more than 16MB, in this case to the 36M block. Worst-fit, which always allocates the largest block on the list hoping that the remainder of the block will be useful for servicing a future request



## 4.1 Prototype

We work locally as this is a prototype.

We ran an experiment on 500 CSV files, with the number of rows ranging from 10 to 10,000. We ran it once on Google Colab and another time locally using the code from this repository. Both runs are timed, and the results clearly show that our merging method reduces the MapReduce runtime.

**MapReduce runtime results:**

Terminal local Run:

- The MapReduce took 108.875 seconds on the merged and large files
- The MapReduce took 145.676 seconds on the small files

Google Colab Run:

- The MapReduce took 261.533 seconds on the merged and large files
- The MapReduce took 301.918 seconds on the small files

**Here is a link to the project on Github:**

https://github.com/avivples/big_data_platforms_final

A jupyter notebook has also been added to the repository as a report of the code. The current checkpoint of the notebook was run on Google Colab.

All files created are placed in the directory output. The code creates csv files, merges them, then runs MapReduce on the large and merged files, and afterwards on the small files.

## 5 Next Steps

The prototype built is a basic demonstration of a solution that can be done in a more complex way. This way we can achieve better results by creating MapFiles that are split in the most efficient way considering the virtual block size. We can do this by considering the most appropriate merge algorithm, i.e. consider the variation of the file sizes when merging. We should also consider parallelization during the merge step in order to achieve a more efficient flow to this additional step that is done before running the MapReduce jobs.

In addition, there are many different types of data, meaning that we are required to take that into consideration when we have a large number of each type and how to merge them. Hence, it is needed to consider how to create a robust solution to many different types of data, like images, audio, etc.

We can also consider additional important points. We'll want to have Fault Tolerance, so it can tolerate machine failures gracefully. This can be done by having the master (the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks) ping every worker periodically, and act accordingly by the response received. We'll want locality since network bandwidth is a relatively scarce resource and we are working with Object Storage and not with a local HDFS storage within the Hadoop Cluster. Merging the small files already allows fewer requests from the Object Storage, however, we might want to consider an option of preloading the data into memory of the cluster, to have some sort of locality[33].

## 6 Conclusion

With the development of big data, Hadoop has become the most popular software framework for big data distributed processing at present and has good performance for large file storage and high-throughput data access. However, there are some defects in handling the large number of small files based on Hadoop. The most contemplated drawback is the memory limitation of NameNode. However, lots of small files further lead to the bottleneck performance at client side, low disk utilization at DataNodes, bottleneck over network, processing overheads, over/under-utilization of resources, data fetching overheads and so forth. Furthermore, we covered the difference between HDFS and Object storage and the benefits and drawbacks in using each one of them. We addressed the problem of using MapReduce over small files and showed that The challenge with MapReduce being executed over small files persisted in Object storage is mostly similar when using HDFS in regards to running the compute functions of the MapReduce. In our prototype we suggested Sequence Files and MapFiles as solution candidates by way of merging large numbers of small files into larger files.  Less number of data blocks means increased efficiency of data processing thus solving the drawback of processing a huge number of small files.

In addition, we reviewed numerous solutions to overcome these challenges, most of the solutions strategies are classified into metadata management optimization and data management optimization. The majority attempted to improve the storage efficiency or cluster scalability to overcome the memory limitations but along with improved memory efficiency, an optimal solution should at the same time be able to improve the access efficiency of the files. Also, solutions should offer to improve the system throughput with minimum latencies. Most of the existing solutions only attempt to improve the storage efficiency or cluster scalability to overcome the memory limitations.

Finally, although various optimization strategies are proposed for small files storage on HDFS at present, there is not a systematic and unified solution. Existing solutions have a lot of limitations, and the effect of the same solution may not be the same on different application scenarios. Consequently, there is still a lot of work to be studied, and the optimization scheme of small file storage based on Hadoop will probably be the research focus and hot topic in the future of big data.

# 7 Bibliography

1. https://www.guru99.com/introduction-to-mapreduce.html
2. https://cloudian.com/blog/object-storage-care/
3. https://blog.scaleway.com/understanding-the-different-types-of-storage/
4. https://searchdatamanagement.techtarget.com/definition/Hadoop-Distributed-File-System-HDFS
5. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
6. https://storageswiss.com/2015/01/06/what-is-hadoop-object-storage/
7. https://www.techtarget.com/searchstorage/tip/Three-reasons-why-object-storage-is-a-good-HDFS-alternative
8. https://sujithjay.com/s3-hdfs
9. https://www.integrate.io/blog/storing-apache-hadoop-data-cloud-hdfs-vs-s3/#one
10. https://blogs.oracle.com/bigdata/post/what-is-object-storage
11. https://blog.westerndigital.com/apache-hadoop-object-storage-data-lake/
12. https://www.ibm.com/cloud/blog/the-future-of-object-storage-from-a-data-dump-to-a-data-lake
13. https://www.slideshare.net/gvernik/hadoop-and-object-stores-can-we-do-it-better
14. C. Vorapongkitipun and N. Nupairoj, "Improving performance of small-file accessing in Hadoop," 2014 11th International Joint Conference on Computer Science and Software Engineering (JCSSE), 2014
15. X. Liu, J. Han, Y. Zhong, C. Han and X. He, "Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS, 2009 IEEE International Conference on Cluster Computing and Workshops, 2009
16. https://www.logicalclocks.com/research/size-matters-improving-the-performance-of-small-files-in-hadoop-middleware-2018
17. G. Mackey, S. Sehrish and J. Wang, "Improving metadata management for small files in HDFS," 2009 IEEE International Conference on Cluster Computing and Workshops, 2009
18. P. Phakade, S. Raut. An Innovative Strategy for Improved processing of small files in Hadoop. International Journal of Application or Innovation in Engineering & Management, 2014
19. 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), 2015

20. V. Gopal and S. Kumar Pamu “Reduction of Data at Namenode in HDFS using harballing Technique", International Journal of Advanced Research in Computer Engineering & Technology, vol. 1, no. 4, pp. 635-642, 2012
21. https://www.programmerall.com/article/48191952307/
22. https://towardsdatascience.com/new-in-hadoop-you-should-know-the-various-file-format-in-hadoop-4fcdfa25d42b
23. https://www.researchgate.net/figure/The-way-of-sequencefile-HBase-is-used-to-complete-data-writing-rather-than-write_fig2_335637597
24. K. Bi, D. Han. Scalable Multiple NameNodes Hadoop Cloud Storage System International Journal of Database Theory and Application, 2015
25. B. Dong, “An Optimized Approach for Storing and Accessing Small Files on Cloud Storage”, Journal of Network and Computer Applications, vol. 35, (2012)
26. Cai, X., Chen, C., & Liang, Y. (2018). An optimization strategy of massive small files storage based on HDFS, 2018.
27. B. Meng, W.B. Guo, G.S. Fan, N.W. Qian. A novel approach for efficient accessing of small files in HDFS: TLB-MapFile, 2016 IEEE/ACIS 17th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2016
28. P.M. Eikafrawy, A.M. Sauber, M.M. Hafez HDFSX: Big data distributed file system with small files support 2016 12th International Computer Engineering Conference, ICENCO 2016
29. https://www.pdl.cmu.edu/PDL-FTP/Storage/CMU-PDL-06-104.pdf
30. http://www.tjprc.org/publishpapers/2-14-1381324008-28%20.A%20Perusual.full.pdf
31. S. Sheoran, D. Sethia and H. Saran, "Optimized MapFile Based Storage of Small Files in Hadoop," 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2017, pp. 906-912, doi: 10.1109/CCGRID.2017.83.
32. https://www.geeksforgeeks.org/program-worst-fit-algorithm-memory-management/
33. Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107–113.
34. https://medium.com/@wleggette/object-storage-hadoop-and-the-data-lake-of-the-future-732260e8b71f