CSP554-02: Big Data Technologies

Research Paper

on

Impact of Small files on Hadoop Performance. An analysis of the NameNode(metadata) Storage Capacity and Processing performance.

Supervisor: Professor Jawahar Panchal

Submitted by:

Albert Mandizha (A20493341),

Illinois Institute of Technology, Chicago

Abstract

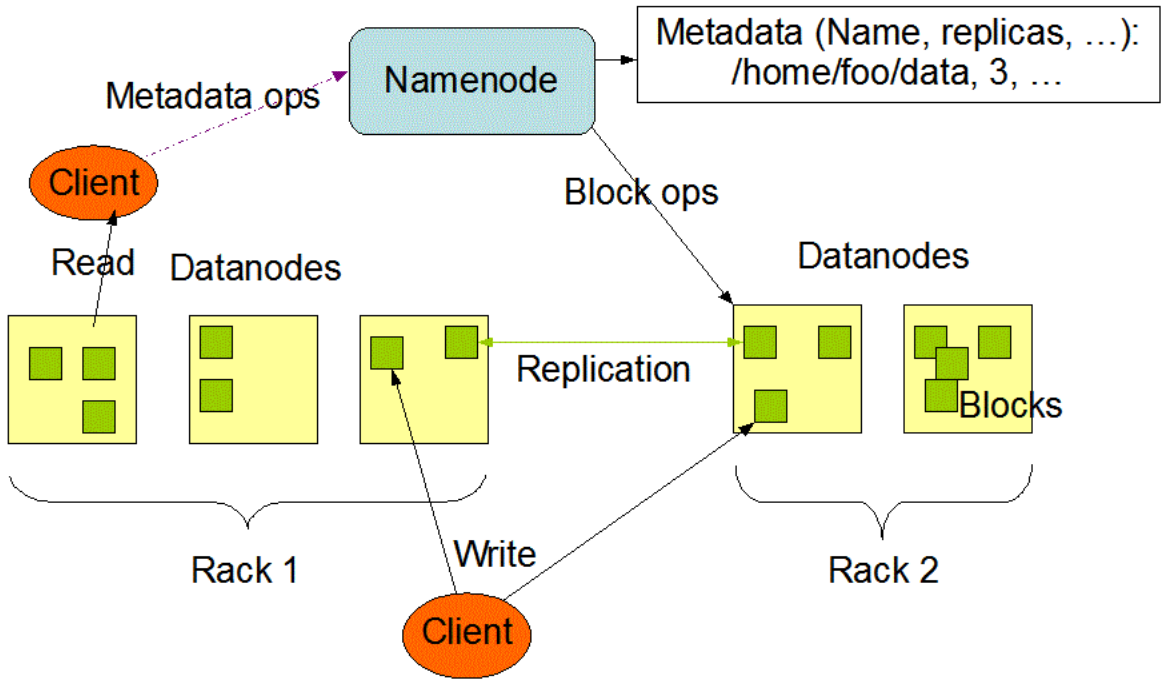
A small file is one which is significantly smaller than the HDFS block size (default 64MB/128MB). If were storing small files, then we probably have lots of blocks, otherwise we wouldn’t turn to Hadoop, and the problem is that HDFS can’t handle lots of files. As a general rule, each file, directory, and block in HDFS is represented as an object in the namenode's memory, which typically takes up 150 bytes. Thus, if 10 million files were used, each requiring a block, 3 gigabytes of memory would be required. With existing gear, scaling up considerably beyond this point is problematic. A billion files are definitely not doable. The paper focused on the two metrics of the Hadoop namenode fsimage loading time and process time to ingest data into the datanode. Four different number of files were loaded on four single node clusters. The files were of different sizes with the first case having 1067 1MB files and case 2 had 267 4MB files, Case 3 had 7 154MB sized files and Case 4 had 1Gig 1 file. All the cases were of the same size 1Gig. The experiments established a positive relationship between the number of files and the time taken by the namenode fsimage to load when a Hadoop cluster is started. The experiment also established it was faster to ingest a single 1 gig file as compared to 1gig of 1000 1MB files. The paper went on to discuss possible sources of small files and established the sequencing file approach to deal with small files in Hadoop. For future research the paper noted the need to use a multi-cluster and also to observe live streaming data processing and measure other metrics that affect the namenode when dealing with small files.

Table of Contents

1.0 Introduction

Small files are a common challenge in the Apache Hadoop world and when not handled with care, they can lead to a number of complications. The Apache Hadoop Distributed File System (HDFS) was developed to store and process large data sets over the range of terabytes and petabytes. However, HDFS stores small files inefficiently, leading to inefficient Namenode memory utilization and RPC calls, block scanning throughput degradation, and reduced application layer performance. The HDFS consists mainly of several DataNodes which are used for storing the data and one NameNode which is used for organizing client access to the DataNodes and manages the metadata of the stored files, Figure 1 represents the HDFS architecture [1-6].

Figure 1: The HDFS Architecture



Adopted from *Menoufia Journal of Electronic Engineering Research*

What are Small Files?

A small file is one which is significantly smaller than the default Apache Hadoop HDFS default block size (128MB by default in CDH). One should note that it is expected and inevitable to have some small files on HDFS. These are files like library jars, XML configuration files, temporary staging files, and so on. But when small files become a significant part of datasets, the problems arise. Hence, in this section, we shall

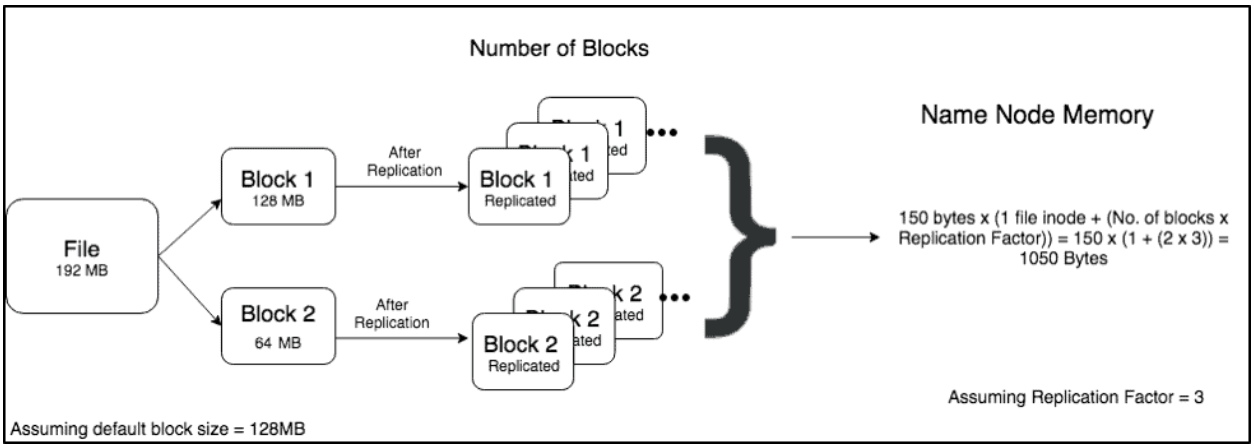
discuss why it is a good goal to have a file size as close to a multiple of the HDFS block size as possible (Sachin Bende 2016).

Hadoop's storage and application layers are not designed to function efficiently with a large number of small files. Before we get to the implications of this, let's review how HDFS stores files.

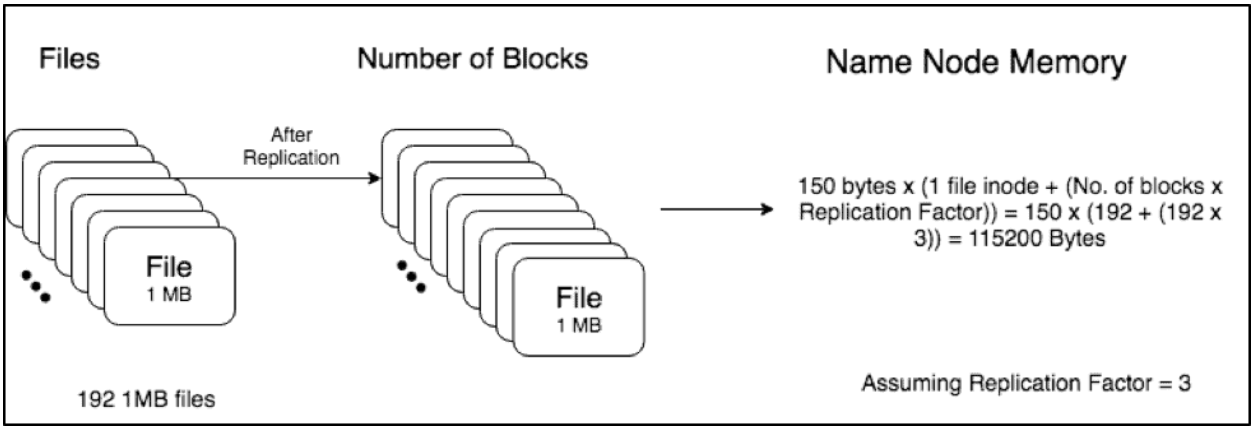
In HDFS, data and metadata are separate entities. Files are split into blocks that are stored and replicated on the DataNodes' local file systems across the cluster. The HDFS namespace tree and associated metadata are maintained as objects in the NameNode's memory (and backed up to disk), each of which occupies approximately 150 bytes, as a rule of thumb. This arrangement is described in more detail in the public documentation here.

The two scenarios below illustrate the small files issue:

Scenario 1 (1 large file of 192MiB):



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



Scenario 1 has one file which is 192MB which is broken down to 2 blocks of size 128MB and 64MB. After replication, the total memory required to store the metadata of a file is = 150 bytes' x (1 file inode + (No. of blocks x Replication Factor)) (Sachin Bende 2016).

According to this calculation, the total memory required to store the metadata of this file on the Namenode = 150 x (1 + (2 x 3)) = 1050 Bytes.

In contrast, scenario 2 has 192 1 MB files. These files are then replicated across the cluster. The total memory required by the Namenode to store the metadata of these files = 150 x (192 + (192 x 3)) = 115200 Bytes.

Hence, we can see that we require more than 100x memory on the Namenode heap to store the multiple small files as opposed to one big 192MB file.

Case 1- Storage

The NameNode is responsible for loading the file system metadata from local disk into memory (RAM). If the NameNode is large, this will affect the starting time it will take long to start. The NameNode must also track changes in the block locations on the cluster. Too many small files can also cause the NameNode to run out of metadata space in memory before the DataNodes run out of data space on disk. The DataNodes also report block changes to the NameNode over the network, more blocks mean more changes to report over the network.

The above scenario 1 and 2 illustrates the effect of many small files on the NameNode.

Case 2- Processing

In general, having a large number of small files results in more disk seeks while running computations through an analytical SQL engine like Impala or an application framework like MapReduce or Spark.

MapReduce /Spark

In Hadoop, a block is the most granular unit of data on which computation can be performed. Thus, it affects the throughput of an application. In MapReduce, an individual Map task is spawned for each block that must be read. Hence, a block with very little data can degrade performance, increase Application Master bookkeeping, task scheduling, and task creation overhead since each task requires its own JVM process.

This concept is similar for Spark, in which each "map" equivalent task within an executor reads and processes one partition at a time. Each partition is one HDFS block by default. Hence, a single concurrent task can run for every partition in a Spark RDD. This means that if you have a lot of small files, each file

is read in a different partition and this will cause a substantial task scheduling overhead compounded by lower throughput per CPU core.

2.0 Literature Review

How do small files "impact" cluster performance?

Everything is a trade-off when dealing with data at scale. The impact of small files, beyond the Namenode pressures, is more specifically related to "job" performance. Under classic MR, the number of small files controls the number of mappers required to perform a job. Of course, there are tricks to "combine" inputs and reduce this, but that leads to a lot of data back planning and increased cluster I/O chatter. A mapper in the classic sense, is a costly resource to allocate. If the actual task done by the mapper is rather mundane, most of the time spent accomplishing your job can be "administrative" in nature with the construction and management of all those resources (Bakul Panchal 2014).

Consider the impact to a cluster when this happens. For example, a client once trying to get more from their cluster but there is a job that is processing 80,000 files. Which lead to the creation of 80,000 mappers. Which lead to consuming ALL the cluster resources, several times over. Following that path, a bit further and we will find that the impact on the Namenode is exacerbated with all of the intermediate files generated by the mapper for the shuffle/sort phases. That's the real impact on a cluster. A little work in the beginning can have a dramatic effect on the downstream performance of our jobs. We have to take the time to "refine" our data and consolidate our files.

3.0 Experimental Setup

In this section we are going to assess two main components of the NameNode on the Hadoop infrastructure for the purpose of ascertaining the runtime to start the NameNode and the used storage space of the NameNode we will load four different sized files in our Hadoop folder on our Hadoop file system.

3.1 Setting Up Hadoop File System on Windows

The Hadoop Distributed file system will be set on four different virtual Machines with one NameNodes and one DataNode, the experiment will focus on the storage and uptime of the NameNode in a given

environment with different number of files and byte size for the NameNodes. Four NameNodes will be set each of the same Size. The different name Nodes will store metadata of the same file size but different number of files. For instance, the table below illustrates loading different number of files of the same size into hdfs.

Table 3.1 Assumed file configuration and their sizes

Case	HDFS Block Size	Number of Files	File Size	Total Size
Case 1	128mb	1	1024mb	1024mb
Case 2	128mb	8	128mb	1024mb
Case 3	128mb	256	4mb	1024mb
Case 4	128mb	1024	1mb	1024mb

Source: *Authors Own Computation*

The Cases 1 to 4 were run on the same Hadoop structure with one NameNode and one DataNode the DataNode will be of 1024 storage capacity. The experiment was carried on using Hadoop on Linux virtual machine to establish the latency and throughput for the NameNode to load the data into the NameNode.

3.2 Sequence Files

The paper will use the filename as key and content as value. We will write a program to put lots of small files in a single Sequence File. They are split table and so MapReduce can operate each chunk independently and in parallel.

Hadoop files are immutable and cannot be appended to, so we need to ingest a large number of the small files at a time in order to make the sequence file works well. In addition, Hive software does not work well with sequence files structure (EL-SAYED, Badawy and EL-SAYED 2019).

4.0 Results Interpretation and Analysis

Our results will be discussed in two forms first case storage and second case the processing of the NameNode.

4.1 Case 1 Storage (NameNode Memory)

4.1.1 Presentation of findings

When a NameNode starts up, it reads HDFS state from an image file, fsimage, and then applies edits from the edits log file. It then writes new HDFS state to the fsimage and starts normal operation with an empty edits file.

FsImage is a file stored on the OS file system that contains the complete directory structure (namespace) of the HDFS with details about the location of the data on the Data Blocks and which blocks are stored on which node.

The experiment was carried out to establish the heap memory used

Table 4.2 Results from experimental test on namenode memory.

Experiment Case:	File Size in kilobytes	No. of Files and directories	Replication factor (single cluster=1)	Number of Blocks created.	Total file system object	Heap Memory in namenode.
Case 1	1014	1126	1	1122	2248	96.55MB
Case 2	4449	261	1	257	518	65.06MB
Case 3	154290	11	1	13	24	54.17MB
Case 4	1082000	5	1	9	14	37.82MB

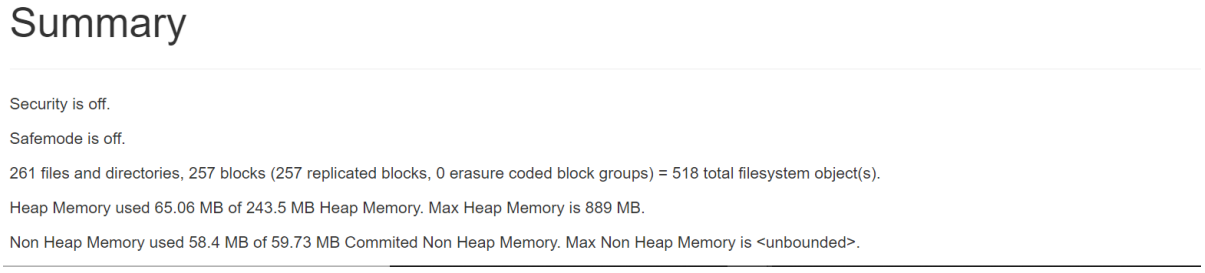
Source *Authors Own Computation*

The number of files include directories in the table above and a replication factor of 1 this is because of a single node cluster with one namenode and one datanode.

4.1.2 Snapshots from experiment

Case 1 (1MB)

Case 2 (4MB)



Case 3 (154MB)

Summary

Security is off.

Safemode is off.

11 files and directories, 13 blocks (13 replicated blocks, 0 erasure coded block groups) = 24 total filesystem object(s).

Heap Memory used 54.17 MB of 211.5 MB Heap Memory. Max Heap Memory is 889 MB.

Non Heap Memory used 55.04 MB of 56.38 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Case 4 (1024MB)

Summary

Security is off.

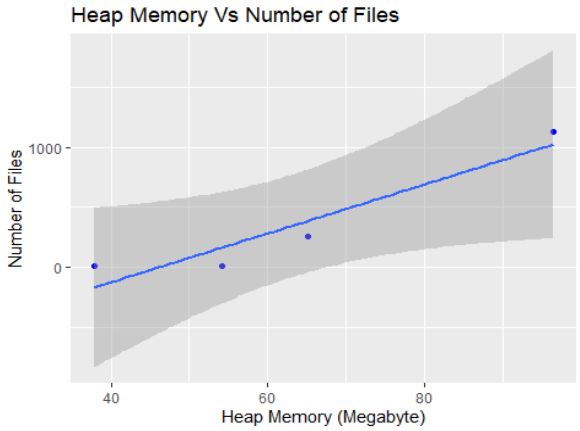
Safemode is off.

5 files and directories, 9 blocks (9 replicated blocks, 0 erasure coded block groups) = 14 total filesystem object(s).

Heap Memory used 37.82 MB of 201.5 MB Heap Memory. Max Heap Memory is 889 MB.

Non Heap Memory used 52.3 MB of 53.59 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

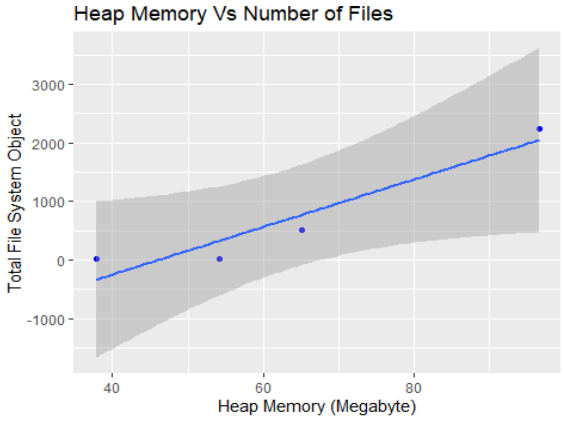
4.1.3 Discussion of Results



Source *Authors Own Computation*

Heap Memory Vs Number of Files

When a Namenode restarts, it must load the file system metadata from local disk into memory. This means that if the namenode metadata is large, restarts will be slower. The Namenode must also track changes in the block locations on the cluster. Too many small files can also cause the Namenode to run out of metadata space in memory before the Datanodes run out of data space on disk. The datanodes also report block changes to the Namenode over the network; more blocks mean more changes to report over the network.



Source *Authors Own Computation*

Total File System Vs Heap Memory

More files mean more read requests that need to be served by the Namenode, which may end up clogging Namenode's capacity to do so. This will increase the RPC queue and processing latency, which will then lead to degraded performance and responsiveness. It can be noted that the more files are run the greater the heap memory required.

4.2 Case 2 Processing (Hadoop Uptime Test)

4.2.1 Presentation of findings

The project Measured four files of the same size but different number of files with each case unique on each form. The table below shows the results in milliseconds taken by the Fsimage to start uptimes on Cases.

Table 1.3 Results from uptime experimental test.

Experiment Case:	File Size in kilobytes	Number of Files	Total File Size loaded (Kilobytes)	Hadoop Uptime(NameNode, Fsimage) milliseconds	Average time on each case
Case 1	1014	1067	1081938	2041	530kb/s
Case 2	4449	243	1081107	1399	772.7k/s
Case 3	154290	7	1080030	941	1147.7k/s
Case 4	1082000	1	1082000	478	2263.59k/s

Source *Authors Own Computation*

4.2.2 Snapshots from experiment

Case 1 (1mb File)

```
22:18:08,292 INFO namenode.FSEditLog: Starting log segment at 8449
22:18:08,521 INFO namenode.NameCache: initialized with 0 entries 0 lookups
22:18:08,526 INFO namenode.FSNamesystem: Finished loading FSImage in 2041 msecs
22:18:09,348 INFO namenode.NameNode: RPC server is binding to localhost:9000
22:18:09,348 INFO namenode.NameNode: Enable NameNode state context:false
22:18:09,388 INFO ipc.CallQueueManager: Using callQueue: class java.util.concurrent.LinkedBlockingQueue, queu
1000, scheduler: class org.apache.hadoop.ipc.DefaultRpcScheduler, ipcBackoff: false.
```

Case 2 (4mb files)

```
2022-12-02 17:07:57,568 INFO namenode.FSEditLog: Starting log segment at 1580
2022-12-02 17:07:57,725 INFO namenode.NameCache: initialized with 0 entries 0 lookups
2022-12-02 17:07:57,726 INFO namenode.FSNamesystem: Finished loading FSImage in 1399 msecs
2022-12-02 17:07:58,323 INFO namenode.NameNode: RPC server is binding to localhost:9000
2022-12-02 17:07:58,324 INFO namenode.NameNode: Enable NameNode state context:false
2022-12-02 17:07:58,364 INFO ipc.CallQueueManager: Using callQueue: class java.util.concurrent.LinkedBlockingQueue, queu
```

Case 3 (154mb files)

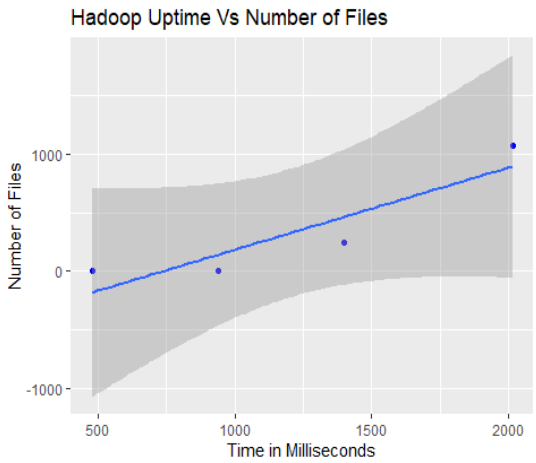
```
2022-12-02 21:37:26,148 INFO namenode.FSEditLog: Starting log segment at 1713
2022-12-02 21:37:26,354 INFO namenode.NameCache: initialized with 0 entries 0 lookups
2022-12-02 21:37:26,355 INFO namenode.FSNamesystem: Finished loading FSImage in 941 msecs
2022-12-02 21:37:27,033 INFO namenode.NameNode: RPC server is binding to localhost:9000
2022-12-02 21:37:27,035 INFO namenode.NameNode: Enable NameNode state context:false
```

Case 4 (1082mb File)

```
2022-12-02 20:36:16,169 INFO namenode.FSEditLog: Starting log segment at 1644
2022-12-02 20:36:16,266 INFO namenode.NameCache: initialized with 0 entries 0 lookups
2022-12-02 20:36:16,266 INFO namenode.FSNamesystem: Finished loading FSImage in 478 msecs
2022-12-02 20:36:16,538 INFO namenode.NameNode: RPC server is binding to localhost:9000
2022-12-02 20:36:16,538 INFO namenode.NameNode: Enable NameNode state context:false
2022-12-02 20:36:16,551 INFO ipc.CallQueueManager: Using callQueue: class java.util.concurrent.LinkedBlockingQueue, queu
eCapacity: 1000, scheduler: class org.apache.hadoop.ipc.DefaultRpcScheduler, ipcBackoff: false.
```

4.2.3 Discussion of Results

Hadoop Uptime Vs Number of Files



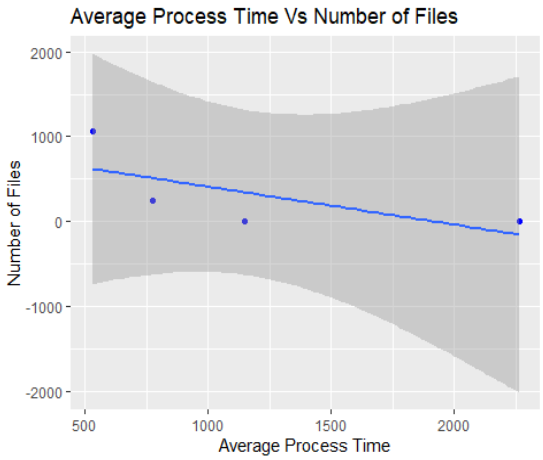
Source *Authors Own Computation*

Hadoop Uptime Vs Number of files

From the graph above it can be observed that as the number of small files increases time that is taken to load the Fsimage increases as well over time. This shows how small files affect the performance of our cluster in terms of the uptime of our Namenode. A good scenario case is that of 1mb files which shows more fsimage time processing.

Average time on each Case

This explains the average number of kilobytes that are loaded per second when starting the namenode by the fsimage. In Table 1.3 it can be observed that the more number of files the namenode has to load the decrease in the processing power or the rate at which the fsimage is loaded. The graph below illustrates this concept.



Source *Authors Own Computation*

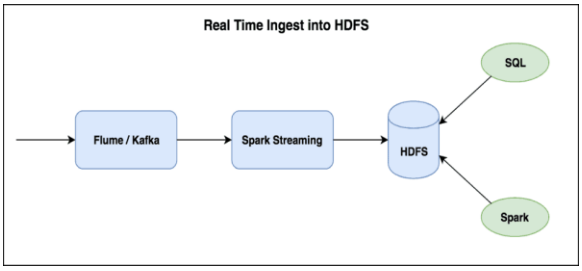
From the graph above it can be noted that the more number of small files we have in our cluster the slower it is the processing time like the smaller files the fsimage loads it takes more time to complete the task of the same file magnitude hence Hadoop is not compactable with small files rather it is with large files. To resolve these two issues of slow processing and Namenode overload we will explore the different ways to deal with small files.

4.3 Dealing with small files issues.

To deal with small files we need to establish how do small files originate. This section discussed some of the common mistakes that give birth to insidious small files.

4.3.1 Streaming Data

Data ingested incrementally and in small batches can end up creating a large number of small files over a period of time. Near real time requirements for streaming data, with small windows (every few minutes or hours) that do not create much data will cause this problem. Below is a typical streaming ETL ingest pipeline into HDFS.



Adopted from *Menoufia Journal of Electronic Engineering Research*

4.3.2 Large Number of Mappers/Reducers

MapReduce jobs and Hive queries with large number of mappers or reducers can generate a number of files on HDFS proportional to the number of mappers (for Map-Only jobs) or reducers (for MapReduce jobs). Large number of reducers with not enough data being written to HDFS will dilute the result set to files that are small, because each reducer writes one file. Along the same lines, data skew can have a similar effect in which most of the data is routed to one or a few reducers, leaving the other reducers with little data to write, resulting in small files.

4.3.3 File Formats and Compression

Using of inefficient file formats, for example TextFile format and storing data without compression compounds the small file issue, affecting performance and scalability in different ways:

- Reading data from very wide tables stored as non-columnar formats (TextFile, Sequence File, Avro) requires that each record be completely read from disk, even if only a few columns are required. Columnar formats, like Parquet, allow the reading of only the required columns from disk, which can significantly improve performance
- Use of inefficient file formats, especially uncompressed ones, increases the HDFS space usage and the number of blocks that need to be tracked by the NameNode. If the files are small in size, it means the data is split into a larger number of files thereby increasing the amount of associated metadata to be stored.

4.3 Proposed Solution to Small files in this Paper

Sequence File: Here we used the filename as key and content as value. We wrote a program to put lots of small files in a single Sequence File. They are splittable and so MapReduce can operate each chunk independently and in parallel.

Going to our the 1000 1MB files, we wrote a program to put them into a single Sequence File, and processed them in a streaming fashion operating on the Sequence File.

Table 5.1 Results after sequencing of files.

Experiment Case:	File Size in kilobytes	No. of Files and directories	Replication factor (single cluster=1)	Number of Blocks created.	Total file system object	Heap Memory in namenode.
Case 1	1.08	1	1	8	9	16MB

It was slow to convert existing data into Sequence Files. However, it is perfectly possible to create a collection of Sequence Files in parallel. Going forward it's best to design our data pipeline to write the data at source direct into a Sequence File, if possible, rather than writing to small files as an intermediate step.

5.0 Conclusion and Recommendation

5.1 Conclusion

The paper discussed the small file issue in Hadoop distributed file system. The two main metrics of the Namenode Hadoop file systems were explored, the uptime of the FS image and the metadata storage (heap memory) that is required to process same sized files of different number of file sizes. It was established that the namenode required more time to load the fsimage when the datanode is stored with many small files and also the more the small files the slower the loading of the files into the storage disk the datanode. The paper went on to discuss the sources of the small files issues noting the streaming data, large number of map reducers and file formats and compression. The paper covered sequencing of files as a way of dealing with small files in Hadoop distributed files.

5.2 Recommendation

The project was performed on a single node cluster, a more robust analysis on the Namenode to establish the effect of small files will be to use multi-node clusters in a Hadoop architecture.

The clusters did not involve live streaming of data into the Datanodes hence l recommend an analysis with huge cluster nodes that have live streaming data to observe the data ingestion process.

Only two metrics of the namenode were considered, in the future a study on other namenode metrics can be carried to establish the relationship between the small files and the Hadoop files system.

Acknowledgements

I would like to express my deep gratitude to Professor Jawahar Panchal for his valuable and constructive suggestions during the planning and development of this research work. His willingness to give his time generously has been very much appreciated.

Appendix

The paper zip file contains the codes, data and experiment snapshots of all the experiments done on this project.

References

Ansari, Shaheer, and Afida Ayob 1. 2021. "Multi-Channel Profile Based Artificial Neural Network Approach for Remaining Useful Life Prediction of Electric Vehicle Lithium-Ion Batteries." *Special Issue Control and Management of Electric Power System in Vehicles)* 14-34.

Bakul Panchal, Parth Gohil. 2014. "Efficient Ways to Improve the Performance of HDFS for Small Files." *Computer Engineering and Intelligent Systems* 205-345.

EL-SAYED, Tharwat, M. Badawy, and Ayman EL-SAYED. 2019. "Impact of Small Files on Hadoop Performance: Literature Survey and Open Points." *Menoufia Journal of Electronic Engineering Research* · 109-120.

Sachin Bende, Rajashree Shedge,. 2016. "Dealing with Small Files Problem in Hadoop Distributed File System." *Procedia Computer Science* 1001-1012.