|  |
| --- |
| New York Institute of Technology |
| Hadoop and map reduce performance analysis |
| Tapan Parikh  Yash Modi  Deep Desai  Computer Network |

|  |
| --- |
|  |

1. **Abstract**

Hadoop is one of the free java based programming framework, which will help to processing large amount of data set from distributed computing systems. Hadoop provides distributed file system called HDFS (Hadoop Distributed File Systems), which stores data across the nodes of the cluster. Mainly Hadoop provides parallel paradigm implementation Map Reduce and is widely used for parallel computation with low response time. Hadoop is used to process extremely large amount of data without bothering of the structure of data. Large data, which means petabytes of Data. The main objective of the project is to measure the performance of the cluster system using Hadoop and implement MapReduce Algorithm.

To measure the performance, we have set up Hadoop cluster with single node or multiple nodes. For measurement, we have used TestDFSIO of hadoop 2.7. Performance parameter calculated will be Throughput, Average I/O rate and Execution time on different environment. HDFS performance will be calculated by writing and reading with small and large data sets. We will implement the MapReduce algorithm to extract data efficiently and fast.

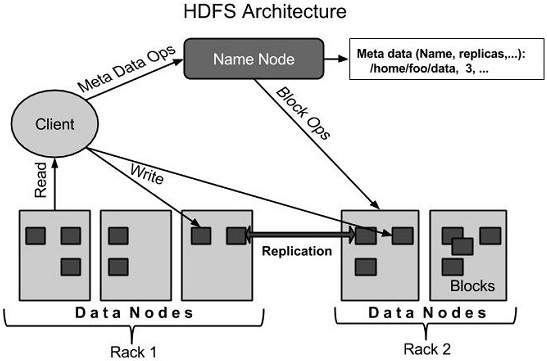
1. **Introduction**

In recent years, data is increasing tremendously in size. Therefore, it is necessary to manipulate data effectively and efficiently. Alongside that, the retrieval of data from such large dataset was also one concerned for some of the big companies such as Facebook, LinkedIn, and Yahoo.[1]

Hadoop has been introduced as an open-source and java-based programming framework that supports the processing of large data sets in a distributed computing environment. Google File System (GFS) papers inspired it. [1] Apache developed Hadoop is now a top level Apache project, being built and used by a community of contributors from all over the world, since it’s easy to install, configure and can be run on many platforms supporting Java. Major players including Google, Yahoo and IBM, largely for applications involving search engines and advertising currently use the Hadoop framework. [2]

The significant commitments of this work are a Hadoop execution assessment on composing and full comprehension about the MapReduce idea and additionally the dispersion of procedures.  
  
Section 3 indicates the HDFS and MapReduce concept with examples. Section 4 depicts the establishment and arrangement of a HDFS single hub bunch. Section 5 describes the test preparation towards our cluster environment. Section 6 has performance measurements based on many test cases and a performance comparison between the HDFS and local file system on the testing cluster.

1. **HDFS Model**
   1. **HDFS Architecture**



**Figure 1: HDFS Architecture Retrieved from http://www.tutorialspoint.com/hadoop/hadoop\_hdfs\_overview.htm**

HDFS works on master-slave architecture. HDFS architecture consists of many components such as NameNode, DataNode, JobTracker and TaskTracker is used several clients.

**Client**

Client is a software, which communicate with the name node.

**NameNode**

NameNode is consider as master server, which contains LINUX operating system. It performs the following task:

* Managing files.
* Client Interaction.
* File operation such as renaming, opening and closing.

**DataNode**

DataNode is consider as slave server in the system. They are managed by anyone of system NameNode. It performs following task

* Read and write operation of file as per client requirement.
* Managing the blocks.

**JobTracker**

Job tracker is the master process, which runs on separate node. It performs the following task.

* Client job is handled.
* It determines location of the data.
* Task allocation to the chosen Task tracker nodes.
* Task tracker is monitored by namenode. Task tracker notifies the name node with results of the job assigned.
* It provides the information to the client.

Job Tracker is only the point of failure in whole hadoop system.[7]

**Task Tracker**

Task tracker is slave process, which accepts the task from the Job tracker. Task tracker runs on separate JVM because during the failure it does not affect the system. It calculates and captures the output. It notifies the job tracker regarding its status and result of the executed task.

**MapReduce**

MapReduce is programming model, which is divided in map and reduce programs. It is implemented according to the user’s requirement.

**Map**

It takes the data from the source generate the intermediate key/value pair. There may be possibility that more one map function are used before passing it to the reduce function.

**Reduce**

It takes the data in form of key/value pair. It works is to merge the data into smaller set of values.

**Example**

In this project, we have implemented the example of creating dictionary, which contains English to other language translation. At last, the result file contains the English word followed by all translation pipe-separated. Below is the code snippet.

Mapper Class [6]

**public** **void** map(Text key, Text value, Context context) **throws** IOException,InterruptedException{

StringTokenizer itr = **new** StringTokenizer(value.toString(), ",");

**while** (itr.hasMoreTokens())

{

word.set(itr.nextToken());

context.write(key, word);

}

}

Above is the Mapper class which reads the every line of the inputfile where it is tokenize and converted to key/value pair.

Reducer Class [6]

**protected** **void** reduce(Text key, Iterable<Text> values, Context context) **throws** IOException, InterruptedException {

String translations = "";

**for** (Text val : values) {

translations += "|" + val.toString();

}

result.set(translations);

context.write(key, result);

}

Reducer class takes value from the Mapper merge the values to the corresponding keys and generates the final dataset.

Main Class [6]

**public** **static** **void** main(String[] args) **throws** Exception {

Configuration configuration = **new** Configuration();

Job configJob = **new** ~~Job~~(configuration, "dictionary");

configJob.setJarByClass(Dictionary.**class**);

configJob.setMapperClass(WordMapper.**class**);

configJob.setCombinerClass(AllTranslationsReducer.**class**);

configJob.setReducerClass(AllTranslationsReducer.**class**);

configJob.setOutputKeyClass(Text.**class**);

configJob.setOutputValueClass(Text.**class**);

configJob.setMapOutputKeyClass(Text.**class**);

configJob.setMapOutputValueClass(Text.**class**);

configJob.setInputFormatClass(KeyValueTextInputFormat.**class**);

configJob.setOutputFormatClass(TextOutputFormat.**class**);

configJob.setProfileEnabled(**true**);

FileInputFormat.*addInputPath*(configJob, **new** Path(args[0]));

FileOutputFormat.*setOutputPath*(configJob, **new** Path(args[1]));

**boolean** result = configJob.waitForCompletion(**true**);

System.*exit*(result ? 0 : 1);

}

It is main class, which contains all the configuration parameter required to run the job.  
  
**Result:  
  
**

1. **Installation**

**Supported Platforms**

Hadoops underpins GNU/Linux as an improvement and creation stage and just backings Win32 as an advancement stage.

**Required Softwares Information:**

• Java VM 1.5.x or newer versions (We used the version 1.7.0 for our test).

• ssh and sshd

• Windows users may install cygwin to have a linux environment for shell support (We used the Oracle Virtual Machine for the Linux Environment).

**For Single Node:**

We can download Hadoop from its official site: [hadoop.apache.org](http://hadoop.apache.org/). We need to extract downloaded version of following:

tar -xvf hadoop-\*.tar.gz

We have used Hadoop-2.7.0 and it is the Hadoop folder’s name too, we called it Hadoop for naming convention. All the configuration files are located in the folder hadoop/conf/.hadoop/conf/hadoop-env.sh

We need to change the environment variables like e.g JAVA\_HOME, HADOOP\_CLASSPATH.

The only thing that we have to take care of is to change the JAVA\_HOME for Hadoop.

We just need to put the JDK path in that. Now, we need to change the default properties from hadoop/conf/hadoop-default.xml.

We needed to change couple of configurations like namenode and JobTracker.

For Example:

<?xmlversion=”1.0”?>  
<?xml-stylesheet type=”text/xsl” href=”configuration.xsl”?>  
<!– Put site-specific property overrides in this file. –>  
<configuration>  
<property>  
  
<name>hadoop.tmp.dir</name>  
<value>/tmp/hadoop-${[user.name](http://user.name/)}</value>  
  
</property>  
  
<property>  
<name>[fs.default.name](http://fs.default.name/)</name>  
<value>hdfs://localhost:10000</value>  
</property>  
  
<property>  
<name>mapred.job.tracker</name>  
<value>localhost:10001</value>  
</property>  
  
<property>  
<name>dfs.replication</name>  
<value>1</value>  
</property>  
  
</configuration>

1. **Test Preparation**

In this chapter, we have explained the writing reading performance of the single node cluster using hadoop benchmark class TestDFSIO.java. At the end of the test, we will come up with three important parameter Throughput, Average I/O rate and Execution time. [5] We have carried out the test by setting different parameter such blockSize, replication factor and nrFiles.

**5.1 Run Command**

The syntax to run the test using TestDFSIO.

$ hadoop jar $HADOOP\_HOME/hadoop-\*test\*.jar TestDFSIO -read | -write | -clean [-nrFiles N] [-fileSize MB] [-resFile resultFileName] [-bufferSize Bytes]

* **Clean**

This parameter is used to delete all previous result from the directory.

* **nrFiles**

This parameter is used to create the control files. According to this parameter map task are decided.

* **fileSize**

This parameter is used to define the size of output/input file based on operation such read or write.

* **resFile**

This parameter is used to define the test result output file. It is optional. By default it created in /benchmark/testdfsio in HDFS.

* **Run the write test**

$ hadoop jar hadoop-\*test\*.jar TestDFSIO -write -nrFiles 2 -fileSize 500

* **Run the read test after write test**

$ hadoop jar hadoop-\*test\*.jar TestDFSIO -read -nrFiles 2 -fileSize 500

* **Run the clean test**

$ hadoop jar hadoop-\*test\*.jar TestDFSIO -read -nrFiles 2 -fileSize 500

* **Result**

----- TestDFSIO ----- : write

Date & time: Fri Dec 04 04:13:55 EST 2015

Number of files: 1

Total MBytes processed: 1200.0

Throughput mb/sec: 88.61974743371981

Average IO rate mb/sec: 88.6197509765625

IO rate std deviation: 0.013640777097642776

Test exec time sec: 18.653

----- TestDFSIO ----- : read

Date & time: Fri Dec 04 04:51:00 EST 2015

Number of files: 1

Total MBytes processed: 1200.0

Throughput mb/sec: 400.6677796327212

Average IO rate mb/sec: 400.66778564453125

IO rate std deviation: 0.050776884918405374

Test exec time sec: 6.417

* **Explore Parameter**

Most notable parameter is *Throughput mb/sec, Average IO rate mb/sec and Test exec time.* All parameter depends on the input fileSize. [5]

***Throughput = Numbers map task \* Total FileSize/Total Time taken*** [5]

***Average IO rate = Total throughput / Total map task* [5]**

From the above formula if we want to find concurrent throughput, we can calculate throughput with the number of map task. Lets’s say if we have throughput X mb/sec throughput and 10 map task the throughput is calculated by 10\*X mb/sec. The number of map task depends upon the **nrfiles** parameter.

**6. Test Results**

**6.1 Cluster Information**

Operating system: Ubuntu (64-bit)

Base Memory: 4096 MB

Processors: 2

Network: Intel PRO/1000 MT Desktop (NAT)

HDD: 40 GB **6.2 Write**

**6.2.1 File size = 1.2 GB, Block size = 64 MB**

nrFilesize = 1, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 88.61 |
| Average I/O rate (mb/second) | 88.61 |
| Execution time (seconds) | 18.65 |

nrFilesize = 1, Replication = 2

|  |  |
| --- | --- |
| Throughput (mb/second) | 70.72 |
| Average I/O rate (mb/second) | 70.72 |
| Execution time (seconds) | 19.373 |

nrFilesize = 3, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 85.50 |
| Average I/O rate (mb/second) | 85.50 |
| Execution time (seconds) | 45.31 |

**6.2.2 File size = 1.2 GB, Block size = 128 MB**

nrFilesize = 1, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 77.13 |
| Average I/O rate (mb/second) | 77.13 |
| Execution time (seconds) | 18.23 |

nrFilesize = 1, Replication = 2

|  |  |
| --- | --- |
| Throughput (mb/second) | 125.091 |
| Average I/O rate (mb/second) | 125.091 |
| Execution time (seconds) | 18.40 |

nrFilesize = 3, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 99.69 |
| Average I/O rate (mb/second) | 99.24 |
| Execution time (seconds) | 40.47 |

**6.2.3 File size = 4 GB, Block size = 128 MB**

nrFilesize = 1, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 97.037 |
| Average I/O rate (mb/second) | 97.037 |
| Execution time (seconds) | 43.64 |

nrFilesize = 1, Replication = 2

|  |  |
| --- | --- |
| Throughput (mb/second) | 98.44 |
| Average I/O rate (mb/second) | 98.44 |
| Execution time (seconds) | 43.45 |

nrFilesize = 3, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 70.30 |
| Average I/O rate (mb/second) | 70.56 |
| Execution time (seconds) | 177.14 |

**6.2.4 File size = 4 GB, Block size = 64 MB**

nrFilesize = 1, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 89.68 |
| Average I/O rate (mb/second) | 89.68 |
| Execution time (seconds) | 48.04 |

nrFilesize = 1, Replication = 2

|  |  |
| --- | --- |
| Throughput (mb/second) | 93.91 |
| Average I/O rate (mb/second) | 93.91 |
| Execution time (seconds) | 45.45 |

nrFilesize = 3, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 81.103 |
| Average I/O rate (mb/second) | 81.49 |
| Execution time (seconds) | 151.15 |

**6.3 Read**

**6.3.1 File size = 1.2 GB, Block size = 64 MB**

nrFilesize = 1, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 400.66 |
| Average I/O rate (mb/second) | 400.66 |
| Execution time (seconds) | 6.417 |

nrFilesize = 1, Replication = 2

|  |  |
| --- | --- |
| Throughput (mb/second) | 378.78 |
| Average I/O rate (mb/second) | 378.78 |
| Execution time (seconds) | 6.34 |

nrFilesize = 3, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 93.399 |
| Average I/O rate (mb/second) | 93.44 |
| Execution time (seconds) | 42.39 |

**6.3.2 File size = 1.2 GB, Block size = 128 MB**

nrFilesize = 1, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 701.34 |
| Average I/O rate (mb/second) | 701.34 |
| Execution time (seconds) | 4.28 |

nrFilesize = 1, Replication = 2

|  |  |
| --- | --- |
| Throughput (mb/second) | 580.55 |
| Average I/O rate (mb/second) | 580.55 |
| Execution time (seconds) | 4.128 |

nrFilesize = 3, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 81.25 |
| Average I/O rate (mb/second) | 81.28 |
| Execution time (seconds) | 47.62 |

**6.3.3 File size = 4 GB, Block size = 128 MB**

nrFilesize = 1, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 82.15 |
| Average I/O rate (mb/second) | 82.15 |
| Execution time (seconds) | 51.57 |

nrFilesize = 1, Replication = 2

|  |  |
| --- | --- |
| Throughput (mb/second) | 78.44 |
| Average I/O rate (mb/second) | 78.44 |
| Execution time (seconds) | 53.63 |

nrFilesize = 3, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 80.67 |
| Average I/O rate (mb/second) | 81.02 |
| Execution time (seconds) | 151.80 |

**6.3.4 File size = 4 GB, Block size = 64 MB**

nrFilesize = 1, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 81.12 |
| Average I/O rate (mb/second) | 81.12 |
| Execution time (seconds) | 52.46 |

nrFilesize = 1, Replication = 2

|  |  |
| --- | --- |
| Throughput (mb/second) | 81.58 |
| Average I/O rate (mb/second) | 81.58 |
| Execution time (seconds) | 51.65 |

nrFilesize = 3, Replication = 1

|  |  |
| --- | --- |
| Throughput (mb/second) | 79.45 |
| Average I/O rate (mb/second) | 79.56 |
| Execution time (seconds) |  |

1. **Conclusion**

From this project we can conclude that Hadoop is used for distributed file system. This article is shows the technique to work with hadoop. For performance of the cluster we can conclude that cluster performance depends on the block Size and replication parameter. When the replication=1 and Block Size=64MB the Throughput=88.61 mb/sec and Average IO rate=88.61 mb/sec, on increasing the replication=2 Block Size=64MB the Throughput = 70.72 mb/sec and Average IO rate=70.72 mb/sec which is less compare to the previous result. So, we can conclude that replication factor affects the performance cluster. Hadoop is developed for the large datasets, when performance is tested with small dataset such 1.2GB of fileSize the throughput=88.61 mb/sec and average IO rate=88.61 mb/sec compare with 4GB of fileSize the throughput=97.03 mb/sec and average IO rate=97.03. We can conclude that Hadoop works better with large dataset compare to small dataset.

**8. References**

**[**1] Dinh, T. D. (2009). Hadoop Performance Evaluation. Retrieved from https://wr.informatik.uni-hamburg.de/\_media/research/labs/2009/2009-12-tien\_duc\_dinh-evaluierung\_von\_hadoop-report.pdf

[2] Hadoop wiki. Top companies using hadoop. http://wiki.apache.org/hadoop/PoweredBy

[3] Tutorialspoint – Simplyeasylearning http://www.tutorialspoint.com/hadoop/hadoop\_hdfs\_overview.htm

[4] Retrieved from https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[5] Noll, M.G. (2011). Benchmarking and Stress Testing an Hadoop Cluster With TeraSort, TestDFSIO & Co. Retrieved from http://www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench/

[6.] Scarioni, C. (2012). Hadoop Basics—Creating a MapReduce Program. Retrieved from https://dzone.com/articles/hadoop-basics-creating

[7] Performance measurement of a Hadoop Cluster. Retrieved from https://www.amax.com/enterprise/pdfs/AMAX%20Emulex%20Hadoop%20Whitepaper.pdf