ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology
Volume 1, Issue 4, June 2012

Reduction of Data at Namenode in HDFS using harballing Technique

Vaibhav Gopal Korat, Kumar Swamy Pamu
vgkorat@gmail.com swamy.uncis@gmail.com

***Abstract*— HDFS stands for the Hadoop Distributed File System. It has the property of handling large size files (in MB's, GB's or TB's). Scientific applications adapted this HDFS/Mapreduce for large scale data analytics [1]. But major problem is small size files which are common in these applications. HDFS manages these entire small file through single Namenode server [1]-[4]. Storing and processing these small size file in HDFS is overhead to mapreduce program and also have an impact on the performance on Namenode [1]-[3].**

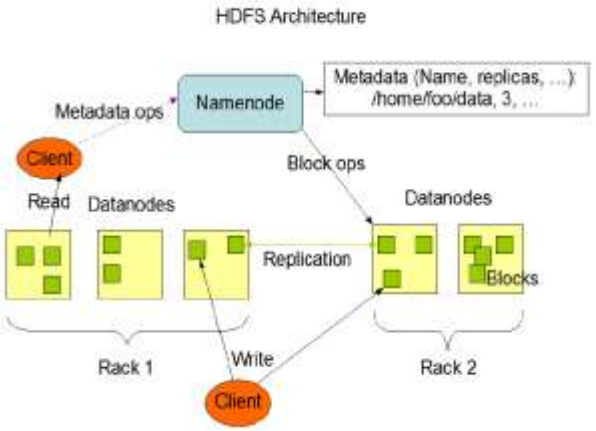
In this paper we studied the hadoop archiving technique which will reduce the storage overhead of data on Namenode and also helps in increasing the performance by reducing the map operations in the mapreudce program. Hadoop introduces "harballing" archiving technique which will collect large number of small files in single large file. Hadoop Archive (HAR) is an effective solution to the problem of many small files. HAR packs a number of small files into large files so that the original files can be accessed in parallel transparently (without expanding the files) and efficiently. Hadoop creates the archive file by using ".har" extension. HAR increases the scalability of the system by reducing the namespace usage and decreasing the operation load in the NameNode. This improvement is orthogonal to memory optimization in NameNode and distributing namespace management across multiple NameNodes [3].

***Key Words*— Distributed File System (DFS), Hadoop Distributed File System(HDFS), Harballing ().**

I. INTRODUCTION

In the past decade, the World Wide Web has been adopted as an ideal platform for developing data-intensive applications. Representative data-intensive Web applications include search engines, online auctions, webmail, and online retail sales. Data-intensive applications like data mining and web indexing need to access ever-expanding data sets ranging from a few gigabytes to several terabytes or even petabytes. Recent trends in the analysis of large datasets from scientific applications show adoption of the Google style programming [1]. So for these large scale scientific application open source version of Google infrastructure is used called, Hadoop. Hadoop architecture uses the Distributed file system, so it takes all the benefits of Distributed File System such as Resources management, Accessibility, Fault tolerance and Work load management.

HDFS has master/slave architecture. An HDFS cluster consists of single Nmaenode and number of Datanodes. A Nameode, known as master manages the file system namespace and regulates access to files by clients and DataNodes manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode. Fig I shows the architecture of the Hadoop Distributed File System [2].



FigureI: HDFS Architecture [2]

The NameNode and DataNode are pieces of software designed to run on commodity machines which typically has GNU/Linux operating system (OS). HDFS is built using the Java language. So any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines. A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The existence of a single NameNode in a cluster greatly simplifies the architecture of the system. The NameNode is the arbitrator and repository for all HDFS metadata. The HDFS architecture is designed in such a way that user data never flows through the NameNode [2].

A. HDFS Design:

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters on commodity hardware.

1. Very large files:

Basically HDFS used to handle large files those which are in gigabytes or terabytes in size. "Very large" in this context means files that are hundreds of megabytes,

gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data [3].

2. Streaming data access:

HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, and then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record [3].

3. Commodity hardware:

Hadoop is designed to run on clusters of commodity hardware. It doesn't require expensive, highly reliable hardware to run on. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure [3].

4. Low-latency data access:

Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. Remember HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency [2].

5. Lots of small files:

Since the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode. As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if you had one million files, each taking one block, you would need at least 300 MB of memory. While storing millions of files is feasible, billions is beyond the capability of current hardware [3].

II. SMALL FILES PROBLEMS IN HDFS

In HDFS files is stored in block and block metadata is held in memory by the Namenode. So it's inefficient to store small files in HDFS as large number of small files can eat up a lot of memory on the Namenode.

The default block size in HDFS is 64MB. Smaller files are one whose size is significantly smaller than the default block size in HDFS. So storing such small files probably have lot of problems, and that's why HDFS cannot handle lots of small files.

As we know that according to the rule of thumb every file, directory and block in HDFS is represented as an object in the namenode's memory, each of which occupies 150 bytes. So 10 million files, each using a block, would use about 3 gigabytes of memory. Scaling up much beyond this level is a problem with current hardware. Certainly a billion files is not feasible [3]-[5].

Also the primary goal of designing the HDFS is for streaming access of large files. HDFS is not geared up to efficiently accessing small files. Reading through small files normally causes lots of seeks and lots of hopping from datanode to datanode to retrieve each small file, all of which is an inefficient data access pattern.

A. Problem with small files and mapreduce:

Hadoop works better with a small number of large files than a large number of small files. If there lot of such small files than the each small file creates a map task and there is lot of such map task which has very little input. Such lots of map task results in extra booking overhead. Compare a 1 GB file broken into sixteen 64 MB blocks, and 10,000 or so 100 KB files. The 10,000 files use one map each, and the job time can be tens or hundreds of times slower than the equivalent one with a single input file [5].

Mapreduce has CombineFileInputFormat which was worked well with small files. Where FileInputFormat creates a split per file, CombineFileInputFormat packs many files into each split so that each mapper has more to process. Crucially, CombineFileInputFormat takes node and rack locality into account when deciding which blocks to place in the same split, so it does not compromise the speed at which it can process the input in a typical MapReduce job to work well with small files [3].

Processing many small files increases the number of seeks that are needed to run a job. if possible, it is still a good idea to avoid the many small files case, since MapReduce works best when it can operate at the transfer rate of the disks in the cluster. Also, storing large numbers of small files in HDFS is wasteful of the Namenode's memory. One technique for avoiding the many small files case is to merge small files into larger files [3].

Consider and java program that counts the appearance of words in the text files which is given as input to the mapreduce program. Following Fig II (a), Fig II (b), Fig II (c), Fig II (d) shows the number of map function to process the large number of small files.

These large number of small files results in lack of performance as it has more number of map function to process each small file as input. Also results in a memory overhead at Namenode by storing these lots of small files.

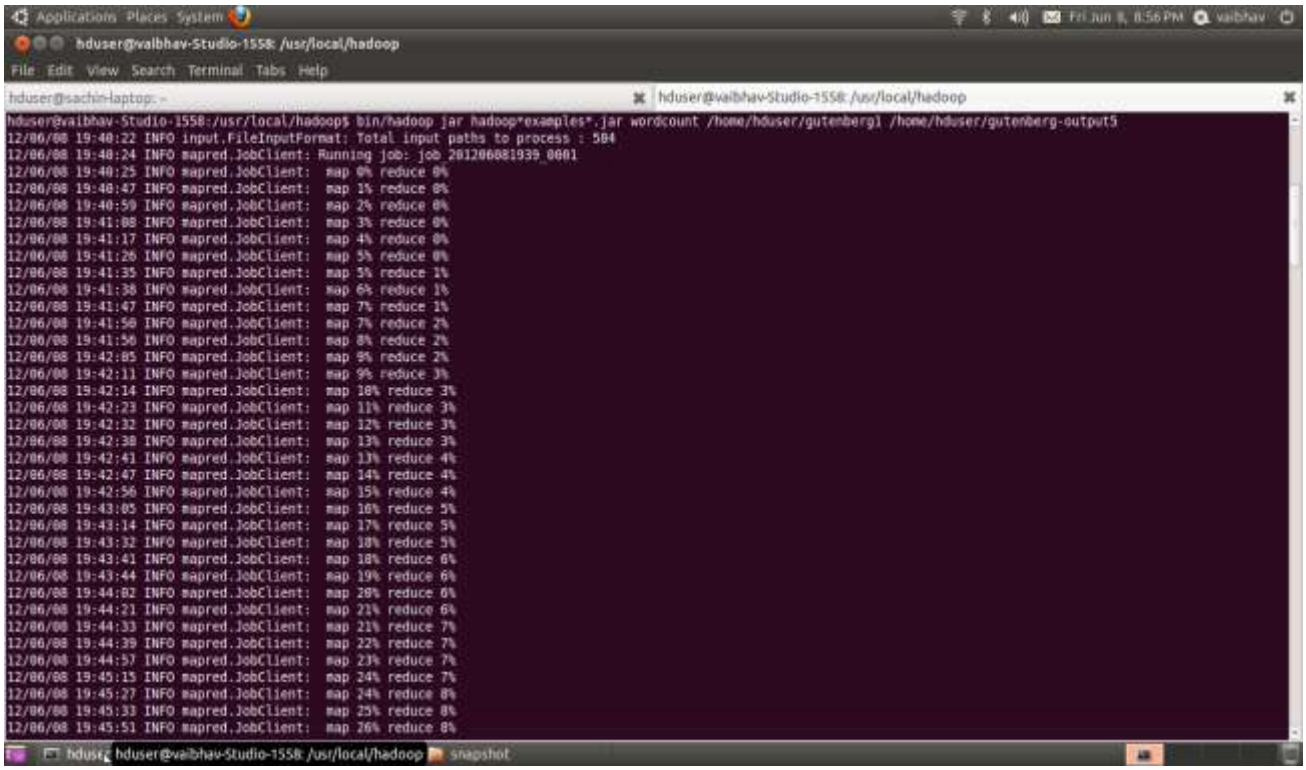


Figure II (a): Processing large no of small files.

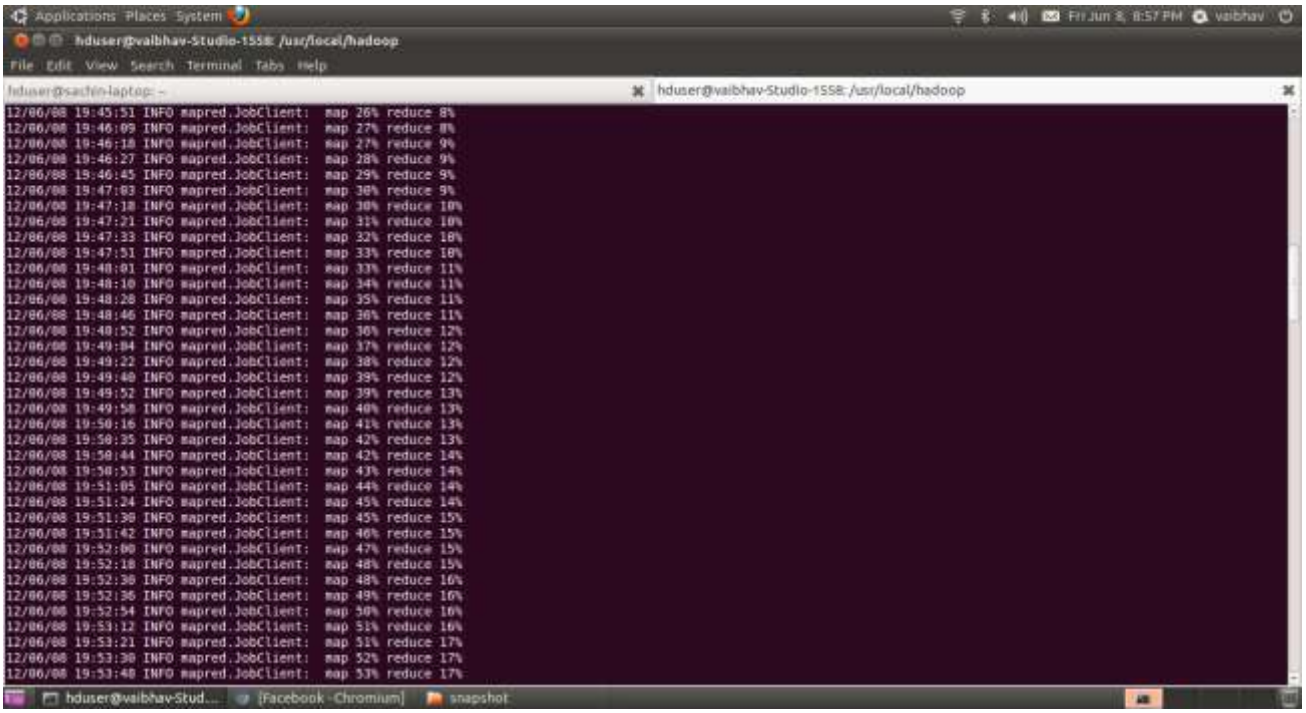


Figure II (b): Processing large no of small files.

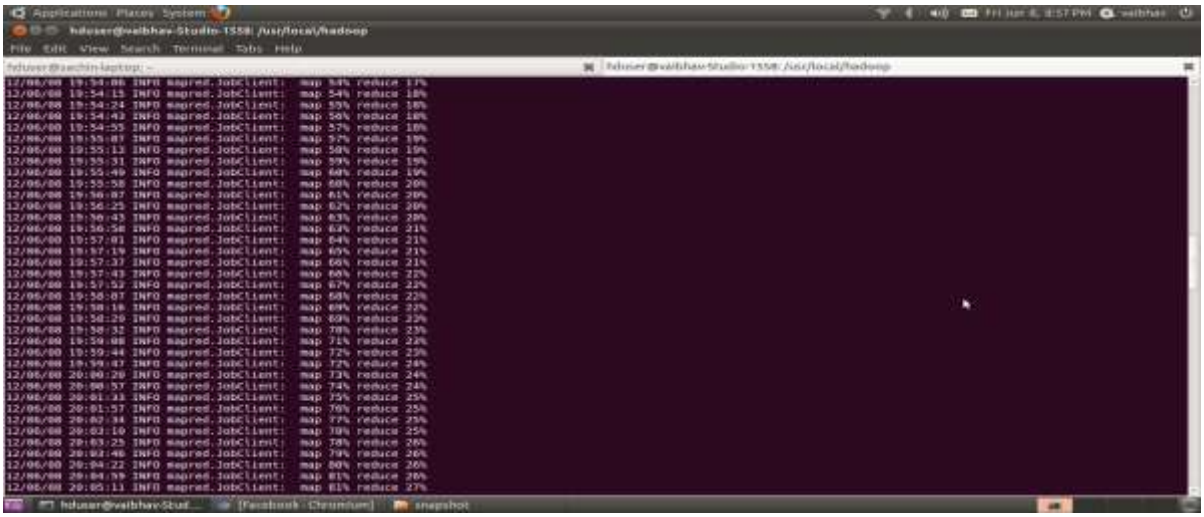


Figure II(c): Processing large number of small files

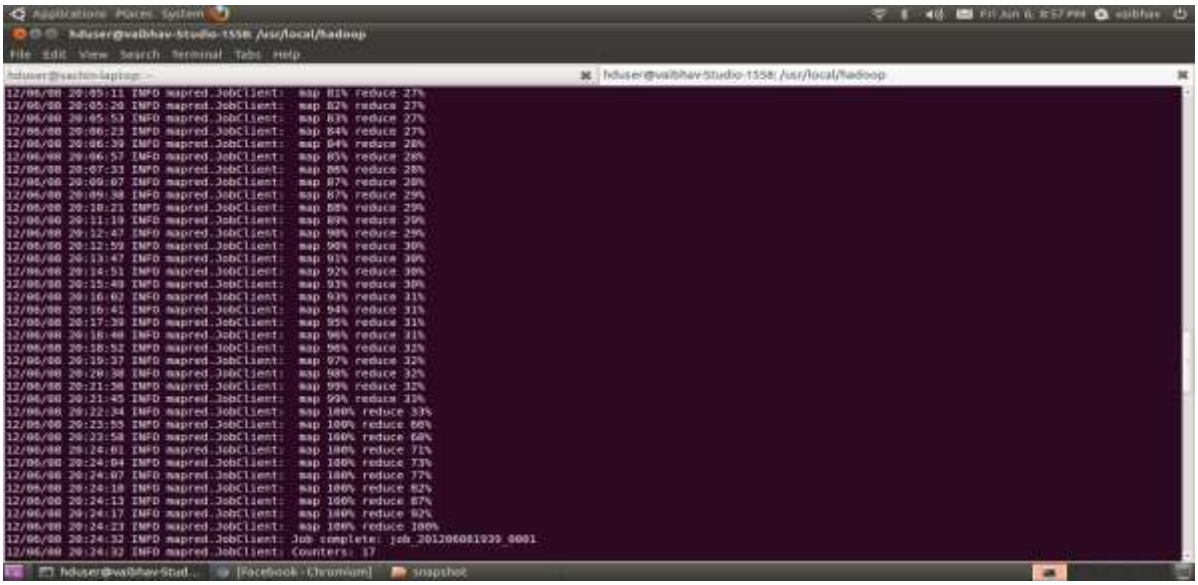


Figure II(d): Processing large no of small files.

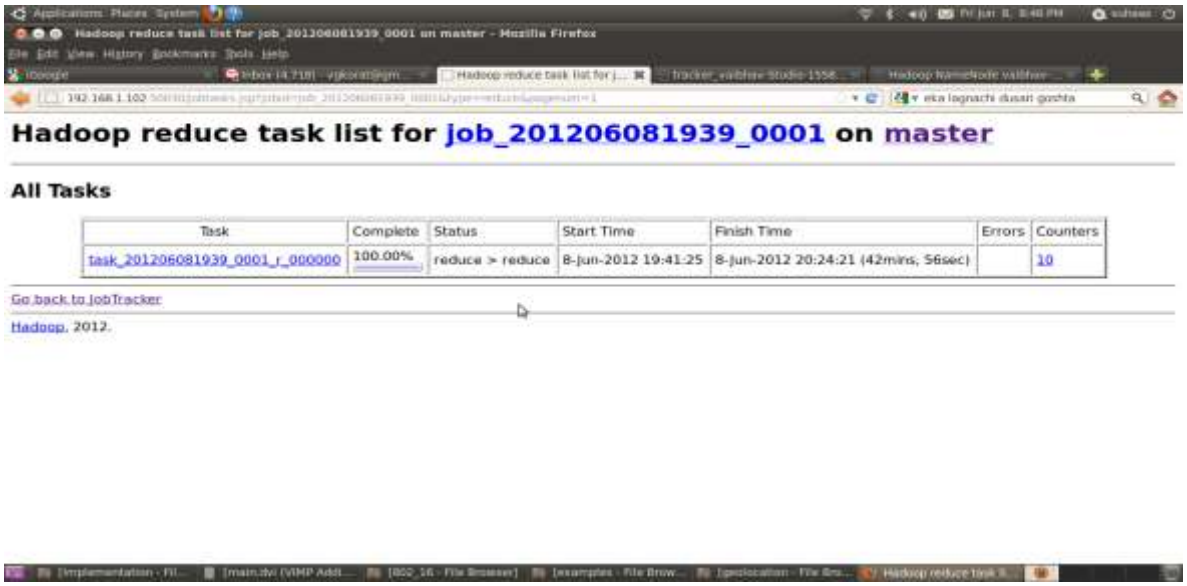


Figure III : Task Details(Before creating HAR file)

The above Fig III shows the task details page shows task statistics. The table provides further useful data, such as the status, start time, Finish time, Error occurred during the reduction of the task and lastly the counters. From the above Fig III we observe that to complete job the reduce operation of machine takes 42mins 56sec.

To overcome the memory overhead problem at Namenode and to increase the performance, Hadoop introduces new archiving technique named as "harballing" technique. Har archiving technique has an advantage of transparency over all the process of archiving. Due to transparency there is no need of de-archive the data then other archiving techniques such as .tar, tar.bz ,zip which is supported by Linux and windows Operating System. Files in a har can be accessed directly without expanding it. This will help in increasing the performance by reducing the step of de-archiving.

III. HADOOP ARCHIVE

HDFS stores small files inefficiently, since each file is stored in a block, and block metadata is held in memory by the Namenode. Thus, a large number of small files can eat up a lot of memory on the Namenode. Hadoop Archives, or HAR files, are a file archiving facility that packs files into HDFS blocks more efficiently, thereby reducing Namenode memory usage while still allowing transparent access to files. Hadoop Archives can be used as input to MapReduce [3]-[5].

HAR files work by building a layered filesystem on top of HDFS. A HAR file is created using the hadoop archive command, which runs a MapReduce job to pack the files being archived into a small number of HDFS files. Hadoop Archive is integrated in the Hadoop's FileSystem interface. The Har FileSystem implements the FileSystem interface and provides access via the har:// scheme. This exposes the archived files and directory tree structures transparently to the users. Files in a har can be accessed directly without expanding it [6].

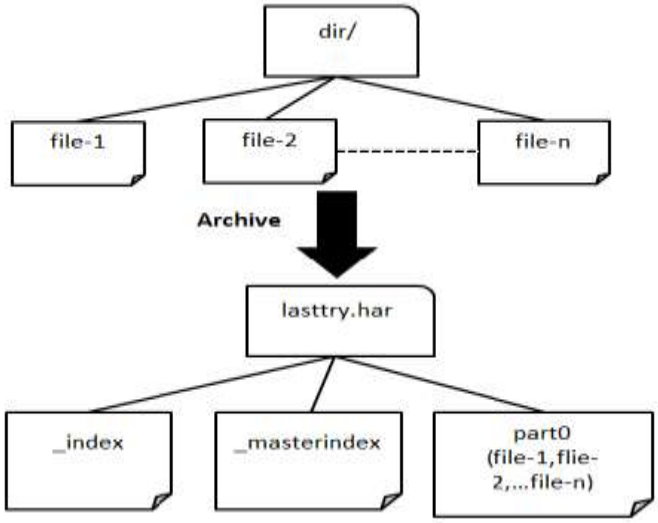


Figure III: Data Model for Archiving Small Files(har Format)

For this reason, there is a slight overhead in referencing files. The request must now go through the metadata of the archive to the index of metadata that the archive contains in order to access the required file. However, this access is done in main memory, so for any harball of reasonable scientific size, the extra layer causes negligible overhead [6].

A. Creating HAR file:

A hadoop archive is created from collection of files using the archive tool. The tool runs a MapReduce job to process the input files in parallel. Following command is used for creating a archive file [3].

%hadoop archive -archiveName lasttry.har /home/hduser/gutenberg1 /home/hduser.

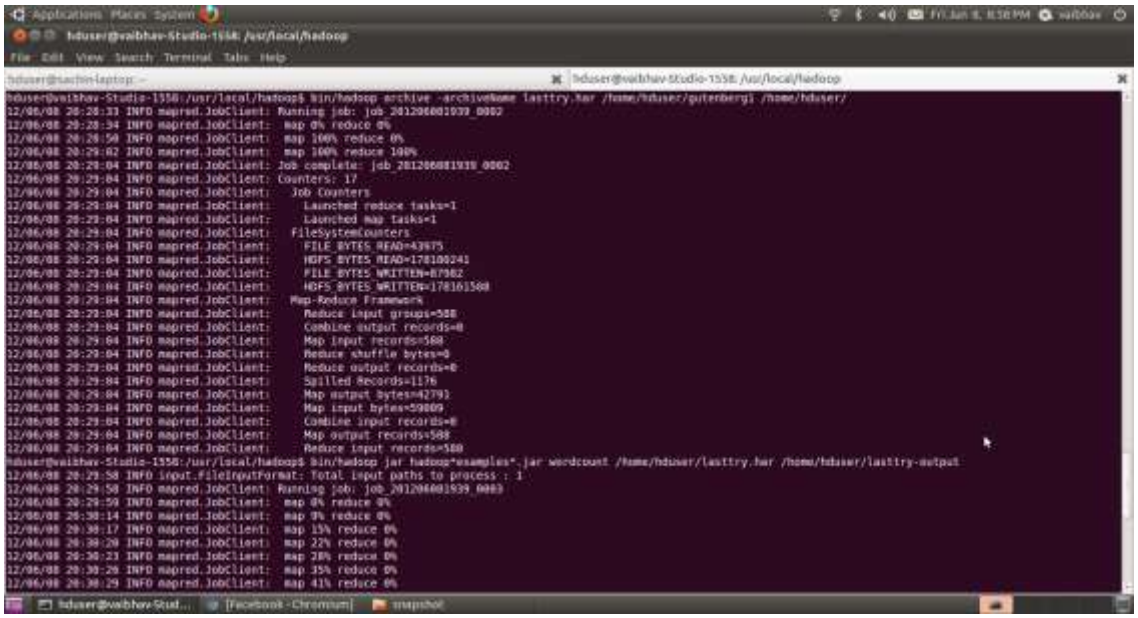


Figure IV: lasttry.har file

To address the Namenode's metadata more efficiently har archival technique is used which is already provided by the Hadoop architecture in new unique way. Increase the amount of files that a Namenode can identify per GB of RAM. By archiving groups of small files into single files, drastically reduce the metadata footprint, thus increasing the possible address space of the Namenode without increasing RAM. The harball introduces another file layer on top of the HDFS. The archive creates a metadata entry for the index of the files it contains. This index serves as a meta-meta data layer for the data in the archive. The output of the har file contains of three file index file, to interact with HAR files,using a HAR fielsystem that is layered on top of the underlying system.

After creating the archive lasttry.har file we again run the same wordcount program using the following command. The following command takes a input as har file and run the worcount example again. Observe the output.

*%hadoop jar hadoop*examples*.jar wordcount /home/hduser/lasttry.har /home/hduser/lasttry-output*

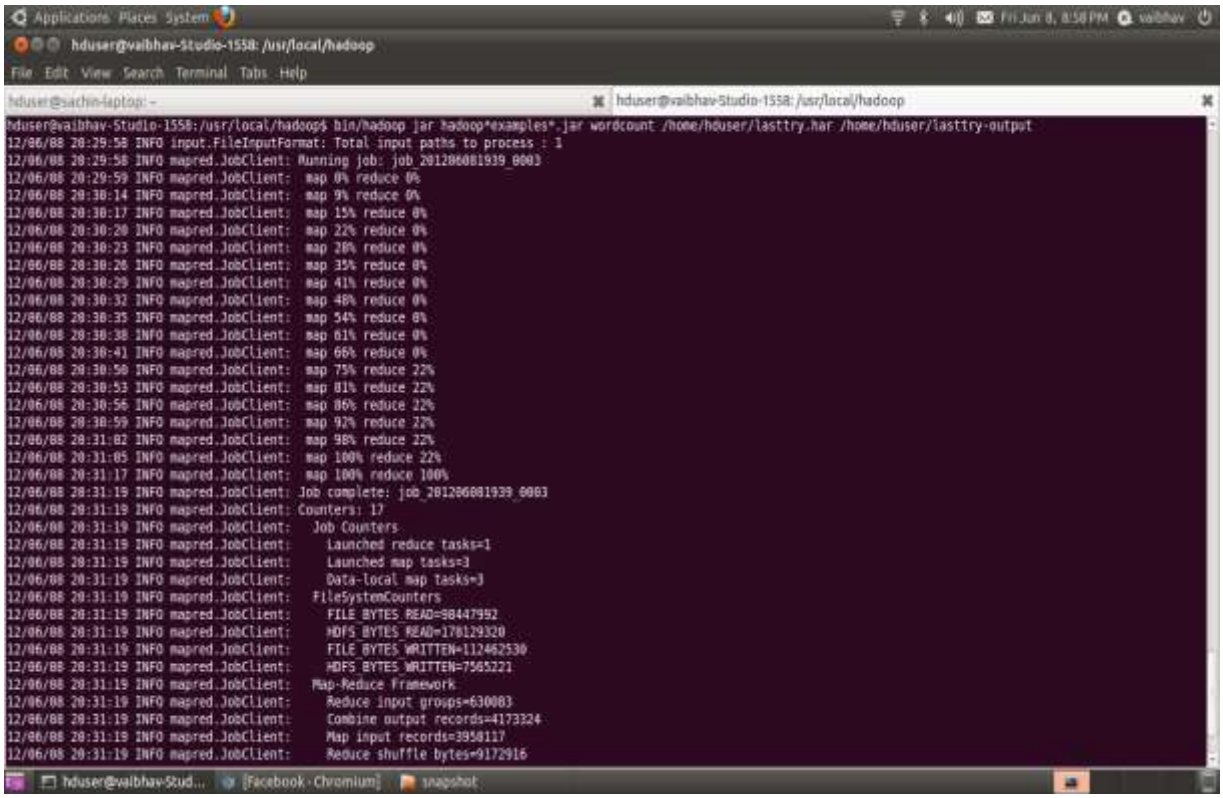


Figure V: lasttry.har file as input to the mapreduce function in HDFS

Masterindex file and part file. Following commands shows the output of the archive file.

%hadoop dfs –ls /home/hduser/lasttry.har

lasttry.har/_masterindex //stores hashes and offset.

lasttry.har/_index //stores file statuses.

lasttry.har/part-[1..n] //stores actual file data

The part files contain the contents of a number of the original files concatenated together. The index file is sorted with hash code of the paths that it contains and the master index contains pointers to the positions in index for ranges of hash codes. All these details are hidden from the application, however, which uses the har URI scheme to,

From the above figure we observe that the parameter Total input path to the process reduces to 1 as compared to Total input path to the process equal to 584 before archiving the (gutenberg1) file which is given as input to mapreduce program, in HDFS. As we combine the small size file in a large size file and then give it as a input to mapreduce program the map and reduce operation also reduce to some extents(as shown in FigV).This helps in increasing the performance of executing the job by reducing the time.

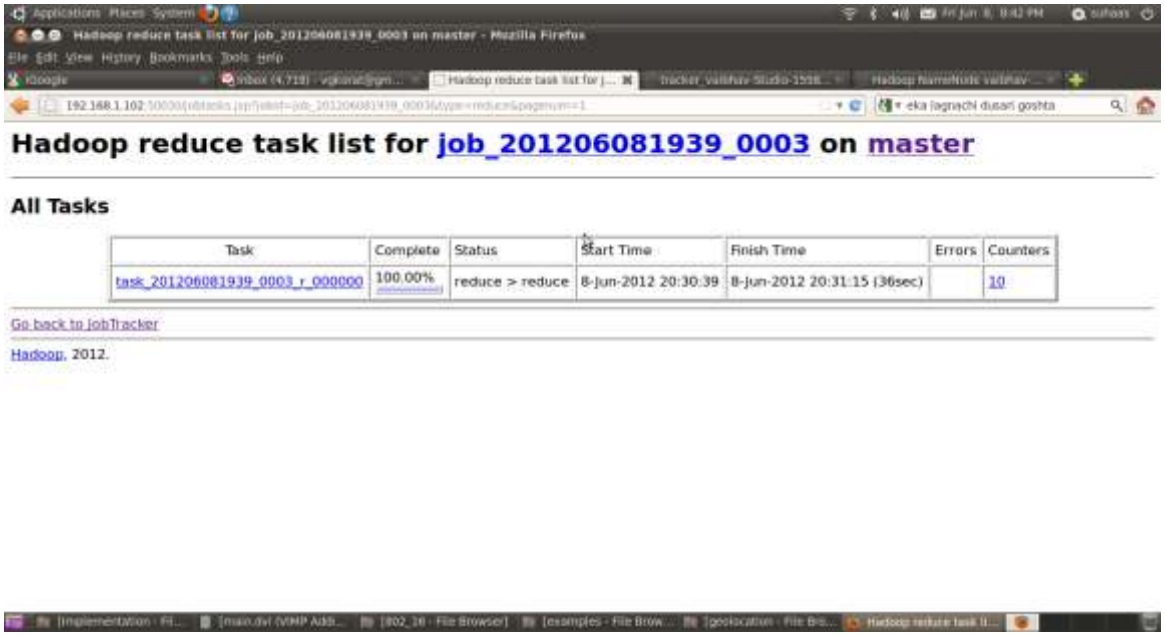


Figure VI: Task Details (After creating a har file)

Fig VI shows the task details combining the small file into a single file using harballing technique (creating a .har file) in HDFS. By comparing the FigureIII and FigureVI we observe the finish time column of the task. In FigureIII to complete a reduce operation on the job it requires 42min 56sec and after archiving the file it takes only 36 sec. It results in increasing the performance.

IV. CONCLUSION

In this paper we give small description about the architecture of Hadoop Distributed File System. From the architecture we know the default block size is 64MB and each file is stored in a block, and block metadata is held in memory by the Namenode. So by storing large number of small size files over Namenode eat up a lot of memory and hence it's inefficient to store small files over Namenode.

To overcome the problem of the small size file we use Hadoop's archival method of .harballs to minimize the metadata storage requirements over Namenode. From the experiment we found that when we give the har archive file as input to the mapreduce function, it completes the job with minimum map and reduce function operations as compared to previous one. So it means we can give har file (which is collection of small size files) as input to mapreduce program to increase the performance time.

REFERENCES

[1] Grant Mackey, Saba Sehrish, Jung Wang Improving Metadata Management for Small Files in HDFS.
[2] Book:The Hadoop Distributed File :Architecture and Design by Dhruba Borthakur. Pages 4-5.
[3] Book:Hadoop The Definnitive guide. Pages 1-13,41-49,71-73
[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung Google. The Google File System.
[5] Small File Problems in HDFS Available: http://www.cloudera.com/blog/2009/02/the-small-files-problem/
[6] Hadoop har archive files idea. Available: http://developer.yahoo.com/blogs/hadoop/posts/2010/07/hadoop_archive_file_compaction/

Vaibhav G. Korat was born in Amravati on 28-Sep-1987. He received BE degree (Computer Science & Engineering) from Prof. Ram Meghe Institute of Technology and Research college of Engineering, Badnera, Maharashtra, India in 2009. He is pursuing M.Tech degree in Computer Technology [Specialization in Network] from V.J.T.I, Mumbai, India. His major field of study is computer networks.

Mr. Vaibhav is a member of Computer Society of India and his paper "Review of A-Star Algorithm for Battery Consumption Problem in mobile Ad-Hoc Networks and Routing Methods" to published in International Journal of Smart Sensors and Ad Hoc Networks (IJSSAN). He also contributes his knowledge in the field of Secure Web System Computer Networks and published a paper "Secure Web System Development" in International Journal of Internet Computing .

Kumarswamy Pamu completed his Master's of Engnieering Mumbai, Maharashtra, India in 2005. His major fields of studies are Wireless Communications, cognitive wireless radio networks. Mr. Kumarswamy Pamu has a research skills in technical field in Solaris, HP-UX, AIX and Redhat Linux implementation, Network administartion of devices such as Cisco (wired or wireless). Mr. Kumarswamy Pamu is a scientific member of IEEE organization. He has a teaching experience of Mtech level courses.