HDFS Pipeline Reconstruction to Avoid Data Loss

Purnachandra Rao. B

Department of Computer Science & Engineering
ANU College of Engg & Technology,
Guntur, India.
pcr.bobbepalli@gmail.com

Nagamalleswara Rao. N

Department of Information Technology
R.V.R. & J.C. College of Engineering & Technology
Guntur, India.
nnmr3654@gmail.com

Abstract - **Hadoop is a popular framework designed to deal with very large sets of data, ranging from gigabytes to terabytes, and large Hadoop clusters store millions of these files. Hadoop Distributed File System (HDFS) uses a pipeline process to write the data into blocks. Namenode then sends the available blocks list so that a pipeline is created based on datanodes having empty blocks. In this paper we customize the datanode replacement policy using configuration parameters in case there is datanode failure in the pipeline process. In this approach the write process resumes even though there are less number of datanodes, i.e. even having single datanode. In the single datanode case, the data is lost since we have only one copy of the data. This paper addresses the issue of having a single datanode in the write operation to take action on clearing up unwanted data like tmp, trash folders and snapshots to have enough space to proceed further in the write operation and avoid the data loss.**

Keywords - DataNode, MapReduce, NameNode, RackAwareness, ReplicationFactor, dataloss, memory space.

I. INTRODUCTION

The Apache Hadoop [1] is explicitly designed to handle large amounts of data, which can easily run into many petabytes and even exabytes (1000^6 bytes). Hadoop data files employ a write-once-read-many access model. Data consistency issues that may arise in an updatable database are not an issue with Hadoop file systems, because only a single writer can write to a file at any time. HDFS [2] [3] architecture is implemented in such a way that only one client can write at a time where as many clients can read at the same time to reach the data consistency. Hadoop is having the master node called NameNode and it is having namespace. Data will be stored in DataNodes where these are connected to Namenode and periodically sends status report to NameNode. For each write operation of client, data will be written to client temporary file. Once the write operation got finished or the temporary file get filled or it crosses the data block boundary, file will get created by Hadoop and the blocks will be assigned to the file. Then the HDFS write operation will be performed by copying block by block to file block. After the copy of first block that will be replicated to two other blocks (default replication factor is 3). The write operation will be completed succesfully only when the Hadoop can write all the blocks of temporary files data to target nodes inclusing replication levels. While writing the data to nodes, there is a possibility of datanode failures. We can have datanodes replacement policy using the config parameters. In this we are addressing the issue of writing data to less number of nodes and the possibility of losing the data. HDFS [2] [3] is designed for shared read operation with exclusive write operation method. There will be number of interactions between NameNode and DataNode to complete the read and write operations and this will decreases the access performance when the system is under a heavy workload. HDFS stores data in HDFS files, each of which consists of a number of blocks (default size is 128MB). This is the minimum amount data that the client can interact. The default block size is customizable, i.e. we can configure it using the HDFS configuration. There will be some time factor associated with retrieving or keeping the data in datanode. Jobs will be scheduled on the same node to minimize the disk access and the data will be replicated to number of datanodes to overcome the data loss issues, improving the throughput and job completion time. . NameNode manages the filesystem namespace by performing the tasks: maintains the metadata pertaining to the file system, such as the file hierarchy and the block locations for each file. It manages the user access to the data files, Mapping the data blocks to the DataNodes in the cluster, Performing file system operations such as opening and closing the files and directories, providing registration services for DataNode cluster membership and handling periodic heartbeats from the DataNodes, determining on which nodes data should be replicated, and deleting over replicated blocks, processing the block reports sent by the DataNodes and maintaining the location where data blocks live. While the NameNode is aware of all the DataNodes that store the data blocks for any HDFS file, it doesn't store the block locations and it will simply reconstructs them from information sent by the DataNodes when you start up the cluster [11]. Like this it retains the information in memory for fast access to it. Based on the directives sent by the NameNode, datanode will provide block storage, read, write request processing based on the client requirement. DataNode will send periodic updates to NameNode, so that NameNode can keep the list of alive nodes with the latest heartbeat datanodes. Using the heartbeat NameNode will decide the healthiness of the Datanode, and the block report will decide the blocks being managed by the DataNode.

II. LITERATURE REVIEW

A. Hadoop Distributed File System

NameNode has metadata of HDFS and the application data is available at DataNodes. DataNode has blocks. The data is stored inside blocks. The hard link from the block will be removed if there is a deletion operation. So the data block will remain in the same directory [4]. DataNode blocks have the file data and the number of blocks data is replicated using the replication factor configured at the property available at configuration file. Namespace in the namenode hs information related to blocks and datanode info of the file. Apache Hadoop core component is a HDFS and provides a virtual file system to client applications. Files are stored across the blocks and the blocks are replicated according to replication factor. Zookeeper is the application which manages and synchronizes distributed clusters and allows coordination among them. Hadoop stores huge volume of data sets constantly. Hadoop is well-liked open source software and can support parallel and distributed data processing. Hadoop is highly scalable computer platform. Hadoop allows users to process and store huge amount which is not possible while using less scalable techniques [14].

Some fault tolerance mechanisms are provided by Hadoop so that system works properly even if some failure occurs in system. For large processing applications data should be provided at high bandwidth. The Hadoop distributed filesystem is having the capability of providing data at high bandwidth to large processing applications. Since Hadoop is performing large volume of data operations it needs to have log file management for tracking the transactions. Flume is a component which will collect all the log activities and keep it at central store. While working with MapReduce applications there may be a necessity of remote access among the clusters. There is a partitioned data placement to avoid the remote access, to lower the destructive performance interference. Job tasks will be scheduled dynamically on any node in a simple cluster. Partitioned data placement substantially reduces remote access rates if task placement is restricted by long running jobs or other prioritizations tasks [12]. The HDFS metadata will be replicated to different NameNodes, so that it can avoid single point of failure. In large volume of data operations if the NameNode goes down then there will be an issue of HDFS goes down because of having single NameNode (metadata). This architecture can't meet the exponentially increased data operations. So we need to maintain the backup of the metedata [13]. Introduce multiple Namenodes in the architecture so that we can avoid this type of issues. Only issue here is the communication among the multiple NameNodes[15]. If you want to increase the cluster capacity, then there is always a bottleneck of having the amount of memory at NameNode. If you increase the cluster then there is a need of accommodating the address space at NameNode. It will always limit the number of files in the memory. So the increase of cluster is not possible if there is a single NameNode [17].

The Data will be written to datanode blocks. When there is a need of data to application that needs to get the data from the client. Instead of that If the data is at nearby location to the application then there is reduction at data access time. Here is the need of Cache memory. If the data is available at the cache location the applications will access the data faster than getting the data from the client location. The cache memory is used to store frequently access data and hence process it much more quickly [6] [18]. Using this policy we can skip the unnecessary trips to hard disk by keeping the data at cache location. Accessing the data with cache will take lesser time than accessing the data without cache. MapReduce is framework which will reduce the task into number of sets and performs the tasks on each smaller set and combines the results based on the key value pair of small sets.

Replication factor property is set up by the application. Whenever the client wants to write data to Hadoop File System, then it will send request to NameNode. NameNode will provide the list of datablocks available at DataNodes (File system) and it will create the output stream. The client will start invoking the output stream followed by writing the data to first datablock there by it will be copied to another datablock in the datanode pipeline (which was already created when there is a request to write operation by client) and son until reaches the end of the datanode pipeline. The write operation is finished only when the copy operation has been completed with all the datablocks in the pipeline. To maintain the high availability of Namenode or the HDFS system, we need to have backup of the metadata at the NameNode, so that we can avoid the single point of failure. The datanode will be declared as dead if the heartbeat will not be received by NameNode at the normal defined period. The default time to declare as dead is 10 minutes [9].

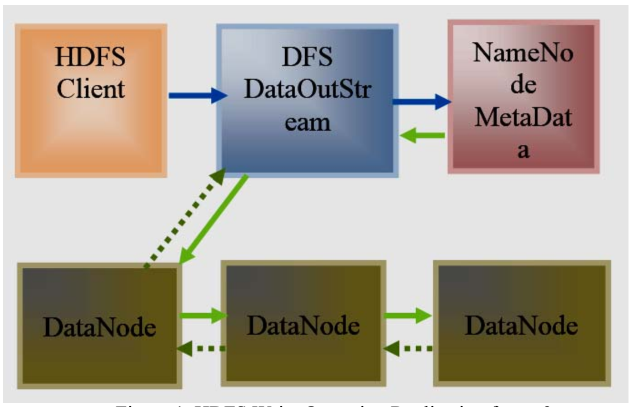


Figure 1. HDFS Write Operation Replication factor 3

Fig.1 shows the HDFS write operation using the default replication factor i.e, 3.

While writing data to the pipeline if there is a failure in the datanode the pipeline will be closed and the packets will be addess to the front of the queue so that the next pipeline will not miss the packets. The new identity will be generated and assigned to new datanodes blocks where the failed datanode blocks will be removed. The data will be transferred to working datanodes while the failed datanode will be removed.

The Namenode will try to arrange the new datanode since it is under replicated. Even if multiple failures are there HDFS client will try to continue even if there is only one datanode. Fig.2 shows the HDFS write operation using replication factor 2 and Fig.3 shows the HDFS write operation using replication factor 1.

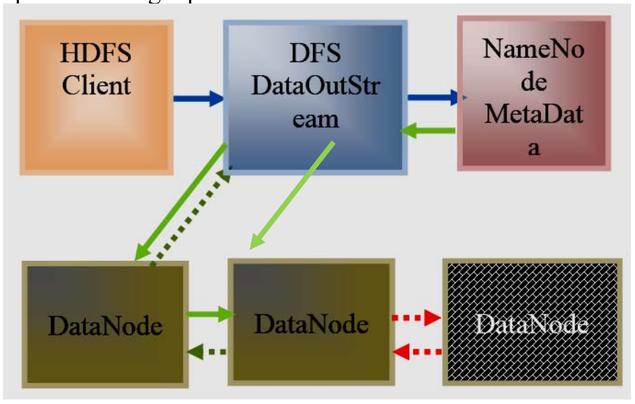


Figure 2. HDFS Write Operation Replication Factor 2

Broken red lines in Fig. 2 and Fig. 3 means that there is no communication because of datanodes were down. Table 1 shows the Data Availability based on the replication factor. If the replication factor is 3 then the data availability is 100%, if it is 2 then availability is 66.66% and 33.33% if the replication factor is 1. Data availability is 0 if the replication factor is 0, that means it is causing data loss. Please observe the same results in Graph 1. This is the problem in the existing architecture.

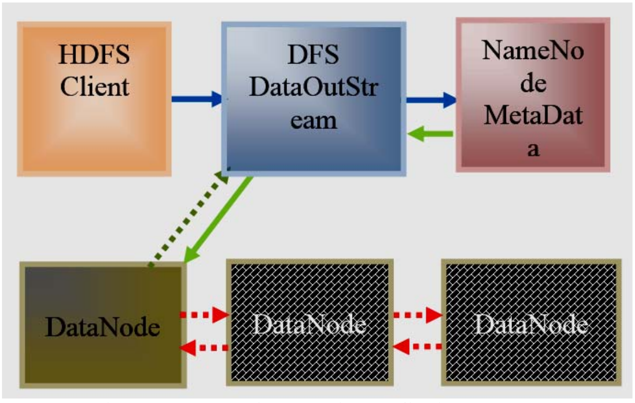
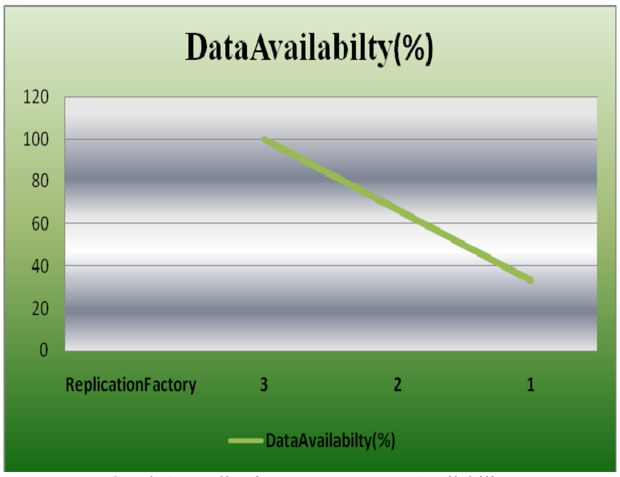


Figure 3. HDFS Write Operation Replication Factor 1.

TABLE I. REPLICATION FACTOR VS DATA AVAILABILITY

Replication Factor	Data Availability (%)
3	100
2	66.66
1	33.33



Graph 1. Replication Factor vs Data Availability

Data availability is going down while the datanodes failures are going up. Please observe the same results in Graph 1.

The parameter dfs.client.block.write.replace-datanode-on-failure.enable If there is a datanode/network failure in the write pipeline, DFSClient will try to remove the failed datanode from the pipeline and then continue writing with the remaining datanodes. Consequence of this is the count of the datanodes will come down. We are using this property to add new datanodes to the pipeline. If the cluster size is very small say 3, then there is a problem of having very less number of datanodes. In this case it is better if the administrators set the policy to NEVER. In this users will face the issue of pipeline failures because of having less number of datanodes. The parameter dfs client block write replace-datanode-on-failure.policy . Please use this policy only if the parameter defined above declared as true. We can use this in two modes one is ALWAYS and the other one is NEVER. That means we can replace the failed datanode with the new datanode or don't replace with new datanode. We need to use the best effort dfs.client.block.write.replace-datanode-on-failure.best-effort property once enabled the property of dfs.client.block.write.replace-datanode-on-failure.enable is true. However, it continues the write operation in case that the datanode replacement also fails. Suppose the datanode replacement fails. The write operation will be failed and in the programming view it will raise an exception. If the replacement is successful then the write operation will be continued with the remaining datanodes. There is always writing to datanodes if we set this property to true irrespective of the number of datanodes, hence it increases the probability of data loss.

B. Namenode

NameNode is having metadata of HDFS and the application data is available at DataNodes. DataNode is having blocks. The NameNode reads fsimage file so that it can get the state of the file system as soon as it started. The Namenbode uses the edit log and updates the edit log with the metadata loaded into memory, fsimage file will be updated by NamNode with the HDFS updated state [16]. DataNode is having the blocks and the blocks are having the data from the client. The data will be read by the client or applications based on the requirement. The data will be available/copied based on the replication factor. That mean same data will be available at number of locations. Namespace in the namenode is having information related to blocks and datanode info information of the file. The NameNode will provide the datablocks as soon as requested by HDFS client. If the client requests for write oparation NameNode nominates list of datanodes available for taking the data. The data will be written to list of datablocks (Datanodes) once it receives from the NameNode [7,8]. We can control the number of datablocks or datanodes using the replacement or best effort parameters available at hadoop configuration files.

C. Datanode

The Rack is having number of clusters and each cluster is having number of datanodes. In the initialization phase of the NameNode it will have handshake mechanism with datanodes, so that it can verify namespace id software version of datanode will be verified. Based on the success of the match the datanode position will be continued with the namenode. In the failure case of the match the DataNode will automatically shuts down. If the new datanode joins the cluster as a replacement policy then it will receive the namespace id.

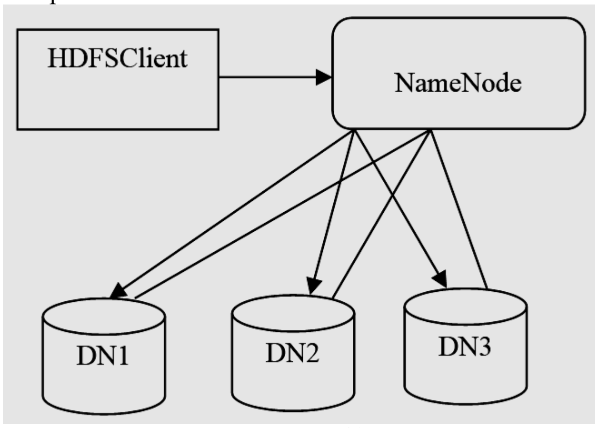


Figure 4. HDFS Architecture.

This id will be verified when there is a start of the NameNode. NameNode uses the block report and heart beat policy to decide the existence of the datablocks/datanode. As soon as the datanode will get initialized it will send the first block report. The subsequent block reports will be sent once in a hour. If there is absence of heart beat for more than 10 minutes then NameNode will decide the failure of datanode and Namenode will schedules the creation of replica of the failed blocks different datanodes. Refer figure.4 for HDFS architecture. As shown in the fig Namenode will send info to the datanodes and datanodes has to sent heartbeats to NameNode in a specified time interval. DN1, DN2 and DN3 are the datanodes.

D. MapReduce

Hadoop has to process the large volume of data. It uses the framework to reduce the input into number of smaller data sets. It will perform the operations on each smaller data set followed by merge operation on the data set . The frame work is called MapReduce. It is using number of algorithms to reduce the task into smaller tasks and assigning them to multiple systems. Sorting, searching, indexing and TF-IDF are the three types of mathematical algorithms. From the mapper output key-value pairs will be sorted based on their keys. Refer figure 5 for MapReduce Architecture.

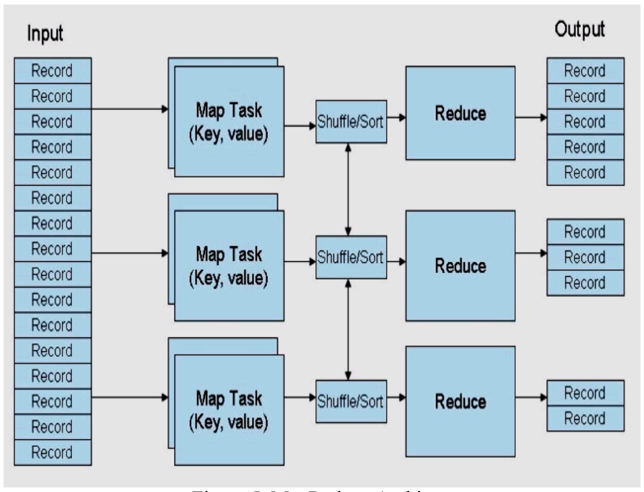


Figure 5. MapReduce Architecture.

E. Data Distribution in HDFS

If HDFS gets the write operation request it will be directed to NameNode for getting the initial list of data blocks available at file system. Then the Outstream will be invoked by the HDFS client to get the address of initial datablock. The pipeline will be created based on the list of available datablocks from the NameNode. The first data block will be written the data from the client and the data will be propagated to second datanode using write operation and so on until the last of datablock in the pipeline. The acknowledgement will travel back till the first block from the last block all the way through the intermediate blocks. If the acknowledgement will not be received by the datanode pipeline then there is an issue in the pipeline (datanode failure) , and the new datanode pipeline will be created based on the availability of the datanodes in the pool. Three is the default replication factor. Refer figure 6 for HDFS write operation.

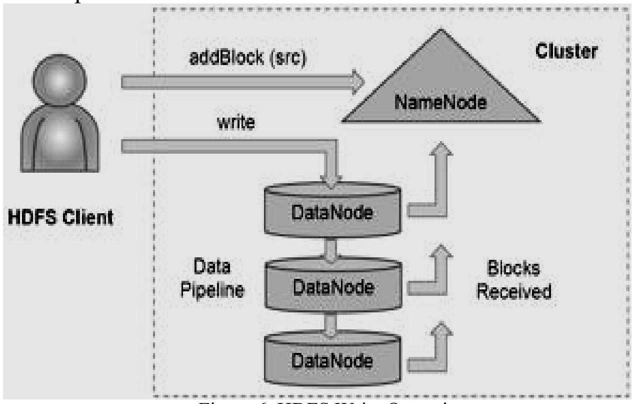


Figure 6. HDFS Write Operation.

F. CheckpointNode

HDFS operations will be copied to edit log file. The log file will copied to fsimage. Suppose if there is a huge amount of data available at edit log file and not yet copied back to fsimage and NameNode has to reinitialize because of some issue, then it will take longer time to get the last status of the file system. To avoid this delay in the initialization of the file system edit log file info and fsimage info will be merged once in a while. This is called Checkpoint Node. After this it will upload to NameNode. We can have the same feature with Secondary NameNode. But it will not upload to NameNode. So the NameNode need to fetch the state from the Secondary NameNode [19]. There is one more functionality called the backup node provides the same functionality as the Checkpoint Node, but is synchronized with the NameNode. It doesn't need to fetch the changes periodically because it receives a stem of file system edits from the NameNode. It holds the current state in-memory and just need to save this to an image file to create a new checkpoint.

G. RackAwareness

Rackawareness components are HDFS and MapReduce. They can have the snapshot view of all the nodes of the cluster rack they belong to and act accordingly. One replica of the block goes to the local data node and two replicas goes to another two data nodes, selected randomly from the cluster in the case of there is no configuration of rack awareness (in this case, all nodes of the cluster are considered to be on the same rack), or if you have a small cluster with just one rack. To avoid this problem, Hadoop (HDFS, MapReduce) supports configuring rack awareness [20]. This ensures that the third replica is written to a data node from another rack for better reliability and availability. Rack availabilty is high because of rack awareness. This will increase the utilization of network bandwidth when reading data because data comes from multiple racks with multiple network switches. The task completion time will be reduced significantly by using the rack awareness, upon the increase of volume of data being processed as well the computational speeds will be reduced [5]. One directional ring structure will be used for data transfer and round robin fashion will be used for inter rack data transfer [10]. Master node uses the script having the network topology to make the rack awareness among the clusters. We need to use the topology script filename parameter in the core-site.xml configuration file. First, you need to change this property to specify the name of the script file. Then you write the script and place it in a file at the specified location. The script should receive a list of IP addresses and return the corresponding list of rack identifiers. It is a one-to-one mapping between what the script takes and what it returns. The simplest way is to read from a file that has the mapping from IP address or DNS name to rack identifier.

H. FileSystem Snapshots

We can take the read only copy the files. These are called HDFS Snapshots. Entire file system or subtree of the file system can be taken as snapshot. We can use this as databackup or error protection. Snapshots can be taken on any directory once the directory has been set as snapshottable. Simultaneously 65,536 snapshots will be accommodated by snapshottable directory. We can have n number of snapshottable directotries . Administrators can set any directory to be snapshottable. We need to delete all snapshots inside any snapshottable directory before deleting the snapshottable directory. Nested snapshottable directories are currently not allowed. If the directory is having snapshots as its ancestors/descendents then we cannot take the sansphot from the directory. If there is an issue in the file system or in the file all the data will be lost because of not having backup. If the snapshot is available then we can take the data from the snapshot.

That means we can persistently save the file system to current state of the file system. We can roll back the file system to particular state in case of failure. It helps to rollback in case of failure during upgrade. Using this HDFS can be resume back to the state as if it was at the time of snapshot. Copy of storage directory will be created by datanode along with hard links. That is why the time of deletion only the hard links will be removed. So the previous data will be remains untouched in their old directory [4]. HDFS will provide shared lock for multi reader policy. If there number of clients are for reading it will allow all the readers at a time, where as a single write it will not allow more than a single client. So it will create a exclusive lock policy. This lease will be revoked once the read or write operation will be completed. Hflush operation will be called by the user once the operation is completed. Hflush operation will wait until the acknowledgement reaches from the last datanode to client. This makes all data written before hflush operation visible to readers [9].

III. PROPOSED METHOD

A. Problem Statement

HDFS will use the pipeline process to write the data into blocks. Namenode will send the available blocks list so that pipeline will be created based on the datanodes having the empty blocks. We can customize the datanode replacement policy in case of any datanode failure in the pipeline process using configuration parameters. In this process write process will be resumed even though there are less number of datanodes i.e, even having single datanode. In Single datanode case we will lose the data since we have only one copy of data. This paper addresses the issue while having single datanode in the write operation and taking action on clearing up the unwanted data like tmp, trash folders and snapshots so that we can have the enough space to proceed further on write operation to avoid the data loss scenario. DataNode replacement property in case of failure dfs.client.block.write.replace-datanode-on-failure.best-effort will be used only when dfs.client.block.write.replace-datanode-on-failure.enable is true. Suppose if the datanode replacement fails then the exception will be thrown so that the write operation will fail. If the replacement successful then the write operation should be resumed with the less number of datanodes. Setting this property to true allows writing to a pipeline with a smaller number of datanodes. As a result, it increases the probability of data loss. This is the problem in the existing environment.

B. Proposal

For the datanode replacement we can use different parameters like datanode replace enable, failure policy and best effort policy. Using these parameters we can control the datanode replacement in case of datanode failure in the pipeline. Failure best effort policy will try to replace the datanode with new datanode with maximum probability. However, it continues the write operation in case that the datanode replacement also fails. Suppose the datanode replacement fails. The write operation will be failed and in the programming view it will raise an exception. If the replacement is successful then the write operation will be continued with the remaining datanodes. There is always writing to datanodes if we set this property to true irrespective of the number of datanodes, and this operation will continue even if there is a single datanode. If there is an issue at the single datanode we will lose the entire data. To avoid the data loss we can remove tmp files based on age criteria, trash files and snapshots.

IV. IMPLEMENTATION

HDFS write operation will work with the datanode pipeline. It will count the number of datanodes in the pipeline. If it is exactly equal to one then we need to invoke the process to clearup the tmp folder based on the age of the files in the folder. As a second step we need to clean up trash folder and snapshots as well.

```
while
  do

  all directory files in hadoop /tmp directory

  find age of the file

  if age of the file is more than 1 day

  delete the file/folder

  repeat the same process till the end of list of files.

  fi

  done

  find all files at trash folder and remove them

  remove all snapshot files.
```

Figure 7 Space Reclaiming process

For this we need to implement the algorithm mentioned at Fig. 7. This will search for the /tmp folder and iterates through the each file and folder recursively and find out age of each file. If the age is minimum one day then we will apply delete operation using the hadoop file system commands. It will apply the delete operation on trash folder and snapshots as well. This is how we can free up some space and provide additional blocks in the pipeline.

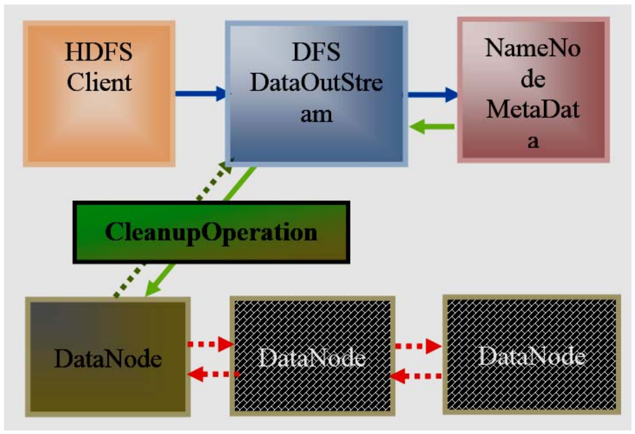
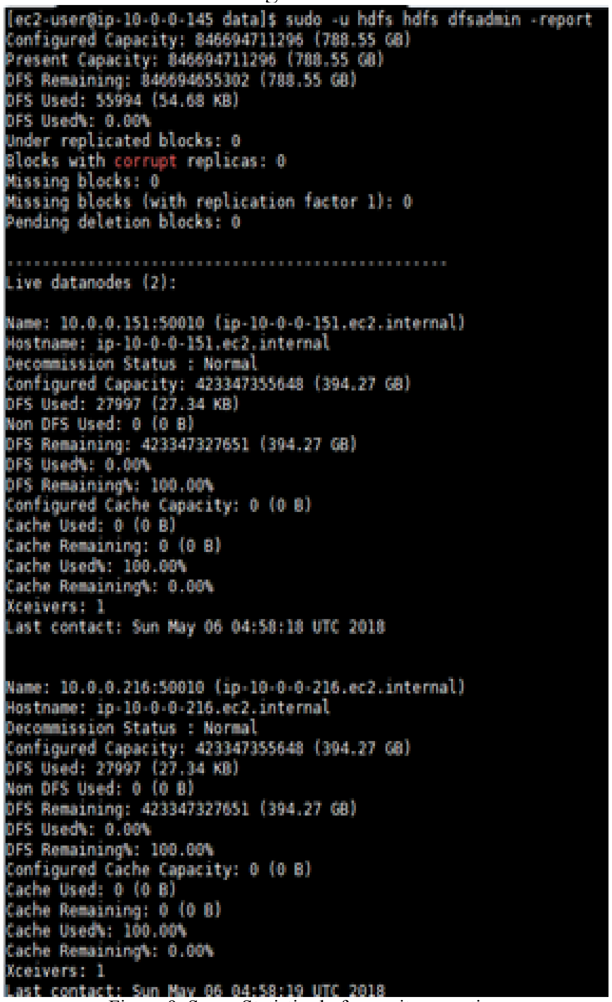


Figure 8 HDFS Write Operation Replication Factor 1.

V. EVALUATION

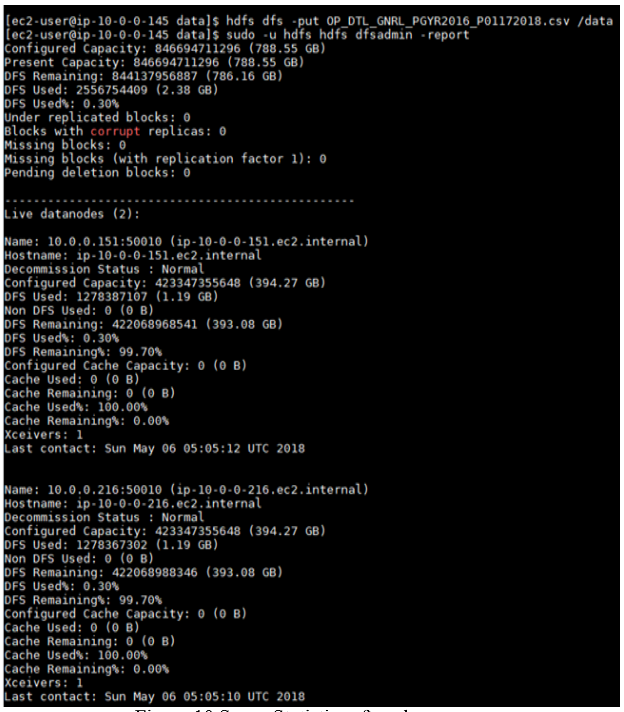
Based on the available datanodes at the datanode pipeline we need to make the script to work on. If the datanodes are exactly one , I mean after having number of failures and the datanode replacements are not happening because of less availability of datanodes at the pool , then we need to the execute the script before proceeding further with write operation. If it is exactly equal to one then we need to invoke the process to clear up the space. Before processing the write the operation the space is 788.55GB. Please refer the statistics as shown in the fig 9



```
[ec2-user@ip-10-0-0-145 data]$ sudo -u hdfs hdfs dfsadmin -report
Configured Capacity: 846694711296 (788.55 GB)
Present Capacity: 846694711296 (788.55 GB)
DFS Remaining: 846694655302 (788.55 GB)
DFS Used: 55994 (54.68 KB)
DFS Used%: 0.00%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0
Pending deletion blocks: 0

-------------------------------------------------
Live datanodes (2):

Name: 10.0.0.151:50010 (ip-10-0-0-151.ec2.internal)
Hostname: ip-10-0-0-151.ec2.internal
Decommission Status : Normal
Configured Capacity: 423347355648 (394.27 GB)
DFS Used: 27997 (27.34 KB)
Non DFS Used: 0 (0 B)
DFS Remaining: 423347327651 (394.27 GB)
DFS Used%: 0.00%
DFS Remaining%: 100.00%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Sun May 06 04:58:18 UTC 2018


Name: 10.0.0.216:50010 (ip-10-0-0-216.ec2.internal)
Hostname: ip-10-0-0-216.ec2.internal
Decommission Status : Normal
Configured Capacity: 423347355648 (394.27 GB)
DFS Used: 27997 (27.34 KB)
Non DFS Used: 0 (0 B)
DFS Remaining: 423347327651 (394.27 GB)
DFS Used%: 0.00%
DFS Remaining%: 100.00%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Sun May 06 04:58:19 UTC 2018
```

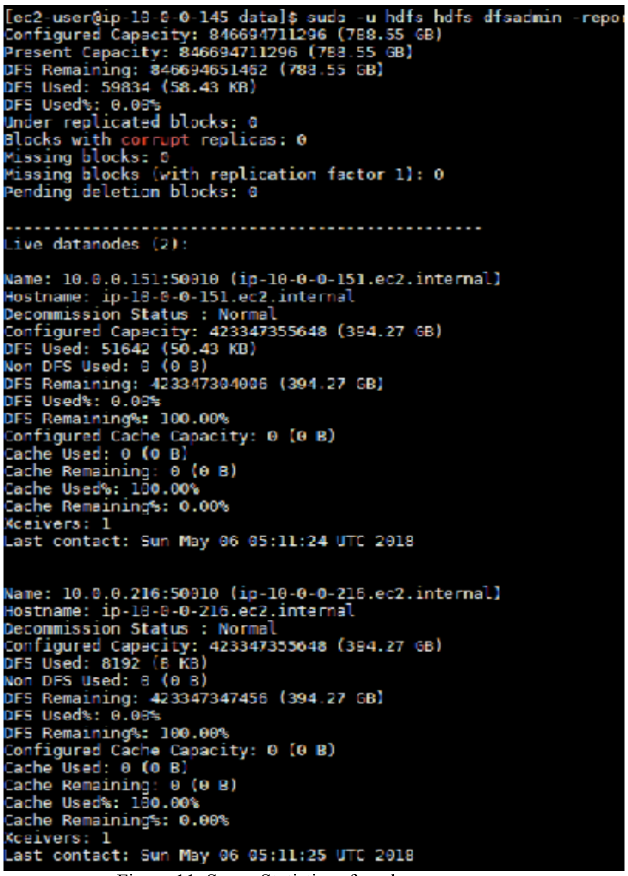
Figure 9. Space Statistics before write operation

Once the write operation done the memory used is 2.38GB. So it is showing as remaining memory. After the /tmp folder clean up operation that will get added to pool memory. This is how we can free up some space and make it available to pool space. We can observe the same behaviour in the fig.9 and fig10 and fig 11. Graph 2 is showing that the space used and getting added to pool memory.



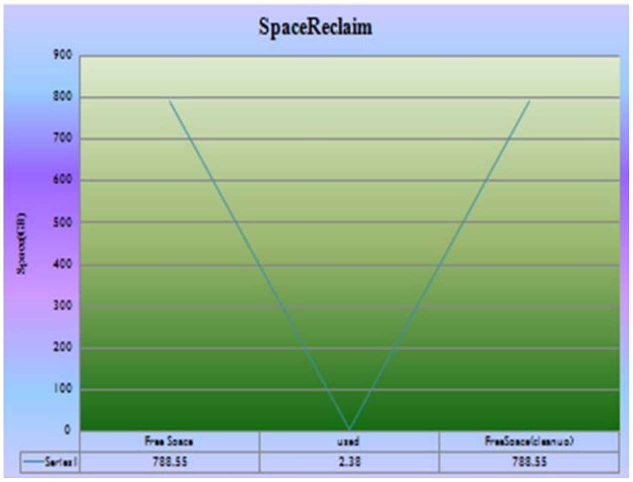
```
[ec2-user@ip-10-0-0-145 data]$ hdfs dfs -put OP_DTL_GNRL_PGYR2016_P01172018.csv /data
[ec2-user@ip-10-0-0-145 data]$ sudo -u hdfs hdfs dfsadmin -report
Configured Capacity: 846694711296 (788.55 GB)
Present Capacity: 846694711296 (788.55 GB)
DFS Remaining: 844137956887 (786.16 GB)
DFS Used: 2556754409 (2.38 GB)
DFS Used%: 0.30%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0
Pending deletion blocks: 0

-------------------------------------------------
Live datanodes (2):

Name: 10.0.0.151:50010 (ip-10-0-0-151.ec2.internal)
Hostname: ip-10-0-0-151.ec2.internal
Decommission Status : Normal
Configured Capacity: 423347355648 (394.27 GB)
DFS Used: 1278387107 (1.19 GB)
Non DFS Used: 0 (0 B)
DFS Remaining: 422068968541 (393.08 GB)
DFS Used%: 0.30%
DFS Remaining%: 99.70%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Sun May 06 05:05:12 UTC 2018


Name: 10.0.0.216:50010 (ip-10-0-0-216.ec2.internal)
Hostname: ip-10-0-0-216.ec2.internal
Decommission Status : Normal
Configured Capacity: 423347355648 (394.27 GB)
DFS Used: 1278367302 (1.19 GB)
Non DFS Used: 0 (0 B)
DFS Remaining: 422068988346 (393.08 GB)
DFS Used%: 0.30%
DFS Remaining%: 99.70%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Sun May 06 05:05:10 UTC 2018
```

Figure 10 Space Statistics after cleanup process



```
[ec2-user@ip-10-0-0-145 data]$ sudo -u hdfs hdfs dfsadmin -repo
Configured Capacity: 846694711296 (788.55 GB)
Present Capacity: 846694711296 (788.55 GB)
DFS Remaining: 846694651462 (788.55 GB)
DFS Used: 59834 (58.43 KB)
DFS Used%: 0.00%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0
Pending deletion blocks: 0

-------------------------------------------------
Live datanodes (2):

Name: 10.0.0.151:50010 (ip-10-0-0-151.ec2.internal)
Hostname: ip-10-0-0-151.ec2.internal
Decommission Status : Normal
Configured Capacity: 423347355648 (394.27 GB)
DFS Used: 51642 (50.43 KB)
Non DFS Used: 0 (0 B)
DFS Remaining: 423347304006 (394.27 GB)
DFS Used%: 0.00%
DFS Remaining%: 100.00%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Sun May 06 05:11:24 UTC 2018


Name: 10.0.0.216:50010 (ip-10-0-0-216.ec2.internal)
Hostname: ip-10-0-0-216.ec2.internal
Decommission Status : Normal
Configured Capacity: 423347355648 (394.27 GB)
DFS Used: 8192 (8 KB)
Non DFS Used: 0 (0 B)
DFS Remaining: 423347347456 (394.27 GB)
DFS Used%: 0.00%
DFS Remaining%: 100.00%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Sun May 06 05:11:25 UTC 2018
```

Figure 11. Space Statistics after cleanup process



	Free Space	used	FreeSpace(cleanup)
Series1	788.55	2.38	788.55

Graph.2. Space reclaiming after cleanup process.

VI. CONCLUSION

HDFS will use the pipeline process to write the data into blocks. Namenode will send the available blocks list so that pipeline will be created based on the datanodes having the empty blocks. We can customize the datanode replacement policy in case of any datanode failure in the pipeline process using configuration parameters. In this process write process will be resumed even though there are less number of datanodes i.e, even having single datanode. As shown in the above results we proved that acting on /tmp ,trash and snapshot folder with hdfs delete oparation we can clain some space and make it available to pool memory so that the hdfs write pipeline will be resumed with maximum number of datanodes (datablocks). This property is used only if the failure property is enabled. As per the best effort propertty policy the write operation should continue irrespective of the datanode replacement. Suppose the datanode replacement fails, the write operation will be failed and in the programming view it will raise an exception. If the replacement is successful then the write operation will be continued with the remaining datanodes. There is always writing to datanodes if we set this property to true irrespective of the number of datanodes, hence it increases the probability of data loss. Suppose the datanode replacement fails, as per the existing architecture if the datanode replacement fails the write operation will be failed. As per the new process which we have provided the datanode replacement will not be failed because this process is patching up the some space to the free pool. If the datanode replacement is true then the write operation will be resumed as usual. So this paper addresses on how to avoid datanode replacement failure, so that we can eliminate data loss.

REFERENCES

[1] Suresh, G., Narayana, K.L., Kedar Mallik, M., Srinivas, V., Jagan Reddy, G. " Processing & characterization of LENS™ deposited Co-Cr-W alloy for bio-medical applications", (2018) International Journal of Pharmaceutical Research, 10 (1), pp. 276-285.

[2] Study of HADOOP, Bhawana Sahare, Ankit Naik, Kavita Patel, International Journal of Computer Science Trends and Technology , IJCST, Vol. 2, Issue-6, ISSN: 2347-8578.

[3] The Hadoop Distributed File System Balancing Portability and Performance , Jeffrey Shafer, Scott Rixner, and Alan L. Cox, Rice University, Houstan, TX.

[4] Distributed Data-Parallel Programs from Sequentila Buildings Blocks, Michael Isard, Mihai Budiu, Yuan Yu.

[5] An Experimental Evaluation of Performance of a Hadoop Cluster on Replica Management. Muralikrishnan Ramane, Sharmila Krishnamoorthy, Sasikala Gowtham.

[6] Hadoop Distributed File System with Cache technology by Archana Kakade and Dr. Suhas Raut, Industrial Science Vol.1,Issue.6/Aug. 2014 ISSN : 2347-5420

[7] J. Dean and S. Ghemawat (2004), "Mapreduce: Simplified Data Processing on Large Clusters". In Proceeding of the 6th Conference on Symposium on operating Systems Design and Implementation (OSDI'04), Berkeley, CA, USA, 2004, pp.137-150.

[8] Shafer J, Rixner S, Cox AL. The Hadoop Distributed Filesystem: Balancing Portability and Performance, in Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2010), White Plains, NY, 2010.

[9] An Efficient dynamic data replication for HDFS using Erasure Coding, Franklin John, Suji Gopinath, Elizabeth Sherly, International Journal of Computer Science and Information Technologies, Vol. 8(2), 2017

[10] A Study of Replica Reconstruction Schemes for Multi-rack HDFS Clusters, Asami Higai, Atsuko Takefusa, Hidemoto Nakada, Masato Oguchi.

[11] Study of Replica Management and High availability in Hadoop Distributed File System, S.Chandra Mouliswaran and Shyam Sathyam, Journal of Science, Vol 2, Issue 2, ISSN 2277-3282.

[12] The Data Recovery File System for Hadoop Clsuter-Review Paper, V.S. Karwande, Dr. S.S.Lomte, Prof, R.A. Auti, International Journal of Computer Science and Information Technologies, Vol. 6, 2015.

[13] Feng Wang et al. Hadoop High Availability through Metadata Replication, IBM China Research Laboratory, ACM, 2009.

[14] Derek Tankel. Scalability of Hadoop Distributed File System, Yahoo developer work, 2010.

[15] "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM" Department of Computer Science Stanford University.

[16] Computer System Architecture, Third Edition, M.Morris Mano

[17] J. Shafer and S Rixner (2010), "The Hadoop distributed file system: balancing portability and performance", In 2010 IEEE International Symposium on Performance Analysis of System and Software (ISPASS2010), White Plains, NY, March 2010. Pp.122-133.

[18] William Stallings (2013), "Computer Organization and Architecture: Desigining for performance", Ninth Edition.

[19] Garry Turkington (2013), Hadoop Beginner's Guide, Learn how to crunch big data to extract meaning from the data avalanche.

[20] SAM R. ALAPATI, Expert Hadoop Administration, Managing, Tuning and Securing Spark, YARN and HDFS, Addison wesley data & analytics series, 2017.