International Research Journal of Management, IT & Social Sciences
Available online at https://sloap.org/journals/index.php/irjmis/
Vol. 6 No. 6, November 2019, pages: 1~16
ISSN: 2395-7492
https://doi.org/10.21744/irjmis.v6n6.739

Hadoop Distributed File System Mechanism for Processing of Large Datasets Across Computers Cluster Using Programming Techniques

Nicholas Jain Edwards [a]
David Tonny Brain [b]
Stephen Carinna Joly [c]
Mariana Karry Masucato [d]

Article history:

Received: 09 May 2019
Accepted: 31 July 2019
Published: 07 September 2019

Keywords:

file;
hadoop;
memory;
pipeline;
system;

Abstract

In this paper, we have proved that the HDFS I/O operations performance is getting increased by integrating the set associativity in the cache design and changing the pipeline topology using fully connected digraph network topology. In read operation, since there is huge number of locations (words) at cache compared to direct mapping the chances of miss ratio is very low, hence reducing the swapping of the data between main memory and cache memory. This is increasing the memory I/O operations performance. In Write operation instead of using the sequential pipeline we need to construct the fully connected graph using the data blocks listed from the NameNode metadata. In sequential pipeline, the data is getting copied to source node in the pipeline. Source node will copy the data to next data block in the pipeline. The same copy process will continue until the last data block in the pipeline. The acknowledgment process has to follow the same process from last block to source block. The time required to transfer the data to all the data blocks in the pipeline and the acknowledgment process is almost 2n times to data copy time from one data block to another data block (if the replication factor is n).

Author correspondence:
Edwards, NJ.,
University of Westminster, London, United Kingdom
309 Regent Street, London W1B 2HW T: +44 (0)20 7911 5000
Email address: edwards.nj@ucl.ac.uk

[a] University of Westminster, London, United Kingdom
[b] University of Westminster, London, United Kingdom
[c] SOAS, University of London, London, United Kingdom
[d] University College London, London, United Kingdom

1. Introduction

Hadoop Distributed File system is having some similarities like normal storage system. It sits on the server storage. Apache Hadoop consists of Hadoop core, Hadoop Distributed File System, Hadoop YARN and MapReduce. Hadoop Distributed File System (HDFS) has several unique features which will be allowed for processing of large distributed data sets. Fault tolerance is one feature where the system failure will not affect the entire system processing or transactions since it is maintaining the multiple copies of the same data using the replication strategy. Hadoop core is having java libraries to start the Hadoop system on the operating system. The file system is having NameNode to store metadata of the file system, DataNodes to store the data. Once the Client sends the read or write request it will be received by Namenode and the list of available blocks info will be returned. DataNodes will send heartbeats to NameNode periodically. Based on this report NameNode will decide the available data nodes or dead data nodes from the file system. Job scheduling and resource management will be taken care by Hadoop YARN. YARN is the processing layer of Hadoop. This is the framework for processing distributed applications running on different machines. It contains a resource manager and job scheduler Resource manager instructs the YARN by distributing the resources inside cluster for each application running inside cluster. NodeManager is running on each data node. ResourceManager and NodeManager work together in the cluster. Each application that is running on cluster is associated with ApplicationMaster. The main responsibility of ApplicationMaster is to negotiate with ResourceManager for resources and collaborate with corresponding NodeManagers to execute the tasks. We can have one ResourceManager per cluster and it will initiate the startup of all YARN applications, distributes resources to all applications cluster-wide to all the DataNodes. The scheduler and the ApplicationsManager are the two main components of the ResourceManager. Allocating resources to applications is handled by scheduler in ResourceManager and the scheduler will allocate the resources based on the availability of resources in the cluster. Resource containers are managed by schedulers to allocate the resources. The ApplicationsManager accepts the requests submitted by client and starts the container for execution of the new ApplicationMaster. ResourceManager will create the container for application in which the Application Master runs the application. ResourceManager tracks the heartbeats from the NodeManagers received by each datanode. It will run the scheduler to schedule the resources inside the cluster. ApplicationMasters will send the requests to ResourceManager for resources. It will always monitors the status of the ApplicationMaster and restart the ApplicationMaster upon its failure. Resource Manager will create the container for application where the ApplicationMaster will run. It manages the requests regarding resources from the ApplicationMaster. Once the application completes it deallocates the containers which was initialized for the same application (Ghazi & Gangodkar, 2015; Hua *et al.,* 2014).

In this paper, we will show that how we can improve the performance of the HDFS read operation using the memory organization process like set associative cache technology and write operation using creation of the datanode pipeline using fully connected diagraph technology.

Literature Review

HDFS I/O Operations

a) Read Operation:

HDFS stores data into datanodes and the data is managed by datablocks. A file can be stored at multiple blocks which might not be from the same datanode. That means the file will be scattered across multiple locations of the multiple datanodes. DatNode is running NodeManager daemon for performing YARN functions. The NodeManager starts interacting with global resource manager using heart beats and container status notification. Block size can be controlled by hdfs-site.xml file. It can be 64MB or 128MB or 256 MB. While receiving data from the file system that will be copied to cache location to process the subsequent requests on the same data, i,e. reduction on memory access time. The interaction between main memory segment and cache memory segment is called as DirectMapping where the main memory locations can only be copied into one location in the cache. We have configured a cluster of four virtual machines with the configuration 2GB, 200GB, each system is connected to remaining systems in the network using password less authentication. Each virtual machine was configured with Ubuntu Linux 16.04, Jdk 1.7, Hadoop 2.7 and updated the block size with 128 MB. We have configured three servers of these as datanodes and the one as NameNode. Every DataNode is having cache memory. To test the existing environment we have used input files with different sizes say 10, 20, 30 and 40 in Kbs and stop words file. We have one driver class, mapper class and reducer class. Driver class for adding the files to cache, Mapper for reading the files from cache and reducer class for having the non stop (valuable words) words (Saraladevi *et al.,* 2015; Bende & Shedge, 2016). We have ran the driver class

using two files Input file , stop words file and please find the values at Table 1. These are the results for direct memory mapped memory organization using different size of files 10,20,30 and 40. Please refer Figure 1 for Read Architecture.

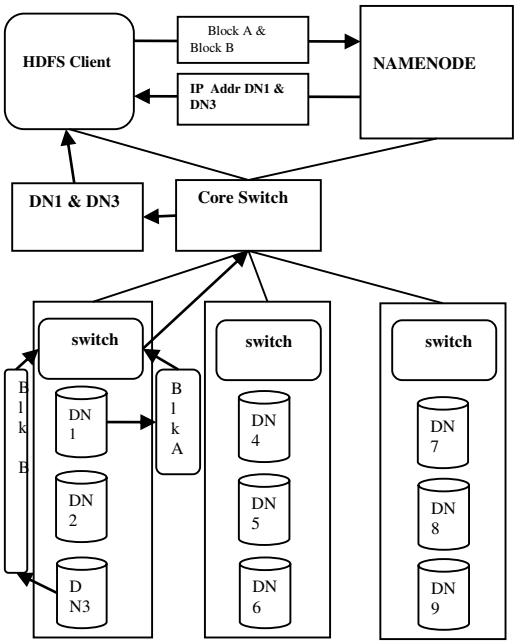

Figure 1. HDFS read architecture

Table 1
File size vs memory access

FileSize(Kb)	Memory Access Time (ms)
10	1724
20	374
30	521
40	633

Please refer Figure 1 for memory access time with respect to number of files. We can optimize the performance of the I/O operation using the set associative cache technology. In this values 10 file size is showing very high value because that is the first time operation started, so it will take some extra time. Apart from this result if you observe the access time is going high as the size of the file is getting increased. The memory access time will be reduced using the set associative cache technology. Please observe the same at Figure 1.

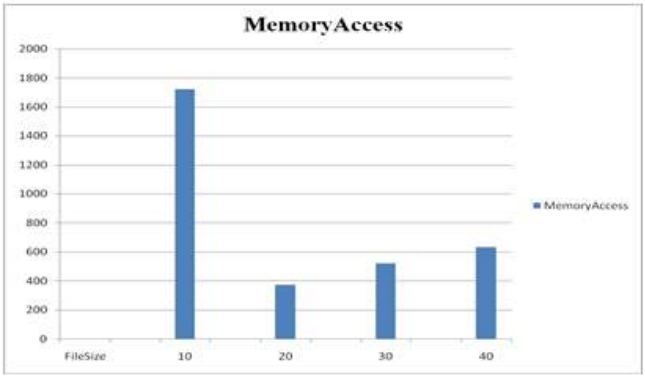

Figure 2. Memory access

b) Write Operation

We have used the existing Hadoop libraries to write the data to distributed File System having different replication factors configured at hdfs-site.xml file. Once the NameNode receives the write request from the HDFS client, it will send the list of available blocks report to Dataoutput stream. The out stream will create pipeline and it will copy the data to source datablock. It will get copied to next datablock from the source datablock of the pipeline. The data will be propagated in the same fashion till the last datablock in the pipeline (Uzunkaya *et al.,* 2015; Jach *et al.,* 2015). The length of the pipeline is equivalent to replication factor which we have configured in the hdfs-site.xml file. Once the data reaches to end as per the replication factor, it sends the acknowledgement back to previous block from where it receives the data. The acknowledgement will be travelled back till the source node of the pipeline. Please find the HDFS write Operation architecture at Figure 2 and acknowledgement at Figure 3. Please find the write access time at Table 2 and you can observe the trend at figure 2.

Table 2
File Size Vs Memory Access

ReplicationFactor	Write Pipeline(ms)
3	389
4	425
5	456
6	487
7	509

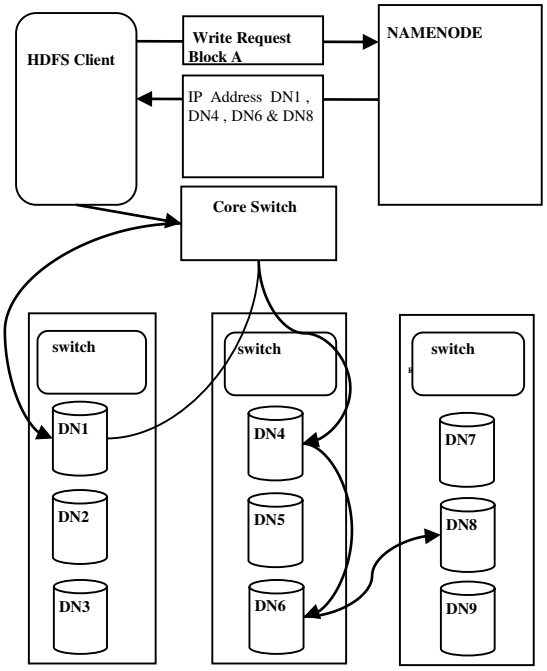

Figure 3. HDFS write operation

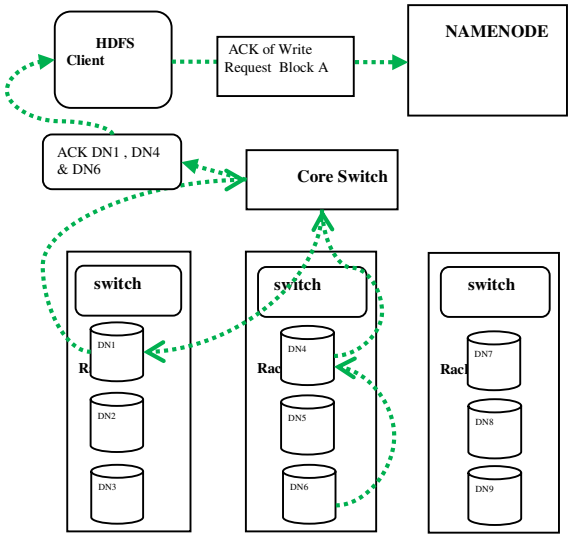

Figure 4. HDFS Write Operation acknowledgement

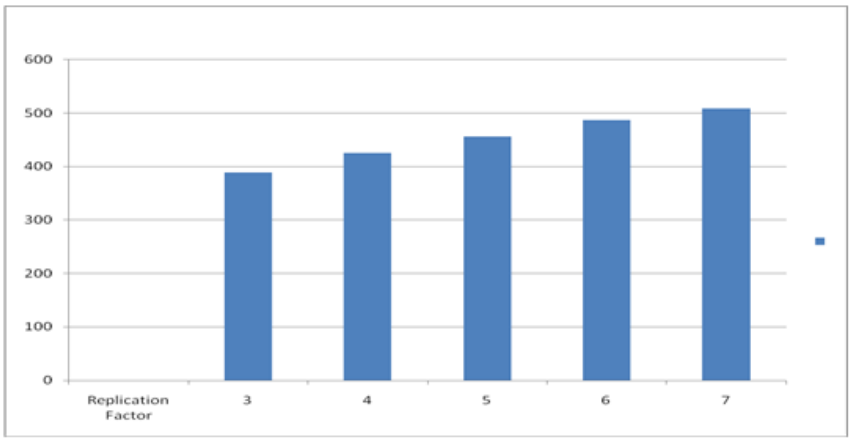

Figure 5. Memory access

2. Materials and Methods

Problem Statement

a) Read Operation
Frequently accessed data will be placed at cache location. Once the entire locations of cache fully occupied with data then there will not be further interaction with the cache to store the new data until there will be minimum one deletion from the cache location. We can say this as swapping the data at cache location with main memory. If the number of swappings increases then it will impact the performance of the read operation negatively. This is one of the problem in the existing environment.

b) Write Operation
When the client wants to write some data to Hadoop Distributed File System, While write operation is in progress the datablock pipeline will be created and the data will be copied to source datablock of the pipeline. From there it will be copied to next datanode in the pipeline and so on until the end of the pipeline. The acknowledgement will be transferred back sequentially till the starting node of the pipeline. Since it is sequential path , the traversal time will be high. If the node breakdowns then the entire pipeline will be reconstructed which will impact the I/O operation performance negatively.

3. Results and Discussions

a) Read Operation

Main Memory frames will be directly mapped to cache location in Direct Memory mapped technology. In this case the swappings are mandatory and in high frequency. So why can't we use extra locations (double or 4 times like power of 2 values) compared to main memory locations at cache. This is what is called Set Associative Cache Memory. Each line in 2 -way set associative is having two words in each line where as 4 - way Set associative is having 4 words in each line. In Direct Memory mapped technology it is having only one word at each line. N- way Set Associative is having n words at each line. The hit ratio will be high in N-way set associative compared to Direct Memory Mapped technology, since we can store lot of words at a time in cache (Anuradha, 2015; Cho *et al.*, 2014).

b) Write Operation

In write operation, we need to use the fully connected digraph datanode technology. That means whatever the list of datablocks we will get from the Namespace (NameNode), will be connected using fully connected directed graph technology. As per the existing environment, the datablocks will be connected sequentially. Once we get the

first datablock the data stream will be initiated. The data will be copied to first datablock. Then the data will be copied to second datablock from the first datablock. The same process will be continued till the last datablock in the list. But the data stream will not be connected till the last block, instead of that each data block will copy the data to follower datablock. The acknowledgement will be propagated back from last block till the starting data block. The data copy operation and the acknowledgement process is running through the sequential process. And if there is any issue in the sequential process, then the whole pipeline will go down. Then the new pipeline will get created using the new datanode (new datanode from the pool) and the old data will be copied. In fully connected directed graph each node is connected to remaining all nodes. So at a time data will be copied to all datablocks and this concept is applicable to acknowledgement as well. In the sequential process the effort will be proportional to replication factor. Where as in the fully connected digraph technology the effort is independent of number of datablocks in the pipeline, i.e, it is independent of replication factor (Wang *et al.,* 2013; Saranya *et al.,* 2015).

Implementation

We have defined the interface which is having map for key, value pairs. One more concrete class needs to implement the interface and having the parameters to define the size of the set associative cache memory. Used this concrete class to read the files i.e, it can store number of files based on the size which we have provided to the class constructor. Using driver, mapper and reducer classes we can read the values from the cache. Here we have provided three inputs, one is input text file, second one is output path, storing the output of the program and third one is file containing the stop words which we will distribute to all the datanodes. Please find the algorithm as follows

1: Read the value from the File system.
2: Keep the value in cache.
3: Read the value from cache system
4: If it is not finding out the value in cache, read the value from file system and copy the same to cache.
5: Please use the Least Recently Used Algorithm to keep the data inset associative cache.

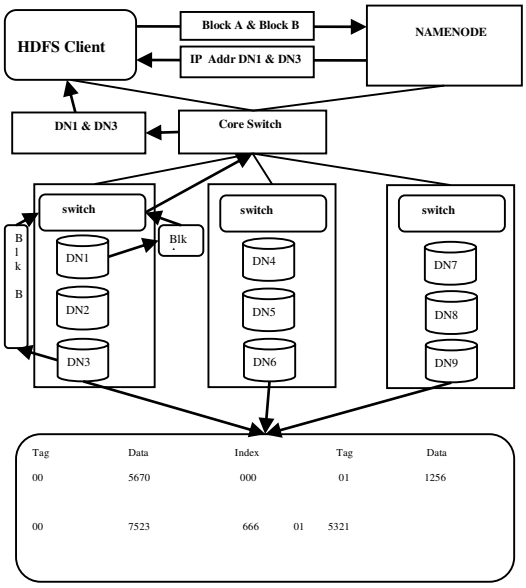

Figure 6. HDFS with Set Associative Cache Memory

Refer Figure 6 for proposed architecture. Hadoop framework provides the cache libraries on all datanodes. We need to integrate the Set associative cache memory implementation inside each datanode. We can access the files using Hadoop APIs (Zhao *et al.,* 2014). We can use these files either in mapper or in reducer job. Cache memory is having two sets of data along with tag info. If the address is 00000 this is the tag followed by index as shown in the figure 9

this will match for tag using 00 address followed by the index 000 , now the data is 5670. In this set associative cache memory we can store two sets of data for the same tag value like 01 000 and 00 000. Cache system checks whether requested file is available in cache local memory or not. If yes then request is fulfilled by cache. If not it will be processes from filesystem and same will be copied to local cache memory for future reference (Liu & Dong, 2012; Uskenbayeva *et al*., 2015).

Table 3 and figure 7 are showing the memory access time for file size 10 and using the associativity in the cache design. Table 4 and figure 8 for file size 20, Table 5 and figure 9 for file size 30, Table 6 and figure 10 for file size 40. While the cache size is getting increased we can observe that the memory access time is going down in almost all the scenarios.

Table 3
Memory access file size 10

Cache Size	FileSize(Kb)	Mmeory Access Time (ms)
1	10	1724
2	10	436
4	10	341
8	10	338
16	10	353
32	10	344

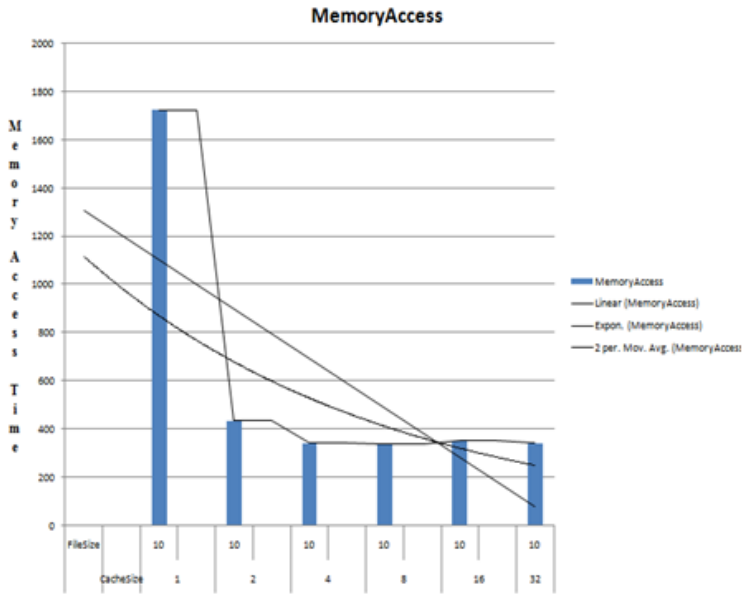

Figure 7. Memory Access Time File Size 10

Table 4
Memory Access Time FileSize 20

CacheSize	FileSize(Kb)	Memory Access Time(ms)
1	20	374
2	20	354
4	20	364
8	20	321
16	20	330
32	20	363

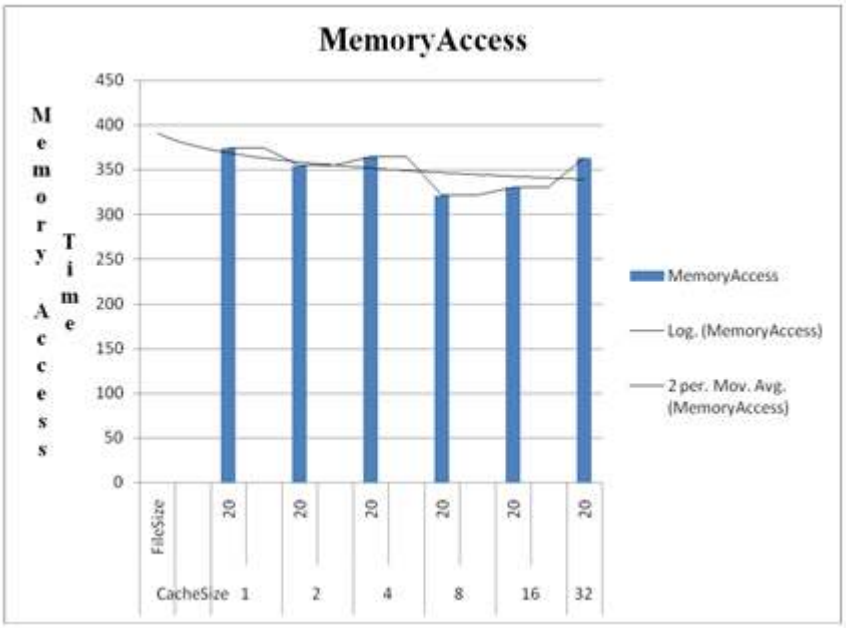

Figure 8. *Memory Access Time FileSize 20*

Table 5
Memory Access File Size 30

CacheSize	FileSize(Kb)	Memory Access Time(ms)
1	30	521
2	30	442
4	30	332
8	30	352
16	30	342
32	30	339

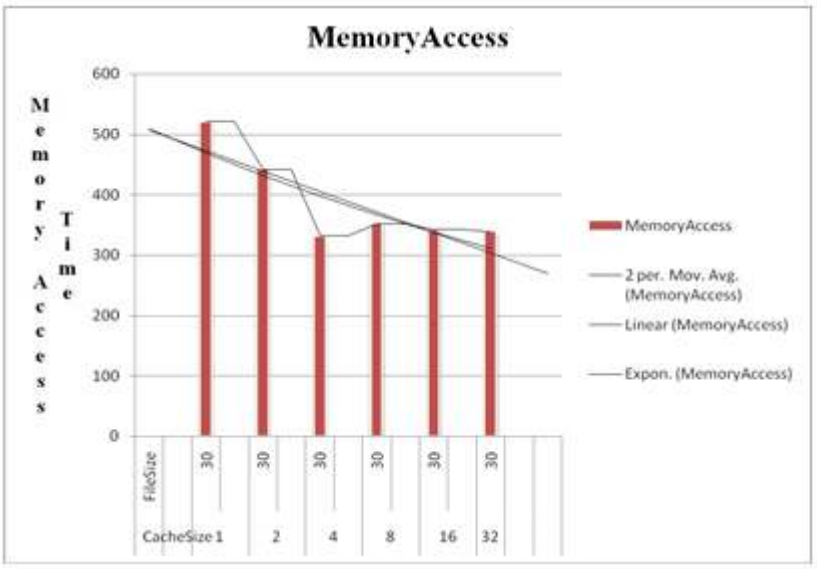

Figure 9. Memory Access Time FileSize 30

Table 6
Memory Access File Size 40

CacheSize	FileSize(Kb)	Memory Access Time(ms)
1	40	633
2	40	346
4	40	346
8	40	351
16	40	349
32	40	341

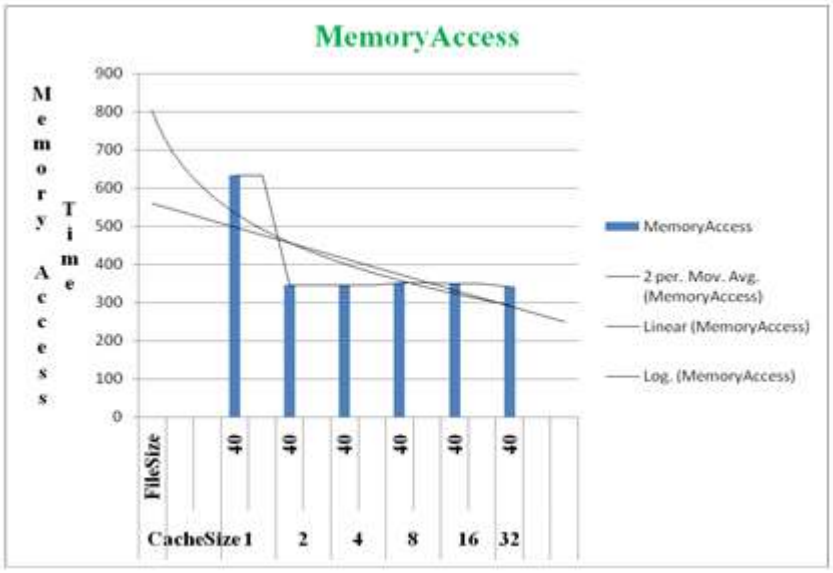

Figure 10. Memory Access Time FileSize 40

Write Operation

We have configured 10 virtual machines with the configuration of 4GB RAM, 400GB HDD and Ubuntu 16.04, JDK 1.8, Maven 3.0.3, Hadoop 2.7.3 on each virtual machine. Using the SSH keys, we have connected one machine to all other machines using password less authentication [15]. As soon as end user sends the write operation request to NameNode, it will return the list of available blocks. We have implemented the class which will retrieve the list of blocks from the NameNode and creates the fully connected graph Using the Dijkstras algorithm we have calculated shortest path from source node to all other nodes in the network (fully connected graph) and initiated the initWriteOperation method from the hadoop libraries on each pair of datablocks. In all the pairs source node is common, where as second one is distinct. Because of the new topology we have added the complexity of O (E Log V) to the existing complexity in place of creation of pipeline in the existing architecture. E is the number of edges and the V is number of vertices, which is equal to replication factor.

Please find the algorithm:

1: Write request from the HDFS Client

2: Request received by NameNode NameSpace.

3: NameNode returns the IP Addresses of the datanodes i.e, block list info.

4: Create the DirectedGraph using the list of available blocks from the step 3.

```
for (i=0; i<=n; i++) {
  for (j=i; j<=n; j++) {
     if (i==j)
     CONTINUE;
     directedGraph.addEdge(i,j);
     }  }
```

5: Declare one of the block as source node and findout the shortest path from source node to all other nodes using Dijkstras algorithm.

6: call the initWriteOperation method from the Hadoop libraries.

Please find the fully connected graph full view at the Fig 5. HDFS client will send write request to Namenode, and the Namenode will respond back with IP addresses of DN1, DN4, DN6 and DN8. As per the existing architecture the pipeline sequence is DN1, DN4, DN6 and DN8. The data will be written to DN1 from the client and DN1 will act as a source of the pipeline. This will write down the block of data to DN4. The block of data from DN4 to DN6 and from DN6 to DN8.

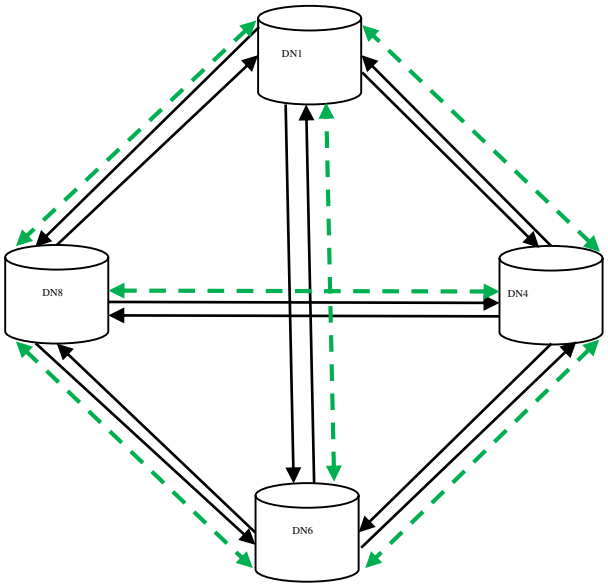

Figure 11. Fully Connected Digraph DataNode Networking technology

If there is any issue in the datanodes available on the pipeline then there is interruption in the data copy as per the replication factor. Then the Namenode has to adjust the list of IP addresses followed by reconstruction of the pipeline to resume the write operation. This is because of having sequential pipeline. And one more issue is time to copy the datablock to all locations as per the replication factor (O'Driscoll *et al.,* 2013; Lee *et al.,* 2014).

As per the proposed architecture Fig 5, the data block will be copied to source node of the pipeline. Here the source datanode will be connected to all other IP addresses parallelly. That means DN1 is connected to DN4, DN6 and DN8 parallelly. As soon as the datablock is available at DN1, the same will be copied to all other datanodes parallelly from DN1. Here the datanodes are connected in fully connected network topology.

As per the proposed solution we have modified the Hadoop library so that it will act on the list of blocks information coming from the namenode. Using the list of blocks info we need to create the directed graph by adding the each block as one vertex of the graph. Degree of the graph is the replication factor. As per the Fig 5, source datanode is DN1, so here the vertex is DN1. Using the Dijkstra's algorithm we can findout the shortest path between the pair of vertices like DN1 -> DN4, DN1->DN6, DN1->DN8. The data copy operation will follow the same path which we get from the Dijkstra's algorithm. Instead of creating the sequential pipeline we need to create couple of arrays so that we can traverse from the first element of the array to second element of the array. Second element is nothing but the last element of the array. If n is the replication factor then there will be n-1 arrays we need to create to copy the data from source node to remaining datanodes as per the replication factor. If there is any network failure and datanode failure using the Dijkstras algorithm we need to re create the shortest from DN1 to DN4, since we have connected using the fully connected digraph datanode network topology.

Table 7
Sequential Vs FullyConnected: Repl 4

FileSize(Kb)	Sequential Pipeline(ms)	FullyConnected (ms)
10	316	72
20	339	74
30	370	82
40	410	93
50	412	93
60	417	98
70	417	113
80	567	119
90	628	123
100	667	134

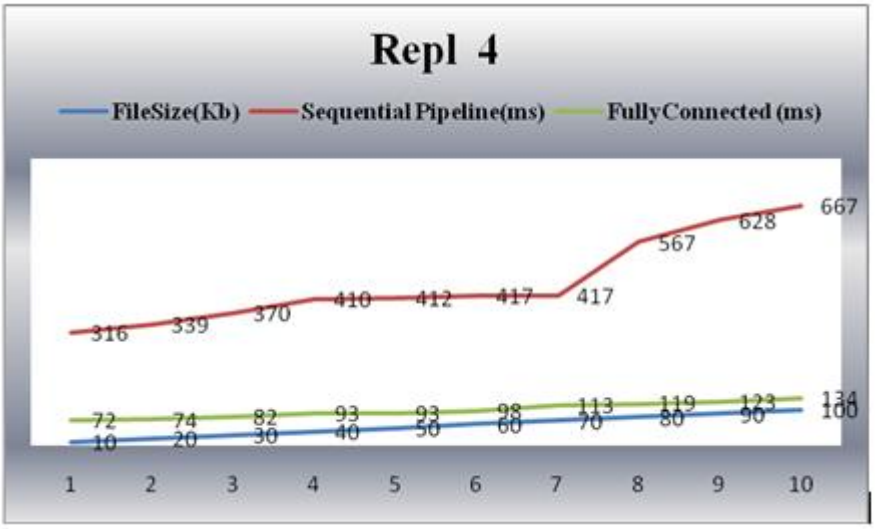

Figure 12. Sequential vs Fully Connected: Repl 4

Table 8
Sequential Vs FullyConnected: Repl 5

FileSize(Kb)	Sequential Pipeline(ms)	FullyConnected (ms)
10	329	71
20	333	69
30	362	74
40	403	65
50	414	79
60	464	82
70	539	75
80	552	83
90	635	76
100	674	89

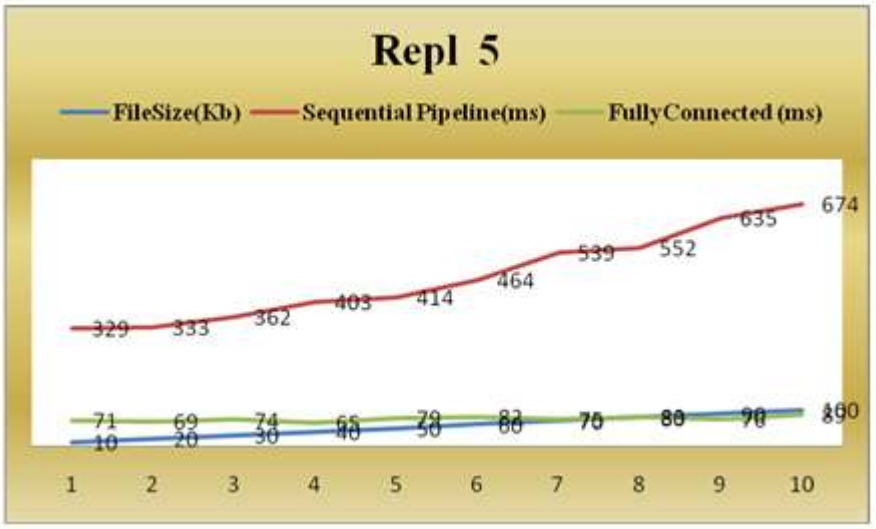

Figure 13. Sequential vs Fully Connected: Repl 5

Table 9
Sequential Vs FullyConnected: Repl 6

FileSize(Kb)	Sequential Pipeline(ms)	FullyConnected (ms)
10	336	74
20	339	73
30	375	76
40	406	82
50	411	82
60	469	95
70	546	102
80	596	96
90	619	107
100	638	124

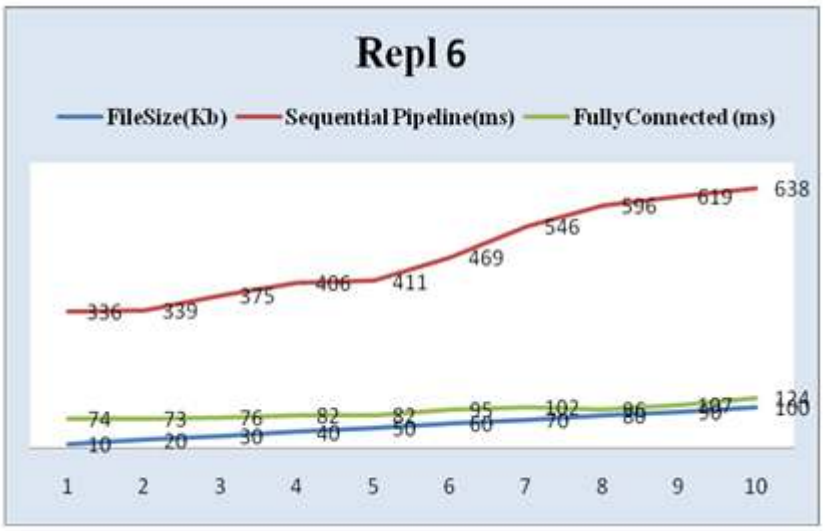

Figure 14. Sequential vs Fully Connected: Repl 6

Table 10
Sequential Vs FullyConnected: Repl 7

FileSize(Kb)	Sequential Pipeline(ms)	FullyConnected (ms)
10	328	82
20	331	79
30	387	85
40	421	87
50	449	75
60	467	92
70	550	89
80	607	96
90	658	105
100	697	109

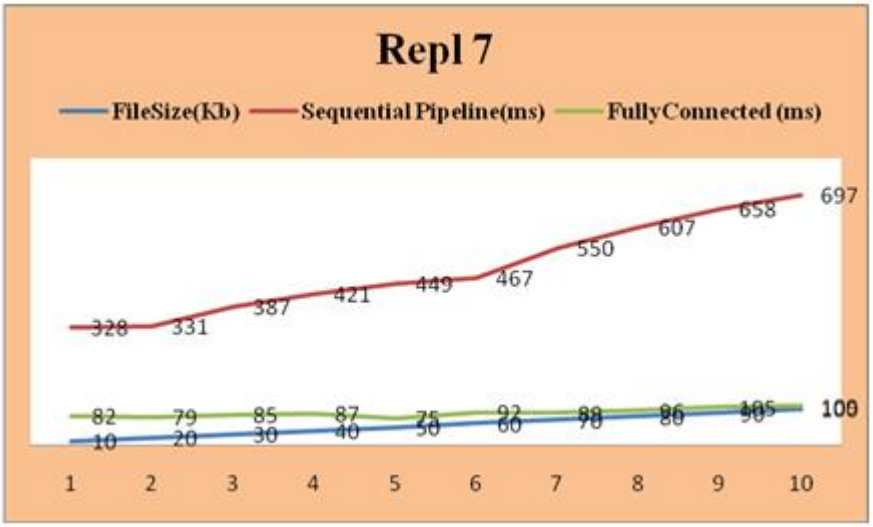

Figure 15. Sequential vs Fully Connected: Repl 7

Table 7, figure 7 are showing the results for sequential pipeline write access vs fully connected datanode write access for the replication factor 4. We can observe that there is huge performance improvement with the proposed architecture. Table 8 and figure 8, Table 9 and figure 9, Table 10 and figure 10 are showing the same results for replications factors 5,6 and 7 respectively.

4. Conclusion

In this paper we have proved that the HDFS I/O operations performance is getting increased by integrating the set associativity in the cache design and changing the pipeline topology using fully connected digraph network topology. In read operation, since there are huge number of locations (words) at cache compared to direct mapping the chances of miss ratio is very low, hence reducing the swapping of the data between main memory and cache memory. This is increasing the memory I/O operations performance. In Write operation instead of using the sequential pipeline we need to construct the fully connected graph using the data blocks listed from the NameNode metadata. In sequential pipeline, the data is getting copied to source node in the pipeline. Source node will copy the data to next datablock in the pipeline. The same copy process will continue until the last datablock in the pipeline. The acknowledgement process has to follow the same process from last block to source block. The time required to transfer the data to all the datablocks in the pipeline and the acknowledgement process is almost 2n times to data copy time from one datablock to another datablock(if the replication factor is n). In this architecture the total amount of time for data transfer and acknowledgement process is always dependent to replication factor. If there is any network failure in the pipeline then write process has to get in touch with NameNode to get the new datablock to reconstruct the pipeline. In this paper we have proved that the data is getting copied to all the datablocks in very less time compared to sequential pipeline write process. In this architecture the total amount of time for transferring data and acknowledgement is always independent to replication factor. So we can increase the performance. In case of any network failure we can reach the datanode from the other paths since each datanode is connected with remaining n-1 datanodes if n is the replication factor. This is how we can increase the performance of the write operation using the fully connected digraph network topology.

Conflict of interest statement
The authors declared that they have no competing interest.

Statement of authorship
The authors have a responsibility for the conception and design of the study. The authors have approved the final article.

Acknowledgments
We thank the editor of IRJMIS for their valuable time.

References

Anuradha, J. (2015). A brief introduction on Big Data 5Vs characteristics and Hadoop technology. *Procedia computer science*, *48*, 319-324. https://doi.org/10.1016/j.procs.2015.04.188

Bende, S., & Shedge, R. (2016). Dealing with small files problem in hadoop distributed file system. *Procedia Computer Science*, *79*, 1001-1012. https://doi.org/10.1016/j.procs.2016.03.127

Cho, J. Y., Jin, H. W., Lee, M., & Schwan, K. (2014). Dynamic core affinity for high-performance file upload on Hadoop Distributed File System. *Parallel Computing*, *40*(10), 722-737. https://doi.org/10.1016/j.parco.2014.07.005

Ghazi, M. R., & Gangodkar, D. (2015). Hadoop, MapReduce and HDFS: a developers perspective. *Procedia Computer Science*, *48*, 45-50. https://doi.org/10.1016/j.procs.2015.04.108

Hua, X., Wu, H., Li, Z., & Ren, S. (2014). Enhancing throughput of the Hadoop Distributed File System for interaction-intensive tasks. *Journal of Parallel and Distributed Computing*, *74*(8), 2770-2779. https://doi.org/10.1016/j.jpdc.2014.03.010

Jach, T., Magiera, E., & Froelich, W. (2015). Application of HADOOP to store and process big data gathered from an urban water distribution system. *Procedia Engineering*, *119*, 1375-1380. https://doi.org/10.1016/j.proeng.2015.08.988

Lee, C. W., Hsieh, K. Y., Hsieh, S. Y., & Hsiao, H. C. (2014). A dynamic data placement strategy for hadoop in heterogeneous environments. *Big Data Research*, *1*, 14-22. https://doi.org/10.1016/j.bdr.2014.07.002

Liu, K., & Dong, L. J. (2012). Research on cloud data storage technology and its architecture implementation. *Procedia Engineering*, *29*, 133-137. https://doi.org/10.1016/j.proeng.2011.12.682

O'Driscoll, A., Daugelaite, J., & Sleator, R. D. (2013). 'Big data', Hadoop and cloud computing in genomics. *Journal of biomedical informatics*, *46*(5), 774-781. https://doi.org/10.1016/j.jbi.2013.07.001

Saraladevi, B., Pazhaniraja, N., Paul, P. V., Basha, M. S., & Dhavachelvan, P. (2015). Big Data and Hadoop-A study in security perspective. *Procedia computer science*, *50*, 596-601. https://doi.org/10.1016/j.procs.2015.04.091

Saranya, S., Sarumathi, M., Swathi, B., Paul, P. V., Kumar, S. S., & Vengattaraman, T. (2015). Dynamic Preclusion of Encroachment in Hadoop Distributed File System. *Procedia Computer Science*, *50*, 531-536. https://doi.org/10.1016/j.procs.2015.04.027

Uskenbayeva, R., Im Cho, Y., Temirbolatova, T., & Kozhamzharova, D. (2015). Integrating of data using the Hadoop and R. *Procedia Computer Science*, *56*, 145-149. https://doi.org/10.1016/j.procs.2015.07.187

Uzunkaya, C., Ensari, T., & Kavurucu, Y. (2015). Hadoop ecosystem and its analysis on tweets. *Procedia-Social and Behavioral Sciences*, *195*, 1890-1897. https://doi.org/10.1016/j.sbspro.2015.06.429

Wang, L., Tao, J., Ranjan, R., Marten, H., Streit, A., Chen, J., & Chen, D. (2013). G-Hadoop: MapReduce across distributed data centers for data-intensive computing. *Future Generation Computer Systems*, *29*(3), 739-750. https://doi.org/10.1016/j.future.2012.09.001

Zhao, J., Wang, L., Tao, J., Chen, J., Sun, W., Ranjan, R., ... & Georgakopoulos, D. (2014). A security framework in G-Hadoop for big data computing across distributed Cloud data centres. *Journal of Computer and System Sciences*, *80*(5), 994-1007. https://doi.org/10.1016/j.jcss.2014.02.006