International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Hadoop Distributed File System Write Operations

Renukadevi Chuppala*[1], Dr. B. Purnachandra Rao[2]

Submitted: 18/05/2023 **Revised**: 27/06/2023 **Accepted**: 05/07/2023

Abstract: Hadoop is an open-source version of the MapReduce Framework for distributed processing. A Hadoop cluster possesses the capacity to manage substantial volumes of data. Hadoop utilizes the Hadoop Distributed File System, also known as HDFS, to manage large amounts of data. The client will transfer data to the DataNodes by retrieving block information from the NameNode. The pipeline configuration will connect the DataNodes that store the blocks. If a DataNode or network fails during the data writing process, the pipeline will remove the failed DataNode. The pipeline will add the new DataNode based on the existing DataNodes in the cluster. If there is a scarcity of spare nodes in the cluster, customers may encounter an abnormally high frequency of pipeline failures due to the inability to locate additional DataNodes or replacements. In the event of a network failure, the data packet is unable to reach the target DataNode due to their interconnected pipeline structure. Interconnecting each DataNode with every other DataNode ensures that multiple pathways are available through other DataNodes, thereby preventing network failure. The copy operation will take longer due to pipeline connectivity. On the other hand, a direct connection between a DataNode and all other DataNodes significantly reduces the time required, as the datapacket doesn't have to traverse through all other DataNodes to reach the final DataNode. This paper presents the utilization of the A* algorithm to enhance the performance of write operations in the Hadoop Distributed File System.

Keywords: *Hadoop Distributed File System (HDFS),NameNode , DataNode, Replica, Rackawareness, Data Packet, Data Packet Transfer Time, Pipeline, Fully Connected Digrapgh Network Topology, A* algorithm.*

1. Introduction

To store very large datasets reliably, have a high degree of fault tolerance, and stream those data sets at high bandwidth for user applications running on commodity hardware, the Hadoop Distributed File System (HDFS) [1] is a distributed file system designed to be deployed on inexpensive commodity hardware. The system uses a master/slave architecture, referring to the master as the NameNode and the slaves as the DataNodes. Although HDFS and current distributed file systems are substantially different, they share many commonalities. A distributed file system comprises a single NameNode and numerous data nodes, ensuring availability and reliability through the maintenance of multiple data replicas. During periods of intense activity, the NameNode and DataNode engage in extensive communication while performing file read or write operations, resulting in a decline in performance. The system's NameNode acts as the primary server. The NameNode is responsible for managing the file system's metadata, including the file structure and block placement for each file. Additionally, it carries out file system tasks such as terminating, initiating, and changing the names of files and directories. The blocks that each storage node's Datanode process manages are under the authority of a master NameNode process running on a different host. To meet the growing storage demands of Hadoop [1], MapReduce [5], Dryad [10], and HPCC (High-Performance Computing Cluster) [15] frameworks, disk-based file systems are the most suitable option. The reason for this is that the data is near, which enables tasks to be executed more efficiently. When working with extensive datasets, executing processes can reduce network traffic and enhance throughput. The Hadoop distributed file system (HDFS) [6] can store immense amounts of data because of the significant number of nodes in each cluster. There is a requirementto incur a time penality when obtaining or storing data in a DataNode. A fully connected network is a communication network in which each of the nodes is connected to each other. Furthermore, data is replicated across multiple nodes using various methods to enhance work completion time. A completely linked network is a type of communication network where every node is directly connected to every other node. It is referred to as a full graph in the field of graph theory. Each DataNode in the cluster will have many alternative paths to connect to other DataNodes. The number of different paths will be equal to the replication factor. In the event of a failure in the existing path between two DataNodes, we can utilize the count of alternative paths available from one DataNode to another DataNode. The copy operation will be faster as we can use parallel copy processes between the datanodes. This is done by connecting the starting datanode directly to all other datanodes involved in the replication process. The proposed method has two main advantages: it reduces the time required for copy operations between datanodes and provides multiple alternative paths to reach the target datanode in case of network failure. In the current process,we configure the parameters dfs.client.block.write.replace-datanode-on-failure.enable and dfs.client.block.write.replace-datanode-on-failure.policy. Once we implement the suggested architecture, there will be no need to modify the DataNodes in the event of a network failure. Instead, we can access the DataNode using an alternative method different routes among the data nodes. This paper includes concepts such as conducting a literature survey on the current mechanism used for HDFS memory operation, describing the components of HDFS, identifying the issues in the existing architecture, proposing a new architecture that utilizes a fully connected digraph DataNode

[1] *Western Union , Financial Services, CA, USA*
[2] *Sr Solution Architect , HCL Technologies, Bangalore, India*
** Corresponding Author Email: renu.chuppala@gmail.com*

network topology, implementing the proposed architecture, and evaluating it through simulation results. This paper introduces a mechanism to minimize the duration of the copy operation to DataNodes, which involves replicating datapackets.

2. HDFS with Linear DataNode Connectivity

When a client requests to read HDFS data, it first contacts the NameNode to collect data regarding the initial blocks of the file it wants to read. The NameNode determines the positions of all DataNodes that store a copy of the original blocks, organizing the DataNodes according to their accessibility to the client. In order to begin writing the data blocks to the pipeline's first DataNode, the client connects to it. The initial DataNode will establish a connection with the subsequent DataNode in the sequence and transmit the data blocks to it as soon as they are received. In the pipeline, the second DataNode then makes a connection and sends the data to the next DataNode. A confirmation packet is delivered over the DataNode pipeline to the client when all three replicas have been entirely written to, letting them know that the block has been successfully written to every node. The client will commence composing the subsequent block at this juncture.The block is committed by the NameNode and recorded as "written" in the edit log once all block replicas have been written. The file is closed by the client after it has finished writing data to it. This requires that all the blocks of the file have been replicated the minimum necessary number of times. The client can encounter a delay in closing the file if there are any DataNode errors during the procedure. The client informs the NameNode that the file writing procedure has been completed successfully. The block replicas are written asynchronously. The client is not required to transmit the data blocks it is sending to every DataNode. The NameNode assigns the data to a particular DataNode from the received list. That DataNode's next task is to send the data blocks to the other DataNodes in the pipeline. In addition, each DataNode will keep a checksum for each data block that it holds. This block's checksum is checked after it has been read to make sure it is accurate and uncorrupted. The block reports that the NameNode gets from the DataNodes are the basis for the metadata that it creates. HDFS stores the data blocks in a way that prevents data loss due to the availability of one or more nodes. Hadoop employs automatic block replication to replace any lost blocks. The premise of Hadoop is to bring processing to the data, not the other way around, as it is in traditional database systems. Data replication helps enforce this philosophy by ensuring both availability and data locality.The existing system is using dfs.client.block.write.replace-datanode-on-failure.enable,dfs.client.block.write.replace-datanode-on-failure.policy parameters to maintain the fault tolerance even when the DataNode /network failure while writing or reading data from DataNodes. DFSClient will request for data blocks from the NameNode. Once the client gets the list of datablocks , client will open the Out stream for write operation. Data will be written to nearest DataNode (block) , and this DataNode will be connected to other DataNodes (number of DataNodes based on the replication factor) in pipeline fashion[7-8].If there is a DataNode/network failure issue in the write operation (pipeline), DFSClient will remove the failed DataNode from the pipeline and then resume write operation with the remaining DataNodes. As a result of this operation, the number of DataNodes in the pipeline will go down. The feature is to add new DataNodes to the pipeline.When the number of nodes in the cluster is (cluster size) extremely small, example number of nodes is 3 or less, cluster administrators may want to set the policy dfs.client.block.write.replace-datanode-on-failure.enable to NEVER in the default configuration file (hdfs-default.xml) or disable this feature. Otherwise, users may face an unusually high rate of pipeline failures since it is impossible to find new DataNodes for replacement. If we have four nodes and a replication factor of 3, each block will have a replica on three of the live nodes in the cluster. If a node dies, the blocks living on the other nodes are unaffected, but any blocks with a replica on the dead node will need a new replica created. However, with only three live nodes, each node will hold a replica of every block. If a second node fails, the situation will result into under-replicated blocks and Hadoop does not have anywhere to put the additional replicas. Since both remaining nodes already hold a replica of each block, their storage utilization does not increase. DataNodes will be connected (based on the blocks from the Namespace) using pipeline. In the existing architecture to deliver the packet it needs to traverse through all the DataNodes to reach the last DataNode (based on the replication factor we need to decide last number).Refer with: Fig. 1, While writing the data if DataNode network fails the failed DataNode will be removed from the pipeline. Adding the new DataNode to pipeline will depend on the available nodes in the cluster. The problem in the existing architecture is users may experience an unusually high rate of pipeline failures since it is impossible to find new DataNodes for replacement. The time required to send the data packet and getting the acknowledgement back to source DataNode will take longer time since the DataNodes are connected in linear pipeline fashion.

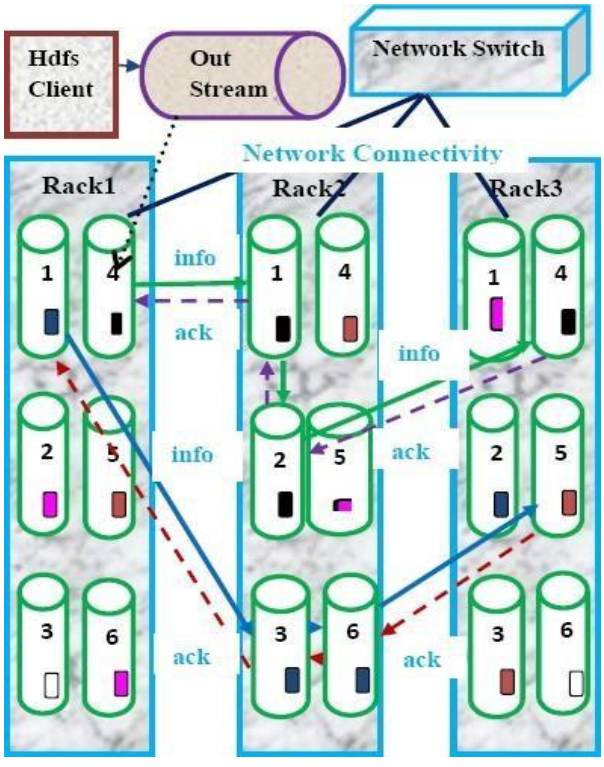


Figure 1: DataNode pipeline connectivity.

If we consider one millisecond is for inter rack DataNode packet transfer and 0.75 millisecond is for intra rack DataNode packet transfer, then (replication factor is 4, DataNode2 and DataNode3 are in second rack, whereas DataNode1 and DataNode4 are in rack1 and rack4 respectively) 1 millisecond to reach to second DataNode , 0.75 to reach from DataNode2 to DataNode3 and

1millisecond from DataNode3 to DataNode4. So the total time is 1+0.75+1 = 2.75 milliseconds. The same is applicable for acknowledgement transfer as well. So total 5.50 milliseconds required to complete one packet copy operation with replication factor 4. Refer with: Table 1 for the time taken for copy operation of one packet using different replication factors.

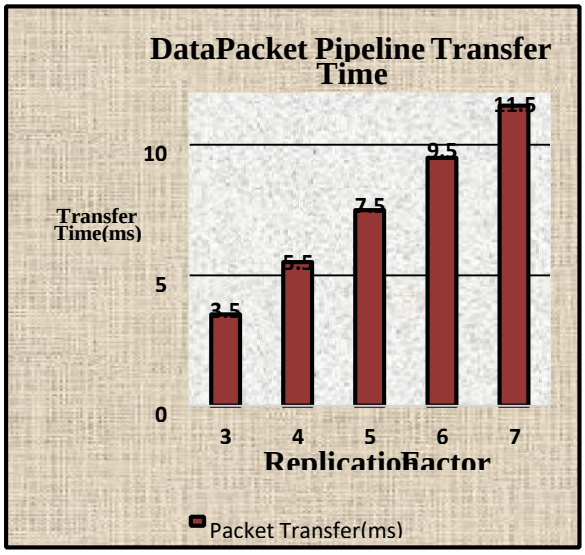
Table 1: Packet Transfer time with different replicationfactors

Replication Factor	Packet Transfer(ms)
3	3.5
4	5.5
5	7.5
6	9.5
7	11.5

The time is growing up while increasing the replication factor,

because the DataNodes will be connected in pipeline fashion[3] (one DataNode to another till the last DataNode and the number of DataNodes in the pipeline depends on the replication factor). For files which are frequently accessed or critical, setting the replication factor improves their tolerance against faults and increases the read bandwidth. In the existing system we are using the parameter dfs.client.block.write.replace-datanode-on-failure.enable[3] to replace the DataNode in case of DataNode network failure. If the number of spare nodes are less or unavailable , then we need to set the parameter to NEVER so that we can externally informing to file system that , there will not be any node replacement in case of any network/node failure. This is having the limitation on number of DataNodes available in the cluster. Refer with: Graph 1 for the time status while increasing the replication factor.

Graph1: ReplicationFactor Vs PacketTransfer time



2.1. *NameNode*

Hadoop Distributed File System's central component is the NameNode. People commonly refer to a NameNode as the master. A NameNode only retains HDFS metadata, including the file hierarchy and the block locations associated with each file. The DataNodes store the physical data and the dataset, leaving the NameNode without them. Typically, we configure the NameNode with a substantial amount of memory, specifically RAM. NameNode is aware of the block list and its location for each file in HDFS. Given its understanding of blocks and locations, NameNode can easily generate files from blocks. The NameNode is a namespace that contains both files and folders. In this context, inodes will serve as the representation for the files and directories. An inode contains information about file permissions, modification and access time, disk space, and namespace. Several DataNodes separately copy each block of the file, typically 128 MB in size. The NameNode holds details about the mapping of file blocks to DataNodes. When the client requests to read some HDFS data, it contacts the NameNode to learn the locations of the first few file blocks it wants to read. In contrast, in a write operation, the client will request from NameNode the set of DataNodes that will house the block copies. The client will carry out the pipeline-style write operation to DataNodes in the upcoming phase [7–8]. The client wants to write the file to HDFS, dividing it into blocks and storing them on different DataNodes. The client establishes a connection with the initial DataNode in the pipeline and commences the process of writing the data blocks on that particular node.

2.2. Data Node

Within a Hadoop cluster consisting of numerous nodes, there will be one or more nodes designated as the master node [15]. The master nodes are in charge of crucial Hadoop operations like naming NameNode and managing resources. The worker nodes that make up the remaining servers in a Hadoop cluster are called DataNodes. These nodes have the responsibility of storing the data blocks. The DataNode executes various tasks as instructed by the NameNode, including the creation and deletion of data blocks, replication of data across the cluster, storage of blocks on the local file system to provide block storage, handling read/write requests from clients accessing the stored data, and maintaining regular communication with the NameNode through heartbeats and block reports. A block report provides information on the blocks that are under the management of the DataNode, while a pulse serves to confirm the presence and well-being of the DataNode. A file system will partition each file into one or more segments and/or store them on distinct data nodes. These items are commonly known as blocks. By altering the HDFS settings, you can increase the default block size of 128 MB. At initialization, the NameNode will shake hands with every DataNode. During the handshaking step, the DataNode's namespace ID and software version will undergo authentication. Once the match is successful, the DataNode will initiate communication. In the event of a discrepancy, the DataNode will promptly initiate an automatic shutdown process. When necessary, DataNodes communicate with the NameNode instead of directly connecting to it. Upon starting or restarting, the DataNode initiates communication with the NameNode, signaling its readiness to perform HDFS read and write jobs. A newly added DataNode without a namespace ID can join the cluster and obtain the group's namespace ID. Each

DataNode retains its own unique storage ID, which makes it easier to identify a DataNode even in the event of a restart with a changed port or IP address. Every DataNode periodically transmits a pulse, containing statistical usage information unique to that DataNode, to a NameNode (with a default frequency of every three seconds). Using this heartbeat signal, the NameNode can provide commands to the DataNodes to cease replication or delete data. When the NameNode does not get a heartbeat for a prolonged duration, it immediately requests a block report from the DataNode. When the NameNode restarts or the DataNode's network connection times out, it asks the DataNode to register again if it doesn't acknowledge the DataNode. The NameNode flags a DataNode as dead and replicates its data on additional DataNodes, increasing the replication factor of the blocks to the specified number of replicas. If the DataNode consistently fails to send its periodic heartbeat for an extended period, such as 10 minutes, this event will occur. The NameNode sends instructions to the DataNodes in response to their heartbeat signals. Refer to Figure 2 to analyze the structure of HDFS.

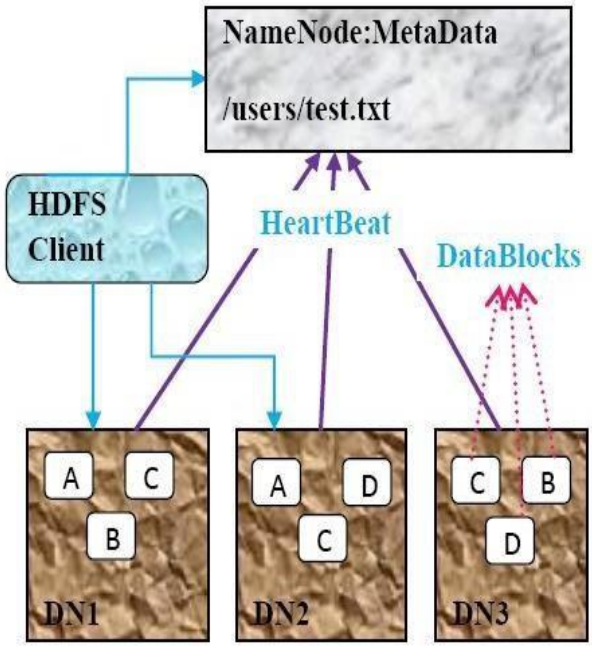


Fig 2: HDFS Write Operation

3. Rackawareness

Hadoop components are designed with consideration for the physical racks in a data center. Usually, Hadoop clusters with more than thirty to forty nodes are housed on many racks. Inter-node communication inside the same rack is more efficient than inter-rack communication. In order to minimize network traffic while reading or writing HDFS in large Hadoop clusters, NameNode chooses DataNodes on the same rack or adjacent racks to read or write requests . A NameNode maintains a record of the rack ID of each DataNode. Rack awareness is the concept in Hadoop that refers to choosing DataNodes based on rack information. Rack awareness is a policy that is simple and direct in its approach to distributing copies among racks. This feature ensures data integrity in the event of a total failure of a rack and allows for the efficient utilization of bandwidth across many racks while reading files. The regulations oversee block replications across several rack clusters, forbidding the arrangement of more than two replicas in the same rack and more than one replica on a single DataNode. Furthermore, the number of racks used for block replication must always be lower than the total number of replicas. Upon creating a new block, the initial replica is stored on the local DataNode, while the second replica is stored on a distinct rack. The third duplicate is stored on a distinct DataNode located within the same local rack. When replicating a block, if there is just one duplicate, position the second duplicate on a distinct rack. If there are two duplicates currently on the same rack, relocate the third replica to a different rack. While reading, the NameNode initially verifies the location of the client's machine within the cluster. Upon the closure of a DataNode, the client is provided with the block positions. This policy aims to minimize the expense associated with data writing while simultaneously improving the speed at which data may be read. This guarantees that data can be accessed in case of a network switch failure or partition occurring within the cluster.

4. HDFS Write Operation

An HDFS cluster is comprised of a NameNode and one or more DataNodes. In this section, we have given a comprehensive analysis about how a client communicates with the NameNode and DataNodes when uploading data to HDFS[7-11].

4.1 File creation into the file system's namespace:

The client first makes a create() HDFS call, which results in a ClientProtocol RPC being invoked to create a new file on the NameNode . Before the creation of the file in the namespace, the NameNode conducts several checks, e.g., whether the file already exists, whether the user has the right to create the file, and whether safe mode is disabled. If all these checks pass, the NameNode would create the corresponding file in the file system's namespace; otherwise it would throw an exception.

4.2 Packets forming from data and inserting into a data queue:

When writing data to HDFS, client applications interpret the data in the file as a conventional output stream. Each block in the data stream has a standard size of 64 MB. We then automatically divide each block into 64KB packets before sending it across the network. Upon the creation of a new block by the client, the DataStreamer thread initiates an addBlock() request to the NameNode, seeking a new block ID and the DataNode IDs for block storage. Once the packets are generated, the client transmits them to a first-in, first-first-out (FIFO) queue, which subsequently forwards them to the DataNodes.

4.3 Packets to DataNodes:

Using the DataNode IDs, DataStreamer builds a pipeline between the client and these DataNodes. It then streams the packets, one at a time, to the first DataNode in the pipeline and saves them in an additional queue called the ACK queue in case some DataNodes need to retransmit because of packet loss. When a packet is received, the initial DataNode validates the packet's checksum, stores it, and then passes it on to the next DataNode in the pipeline. Until the packet reaches the last DataNode at the end of the pipeline, this process will be repeated.

4.4 Sending acknowledgement (ACK) to the client:

Once the final DataNode receives the packet, it will transmit an acknowledgment (ACK) along the pipeline in a reverse sequence. The client's PacketResponder thread receives acknowledgment (ACK) responses. When the PacketResponder thread receives an acknowledgment (ACK) packet from all DataNodes, it removes it from the ACK queue.

4.5 Close the output stream:

The client runs close() on the stream and waits for all packet ACKs after flushing all data into the output stream.

4.6 Completing file write:

The PacketResponder thread wakes up the client when it receives all of the packets' ACKs. The client would provide a comprehensive signal to the NameNode in order to finalize this file-write operation. Consult using: Figure 3 illustrates the HDFS write operation.

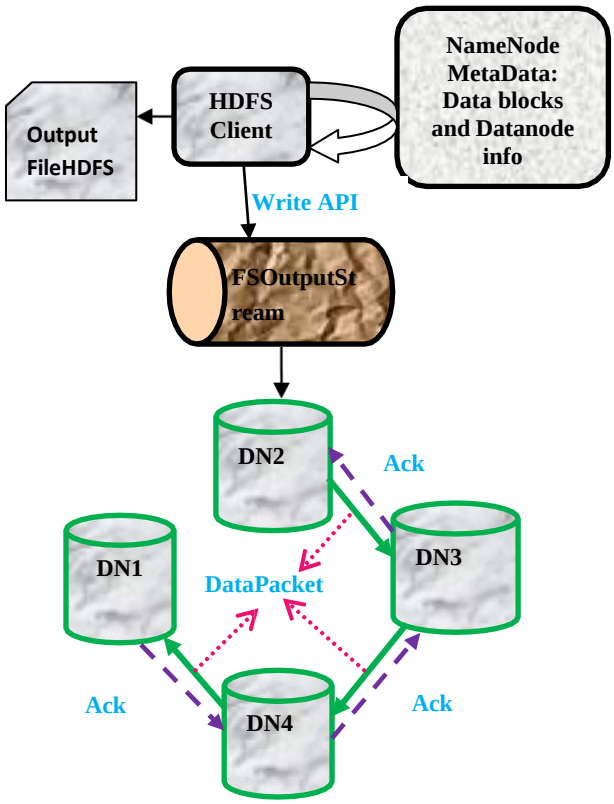


Figure 3: HDFS Write Operation

As per the namespace DN1,DN2,DN3 and DN4 are connected in a pipeline fashion. Once the packet has been written to DN2 by FSDataOutputStream, then it will be transferred to DN3 by DN2 , DN3 to DN4 and DN4 to DN1. Acknowledgement will be transferred back from DN1 to DN4, DN4 to DN3, DN3 to DN2.The solid lines from DN2 ->DN3->DN4->DN1 in the figure is showing data packet transfer and dotted lines from DN1->DN4 ->DN3->DN2 is showing acknowledgement. Using the configuration (hdfs-site.xml) we can set the replication factor of a file. By defaultthe replication factor is three. Here we are taking the replication factor as four. DN1 is in rack1, DN2 and DN3 are in rack2 and DN4 is in rack3. As discussed assume that one millisecond is for inter rack DataNode packet transfer and 0.75 millisecond is for intra rack DataNode packet transfer, then (replication factor is 4, DataNode2 and DataNode3 are in second rack, whereas DataNode1 and DataNode4 are in rack1 and rack4 respectively) 1 millisecond to reach to second DataNode , 0.75 to reach from DataNode2 to DataNode3 and 1 millisecond from DataNode3 to DataNode4. So the total time is 1+0.75+1 = 2.75 milliseconds. The same is applicable for acknowledgement transfer as well. So total 5.50 milliseconds required to complete one packet copy operation with replication factor 4. We can reduce this time using fully connected network topology among the DataNodes.

5. Problem Statement

Based on blocks from the Namespace, a pipeline joins DataNodes. In the event of a network failure in the DataNode, the pipeline will eliminate the non-functioning DataNode. Drawing from the available DataNodes inside the cluster, the pipeline will incorporate a new DataNode. Because it is unable to identify new DataNodes to replace the lost ones, customers may encounter an abnormally high rate of pipeline failures if the cluster has relatively few spare nodes. The pipeline connection keeps the data packet from getting to the intended DataNode in the event of a network failure. Because the copy operation must go via each connected DataNode in the pipeline until it reaches the last DataNode, it takes longer when pipeline connectivity is present. Users may encounter an uncharacteristically high rate of pipeline failures, and the copy process takes longer since pipeline connectivity makes it impossible to identify new DataNodes to replace the ones in the current architecture.

5.1. Proposal

We can connect the DataNodes using fully connected digraph network topology[4] , where each DataNode is connected to every other DataNode as per the list from the NameNode. We can have number of alternative paths in case of network failure in the current part and we can improvethe write operation performance by decreasing the operation time using the new architecture. DFSClient will request for data blocks from the NameNode . Once the client gets the list of data blocks , client will open the Out stream for write operation. Data will be written to nearest DataNode (block) , and this DataNode will be connected to other DataNodes (number of DataNodes based on the replication factor) in pipeline fashion. If there is a network failure in the write pipeline, the operation cannot be completed . To avoid this connectivity issues , we can use the DataNodes using fully connected digraph network topology[4] , where each DataNode is connected to every other DataNode as per the list from the NameNode . Total number of edges are n(n-1) if there are n DataNodes in the pipeline. Each DataNode is having n-1 outgoing edges to connect to n-1 DataNodes.The existing architecture each

DataNode is having 2(n-1) edges where as n-1 edges for datapacket copy operation and the other n-1 for acknowledgement. Solid lines are datapacket transfer operation and dotted lines for acknowledgement operation. Here the dotted lines mentioned with double direction.

Refer with: Fig 4. for proposed architecture. Replication factor is the number of copies the data block will be copied in cluster. The replication factor 4 has been used here, so the data is available in four DataNodes. The cluster is having three racks Rack1, Rack2 and Rack3 and each rack is having 6 DataNodes. DN1, DN2, DN3,

DN4, DN5 and DN6 are represented as 1, 2, 3, 4, 5 and 6 respectively. The representation is same for each rack. As shown in the figure the data packet (using distinct colors to distinguish datapackets) is stored into DN4 in Rack1 , DN1 and DN2 in Rack2 followed by DN4 in Rack3(Replicationfactor is 4). Here single direction lines are for datapacket transfer operation and bidirectional dotted lines for acknowledgement operation. DataNode DN4 in Rack1 is connected to three DataNodes DN1(Rack2),DN2(Rack2) and DN4(Rack3). Once the client writes data packet to DataNode DN4 in Rack1 this will get copied to all other DataNodes in the list DN1 in Rack2, DN2 in Rack2 and DN4 in Rack3 simultaneously. The acknowledgement packet will be transferred back to DN4 in Rack1 from all other DataNodes simultaneously. Since this is parallel operation both in forward (sending packet) and backward (acknowledgement) direction , the time required to complete one packet copy operation is just twice the time required for inter rack packet copy operation , and if there is intra rack DataNode is available in the replication list then the total time will be lesser than the time which we have counted in inter rack packet transfer. If there is any network failure while copy operation is in progress we can reach the destination DataNode using number of alternative paths, i.e,if the replication factor is 3 we can have one alternate path, if it is 4 we can have 4 alternative paths , for 5there will be 15 alternate paths and for replication factor 6 we can have 40 alternate paths. The complexity of network implementation[9] is high compared to existing architecture but we can nullify the network issues and we can decrease the time required to write the datapacket. In the existing system we are using the parameter dfs.client.block.write.replace-datanode-on-failure.enable [3] to replace the DataNode in case of DataNode network failure. We need to set this parameter to NEVER in case of cluster size is very small like having three DataNodes. If the number of DataNodes are three then in case of network failure there will be any choice to replace , instead of that we need to face the consequences of failure. Need to set as true in case of having more number of nodes in the cluster so that we can replace with new DataNode . In the proposed architecture in case of network failure we no need to depend on the replacement of the DataNode with new DataNode , instead of that we can reach the target DataNode using the shortest path from the remaining paths. We can find the shortest path from single source DataNode to all other DataNodes using Dijkstras's shortest path algorithm[9]. In the context of the

research paper focused on HDFS write operations utilizing a fully connected digraph data node network topology [10], the current approach employs Dijkstra's algorithm. To enhance performance further, I propose leveraging the A* algorithm.

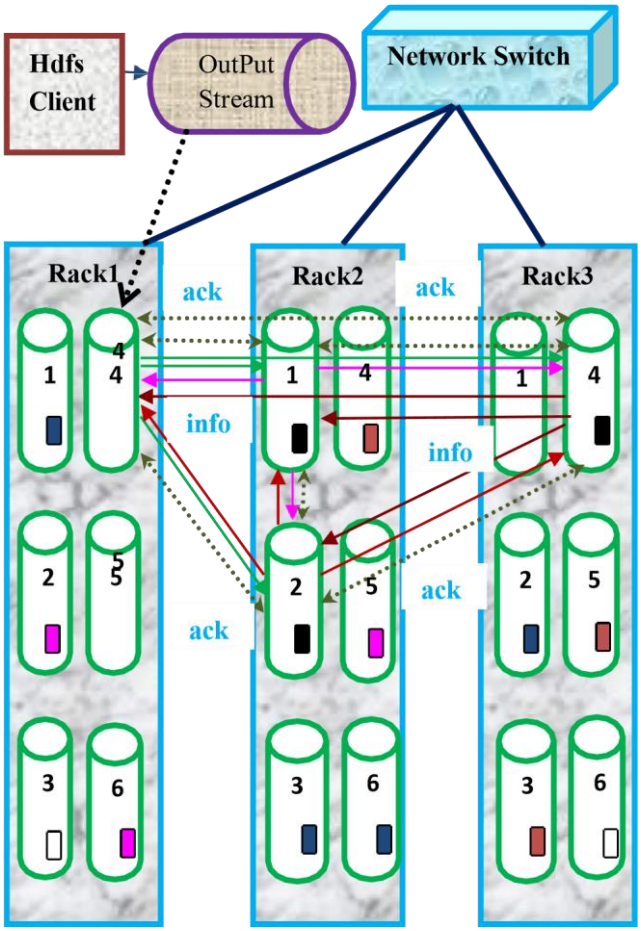


Figure 4: DataNodes with fully connected digraph network topology

A* algorithm [11], renowned for its efficiency in pathfinding, introduces a heuristic component to intelligently navigate the data node network, potentially optimizing the overall HDFS write operation process. This strategic integration of A* algorithm aims to elevate performance to the next level, surpassing the capabilities of the existing Dijkstra's algorithm implementation.

Table2:A* algorithm heuristic values

Node	Heuristic value
1	20
2	8
3	14
4	6
5	11

f(x)=g(n)+h(n), f(x) is final value where as g(n) and h(n) are actual distance(value),heuristic value. Please find the A* algorithm heuristic values. As shown in figure 5 the distance between nodes 1 and 2 is 10, but the traveling distance is heuristic value and actual distance is 30. The traveling distance between nodes 1 and 4 is sum

of actual distance and sum of heuristic values is 10+1+20+8 = 39. We have other options to reach node 4 from 1(1-3-4,1-3-5-4,1-2-3-4) and the final f(x) values for all the mentioned paths are 48 , 58 and 63. In the above set of minimal values 39 is the least value. We can consider path 1-2-4 is the minimal distance path. Similarly considering the distance between nodes 1 to 5(1-3-5) is 41. We have other options to reach node 5 from 1(1-2-3-5) is 56 . In the above set minimal values 41 in least value. So this is the best path

1-3-5. The traveling distance between nodes 3 and 4 is sum of actual distance and sum of heuristic values is 9 + 14 = 23.We have other options to reach node 4 from 3(3-5-4) and the final f(x) values for all the mentioned paths are 33. In the above set of minimal values 23 is the least value. We can consider path 3-4 is the minimal distance path. The traveling distance between nodes 2 and 5 is sum of actual distance and sum of heuristic values is 2+2+8+14 = 26. In this case we can consider this is the only option as per the given graph. Since we are taking fully connected directed graph each node is having n-1 edges. The traveling distance between nodes 1 and 5 is sum of actual distance and sum of heuristic values is 5+2+20+14 = 41. We have other option to reach 5 from 1 , it is 1-2-3-5. The total value is 10+2+2+20 + 8 +14=56. In the above set of minimal values 41 is the least value. We can consider path 1-3-5 is the minimal distance path. source to this node can be reduced while using the selected edge. If this is going to happen then the distance is updated and the node is added to the nodes which need evaluation. This is how can have number of alternative paths so that users will escape from experiencing an unusually high rate of pipeline failures.

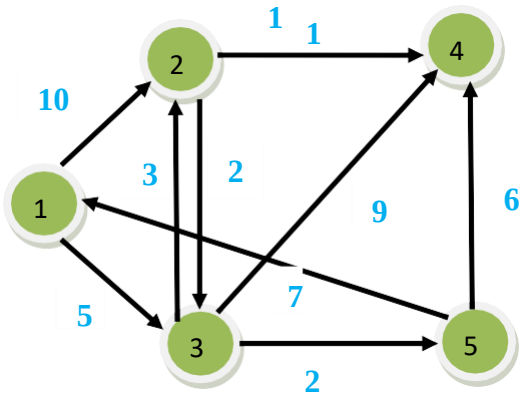


Figure 5: Input Graph

Here I am considering the weights as real numbers just to explain the A* algorithm instead of taking one millisecond and 0.75 millisecond. Please Refer with: Fig 5.

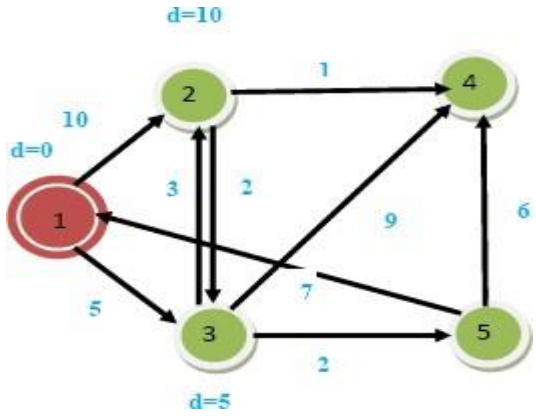


Figure 6: Input Graph

Node being considered: 1

Nodes Not yet finalized: {2,3,4,5}

Distances={INF,INF,INF,INF,INF}

Please Refer with: Fig 6. In this we are considering node 1. The remaining we are not considering. So the distances are INFINITE.

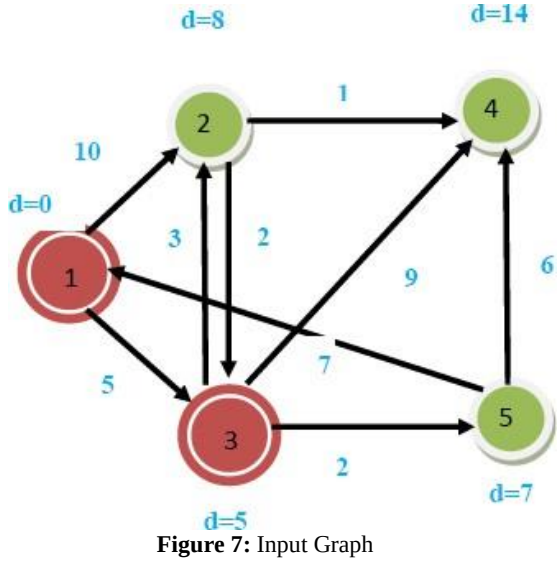


Figure 7: Input Graph

Node being considered:3

Nodes Not yet finalized: {2,4,5}

Distances={0,10,4,INF,INF

Please Refer with: Fig 7. Here considering node 3 after 1.We will we take the minimum i.e, 3 and proceed.

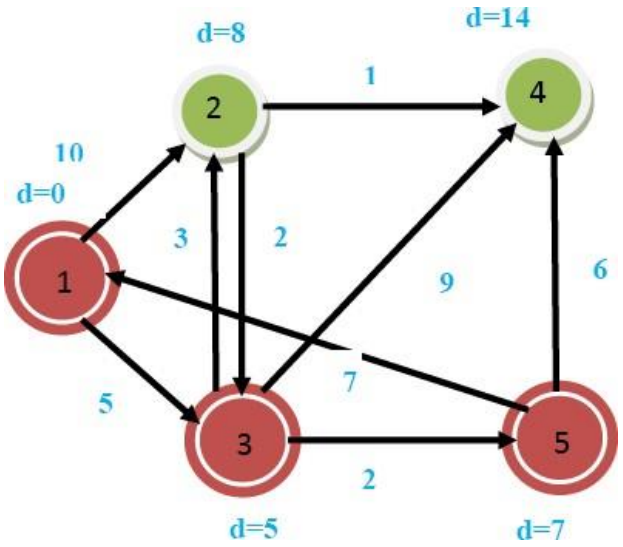


Figure 8: Input Graph

Please Refer with: Fig 8 for the status of nodes and distances while considering node 5. Nodes Not yet finalized: {2,4}

Distances={0,8,5,14,7}

Distance[2]=Distance[3]+wt(3,2)=8

Distance[4]=Distance[3]+wt(3,4)=14

Distance[5]=Distance[3]+wt(3,5)=7

We need to add heuristic values as well and finalize the minimum distance node. For example we consider minimum node5 and proceed.

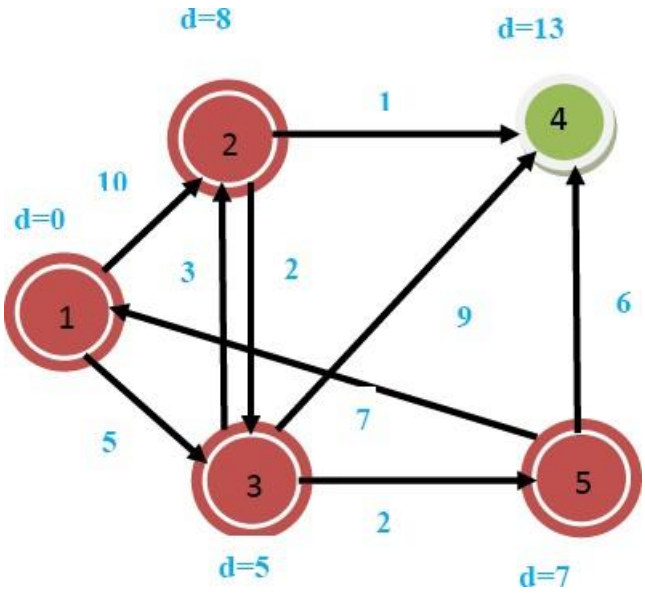


Figure.9: Input Graph

Please Refer with: Fig 9 for the status of nodes and distances

while considering node 2. Nodes Not yet finalized: {2,4}

Distances={0,8,5,13,7}

Distnace[4]=Distance[5]+wt(5,4)=13

We need to add heuristic values as well and finalize the minimum distance node. For example we consider minimum node2 and proceed.

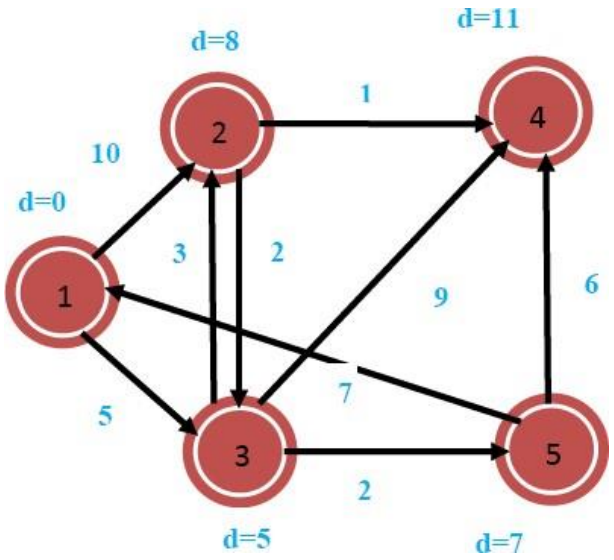


Figure 10: Input Graph

Please Refer with: Fig 10 for the status of nodes and distances while considering nodes 2,4,5

Nodes Not yet finalized: {2}

Distances={0,8,5,11,7}

No update is required. hence distance between 1 and all other nodes are given in distance.

For obtaining the shortest paths in a weighted network with positive or negative edge weights (but no negative cycles), an additional approach is the Floyd-Warshall algorithm [9]. One iteration of the algorithm will calculate the total weights of the shortest pathways between every pair of vertices. Alternative names for this algorithm include the Roy-Warshall algorithm, the Roy-Floyd algorithm, and the WFI algorithm. Let's examine a graph G that consists of vertices M, which are numbered from 1 to N. Think of a function called shortestPath(i,j,k) that finds the shortest path between two points, I and j, using only vertices from the set {1,2,3,4...k} as intermediate points. The weight of the edge between vertices i and j can be defined as w(i,j).The function shortestPath(i,j,k+1) is computed using the recursive formula: shortestPath(i,j,0) = w(i,j). The equation shortestPath(i,j,K+1) is defined as the minimum value between shortestPath(i,j,k) and the sum of shortestPath(i,k+1,k) and shortestPath(k+1,j,k). The algorithm operates by performing computations. The algorithm calculates the shortest path between all pairs of (i,j) by iterating from k = 1 to k = N. This process persists until the value of k reaches N.

Let the dist be |M| * |M| array of minimum distnaces initialized to INFINITY.

```
for each vertex i
        dist[i][i] <- 0
for each edge (a,b)
          dist[a][b] <- w(a,b)
for k from 1 to |M|
for i from 1 to |M|
          for j from 1 to |M|
                if dist[i][j] > dist[i][k] + dist[k][j]
      dist[i][j] <- dist[i][k] + dist[k][j]
    endif
```

Let n represent the number of vertices, which is equivalent to the cardinality of set M. It takes n2 operations in total to separate the n2 instances of shortestPath(i,j,k) from those of shortestPath(i,j,k-1). The sequence of n matrices shortestPath(i,j,1), shortestPath(i,j,2), and shortestPath(i,j,n) are computed starting with shortestPath(i,j,0) = edgeCost(i,j). The total number of operations utilized is equal to n multiplied by 2 raised to the power of n squared, which can be simplified as 2n cubed. The overall complexity is O(n3). When all nodes are run through the A* algorithm, the complexity is O(E log V), whereas Floyd's difficulty is O(V3). Here, V represents the number of vertices and E the number of edges. If the time complexity of algorithm E is O(V2), then these two algorithms are mathematically equivalent. However, in practice, Floyd's technique is more efficient. E = O(V) indicates that it is better, practically and theoretically, to run A* for every node. If the graph is complete, Floyd's technique should be used; if not, run A* from every node if the number of edges is equal to the number of nodes. If you possess sufficient memory and time

resources, Floyd's approach is evidently superior due to its significantly reduced coding time. Nevertheless, if you have no interest in obtaining all potential routes, the Floyd-Warshall algorithm may consume unnecessary time by computing numerous undesired shortest paths. Given those circumstances, we can employ the A* algorithm. A different algorithm is the Bellman-Ford algorithm, which determines the shortest path between each vertex in the graph and the source. The graph has the potential to include negative edges. Nevertheless, the negative edge holds little significance. The Bellman-Ford algorithm is less complex than the A* algorithm and is highly compatible with distributed systems. The A* algorithm is not as time-efficient as the Bellman-Ford algorithm, with O(VE) time complexity.

5.2. Implementation

For an implementation of a fully connected digraph network topology [4] among the DataNodes within the Hadoop Distributed File System, refer to Fig. 17. The HDFS client sends creation requests to the DistributedFileSystem APIs. The DistributedFileSystem initiates an RPC (Remote Procedure Call) to the NameNode in order to create a new file within the namespace of the filesystem. The NameNode performs numerous verifications to ensure that the file does not already exist and that the client has the necessary rights to create it. If the result of the check is positive, the NameNode creates a record of the new file. Otherwise, if the check fails, the file creation is unsuccessful, and the client receives an IOException. The DistributedFileSystem provides the client with an FSDataOutputStream to initiate the process of writing data to the DataNode. As the client writes data, DFSOutputStream divides it into packets and adds them to an internal queue called the data queue. The DataStreamer then processes this queue, requesting the NameNode to allocate new blocks. The DataStreamer selects a list of appropriate DataNodes to store the data replicas. A fully connected digraph network topology connects the DataNodes, enabling the simultaneous transfer of data packets to all DataNodes.

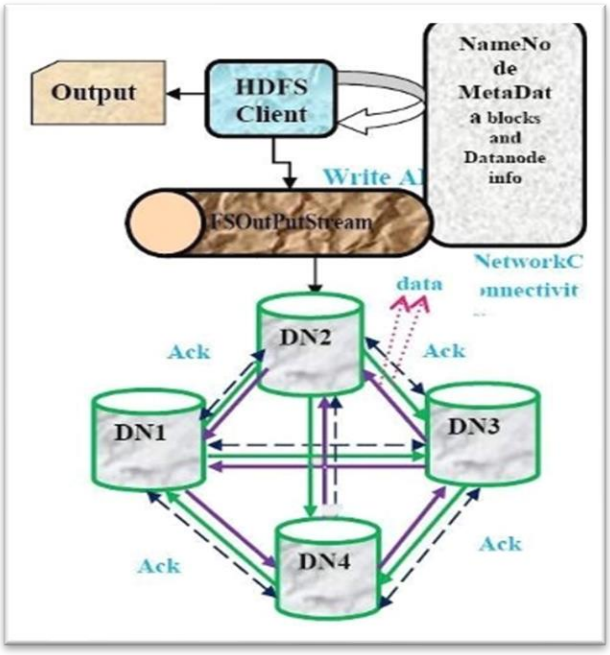


Figure.11: Fully Connected Digraph network topology implementation in HDFS

In the existing architecture the data packet needs to traverse through all the DataNodes to reach the last DataNode (based on the replication factor we need to decide last number). If we consider one millisecond is the time required to transfer packet from one DataNode to another DataNode between two racks and 0.75 for intra rack DataNode transfer, then to reach the 4th DataNode (replication factor is 4) is 2.75 milliseconds. The same is applicable for acknowledgement transfer as well. So total 5.5 milliseconds required to complete one packet copy operation with replication factor 4. Whereas in fully connected digraph network topology data packet will be transferred in parallel fashion i.e, it will take one millisecond to transfer the packet to all DataNodes irrespective of replication factor. Acknowledgement as well will be transferred to source DataNode in one millisecond. So total 2 milliseconds required for successful one packet copy operation irrespective of replication factor. If we consider intra rack DataNode transfer less than one millisecond then the total time is max 2 milliseconds. DN1 is connected to all DataNodes and the same is applicable to all DataNodes. So the total number of connections are n(n-1) excluding acknowledgement edges. If there is any network failure while writing datapacket to DataNodes which were connected using fully connected network topology, no need look for the new DataNode for replacement , instead of that there will be number of alternative paths to reach the target DataNode . As shown in the Fig.11 DN2 will receive the write request (datapacket) from the IOstream. DN2 will send the packet to DN1,DN4 and DN3 simultaneously. So the total time is max one millisecond (considering DN1 in Rack1, DN2,DN3 are in Rack2 and DN4 in Rack3) for writing datapacket and the

acknowledgement time is max one millisecond. While writing datapacket to DN4 from DN2 if there is network failure issue , using the parameter no need to replace the new DataNode , instead of that,datapacket can reach DN4 using DN2->DN1->DN4, DN2->DN3->DN4, DN2->DN3->DN1->DN4, DN2->DN1->DN3->DN4. Like this depends on the replication

factor we can have multiple number of paths i.e, if the replication factor is 3 we can have one alternate path, if it is 4 we can have 4 alternative paths (as shown above) , for 5 there will be 15 alternate paths and for replication factor 6 we can have 40 alternate paths from DN2 to DN4. Since we have

alternative paths we no need to think about replacement .This is how we can avoid replacement of new node in case of network failure.

5.3. Evaluation

The simulation results are here with the assumption that inter rack datanodepacket transfer will take one millisecond and intra rack DataNode packet transfer will take 0.75 millisecond. There is best case scenario, medium and worst case scenariosbased on the path which we consider toreach to target node. Suppose DN2 to DN4 datapacket needs to be copied and considering DN1 in Rack1, DN2,DN3 are in Rack2 and DN4 in Rack3 Direct path from DN2 to DN4 is the best case scenario where it will take max one millisecond to copy the data and one millisecond for acknowledgement. So total 2 milliseconds required for data packet copy operation including acknowledgement. Whereas in medium

case scenario, DN2->DN1->DN4, DN2->DN3-DN4 it will take one millisecond for DN2->DN1 different rack, DN1->DN4 one millisecond for different rack . So total 2 milliseconds required for copy and 2 milliseconds for acknowledgement. So total 4 milliseconds in medium case scenario. In the worst case scenario DN2->DN1->DN3->DN4(3+3), DN2->DN3->DN1-DN4 (2.75+2.75) total time is max 6 milliseconds and min 5.5 milliseconds including acknowledgement. Refer with: Table 3 for the Access Time Analysis using Fully Connected DataNode Topology. In the linear pipeline connectivity time required for

datapacket copy operation including acknowledgement is 5.5 milliseconds, which is almost two times to worst case scenario of fully connected digraph network topology. Refer to: Table 4 for the results of linear fashion DataNode pipeline connectivity and a fully connected digraph network topology with different level replication factors using the best casescenario of fully connected digraph network topology. That means no network failure and using the direct path from source DataNode to destination DataNode . In this proposed architecture the time required to complete one packet copy operation is 2milliseconds if there are no intra rack DataNodes , and max 2milliseconds in case of the DataNodes list includes intra rack DataNodes. Based on the results mentioned here for replication factor 4 FullyConnectedDataNode Digraph topology is better than Liner Pipeline data packet transfer time.

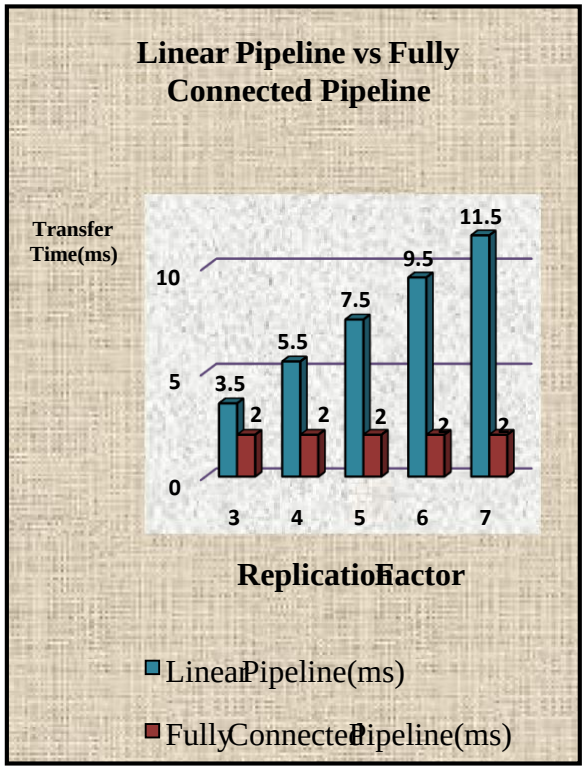
Table 3: Access Time Analysis using Fully ConectedDataNode Topology.

BestCase Scenario	**DataNode Connectivity**	**Transfer Time**
Rack1 DN1 ck2 DN2,DN3 Rack3 DN4	**DN2->DN4 : 1 ms, different rack**	**1 +1 : 2ms (copy+ack)**
ediumCase Scenario	**DataNode Connectivity**	**Transfer Time**
Rack1 DN1 , ck2 DN2,DN3 Rack3 DN4	**DN2->DN1->DN4, DN2->DN3->DN4 DN2->DN1 : 1 ms DN1->DN4 :1 ms DN2->DN3: 0.75ms same rack**	**1+1 : 2 2+2:4 (copy+ack) 0.75+1:1.75 1.75+1.75:3.50 (copy+ack) 0.75+0.75:1.5 1.5+1.5=3ms (copy+ack)**
WorstCase Scenario	**DataNode Connectivity**	**Transfer Time**
Rack1 DN1 ck2 DN2,DN3 Rack3 DN4	**DN2->DN1->DN3->DN4 DN2->DN3->DN1->DN4**	**1+1+1:3 3+3 : 6 ms (copy+ack) 0.75+1+1:2.75 2.75 + 2.75 : 5.50 ms (copy+ack)**

Table 4: LinearPipeline vs FullyConnectedPipelinePacket Best Case Transfer time

eplication Factor	**Linear Pipeline(ms)**	**Fully Connected Pipeline(ms)**
3	**3.5**	**2**
4	**5.5**	**2**
5	**7.5**	**2**
6	**9.5**	**2**
7	**11.5**	**2**

Refer to: Graph 2 for the time required to complete copy operation in Linear Pipeline DataNode connectivity is increasing once we increase the replication factor.Whereas in Fully Connected digraph DataNode network topology the time is constant irrespective of the replication factor.



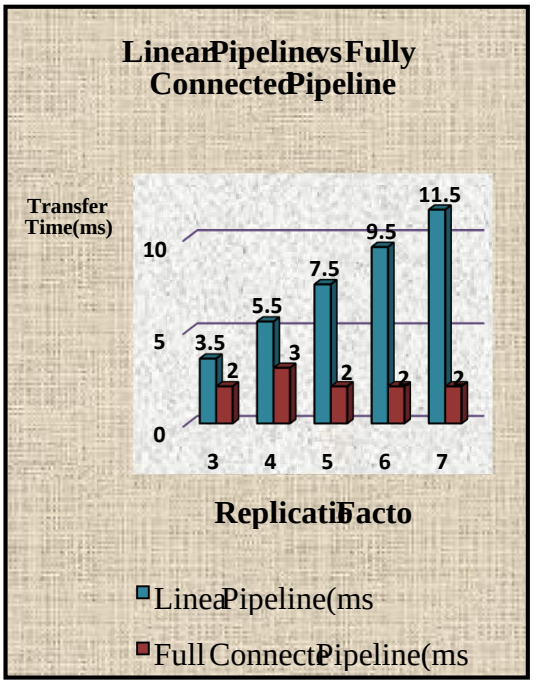
Graph 2:.PacketTransferTime for Linear Pipeline vsFullyConnectedDatanode Pipeline.

Possible paths from DN2 to DN4 using replication factor 4 incase of network failure is there from DN2->DN4 is DN2->DN1->DN4, DN2->DN3->DN4. Refer with: Table 5 for the results of linear fashion DataNode pipeline connectivity and the fully connected digraph network topology with different level of replication factors especially with medium case scenario of fully connected digraph network topology for replication factor 4. In this proposed

architecture the time required to complete one packet copy operation from DN2->DN1 is 1 millisecond and DN1->DN4 is 1 millisecond. So total is is 2 milliseconds and including acknowledgement is 4 milliseconds. In case of DN2->DN3->DN4 the total time including acknowledgement is 3 milliseconds. Based on the results mentioned here for replication factor 4 Fully Connected DataNode Digraph topology is better than Liner Pipeline data packet transfer time.

Table 5: LinearPipeline vs FullyConnectedPipelinePacket medium case Transfer time

Replication Factor	Linear Pipeline(ms)	Fully Connected Pipeline(ms)
3	3.5	2
4	5.5	3
5	7.5	2
6	9.5	2
7	11.5	2



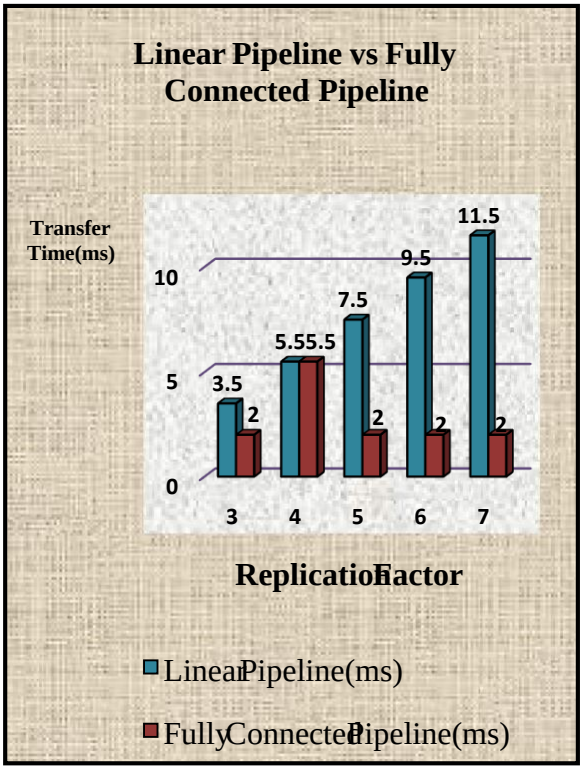
Graph 3: PacketTransferTime for Linear Pipeline vsFullyConnectedDatanode Pipeline.

Refer to: Graph 3 for the time required to complete copy operation in Linear Pipeline DataNode connectivity is increasing once we increase the replication factor. Whereas in Fully Connected digraph DataNode network topology the time is constant irrespective of the replication factor. If we consider the network failure from DN2-DN4 in two combinations DN1->DN4 or DN3->DN4, then the possible paths from DN2->DN4 are DN2->DN1->DN3-DN4 and DN2->DN3->DN1->DN4.

Refer with: Table 6 for the results of linear fashion DataNode pipeline connectivity and the fully connected digraph network topology with different level of replication factors especially with worst case scenario of fully connected digraph network topology for replication factor 4. In this proposed architecture the time required to complete one packet copy operation from DN2->DN1 is 1 millisecond , DN1->DN3 is 1 millisecond and DN3->DN4 is 1 millisecond. . So total is 3 milliseconds and including acknowledgement is 6 milliseconds. In case of DN2->DN3->DN1->DN4 ,DN2->DN3 is 0.75 millisecond , DN3->DN1 is 1 millisecond and DN1->DN4 is 1 millisecond. The total time is 2.75 millisecond and including acknowledgement is 5.50milliseconds . Only in the worst casescenari Linear pipeline connectivity id equal to FullyConnectedDataNode Digraph topology. Two network failures I have taken to create the worst case scenario. But this very rare case. So we can conculde that FullyConnectedDataNode Digraph topology is giving always better results than liner fashion DataNode pipeline connectivity.

Table 6:.LinearPipeline vs FullyConnectedPipelinePacket worst caseTransfer time

Replication Factor	Linear Pipeline(ms)	Fully Connected Pipeline(ms)
3	3.5	2
4	5.5	5.5
5	7.5	2
6	9.5	2
7	11.5	2



Graph 4: . PacketTransferTime for Linear Pipeline vsFullyConnectedDatanode Pipeline.

Refer with: Graph 4 for the time required to complete copy operation in Linear Pipeline DataNode connectivity is increasing once we increase the replication factor. Whereas in Fully Connected digraph DataNode network topology the time is constant irrespective of the replication factor.

6. Conclusion and Future Work

Based on the analysis of the values using different replication factors we can say that the time required to copy data packet to all DataNodes as per the list available from the metadata from the NameNode is constant. Whereas in linear Pipeline DataNode connectivity the time increases while increasing the replication factor. In Linear Pipeline DataNode connectivity we need to support network failure by using the parameterdfs.client.block.write.replace-datanode-on failure. enable using true or NEVER options based on the available DataNodes in the cluster (cluster size), whereas in fully connected digraph DataNode network topology if there is any chance of network failure in one edge we can have multiple paths to reach to destination node, i.e, if the replication factor is 3 we can have one alternate path, if it is 4 we can have 4 alternative paths , for 5 there will be 15 alternate paths and for replication factor 6 we can have 40 alternate paths. So we can nullify the network failure issues.

In this architecture the time required to copy the datapacket to

DataNodes in the network is max one millisecond and acknowledgement is max one millisecond. With the replication factor 4 max 2 milliseconds required to complete the datapacket write operation in the best casescenario, that means there is no network failure , whereas 3 milliseconds for the same operation in medium case scenario , that is where there is one network failure issue and 5.50 milliseconds requiredin worst case scenario where there are two network failure issues. Here we have verified that no need of replacement of DataNode in case of network failure issues. So the usage of dfs.client.block.write.replace,datanode-on-failure.enable,dfs.client.block.write.replace-datanode-on-failure parameters is not required. This is how we can reduce or nullify the network failure issues among DataNodes. Since we have number of alternative paths among the DataNodes, users can escape from experiencing an unusually high rate ofnetwork failures. Using this shortest paths we can reduce the copy operation time as well as we have proved using the Access Time Analysis using Fully Connected DataNode Topology. As we change the architecture to fully connected digraph DataNode network topology the complexity and the cost to implement the architecture will also increase. We can ignore this cost and complexity since there is an improvement in data packet write operation performance and nullifying the network failure issues among the DataNodes. The future work includes reducing the cost of the network by using network cost optimization techniques.

References

[1] Apache Hadoop. Available at Hadoop Apache.

[2] Deepak Vohra, Practical Hadoop Ecosystem:A Definitive Guide to Hadoop-Related Frameworks and Tools, Appress; 1st ed. edition ,October 1, 2016

[3] Tom White, "Hadoop:The Definitive Guide", Storage and Analysis at Internet Scale, Second ed., Yahoo Press, 2010

[4] J.L.Mott, A.Kandel, Mott & Kandel, Discrete Mathematics For Computer Scientists And Mathematicians (English) , 2 Ed, Pearson India, (2015)

[5] Hadoop Distributed File System with Cache system – a paradigm for performnace improvement by Archana Kakade and Dr. SuhasRaut, International journal of scientific research and management (IJSRM), Vol.2,Issue.1: Pp,1781-1784 /Aug. 2014.

[6] KonstantinShvachko, HairongKuang, Sanjay Radia, Robert Chansler , "The Hadoop Distributed File System". Vol.1, No.1, pp.1-10, 2010.

[7] Debajyoti Mukhopadhyay, Chetan Agrawal, Devesh Maru, Pooja Yedale, Pranav Gadekar, Addressing NameNode Scalability Issue in Hadoop Distributed File System using Cache Approach.Vol.1, pp.1-6, 2014

[8] Feng Wang, Jie Qiu, Jie Yang, Bo Dong, Xinhui Li,Ying Li, " Hadoop High Availability through Metadata Replication", IBM China Research Laboratory, ACM, pp 37-44 ,2009.

[9] Ellis Horowitz and Sartaj Sahni, Sanguthevar Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications, 2010.

[10] B. Purnachandra Rao, Dr. N. Nagamalleswara Rao, HDFS Write Operation Using Fully Connected Digraph DataNode Network Topology, International Journal of Applied Engineering Research ISSN 0973- 4562 Volume 12, Number 16 (2017) pp. 6076-6090, © Research India Publications.

http://www.ripublication.com

[11] A Systematic Literature Review of A* Pathfinding,

https://www.sciencedirect.com/science/article/pii/S187

7050921000399

[12] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman “The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM” Department of Computer Science Stanford University, Vol. 43, No. 4, pp. 92-105, December 2009

[13] Hong Zhang1, Liqiang Wang1, and Hai Huang2, "SMARTH: Enabling Multi-pipeline Data Transfer in HDFS", in: Proc of. Parallel Processing (ICPP), 2014 43rd International Conference on, HDFS DataTransfer

[14] 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 andSoftware (ISPASS2010), White Plains, NY, Pp.122-133, March 2010.

[15] SAM R. ALAPATI , Expert Hadoop Administration, Managing, Tuning and Securing Spark, YARN , and HDFS, Addision Wesley Data Analytics series, 2017.