HIGH SCALABILITY OF HDFS USING DISTRIBUTED NAMESPACE

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science and Engineering

Submitted By:
HARCHARAN JIT SINGH
Roll No. 820932001

Under the supervision of:
Dr. V. P. SINGH
(Assistant Professor)

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2012

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "*High Scalability of HDFS Using Distributed Namespace*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Computer Science and Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. V.P. SINGH* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

(HARCHARANJIT SINGH)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Dr. V.P. SINGH)
Assistant Professor,
CSED

Countersigned by

(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean
Academic Affairs
Thapar University
Patiala

ACKNOWLEDGEMENT

First of all I am thankful to God for blessings and showing me the right decision. With his mercy, it has been possible for me to reach so far.

I would like to express sincerest thanks to my thesis supervisors Dr. V.P. Singh, for the inspiration, guidance, stimulating suggestions, immense help and support throughout the period of this research work. He has provided me all the necessary resources including motivation and research environment without which it would not have been possible to complete this work. It was a great opportunity for me to do this work under his supervision.

I am thankful to the authors whose work I have consulted and quoted in this work.

I lack words to express my cordial thanks to all my friends for their useful comments and constructive suggestions during all the phases of my life.

Finally, I convey deep sense of gratitude towards my family members for their moral support and encouragement without which it would not have been possible to bring out this thesis.

HJSingh

HARCHARAN JIT SINGH

ABSTRACT

In data intensive computing, Hadoop is widely used by organizations. The client applications of Hadoop require high availability and scalability of the system. Mostly, these applications are online and their data growth rate is unpredictable. The present Hadoop relies on secondary namenode for failover which slows down the performance of the system. Hadoop system's scalability depends on the vertical scalability of namenode server. As the namespace of Hadoop distributed file system grows, it demands additional memory to cache. If namenode server does not have enough primary memory to cache the namespace, its performance and availability effects.

A new Hadoop architecture has been proposed to address the issues of namenode scalability, single point of failure and Implementation security of Hadoop. This approach is based on distribution of namespace using distributed hash tables. The growing size of namespace of HDFS is divided into multiple name node servers. The proposed architecture of Hadoop is simulated by using the multiple name node servers. The name node are arranges in chord ring. This allows HDFS to scale up horizontally. By simply adding namenode to name node ring, the scalability, availability and performance of HDFS improves. The system provides decartelize managed approach for namespace distribution which gives consistent performance. The results of HDFS namespace to store 1 billion or above files are discussed in this research work. The proposed architecture has shown high availability and adapts to name node failure.

Table of Contents

List of Figures

List of Tables

CHAPTER 1
INTRODUCTION

The phenomenal growth of internet based applications and web services in last decade have brought a change in the mindset of researchers. The traditional techniques to store and analyze voluminous data have been improved. Organization providing information technology solutions are having great concerns to the amount of data their machines are producing. These organizations are ready to acquire solutions which are highly reliable to store and process large data bases. Such organizations are required to index huge volume of contents and analyze terabytes of data to extract patterns.

The size of program code is very small compared to data. The client applications move the program code to data. The program instructions are executed on the node where data resides. This provides data locality to client application's code. The results from all machines are sent back, merged and streamed to client application.

1.1 Data Intensive Computing

The rapid growth of the Internet and its applications like Google, Facebook, Yahoo, Youtube, Twitter, ebay, IBM, Amazon etc has led to vast amount of information available online. Now, the challenge is to store and process this huge amount of data. Parallel processing approaches are used to analyze such a large amount of data.

Parallel processing approaches can be generally classified as either compute-intensive or data-intensive. Compute-intensive is used to describe application programs that are compute bound. Such applications devote most of their execution time to computational requirements as opposed to I/O and typically require small volumes of data. Data-Intensive Computing is a class of parallel computing applications which use a data parallel approach to process large volumes of data which are typically terabytes or petabytes in size.

Several system architectures have been implemented for data-intensive computing and large-scale data analysis, such as applications including parallel and distributed relational database management systems. But, most of data growth is unstructured form of data. MapReduce is a programming paradigm architecture pioneered by Google [1, 2]. Now it is available in an open-source implementation called Apache Hadoop [12]. It is used by organizations like Yahoo, Facebook and other online shopping marts.

Data-Intensive Computing Systems have approaches to parallelize the processing of data. The goal to design such platform is to provide high levels of reliability, efficiency, availability and scalability. Hadoop is one such architecture which exploits above mentioned features.

1.2 HADOOP Architecture

Hadoop parallelizes data processing across many nodes (computers) in a cluster. It speeds up large computations and hides I/O latency. Hadoop is especially well-suited to large data processing tasks like searching and indexing because it has powerful distributed file system. The Hadoop distributed file system has namenode servers and data nodes. The namenode server maintains the metadata called namespace. Namespace has information about namenode servers, file, blocks, replica, data nodes and running jobs. HDFS is highly reliable as it replicates chunks of data to nodes in the cluster. The replica decisions are used to improve the availability of system.

Hadoop is a flexible and available architecture for large scale computation and data processing on commodity hardware [12]. It does not require expensive and highly reliable hardware. It is designed to run jobs on clusters of commodity hardware i.e. commonly available hardware. Nodes are commodity PCs of Gigabit Links and are connected to rack switches of Gigabit Connectivity. The basic figure of two level architecture of Hadoop cluster is shown in Figure 1.1.

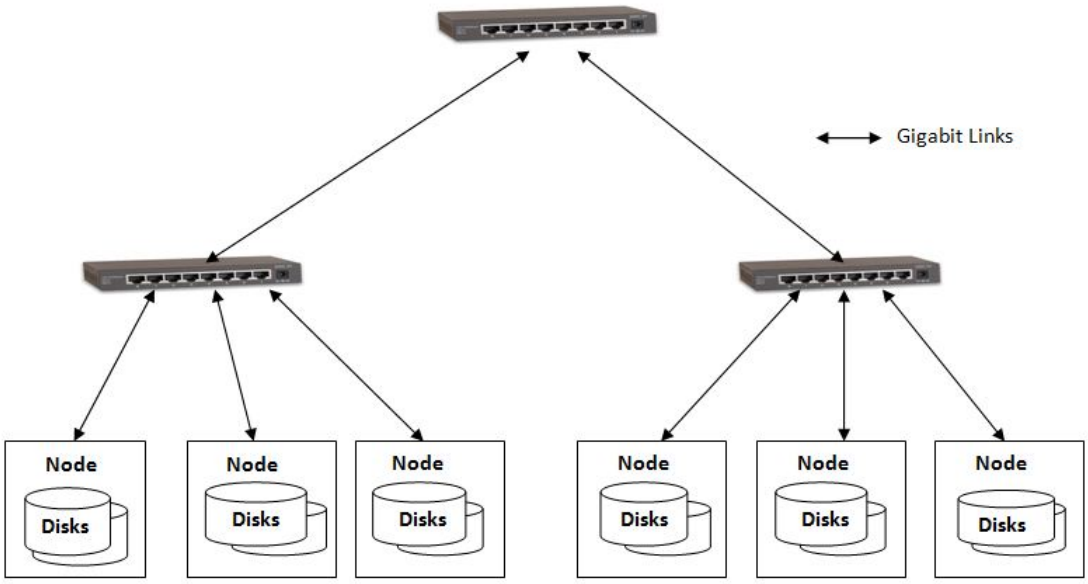

Figure 1.1 Hadoop on Commodity Hardware.

Hadoop is used for various applications such as searching, log processing, business intelligence/data warehousing, video and image processing, sound processing, NLPs and applications in which data is divided into sub data sets.

Hadoop has emerged as a data mining platform and is becoming an industry standard for large data processing. Hadoop is successfully used in science and a variety of industries. Scientific applications include mathematics, high energy physics, astronomy, genetics, and oceanography. The platform adoption has scaled far beyond expectations [13].

The goal of Hadoop distributed file system is to address the issues of hardware failure, high throughput (Streaming) data access and process large data sets of applications [4]. It has simple coherency model named write-once-read-many for files and works on idea of moving computation to data. It is portable across heterogeneous hardware and software platforms.

1.2.1 Hadoop Cluster

There are mainly two service components. One is called namenode which is responsible to hold the metadata and coordinate activities. The metadata is called namespace which

holds the information about name nodes, files, blocks, replica, jobs and data nodes. Other component is data node which is used to store data blocks locally and which is also responsible to execute the tasks. This decoupling of metadata and data provides an opportunity to run independent jobs on Hadoop cluster. The decoupling provides flexibility to the architecture to manage more datanodes in the cluster.

As shown in Figure 1.2, HDFS cluster consists of a single name node called a master server that manages the file system of namespace and regulates access to files by client applications.

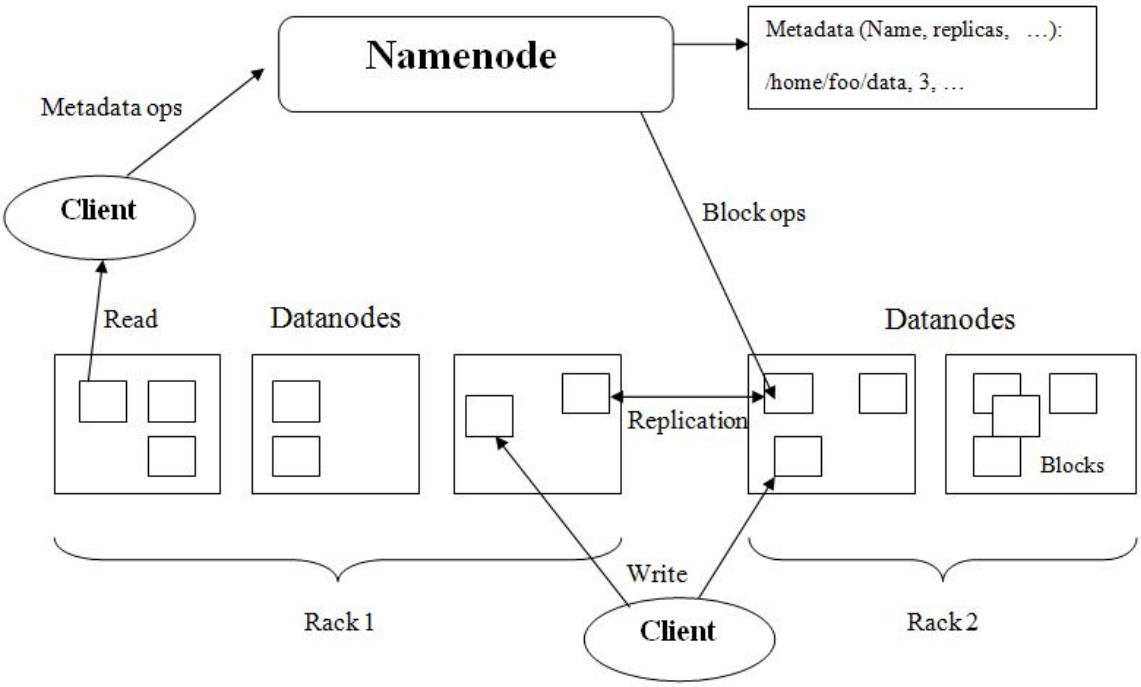

Figure 1.2 Hadoop Distributed File System Architecture [12]

In addition, there are a number of datanodes that are responsible to manage storage. Only Data node knows the physical location of the data. HDFS exposes a file system namespace and allows client applications to store data on data nodes. Internally, a file is divided into one or more blocks of 128 Megabytes. These blocks are stored in a set of data nodes. The name node executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to data nodes. The data nodes are responsible for serving read and write requests from the

file system. The data nodes also perform block creation, deletion, and replication upon instruction from the Namenode.

The client application interacts with the namenode and sends a request for metadata of file. The name node responds with metadata and reply contains the information about datanode, blockid and its replica. Now, the client application directly sends Mapreduce jobs to the data node where the block ids are placed. These jobs run in parallel on each related data node and the results are sent back to the client application. The client application merges the resultant records and gives the output data to the user. In this process, the jobtracker runs on the namenode server and coordinates the execution of jobs running on the datanodes. The data node runs task tracker to monitor the tasks running on datanode. In case of failure, the task tracker sends failure notice to client application and client initiates same job on other replica data node and inform the jobtracker to update the status.

1.3 Hadoop Distributed File System (HDFS)

HDFS is a large scale data storage system. Similar to other common file systems, HDFS is also hierarchal in nature [13]. The namenode splits the large file into fixed size data blocks of 128 MB in size. These blocks are scattered across the cluster. This type of data storage follows the write once read many (WORM) model. The files once written are only appended, deleted and cannot be modified to maintain the data coherency.

Data node failure is inevitable and appears as part of process rather than exception. To achieve high availability, data blocks are replicated to multiple datanodes [8]. HDFS provides replication, fault detection and implicit data block recovery. It maintains auto corrective and uninterrupted storage access.

1.3.1 Namenode

The Namenode maintains the file system metadata as HDFS directory tree and operates as centralized service in the cluster. It provides the mapping between filename, data block

locations and datanodes on which data blocks are stored. It also maintains the transaction logs to record the modifications in the file system. Clients interact with the namenode for common file system operations like open, close, rename and delete.

The namenode is responsible to maintain the records of jobs running on the data node. It also checks the heartbeat of the datanodes and maintains a catalog of running data nodes. Namenode provides the membership to the data node to participate in the cluster. So namenode is the heart of the HDFS.

1.3.2 Datanode

In addition to name node, there are number of data nodes. Datanodes stores the blocks of files [13]. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of data nodes. The data nodes are responsible for serving read and write requests from the file system. The data nodes also perform block creation, deletion, and replication upon instruction from the NameNode. Datanodes are responsible to the track the running jobs through a process called tasktracker. For keeping the records upto date, the data node periodically reports all of its data block information to the namenode. Periodically, Data nodes send their heartbeat signal to the name node server.

1.3.3 Namespace

The namespace is a live record of the HDFC located on the centralized name node server. It is a directory tree structure of the file system which documents various aspects of the HDFS such as block locations, replication factor, load balancing, client access rights and file information. The namespace serves as a mapping for data location and helps HDFS clients to perform file system operations.

The metadata is stored as a file system image (fsimage) files which is a persistent checkpoint of the file system [4]. The edit log records the write operations submitted by the file system clients. When the edit log size exceeds a predefined threshold, the name

node moves the transaction log into live memory and applies each operation to the fsimage.

A backup copy is periodically stored on the secondary name node to guard it against failure. The whole namespace is in the cache to provide high availability to client applications and data nodes.

1.3.4 MapReduce

MapReduce is a programming paradigm. MapReduce jobs are submitted to namenode. Then, these are forwarded to appropriate datanodes where the data blocks reside. Datanodes execute the jobs on local machine and return the results.

The computation in MapReduce is divided into two tasks, one is called Mapper and other is called Reducer [2, 18]. In the mapping task, the data is processed into key and the value pairs with a minimal coordination of data nodes. In the reducing task, each output from data nodes is combined to produce single output for the application.

1.4 Architectural Limitations

HDFS has architectural limitations that restricts its features like high scalability, high availability because the namenode server becomes a bottleneck for the Hadoop cluster. So it affects the performance of large distributed storage system.

The architectural limitations of HDFS are as follows:

1.4.1 Scalability Issue

The centralized namenode server stores the namespace of HDFS in live memory for high performance. As the storage capacity of a cluster grows, more namenode server memory is required. Shvachko [3] estimates that 1 Gigabyte(GB) memory is required to cache namespace of 1 Petabyte (PB) of data in cluster. One petabytes of data storage requires approximately 100 million data blocks in HDFS. In order to accommodate data of 100 million blocks, the HDFS cluster needs 10 thousand of nodes with eight 1 TB drives. In

case of total storage capacity of cluster as 60 PB, it requires a minimum 60 GB of memory in namenode server to provide full caching of namespace. Beyond 60 PB of data, it requires additional memory (RAM). The additional RAM achieves vertical scalability. It is unrealistic to add any amount RAM to the name node server.

As the number of data nodes increases, the work load on a single centralized namenode server increases and has a great impact on the performance and availability of the cluster. It is therefore, increasing work load and memory that restricts the scalability of the Hadoop cluster [6, 20].

1.4.2 Availability Issue

A single centralized namenode server is more prone to failures. In case namenode server fails, the whole cluster data nodes are unavailable to client applications. The recovery time depends on the amount of metadata data [8]. As the name node server starts, it takes time to load the namespace to cache. This adds up to the unavailability of service. So a single centralized namenode server becomes a Single point of failure (SPOF) and this failure may be attributed because of denial of service or distributed denial of service attack.

1.4.3 Performance Issue

Namenode is responsible to check the heartbeat of data nodes, maintaining the namespace of the cluster and responding to client application queries for metadata. As the data nodes and data blocks grow in the cluster, the performance reduces after reaching the thresholds. So a single centralized namenode becomes a bottleneck. The whole cluster performance depends on the performance of name node server.

1.4.4 Quality of Service

Different client applications require different storage limits, computation and bandwidth. In the present HDFS, there is no facility to provide QoS policy to client applications. The client applications are no able to control the data storage and data nodes to perform

computations. The present architecture is build for single application and has full access to datanodes. The architecture does not support to prioritize applications to store data and execute jobs.

Managing many small clusters for different application is cumbersome and costly affair. A single big cluster is always easy to manage for different applications. But, different applications have different resource requirements like storage, processing etc and it also becomes the basis on which companies generally charge from the clients. So, quality of service is important from client's application point of view. Moreover, different client applications are storing their data on single cluster and permissions/access control is an important issue that needs to be taken care off.

1.4.5 Security

Application data security is not addressed throughout the life cycle of Hadoop. Hadoop cluster was mainly designed for single application data. The client application has full access of datanodes and they can read/write any requested blocks on the data nodes.

1.4.6 Client Application Intimacy

Client application intimacy depends on the number of queries for a given file on the name nodes. As the size of namespace grows these queries take more time. Client application intimacy is an approach to improve the performance of repeated queries for same filename, datablock and datanode. Otherwise, the applications have no advantage and have to search through the namespace.

1.5 CHORD

Chord's main goal is the location of entities in P2P environments, like documents, files, or any resource that one might want to share in a computer network [20,21,25]. It is a distributed lookup protocol. It is done by means of a single operation that maps a given key onto a node. Data location can thus be easily implemented on top of Chord by associating a key with each resource item.

Chord shows adaptation feature when node failures occur, when nodes join and leave the network in small interval. Another feature along with the adaptation of the network is efficient query processing. In chord, namespace data is distributed by distributed hash tables and it addresses the limitations of single namenode server like scalability, failover and performance. Chord is a simple and decentralized environment which is symmetric, auto adaptive and provides consistent performance of resource lookup.

Chord uses two mathematical concepts hash functions and modular arithmetic [25]. The maximum number of nodes $N = 2^m$ where *m* refers to the number of bits that limit the identifier space.

1.5.1 Hash functions

Each node belonging to the network is assigned a number through the use of a hash function. Each item that is going to be looked up has such a numeric association. Hash functions usually convert an input from a large domain into an output in a smaller range. The domain can be from any number, or data but can be represented in a numeric way. In the case of the IDs of nodes belonging to a Chord network, the IP address, or the IP and port pair is a good candidates as input and thus serves as a value from the domain in the hash function. In the case of resources, the name of a file or resource, or even their contents can also be represented in a numeric way. The hashing is only possible because of numeric association with input domain. The hashing is used to cut down the range of disperse input values.

1.5.2 Modular Arithmetic

Chord is a protocol whose behavior is based entirely on the topology that the network forms. Modular arithmetic is the cornerstone upon which this topology lies. So hashing provide the numeric representation for both nodes and items which belongs to a certain range of numbers [0, X). This numbers are operated in a modulo arithmetic. Table 1.1 shows an example of a modulo 8 table of 64 numbers.

Table 1.1 Table of input values (0-63) mapped to output value (0-7) using modulo 8

X	mod(X,8)	X	mod(X,8)	X	mod(X,8)	X	mod(X,8)
0	0	16	0	32	0	48	0
1	1	17	1	33	1	49	1
2	2	18	2	34	2	50	2
3	3	19	3	35	3	51	3
4	4	20	4	36	4	52	4
5	5	21	5	37	5	53	5
6	6	22	6	38	6	54	6
7	7	23	7	39	7	55	7
8	0	24	0	40	0	56	0
9	1	25	1	41	1	57	1
10	2	26	2	42	2	58	2
11	3	27	3	43	3	59	3
12	4	28	4	44	4	60	4
13	5	29	5	45	5	61	5
14	6	30	6	46	6	62	6
15	7	31	7	47	7	63	7

The objects of namespace become input and are mapped to 8 nodes using modulo 8 operations.

In case, a chord ring has m=6, it means that the chord ring has maximum 64 name nodes. If the keys are also 64 then there is one to one mapping between key and name node as for 640 million keys, there are 10 million keys per node mapping.

In the figure 1.3, there are 10 nodes, with identifiers 1, 8, 14, 21, 32, 38, 42, 48, 51 and 56 live from 64 node chord ring. Some of the nodes are responsible for a set of keys present in the network. The keys 10, 24, 30, 38 and 54 are in the network. Each key is held by its responsible node. A node with identifier id d is responsible for document p such that d=sucessor(p). With successor list only, chord provides a node lookup of linear complexity. This can be brought down to O(log N) by adding finger table for each name node.

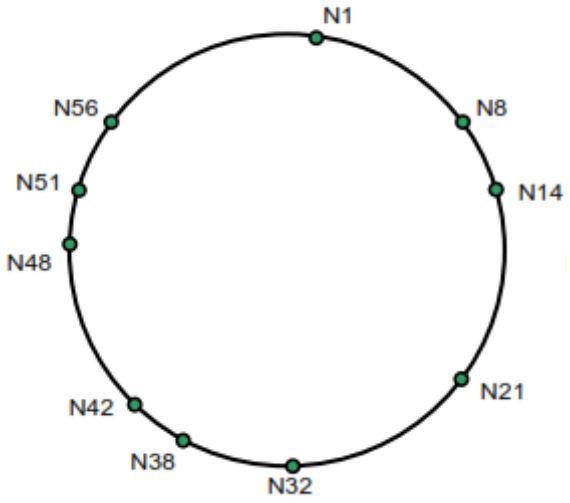

Figure 1.3 A chord ring of ten nodes

Finger Table

A fingers table is a data structure with m entries where m refers to the number of bits that limit the node identifier space [25]. Each ith entry in the finger table holds, $0 <= i < m$ the identifier of successor($id+2^i$). This structure offers improvement in node lookup performance. This structure is the one that will ensure that lookups will be performed with cost O(log N). Table 1.2 shows an example of the finger tables of nodes 8 for the chord ring shown in Fig. 4.4. It has six entries as m=6.

Table 1.2 Finger Table of namenode 8 as per chord ring

Finger Table of Nanenode 8			
Finger Level	**Aim Node Id**		**Successor of Aim Node**
0	8+power(2,0)	=9	14
1	8+power(2,1)	=10	14
2	8+power(2,2)	=12	14
3	8+power(2,3)	=16	21
4	8+power(2,4)	=24	32
5	8+power(2,5)	=40	42

Node Join Operation

When a node joins the namenode cluster, its successor and predecessor are NULL. The node X joins with his identifier and request any node Y present in the network for its successor [25]. When X receives a reply, it stores its successor's id. In bidirectional Chord ring it also looks for predecessor. Stabilization and fixFingers routines are than called and executed periodically. This ensures that the predecessor and finger tables are up to date. Here is a part of the pseudo code involved in the creation of a Chord ring with a node. Here, x joins a chord ring containing node y.

```
//Creating a new Chord ring

n.create( )
{
  predecessor := nil;
  successor := nil;
  schedule(stabilize);
  schedule(fixFingers);
}

 // x joins a Chord ring containing node y
x.join(y)
{
  predecessor := nil;
  successor := y.findSuccessor(x);
  schedule(stabilize);
  schedule(fixFingers);
}
```

The key exchange is than happen and allows the namenode to go down or come up with minimal of disruption. In stabilization procedure, if node 46 is responsible for keys 35, 38

and 40, and a node join with identifier 39, it takes the responsibility of keys of node 35 and 38. Then, it starts building finger table to improve the resource lookup performance.

Resource Lookup

In resource lookup, a name node with identifier x is interested in locating key k. If k lies between x and x's successor, the answer is x's successor. Else, the lookup sent to the closest preceding node in the network that x knows about by inspecting the fingers table [25]. In this way, by forwarding request, the node lookup operation steps closer its destination key k. In each request forwarding step, the request is processed on the local node then forwarded. In case key is found, it stops the lookup process otherwise request is forwarded to next hop in finger table. It achieves resource lookup in O(log N) complexity.

Node Failure

A node sends periodically a message to its predecessor to check its heartbeat. If a node does not receive a reply in a specified time, it considers that its predecessor is out of order and set its predecessor pointer to NULL and runs the stabilization process.

1.6 Our Contribution

The namespace is distributed using distributed hash table. It has improved its scalability and tolerance to failures. It provides quality of service to different client applications and application data security. In this research, the bottlenecks of traditional Hadoop are discussed and are addressed in proposed architecture.

The new evenly distributed namespace management system is designed to achieve high scalability without compromising the performance much.

Hadoop Architecture and Its Limitations

In this thesis, the limitations of Hadoop architecture are discussed and analyzed. The limitations in Hadoop and its extensive use in data intensive computing has forced it to

achieve high scalability, guard from single point of failure, high availability, quality of service and security of application data. These limitations are as follows:

High Scalability through Distributed Namespace

Different namespace partitioning techniques, their limitations and performance issues are discussed and analyzed. A new technique to partitioning the namespace of Hadoop is proposed. The namespace is hashed on the basis of parent directory. Then, hash values are distributed evenly with peer name nodes.

Evaluating New Design of Hadoop Distributed File System with Previous One

The new design is evaluated with the single namenode architecture of Hadoop distributed file system and other proposed techniques. The comparisons are based on scalability, availability, failover and performance of namespace queries.

1.7Thesis Organization

The thesis report comprises of six chapters. The first chapter gives introduction to the data intensive computing, Hadoop architectures, its components and MapReduce jobs. In this chapter, the architectural limitations like scalability, availability, performance issues, quality of service and client intimacy to improve the performance are discussed. Second chapter two concentrates on the efforts of researchers and their recent published approaches to overcome the limitations. A brief discussion of these design and techniques is presented. In third chapter, the problem of statement is stated. In fourth chapter, a new approach to overcome these issues is proposed. In this proposed Hadoop architecture the namespace is distributed by using distributed hash tables. Fifth Chapter presents the performance evaluation of proposed Hadoop with the existing approaches. Last chapter concludes our work and presents the future scope.

CHAPTER 2

Literature Review

Since the weakness of the centralized namespace storage of Hadoop has surfaced up, there have been attempts in research publications providing strategies for eliminating the single point of failure and address the scalability issue of the architecture. In this chapter the techniques to address these issues are discussed.

Dhruba Borthapur discussed the issue of single point of failure of Hadoop and suggests improvements in failover of Namenode server. The AvtarNode [8] was developed to address the issue of failover and a mechanism to deal with the single point failure of the Namenode. The primary AvtarNode runs exactly same as the Namenode and writes its transaction logs into the shared NFS filer. Another instance of AvtarNode is instantiated that runs in standby mode. Using NFS, It keeps reading the transaction logs from the same shared NFS filer and keeps feeding the transaction logs to the encapsulated Namenode instance. The name node within standby AvtarNode is kept in safe mode to prevent it from participating in the cluster activities. The approach is show in figure 2.1.

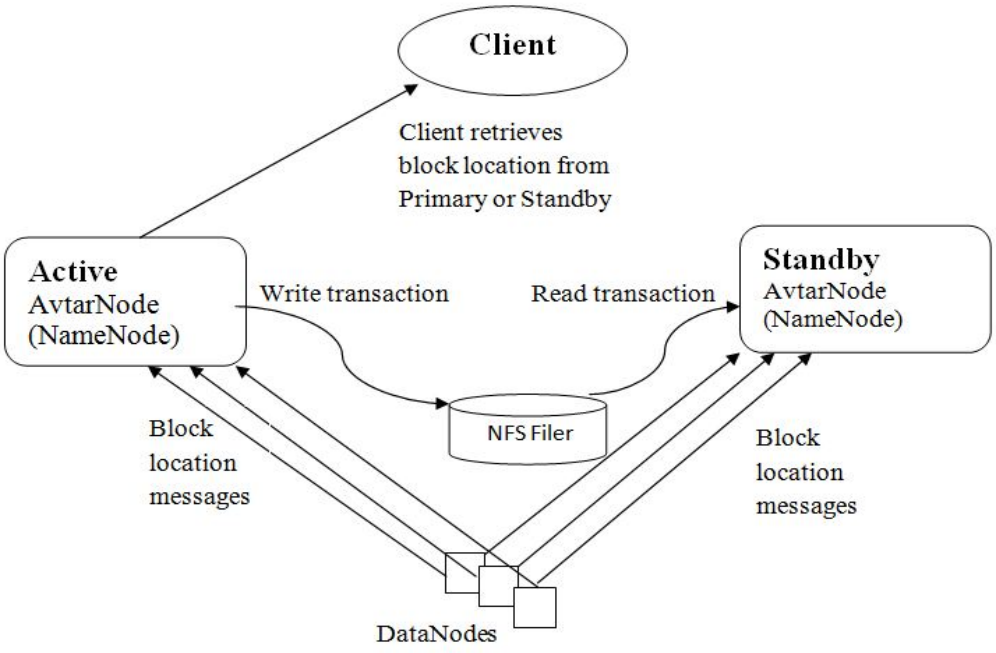

Figure 2.1 AvatarNodes: the Active/Standby AvatarNode [8]

HDFS clients are configured to access the AvatarNode via virtual IP address (VIP). If the primary AvatarNode fails, the failover is performed by switching the VIP to the Standby AvatarNode. As the clients receive the entire data block list and replica locations at the time of file open operation, file read operations are not affected by the failover period. When the file is being written during the time of failover, client receives an I/O exception after the failover event. The failover does not affect MapReduce task execution as the framework is designed to retry any failed tasks. Hence the failover happens in minimum time and kept all functionalities intact.

The AvatarNode is effective mechanism to guard against Namenode failures and keeps the namespace data protected. However, the AvatarNode does not address the high scalability of the architecture and still has the single point of access to the cluster. As the namespace grows, the two name node servers do not load balance the work. This approach provides failover and not able to accommodate the large namespace.

Feng Wang's discussed the metadata replication based solution to provide high availability and failover technique [9]. The solution consists of three major phases: the first is the initialization phase which initializes the execution environment of high availability. The second is the replication phase which replicates metadata from critical node to corresponding backup node at runtime. The third is the failover phase which resumes the running of Hadoop despite the critical node being out of work. As the file system information and EditLog transactions are stored as a backup copy on the Namenode, the solution provided in this paper only concentrates on the replication of critical metadata

In the initialization process, multiple slave nodes register with the primary node for the up-to-date metadata information. The second stage of replication resumes after initialization of the slave nodes. Figure 2.2 explains the architecture designed for critical metadata replication. The primary node collects metadata from clients request processing threads and sends it to the slave node. The slave node handles the processing of the received metadata which includes in memory and in disk processing of clients requests.

The slave node sends a heartbeat signal to the primary node to keep track of its live status.

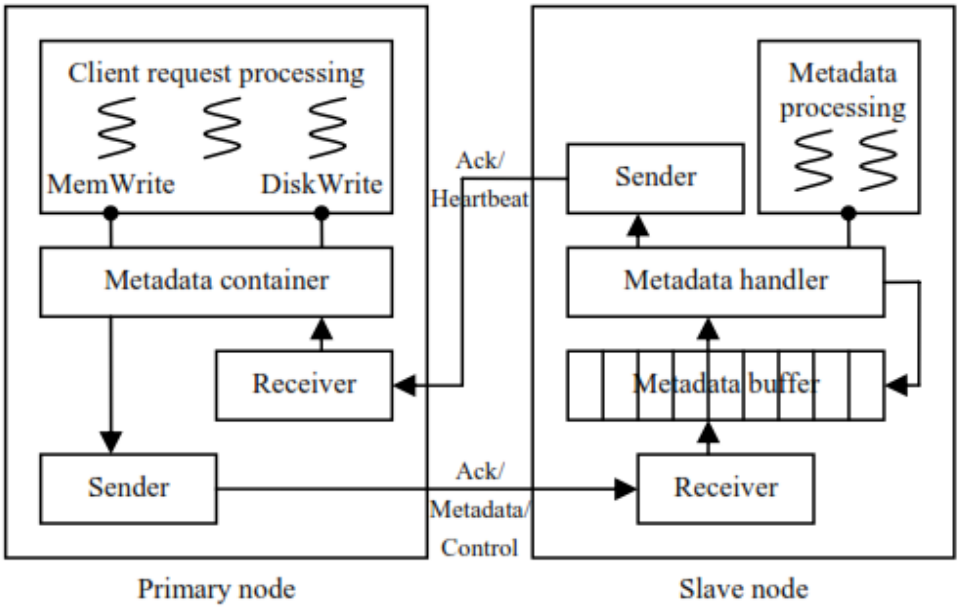

Figure 2.2 Replication of namespace [9]

In the failover state, when slave nodes fails to receive the acknowledgement of its heartbeat message, leader election algorithm is used to decide which slave node takes place of the primary node. Upon selection of a slave node as the primary node, it changes its IP address to the IP address of the failed primary node, thus finishing the failover process.

It presents an adaptive method for failure recovery of the Namenode by metadata replication with further reduces failover duration, but it does not solve the issue of single point of failure with Hadoop. The solution is suitable for medium amounts of files but not for higher amounts of I/O requests. The metadata replication does not improve the scalability of the architecture.

The Hadoop RPC server implementation [7] has a single listener thread that reads data from the socket and puts them into a call queue for the Namenode threads. Namenode

gets to process the RPC requests only after all the incoming parameters copied and de-serialized by the listener thread. For a heavy load cluster, a single thread operating the RPC listener tasks is a bottleneck causing the clients receiving RPC socket errors. Due to this bottleneck, clients are unable to utilize power of the Namenode.

It is observed that most workload of the Namenode appears from read/write operations of the data. The Namenode metadata management was enhanced by creating a pool a RPC reader threads which works to decentralize the RPC requests from the clients. Most of the file system operations are read only and do not trigger any synchronous transactions. By changing the current File System name system lock to readers-writer lock, the performance of the Namenode improved significantly.

The solution improves the performance of a heavy usage Hadoop cluster by improving the bottleneck of RPC listener thread. The solution was effective in improving the performance of the Namenode to handle heavy workload, but it fails to provide a solution to the linear scalability and single point of failure of the Namenode. To store thousands of petabytes, the cluster is not able to accommodate metadata in cache.

George Porter [5], provides a solution to meet the increasing demands of namespace storage of the cluster. Porter discusses the use of a decoupled Datanode architecture to provide increased data storage and computation in Hadoop [5]. The paper introduces SuperDataNodes which are servers containing more disks than the regular nodes in Hadoop. It host amounts of data equivalent to the storage capacity of many DataNodes. The design is a storage-rich architecture of Hadoop.

Figure 2.3 shows a SuperDataNode with a storage pool made up of several disks. From this pool, several file systems are built with the help of virtual machines with its own network interface assigned. The SuperDataNode is much richer than average storage layer with large magnitude of disks and network bandwidth. Each virtual machine forms a separate DataNode in the network where the jobs are executed individually by separate task tracker servers.

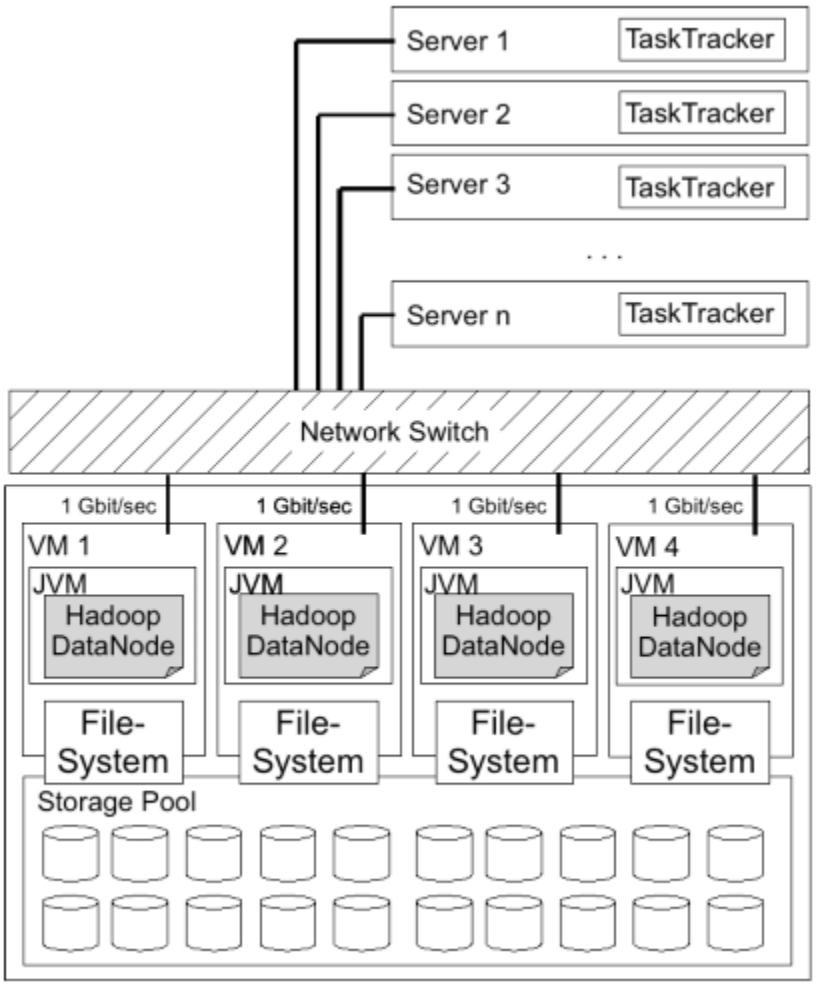

Figure 2.3 Architecture of SuperDataNode [5]

For storage of N bytes in the cluster with c being an average storage capacity of a single DataNode, the total number of DataNodes required to form the Hadoop cluster is N/c. The use of SuperDataNode decouples the amount of storage in HDFS from the number of DataNodes providing increased storage capacity with less number of DataNodes. By separating the storage from task trackers, the task tracker servers are turned off when not in use in order to lower the power consumption of the cluster.

Since a single SuperDataNode accommodates data worth many DataNodes, its failure has significant impact on the storage. The use of SuperDataNode has no impact on the

metadata storage. As a result, it does not improve scalability of the architecture. The use of SuperDataNodes is not a cost effective solution to improve the storage capacity. Also, the network bandwidth of the architecture is affected due to single point of access to a large amount of data.

Apache group came up with a solution to called federation that means the name nodes are independent and don't require coordination with each other. In order to scale the name service horizontally, federation uses multiple independent name nodes servers. The name nodes are federated [22] that means the name nodes are independent and don't require coordination with each other. The datanodes are used as common storage for blocks by all the name nodes. Each data node registers with all the name nodes in the cluster. Data nodes send periodic heartbeats and block reports and handles commands from the name nodes.

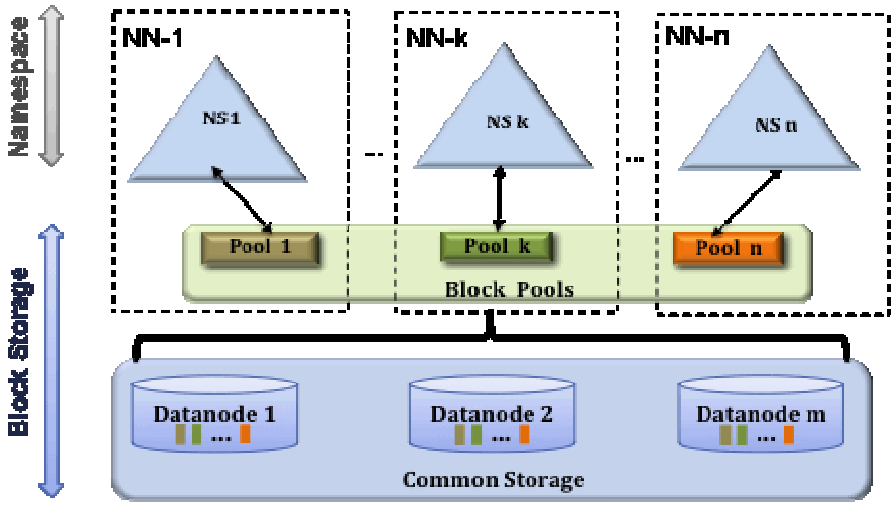

Figure 2.4 HDFS Federation [22]

A Block Pool is a set of blocks that belong to a single namespace. Datanodes store blocks for all the block pools in the cluster. It is managed independently of other block pools. This allows a namespace to generate Block IDs for new blocks without the need for coordination with the other namespaces. The failure of a Namenode does not prevent the datanode from serving other Namenodes in the cluster.

A Namespace and its block pool together are called Namespace Volume. It is a self-contained unit of management. When a Namenode/namespace is deleted, the corresponding block pool at the datanodes is deleted. Each namespace volume is upgraded as a unit, during cluster upgrade.

This approach is suitable for running many independent namespace in one cluster. All these namespaces are still not contributing to the scalability of single namespace and big cluster deployment still depends on single name node server. Though the namespace data is store on independent datanodes but still namespace cache is not distributed and has scalability limitations. This idea is good to have multiple namespace in a single cluster.

A federated cluster is able to store more data and handle more clients, because it has multiple name nodes. However, each individual name node is subject to the same limits and shortcomings, such as lack of High Availability (HA), as a non-federated one. The federated approach provides a static partitioning of the federated namespace. If one volume grows faster than the other and the corresponding Namenode reaches the limit, its resources cannot be dynamically repartitioned among other name nodes except by manually copying files between file systems.

CHAPTER 3

PROBLEM STATEMENT

HDFS is based on an architecture where the namespace is decoupled from the data. The namespace forms the file system metadata, which is maintained by a dedicated server called the namenode. The data itself resides on other servers called datanodes. The namespace consists of files and directories. Files are divided into large blocks. To provide data reliability, HDFS uses block replication. Each block by default is replicated to three datanodes. Once the block is created, its replication is maintained by the system automatically. The block copies are called replicas. The name node keeps the entire namespace in RAM. This architecture has natural limiting factors on the memory size, the number of namespace objects (files and blocks).

The growing namespace limits the scalability of Hadoop with single name node server. It becomes a problem for big cluster deployment. So far, Hadoop has successfully run on 10,000 data nodes with approximate 60 GB of RAM on single name node server. The growing size of namespace requires additional memory to cache and data nodes to store data blocks. Periodically, each data node sends block status report and heart beat signal to name node server. So, the growth of namespace has become a challenge for the scalability of Hadoop.

It is obvious from the statistics mentioned in literature that to store 1000 billion of files, it requires a higher amount of memory which is normally not available in a single server. To store same number of files, it requires approximately 100 thousand data nodes. HDFS is unable to cache the metadata of 1000 million files and is susceptible to single point of failure. A new Hadoop architecture has been proposed to address the issues of namenode scalability and single point of failure. This approach is based on distribution of namespace using distributed hash tables. The different types of applications require a special amount of memory for their computation. The proposed architecture of Hadoop is simulated on multiple name node servers.

CHAPTER 4

Solution Design

The large size of namespace catering millions of clients and billons of files and directories imposes a big challenge to provide high scalability and performance of metadata services. In such systems a structured, decentralized, self organizing and self healing approach is required.

The proposed architecture addresses the issues to achieve high scalability, SPOF (Single Point of failure), high availability, load balancing, security and quality of service without compromising the performance.

4.1 Distributed Hash Table Based Namespace

Distributed hash table divides the namespace of HDFS to multiple namenode servers. To achieve very high scalability and availability, divided namespace is replicated on different name nodes. The main goal for building such a system is to cater the growing demands of namespace and seamless support to high scalability. The current namespace limit is 100 million files. Static partitioning allows it to scale the federated namespace to billions of files. Estimates shows that implementation of a dynamically partitioned namespace will be able to support 100 billion objects. DHT is a managed and structured approach for the scalability of HDFS.

Availability is another strong motivation for the distributed hash table based namespace. A HDFS installation with a NameNode operating in a large JVM is vulnerable to frequent full garbage collections, which may take the Namenode out of service for several minutes. This leads to saturation of client's application for a while. A failure of the namenode makes the file system inaccessible and takes time to recover.

There are structured, decentralized and peer to peer approaches like Pastry from Rice, Microsoft, CAN from UC Berkeley ICSI/ICIR, Chord form UC Berkeley MIT, Tapestry from Berkeley, Symphony, Viceroy and Kademlia.

Considering all above mentioned concerns, the thesis proposes an improved Hadoop system architecture that will provide dynamic distribution of namespace to achieve very high scalability, availability and guard the system from single point of failure. The design not only eliminates the limitations but also improve the Hadoop core functionalities.

Namespace Distribution by Hashing:

Firstly, the hashing is done on the basis of parent directory path. This approach controls the migration that happens due to renaming of directory. The directory structure is hierarchical. An example of a file under directory /prod/data/result/result1.txt or /prod/logs/ is shown in Figure 4.1.

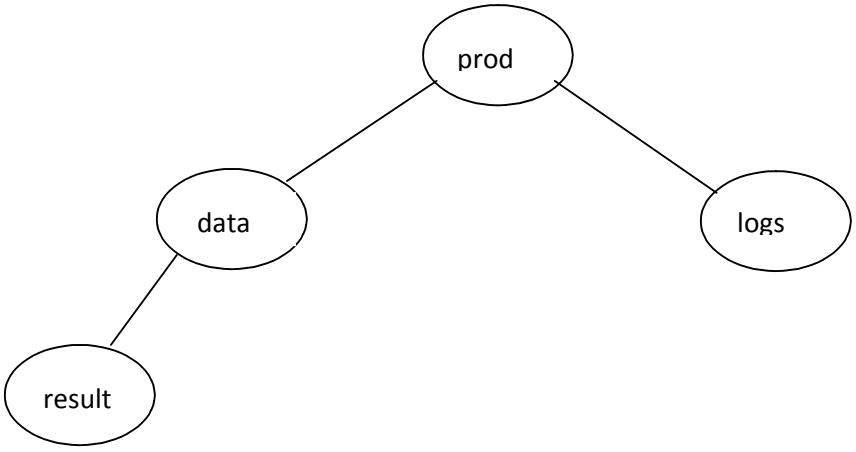

Figure 4.1 Directory Structure

The distributed structure parameters are given in Table 4.1.

Table 4.1 Metadata of objects

ObjectID	ObjectName	ParentId
1101	/prod	0
1150	/prod/data	1101
1110	/prod/logs	1101
1190	/prod/data/results	1150

The namespace of directory structure described in Figure 4.1 contains other information than the above mentioned attributes like the access permissions. It separate tables for mentioning the userid, permission level and relation of access log of the objects. The objects are added and removed. / is the root of hierarchy and its object is 0. The user-object relation depicted in Table 4.2.

Table 4.2 User-Object Relationship

Userid	Objectid	Permissions	With grant option
501	1101	7	Y
501	1190	7	Y
502	1101	5	-
502	1190	5	-
….	….	….	….

The relation given in Table 4.2 helps in defining access and privileges to users. The admin of cluster or application owner grants or revoke permissions to users. Number seven is regarded as full read, write and executions permissions. The permissions value is same as it is in Linux File System. This relation may contain more attributes like admin option under which a user can grant permissions to other users. The objects are distributed to namenode servers as per Hash Index Values (HIV) and are given in table 4.3:

Table 4.3 Distributed hash value Index for namenode server

HIVFrom	HIVTo	Namenode Server
0	400	NN0
401	800	NN1
801	1200	NN2
1201	1600	NN3
…..	…..	….

This relation contains range of values for which the namenode is responsible. This relationship is also used to find the namenode sever which contains the specific hash value. The query for the metadata is sent directly to the namenode. The size of this relation is small and namenode do not often goes down and come up frequently. This relation not only speeds up the search but also perform the load balancing of namespace to namenode.

4.2 Proposed System Architecture of Hadoop

The namenode servers form a ring and namespace is distributed on the namenode servers. This is different than the default Hadoop which is prone to single point of failure.

Each namenode pre-fetch the namespace it is responsible for and caches it. This caching is on the server side. The namenode also caches the relations required to hash object and HIV distribution table. As the namenode has limited set of namespace, it can easily cache and process the user request. All the metadata for a namenode server is stored in the database that may be Mysql or any other, as database gives higher throughput and control for transaction processing.

The caching of metadata is implemented by B+ tree which places the metadata in memory while the system starts and joins the Namenode cluster. Each namenode maintains its successor and predecessor namenode lists. So a namenode is aware of its predecessor and successor and can communicate to both sides. It forms a bidirectional ring of namenode. Periodically, each namenode is monitored by its successor. So successor finds whether the namenode is up or down. If it finds it down, then it will notify to all other namenode that its predecessor is out of service. Predecessor of down node becomes predecessor of namenode.

The proposed architecture of Hadoop is described figure 4.2:

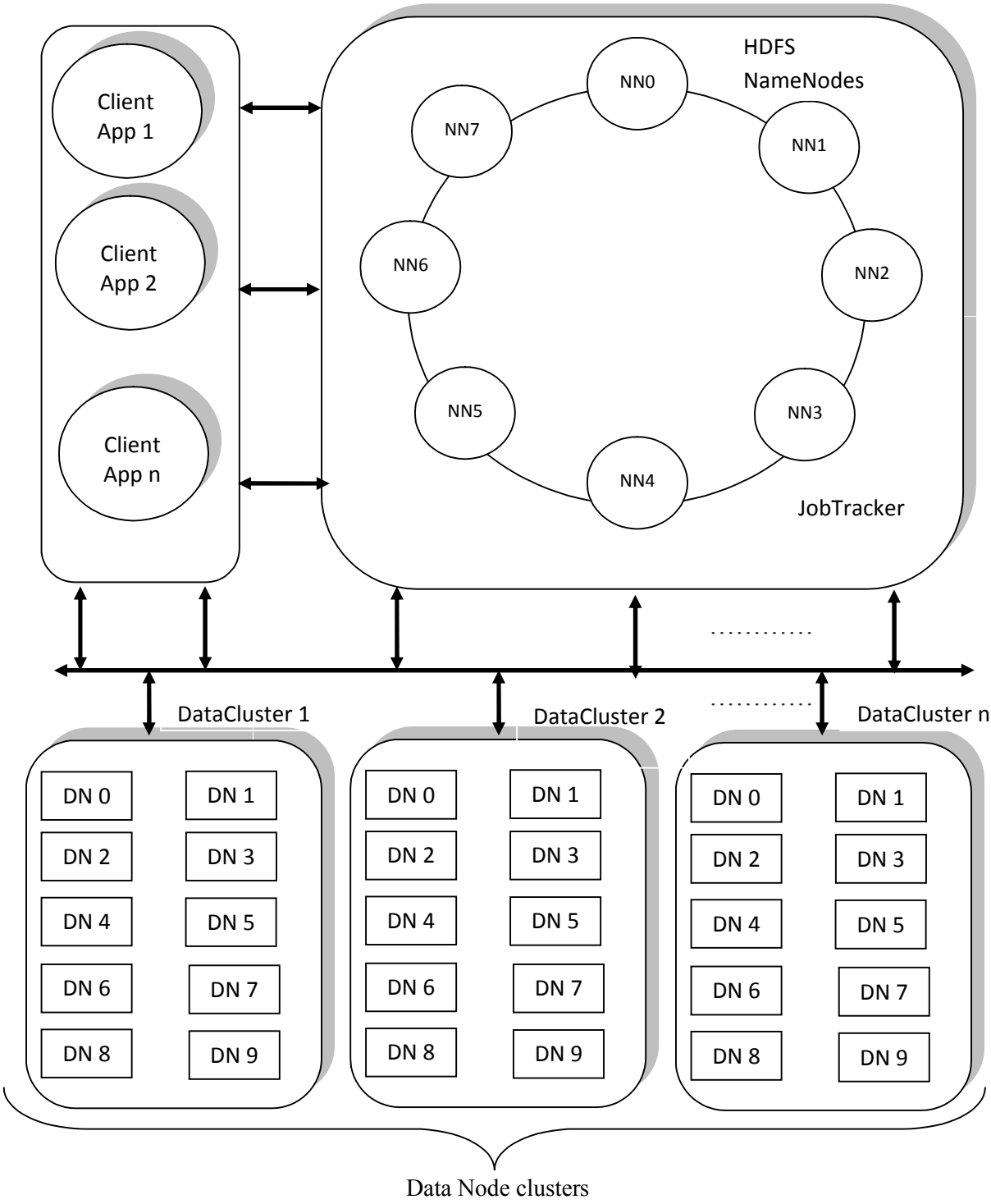

Figure 4.2 Proposed Architecture of Hadoop

4.2.1 Management of files and blocks

The files are divided into blocks of size 128 MB and are placed on the data nodes and its replication is placed on the other data node at minimum three places to give high availability. The metadata about the datanode and blockid is placed in the name node servers. In the present case, the metadata store in namenode which covers the hashing range of its parent directory and is responsible to handle the queries of clients. The client hashes the parent directory of file and lookup the namenode that maintains the range in which it falls. It contacts any of the namenode and move clockwise (increasing) or anti clockwise (decreasing). If distributed hash value table of keys is replicated on all nodes then the name node lookup complexity is O(1) otherwise it is O(log N).

4.2.2 Uniform Distributed Namespace Caching

It has been a big challenge to distribute the namespace data uniformly to load balance all namenodes. The metadata is prefetched and cached using B+ tree. A client querying the namenode is answered right away from the perfected cache. So, metadata distribution is very important to load balance the namenode activities. Here, the uniform hash key distribution is used for metadata. An uneven approach may lead to inconsistent performance and lead to underutilization of cluster capabilities. In this approach caching is symmetric and all namenodes are evenly loaded. In case a namenode joins, it takes load from the neighboring namenodes on the other hand if a node leaves; its successor namenode takes the burden of metadata and informs the other namenodes that ancestor namenode is changed.

4.2.3 Namespace Backup and Recovery

The blocks on datanode are replicated to guard against failure but the crucial namespace is kept in active and passive mode to guard the single point of failure. In earlier approaches, the namenode data was stored in files and fetched to cache. The transaction processing in files for a huge metadata is cumbersome and time consuming. If a namenode goes down, it takes a lot of time to check the integrity and then it takes huge time to cache it in the namenode server.

In the present approach, the structured metadata is stored in the database. This database gives high throughput to transactions. Using database backup and recovery tools, the backup of metadata becomes easy. Also, it has no effect on the performance of the namenode. It is another edge to proposed architecture. Though the database server takes some portion of the memory and CPU but it still compensate it by improving the availability, high transaction throughput and backup recovery features.

4.2.4 High Availability

The proposed architecture has been designed to provide high availability. A big HDFS installation with a Namenode operating in a large JVM is vulnerable to frequent full garbage collections, which may take the Namenode out of service for several minutes. In the present design, the namespace is distributed using hashing and a namenode is responsible for a small set of namespace data and that set is well guarded by replication. Hence, if a namenode goes off or is unavailable, then the successor takes charge for a while without any downtime. As the down namenode comes up its successor again redistribute the load. So it provides maximum availability.

If there are multiple namenode failures and the running namenode do not have the cache to accommodate the whole namespace, it then affects the availability. So administrator has to check that the total available memory is always greater than the namespace size. So, an administrator needs to set a critical value of running namenode server. The administrator has to ensure this critical number under which the availability and performance suffers.

4.3 NameNode

The detailed operation of namenode servers in bidirectional circular ring is discussed. The idea is based on the working of chord ring with some modification to the requirement of Hadoop. Chord is a peer to peer, decentralize, symmetric, self healing and self organizing project. The new proposed design brings in some reworking on the initialization and namenode servers. The coordination is very important among each namenode servers while the node server goes down and comes up or while new file and

directories or objects/ blocks are added to the namespace. The dynamism of chord and dynamism of namespace coordinates to achieve best possible performance.

In proposed Hadoop design, the files are mapped by their parent directory id so that a single node has the entire directory element. The probability of application lookup for same kind of file is high and the probability of these files under single directory is also high. The hashing is done on parent directory id. It allows same kind of files hashed to single name node server. The lookup for metadata is even fast as application finds all its file metadata under one namenode server and it cuts down the name node lookup for resources.

So the chord has nodes and key of resources that name nodes want to share. In Hadoop the namenodes is in the range of $[0, 2^m)$ where m is space identifier. Each namenode is having a namenode id. In Figure 4.2, eight namenodes were shown with id NN0-7. Each namenode is responsible for a set of keys. Each name node has a finger table of three entries. When a data node creates a block of a file, a key is generated for that block and is assigned to the namenode server.

The namenodes and objects/data blocks organization is explained with respect to each other in a 2^m name nodes numbered from 0 to $2^m - 1$. Key k is assigned to the first node whose identifier is equal to k in the identifier space, regardless of which name node the owner of the resource is originally, that generated this key. This namenode is called the successor node of key k, denoted by successor (k).

4.4 Chord for Namespace Management

Here, Chord is proposed with some modifications of chord ring that help to improve the performance of namespace management of Hadoop. Apart from finger table and successor list, the namenode server also maintains predecessor's pointer to make the ring bidirectional. As the namenode identifiers are in serial order, it becomes easy to check the identifier of the namenode and compare it with the requester. If it is less in value, then it goes one step back or to node's predecessor. Otherwise, it has to go clockwise, jumping

from node to node as in finger table and reaches the destination. The same thing happens when a key resource is searched. Hence bidirectional ring gives an edge to improve the lookup performance.

CHAPTER 5

Results and Performance Evaluations

The primary component of the design is distributed name node server in chord ring. To evaluate the performance of proposed design, planetsim simulator is used to construct the chord ring and its related data structures. Implementation of simulator is done on Oracle VM VirtualBox 4.1.14 with windows professional 32 bit machine environment. Ubuntu 10.4 is installed in the virtual environment with kernel 2.6.32 with 1 GB of RAM.

Planetsim is an object oriented simulation framework for overlay networks and services. This framework presents a layered and modular architecture with well defined hotspots documented using classical design patterns. Planetsim has been developed in the Java language. Following are the steps required to install Planetsim:

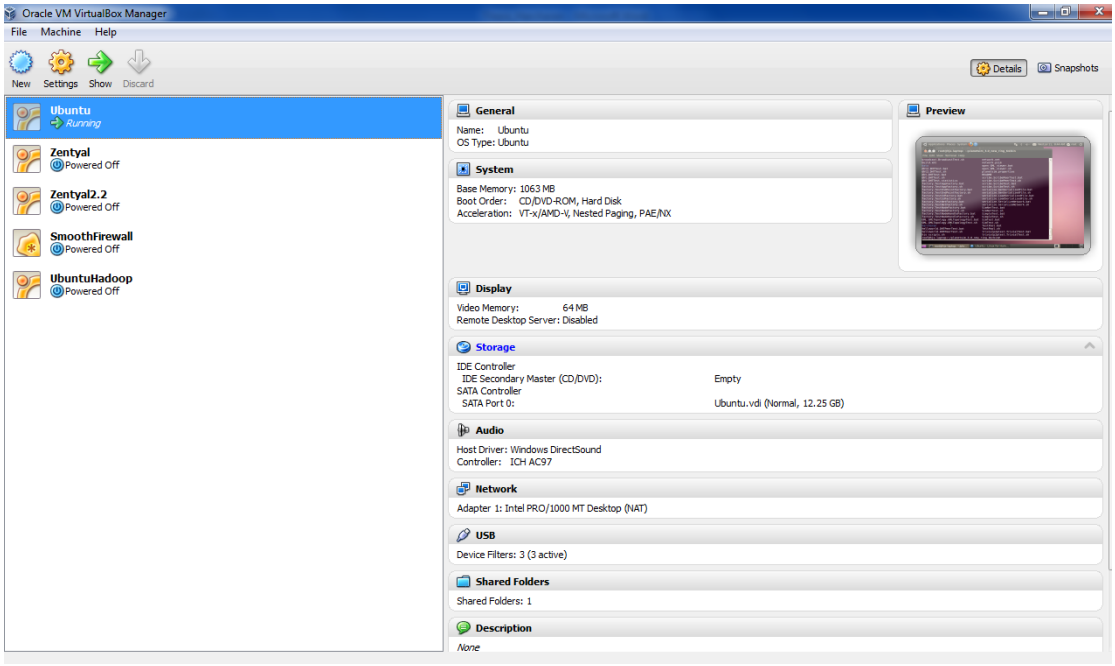

Figure 5.1 Oracle VM VirtualBox Manager

Planetsim simulator has dependencies which are installed prior to Planetsim's installation. These dependencies are Java 1.4 or higher and Ant utility. The installation

follows three steps as download the distribution from http://projects-deim.urv.cat, unzip and run the desire test. An example of running scripts on a planetsim is shown in Figure 5.2

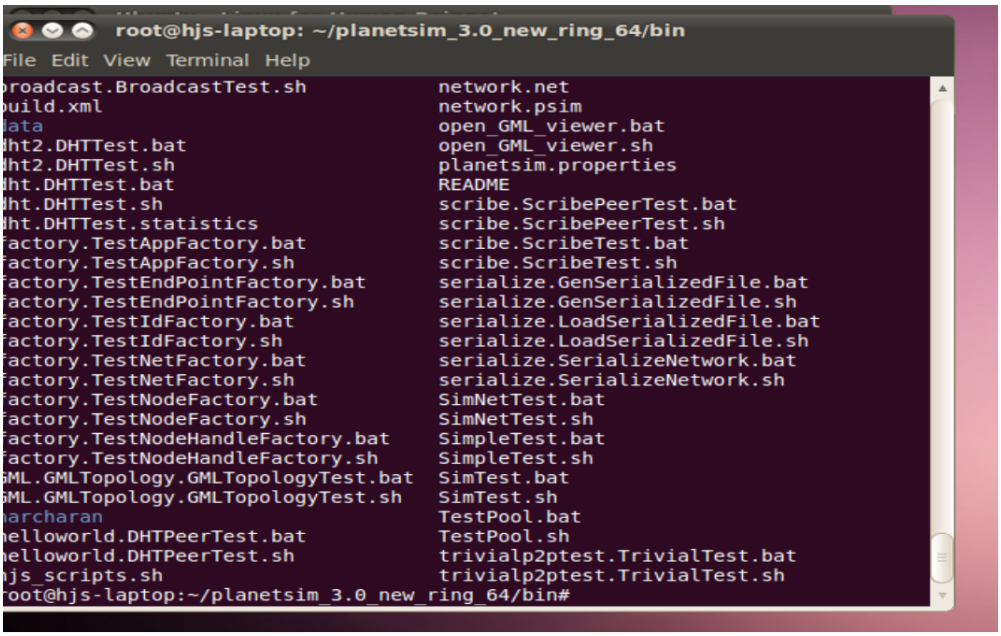

Figure 5.2 Scripts of Planetsim 3.0

A 64 chord ring is constructed having namenode shown in Figure 5.3.

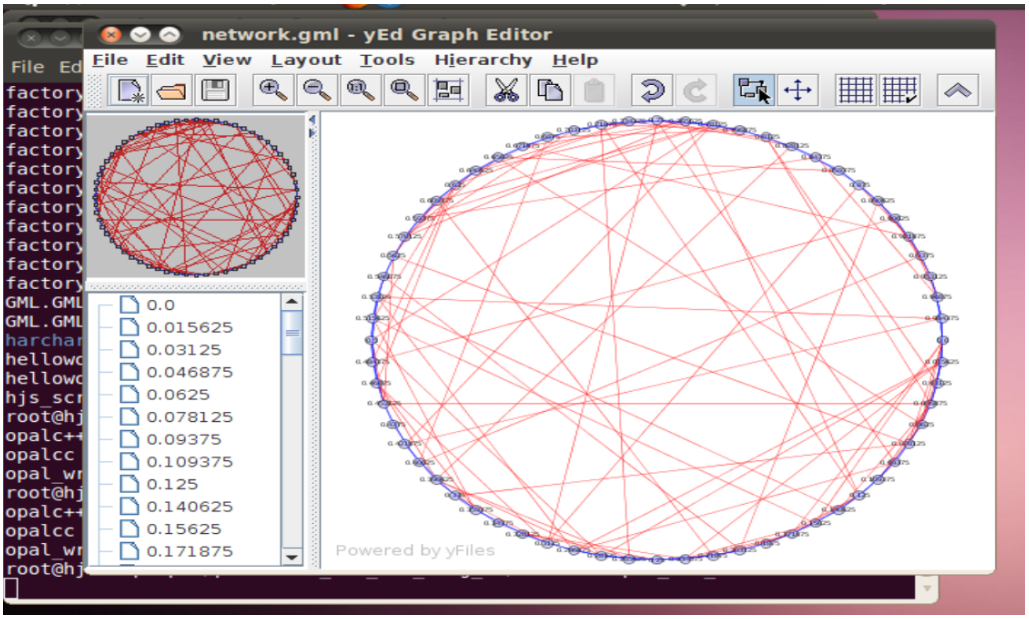

Figure 5.3 Chord Ring of 64 Name Node Servers

The test results are obtained for chord ring of different sizes for parameters such as creation time, broadcast time, random key lookup and uni-cast time and are shown in tables from 5.1 to 5.4.

The time required in creating a chord ring of name node servers and their lookup data structures like finger table, successor list and predecessor list. The network creation time affects the availability and increases mean time to recovery (MTTR).

Table 5.1 Network creation time and number of name node servers

Number of namenode servers	Network Creation time in seconds	Number of Steps
8	0.076	857
16	0.111	937
32	0.153	1115
64	0.251	1435
128	0.54	2075
256	2.472	3355

This time is proportionate to the number of namenode server. A relation of the network creation time and different name nodes is shown in table 5.1

The time required to broadcast a message in chord ring of name node servers depends upon the number of nodes. The broadcast is required when a namenode joins or leave gracefully. Name node announces its present as it joins. The relationship between the broadcast time and number of namenode servers is shown in Tables 5.2.

Table 5.2 Broadcast time on different number of name node servers

Number of namenode servers	Broadcast time in seconds	Number of Steps broadcast
8	0.021	4
16	0.026	5
32	0.03	6
64	0.038	7
128	0.055	8
256	0.108	9

The smaller time improves the performance of node joining and leaving operation.

The random key lookup time and steps is very important to find the object metadata. The lookup first searches the local name node namespace, then forwards the request to other using finger table, successor list and predecessor list. Using these data structures, name node lookup gives a managed performance and need not to use broadcasting. The lookup time of six random keys on different number of name node server is shown in Table 5.3. The smaller lookup time gives higher metadata lookup performance.

Table 5.3 Random key lookup time in a chord ring of different name node servers

Number of namenode server	No of Random key lookups	No of steps in lookup	Time
8	6	288	0.015
16	6	310	0.007
32	6	355	0.010
64	6	448	0.012
128	6	623	0.012
256	6	976	0.030

The chord ring is a managed network. Each name node server has its own lookup data structures. The key lookup requests are passed to other name node server using unicasting. Hence, unicast time affects the performance of key lookup operation. A relationship between number of name nodes and unicast time in chord ring is shown in Table 5.4. The unicast time depends upon the number of name node server and directly improves the performance of key lookup operation.

Table 5.4 Unicast time in a chord ring of different number of name node servers

Number of namenode servers	Uni-cast Time in seconds	Steps
8	0.056	47
16	0.056	56
32	0.073	66
64	0.092	67
128	0.120	75
256	0.302	89

From the above tables, it is clear that choosing the right number of namenode servers in chord ring is very important for operations mentioned above. A wrong choice of number affects the overall performance of HDFS.

To test the performance of the caching of name node servers in chord ring, a B+ tree has been implemented in Python 3.2 to cache the namespaces. Ninja IDE for python is installed to program and capture output. A snapshot is shown in Figure 5.4.

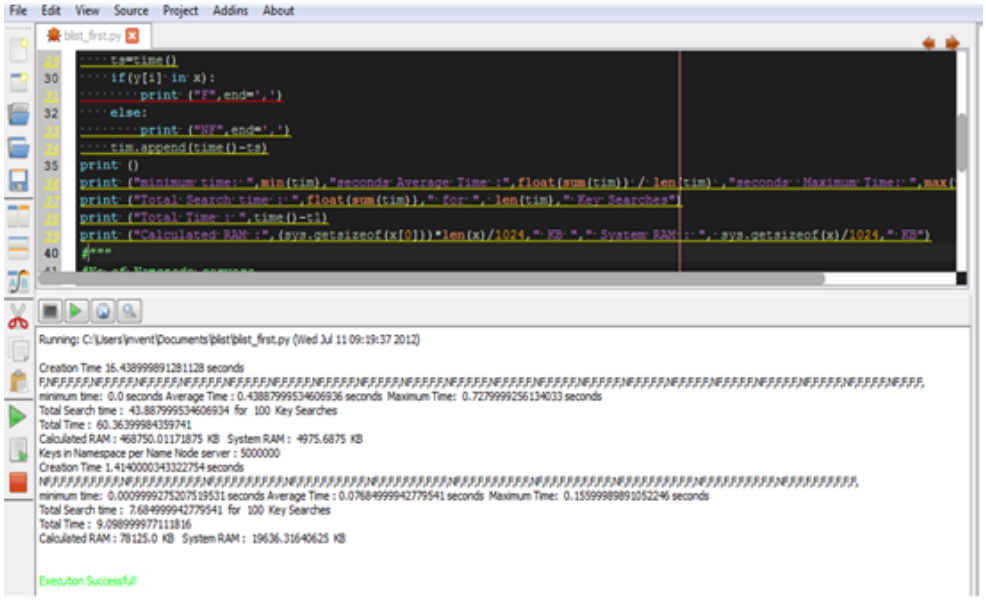

Figure 5.4 Python B+ Tree implementation for namespace caching

The B+ tree based caching of namespace for single namenode and multiple namenode servers with different number of namespace objects has been simulated in python program. To test the performance, random key searches are performed on a namespace of single and multiple name node servers. Results are collected and are show below in tables.

The initialization time is the time to fetch metadata into the memory. Pre-fetch and caching of metadata is done to improve the performance of lookups. B+ tree is in-memory data structure that Hadoop uses to cache the metadata. A relationship of number of name node server, number of objects in metadata, metadata per name node server and cache initialization time in seconds are shown in Table 5.5

Table 5.5 Cache initialization Time and metadata objects per of name node servers

Number of Namenode servers	Number of Objects in metadata	Number of Metadata per Name Node Server	Cache Initialization time in seconds
1	10000000	10000000	3.169000149
1	12500000	12500000	4.167000055
1	25000000	25000000	8.967000008
1	40000000	40000000	15.52300000
1	50000000	50000000	20.00600004
8	10000000	1250000	0.358000040
8	12500000	1562500	0.424999952
8	25000000	3125000	0.883000135
8	40000000	5000000	1.407999992
8	50000000	6250000	1.698999882
16	10000000	625000	0.176000118
16	12500000	781250	0.221000195
16	25000000	1562500	0.443000078
16	40000000	2500000	0.71600008
16	50000000	3125000	0.84100008
32	10000000	312500	0.088999987
32	12500000	390625	0.220000029
32	25000000	781250	0.209000111
32	40000000	1250000	0.358999968
32	50000000	1562500	0.417000055
64	10000000	156250	0.042000055
64	12500000	195312	0.055000067

64	25000000	390625	0.110999823
64	40000000	625000	0.167000055
64	50000000	781250	0.212999821
128	10000000	78125	0.020999908
128	12500000	97656	0.026000023
128	25000000	195312	0.055000067
128	40000000	312500	0.088000059
128	50000000	390625	0.10800004
256	10000000	39062	0.00999999
256	12500000	48828	0.013000011
256	25000000	97656	0.026000023
256	40000000	156250	0.039000034
256	50000000	195312.5	0.053999901

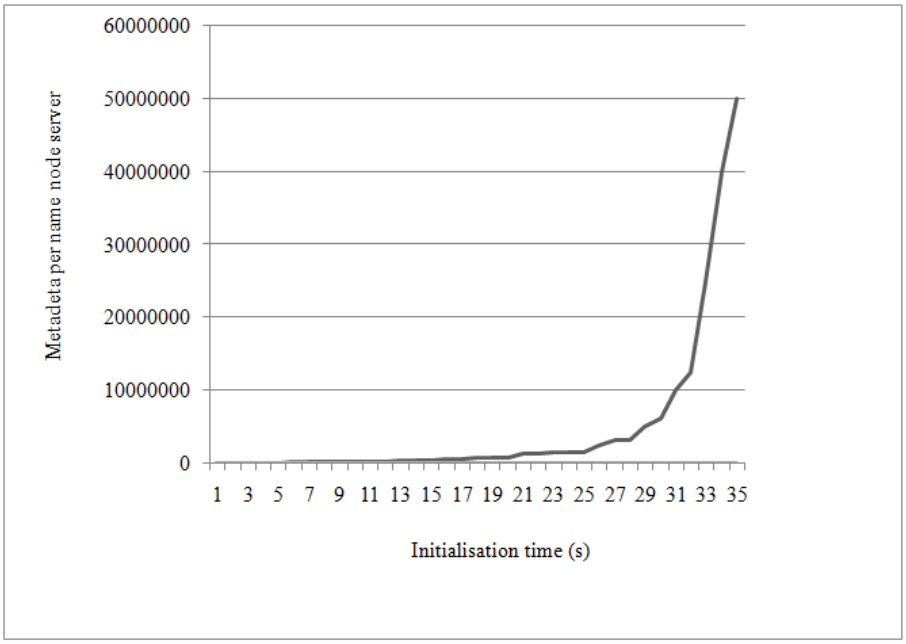

Figure 5.5 Initialization time vs Metadata objects per name node server

To evaluate the availability of the proposed system, the cache initialization has been implemented. Figure 5.5 shows that there is an exponential rise in cache initialization time for metadata objects greater than 3.5 millions. Under this number, the growth is almost linear. The system will remain unavailable while whole metadata is not cached. The large cache initialization time affects the availability of the system and mean time to recovery of the system.

The time consumed to search 100 random metadata key queries from the cache is directly related to performance of the name node server. A relation is shown among the number of name node servers, number of metadata objects, number of metadata objects per name node, number of object searches and time consumed in Table 5.6.

Table 5.6 Time consumed to search 100 keys and metadata objects per namenode server

Number of name node servers	Number of metadata objects	Number of metadata objects per name node	Number of object Searches	Time Consumed
1	50000000	50000000	100	52.1400001
1	40000000	40000000	100	40.66700006
1	25000000	25000000	100	24.65600014
1	12500000	12500000	100	12.22799993
1	10000000	10000000	100	9.644999981
8	50000000	6250000	100	9.288000107
8	40000000	5000000	100	7.944000006
8	25000000	3125000	100	4.742000103
8	12500000	1562500	100	2.32400012
8	10000000	1250000	100	1.95600009
16	50000000	3125000	100	5.146000147
16	40000000	2500000	100	4.133999586
16	25000000	1562500	100	2.315000057
16	12500000	781250	100	1.582000017
16	10000000	625000	100	0.977999926
32	50000000	1562500	100	2.35800004
32	40000000	1250000	100	1.995000124
32	25000000	781250	100	1.194000006
32	12500000	390625	100	1.372999907
32	10000000	312500	100	0.526000023
64	50000000	781250	100	1.197000027
64	40000000	625000	100	0.87500000
64	25000000	390625	100	0.679000139
64	12500000	195312	100	0.285000086
64	10000000	156250	100	0.230000019
128	50000000	390625	100	0.621000051
128	40000000	312500	100	0.478999853
128	25000000	195312	100	0.296999931
128	12500000	97656	100	0.147000074
128	10000000	78125	100	0.105999947

256	50000000	195312.5	100	0.319000006
256	40000000	156250	100	0.225999832
256	25000000	97656	100	0.14199996
256	12500000	48828	100	0.072000027
256	10000000	39062	100	0.058000088

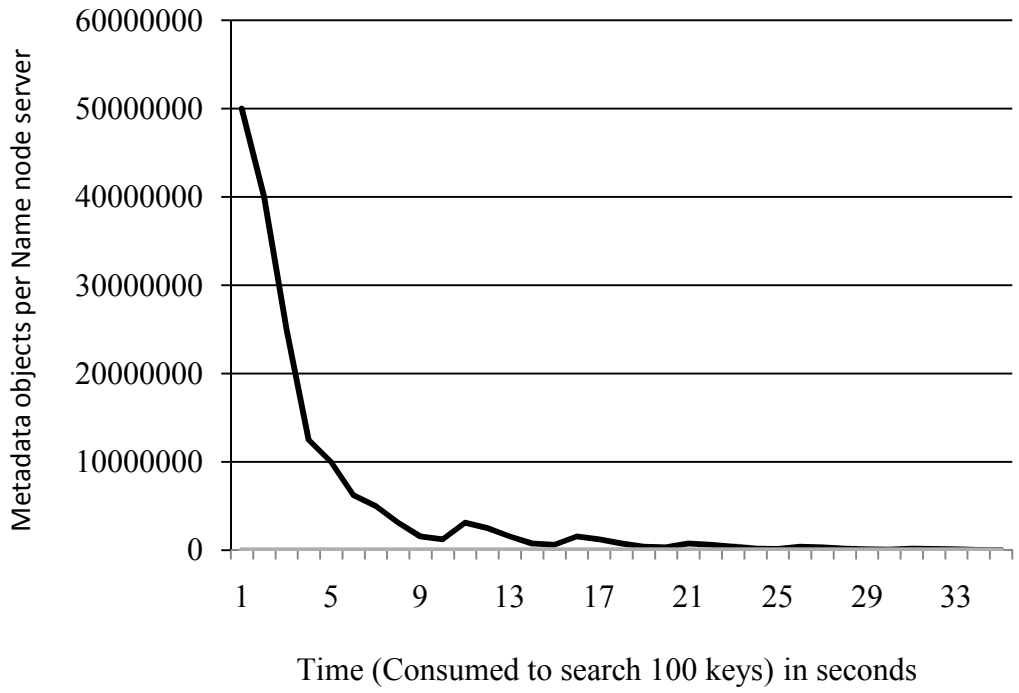

Figure 5.6 Time to search 100 random keys vs metadata objects per namenode server

The larger search time affects the name node server performance and slows down the performance of client application. From Figure 5.5, it is clear that the time consumed to search 100 keys falls substantially from 50 million metadata object per name node server to 3 millions. This fall is almost linear. It shows that around 3 million or below object per name node is best for the system performance.

The queries per system per second indicates the performance of whole system comprising of n name node server. This parameter indicates the number of client requests processed in a second. A relationship between number of name node servers, number of metadata objects, number of metadata objects per name node and maximum number of queries per system per second is shown in Table 5.7.

Table 5.7 Maximum number of queries per system per second and number of namenode servers

Number of Namenode servers	Number of metadata Objects	Number of metadata objects per name node	Max. number of Queries per system per second
1	50000000	50000000	1.917913306
1	40000000	40000000	2.458996234
1	25000000	25000000	4.055807894
1	12500000	12500000	8.177952291
1	10000000	10000000	10.36806638
8	50000000	6250000	86.13264328
8	40000000	5000000	100.7049345
8	25000000	3125000	168.705184
8	12500000	1562500	344.2340614
8	10000000	1250000	408.9979363
16	50000000	3125000	310.9210949
16	40000000	2500000	387.034388
16	25000000	1562500	691.1446913
16	12500000	781250	1011.377992
16	10000000	625000	1635.991944
32	50000000	1562500	1357.08225
32	40000000	1250000	1604.009925
32	25000000	781250	2680.066989
32	12500000	390625	2330.662941
32	10000000	312500	6083.649925
64	50000000	781250	5346.7
64	40000000	625000	7314.285714
64	25000000	390625	9425.623988
64	12500000	195312	22456.13359
64	10000000	156250	27826.08465
128	50000000	390625	20611.91455
128	40000000	312500	26722.3464
128	25000000	195312	43097.65306
128	12500000	97656	87074.78587
128	10000000	78125	120754.7778
256	50000000	195312.5	80250.78226
256	40000000	156250	113274.4204
256	25000000	97656	180281.741
256	12500000	48828	355555.4237
256	10000000	39062	441378.6427

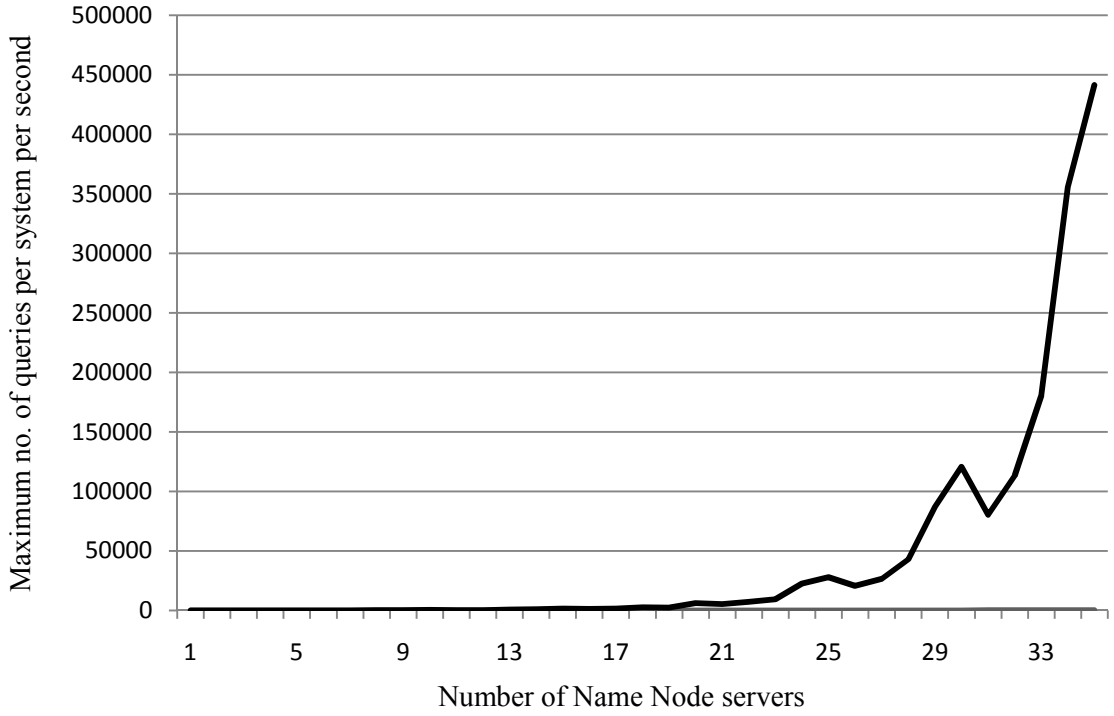

Fig 5.7 Number of name node server vs maximum number of queries per system

The larger the number of queries a system can take, improves the system performance and client user experience. Adding name node servers in chord ring improves the query performance and metadata caching capacity of HDFS.

High Scalability of Proposed Hadoop Architecture

HDFS single namenode server has limited support for metadata objects and data nodes. Single name node saturates at 100 billions of files and 10 thousands. The proposed approach distributes the metadata and data node load on multiple name nodes using distributed hash tables. This approach gives limitless support to the scalability of HDFS. The size of namespace is computed on the bases of number of files, average number of metadata objects per file, total metadata objects and number of bytes per objects as shown in Table 5.8.

Table 5.8 Size of Namespace for 1 billion and 2 billion files

NUMBER OF FILES	NUMBER OF METADATA OBJECTS PER FILE	TOTAL METADATA OBJECTS	METADATA BYTES PER OBJECTS	SIZE OF NAMESPACE IN GB
1000000000	4	4000000000	200	745.06
2000000000	4	8000000000	200	1490.12

The Table 5.8 shows that approximately 746 GB is required to store and cache 1 billions of files. This amount of metadata is distributed on the name node servers. Table 5.9 is depicted to show the size of metadata per namenode server for different number of name node in chord ring.

Table 5.9 Distribution of metadata for different number of namenode servers

Namespace (GB)	Number of name node servers	Size Metadata per namenode server (GB)
745.06	32	23.28
745.06	64	11.64
745.06	128	5.82
1490.12	32	46.57
1490.12	64	23.28
1490.12	128	11.64

Other important factor for scalability is to manage the internal load due to data node. The large number of datanodes per name node saturates the server dues to high number of status update requests. The number of data node is proportional to number of files as shown in Table 5.10.

Table 5.10 Number of Data Nodes and Number of files

NUMBER OF FILES	AVERAGE BLOCKS PER FILE	BLOCK SIZE	NUMBER OF REPLICA	TOTAL STORAGE REQUIRED ON DATA NODES in TB	NUMBER OF DATA NODE WITH 8 TB OF STORAGE
1000000000	3	128	3	1098633	137329
2000000000	3	128	3	2197266	274658

The large amount of storage requires large number of data nodes. According the proposed architecture, the data nodes are also resources and are divided among name nodes severs.

Table 5.11 Load of Data Nodes per Name Node Server

NUMBER OF DATA NODE	NUMBER OF NAME NODE SERVERS	DATA NODE PER SERVER
137329	32	4292
137329	64	2146
137329	128	1073
274658	32	8583
274658	64	4292
274658	128	2146

The data given in tables from 5.8 to 5.11 are computed to manage 1 billion and 2 billion files on 32, 64 and 128 name nodes. The data is well within the limit for memory and internal load of data nodes. The proposed architecture has broken the limit of 100 million files and 10,000 data node per Hadoop cluster.

It is clear from Table 5.8 that the 1 billion files require 745 GB of memory and this memory is made available to HDFS from 32 name node servers by using 23.28 GB memory of each. One billion files require 1098633 TB of storage. To store the same amount of data 137329 data nodes with 8 TB of storage are required. Each name node carries an internal load of 4292 data nodes.

Further, the scalability is improved by adding more name node servers to manage higher size of namespace.

Name Node Failover in Proposed Hadoop Architecture

When a Namenode fails or gracefully leave the chord ring, its load is passed to the next active successor node in the ring. Although, Namenode failures are not frequent still the proposed system has the resilience and adaption to these failures. A relationship of metadata and datanode load on successor node due to two consecutive name node failures is shown in Table 5.12 and 5.13 respectively.

Table 5.12 Metadata load on successor node due to two consecutive name node failures

SIZE OF METADATA PER NAMENODE SERVER	NUMBER CONSECUTIVE NAME NODE FAILURES	METADATA LOAD	SIZE OF SUCCESSOR METADATA
23.28	2	46.56	69.84
11.64	2	23.28	34.92
5.82	2	11.64	17.46

Table 5.13 Data node load on successor node due to two consecutive failures

NUMBER OF DATA NODE	NUMBER OF NAME NODE SERVERS	DATA NODE PER SERVER	CONSICUTIVE FAILOVER NODES	EXTRA NODE PER SERVER	TOTAL NODES AFTER FAILOVER
137329	32	4292	2	8583	12875
137329	64	2146	2	4292	6438
137329	128	1073	2	2146	3219
274658	32	8583	2	17166	25749
274658	64	4292	2	8583	12875
274658	128	2146	2	4292	6438

The metadata is replicated to multiple namenode servers using its finger table. The successor node has the replication copy of metadata of failure name node. It takes the load of failure name node. In case the failure name node joins again. It searches for its successor and gets its metadata and internal load back.

CHAPTER 6

Conclusion and Future Scope

The proposed architecture has resolved the issues of namespace scalability, failover and availability. The focus of the work is based on namespace distribution using distributed hash tables. The system has achieved high scalability as namespace is distributed among namenodes by using distributed hash tables. The centralized namenode has been prone to single point of failure for HDFS. In proposed design, the failover technique has been discussed that guards it from single point of failure. In previous approaches, the growing namespace of HDFS affects the availability and performance of Hadoop cluster. In this research work, the performance and availability of namespace is scaled up by adding namenode server. The results show that the proposed architecture has improved the performance in terms of system initialization time, key lookup operation and the load capacity of HDFS.

Only vertical scalability was possible in previous approaches. Now, the proposed architecture of has self adaptive and healing feature that allows it to scale up the namespace horizontal by adding namenode server. The resource lookup cost is O(log N).

The issue of single point of failure has been addressed. Now, namenode may leave and join without any downtime and much overhead. This has improved the availability of cluster as other name nodes had replication of distributed namespace.

Finally, the provisioning of services on the proposed architecture improves the performance of client applications, gives the scalability in terms of data storage, load capacity of data nodes and overall performance of the HDFS.

Future Work

The proposed architecture has distributed the growing namespace load from single namenode to multiple name nodes. The research work addresses the issues of scalability, namenode failover and availability of the system. The performance of the system can further be improved by strategies the replication of distributed namespace on other namenode servers.

The client applications interacts the namenode servers for metadata lookup for files. The metadata of these files lies on few sets of namenode servers. The intimacy between client application and these namenode can be developed to further improve the performance of the HDFS.

A single large Hadoop cluster deployment is always cheaper and easy to manage than many small Hadoop clusters. So many client application data resides on a single large Hadoop deployment. It requires careful review of the security. Each client applications have different storage requirements. More advanced technique could be designed to address the issue of quality of service and client application data security.

References

[1] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, "The Google File System", Google, 2003.

[2] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Google, 2004.

[3] Konstantin V. Shvachko, "HDFS scalability: the limits to growth", usenix vol 35 no 3, May 2010, www.usenix.org/publications/login/2010-04/openpdfs/shvachko.pdf.

[4] Dhruba Borthakur, "The Hadoop Distributed File System: Architecture and Design", November 2007.

[5] George Porter, "Decoupling Storage and Computation in Hadoop with SuperDataNodes", ACM SIGOPS Operating Systems Review, Volume 44 issue, April 2010.

[6] Derek Tankel, "Scalability of Hadoop Distributed File system," Yahoo developer work, May 2010.

[7] The RPC server Listener thread is a scalability bottleneck, Apache Jira, https://issues.apache.org/jira/browse/HADOOP-6713.

[8] Dhruba Borthapur, "Hadoop AvatarNode High Availability", Facebook, http://hadoopblog.blogspot.com/2010/02/hadoop-namenode-high-availability.html.

[9] Feng Wang,Jie Qiu, Jie Yang, Bo Dong, Xinhui Li, Ying Li, "Hadoop High Availability through Metadata Replication", IBM China Research Laboratory, ACM, November 2009.

[10] Yixue Wang, HaiTao LV, "Efficient Metadata Management in Cloud Computing", IEEE 3rd International Conference on Communication Software and Networks, 2011.

[11] A. Vijay Sriniwas, M. Venkateswara Reddy, D. Janakiram, "Distributed Wisdom: Designing a Replication Service for Large Peer to Peer Data Grids", IEEE Distributed Systems Online Vol. 7, No. 3, March 2006.

[12] Apache Hadoop Project: http://hadoop.apache.org

[13] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, "The Hadoop Distributed File System", Mass Storage Systems and Technologies (MSST), IEEE 26th Symposium, 2010.

[14] Garhan Attebury, Andrew Baranovski, "Hadoop Distributed File System for the Grid" Nuclear Science Symposium Conference Record (NSS/MIC), IEEE 2009

[15] SONG Guang-hua , CHUAI Jun-na, "QDFS: A Quality-Aware Distributed File Storage Service Based on HDFS**"** Computer Science and Automation Engineering (CSAE), IEEE International Conference 2011

[16] Konstantin Shvachko, Hairong Kuang, "The Hadoop Distributed File System" Mass Storage Systems and Technologies (MSST), IEEE 26th Symposium 2010

[17] An Introduction to HDFS Federation , http://hortonworks.com/blog/an-introduction-to-hdfs-federation/

[18] The Next Generation of Apache Hadoop MapReduce, http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen/

[19] Konstantin V. Shvachko, "Apache Hadoop: The Scalability Update" , https://www.usenix.org/publications/login/june-2011-volume-36-number-3/apache-hadoop-scalability-update USENIX, The advanced computing system association June 2011

[20] Flocchini, P. , "Enhancing Peer-to-Peer Systems Through Redundancy" Selected Areas in Communications, IEEE Journal, Volume 25 , Issue: 1 Jan 2007

[21] Haiping Huang, Yan Zheng, PChord: a distributed hash table for P2P network, Frontiers Of Electrical And Electronic Engineering In China Volume 5, Number 1 2010

[22] HDFS Federation, http://hadoop.apache.org/common/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/Federation.html

[23] Tom White, Hadoop: The Definitive Guide

[24] Jason Venner, Pro Hadoop

[25] Quang hieu Vu,Mihai Lupu,Beng Chin Ooi, "Peer to Peer Computing principles and Applications " Springer

List of Paper Publications

[1] Harcharan Jit Singh,V. P. Singh. “High Scalability of HDFS Using Distributed Namespace” International Journal of Computer Applications, 2012
(Communicated)