High Scalability of HDFS using Distributed Namespace

Harcharan Jit Singh V. P. Singh

Department of Computer Science and Engineering,
Thapar University,
Patiala, Punjab, India

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. A 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 availability of Hadoop. This approach is based on distribution of namespace using distributed hash tables. The growing size of namespace of HDFS is distributed 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. 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.

General Terms
Data intensive computing, Scalability, Failover, Availability

Keywords
HDFS, Hadoop, Chord, Namespace, namenode, datanode

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 sets. Such organizations are required to index huge volume of contents and analyze terabytes of data to extract patterns [23, 24]. 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 [2, 18, 23, 24].

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 in 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 [23, 24, 27].

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 [1, 2, 12, 18, 23, 24]. The Hadoop distributed file system (HDFS) 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 has emerged as a data mining platform and is becoming an industry standard for large data processing [23,24]. Hadoop is successfully used in science and a variety of industries. Scientific applications include mathematics, high energy physics, astronomy, genetics, and oceanography. The platform has been in action in many areas [13].

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

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, 26]. It is a distributed lookup protocol. It maps a given key onto a node. Data is easily placed in a 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 is its 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 decentralized environment which is symmetric, auto adaptive and provides consistent performance of resource lookup. Chord ring maintains data structures like finger table, successor list and predecessor list to five consistent resource lookup performances.

The remainder of this paper is organized as follows. Background and related work are described in Section 2. Section 3 states the details of the proposed architecture of Hadoop. Evaluation of the proposed architecture is described in Section 4. Finally, conclusions and future work are described in Section 5.

2. BACKGROUND AND RELATED WORK

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 section, the background of this research field and related studies are reviewed.

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 address the single point failure of the Namenode. The primary AvtarNode is a Namenode and writes its transaction logs into the shared NFS filer. Another instance of AvtarNode is running and called standby node which continually reading the transaction logs from the same shared NFS filer. The standby namenode donot participate in the functioning of HDFS. 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 has phases: the first is the initialization phase which initializes the execution environment of high availability. The second phase replicates metadata from critical node to corresponding backup node at runtime. The last phase resumes the running of Hadoop. As the file system information and Edit Log transactions are stored as a backup copy on the Namenode, the solution emphasizes on the replication of critical metadata. 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 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. The Namenode metadata management was enhanced by creating a pool a RPC reader threads which works well 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 [7].The solution was effective in improving the performance of the Namenode to handle heavy workload, but it fails to provide a solution to the scalability and single point of failure of the Namenode.

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. The design is a storage-rich architecture of Hadoop. As a single SuperDataNode accommodates data worth many DataNodes, its failure has significant impact on the storage. The use of SuperDataNode has no change on the metadata storage. As a result, it does not improve scalability of HDFS the architecture. The use of SuperDataNodes is not a cost effective solution to improve the storage capacity. The single point of access puts load on it bandwidth to access big 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. 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 designed 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.

3. THE PROPOSED ARCHITECTURE

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.

3.1 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, it takes the Namenode out of service frequently. During this time client application waits for the namenode server. A failure of the namenode makes the file system inaccessible and takes time to recover. 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 1.

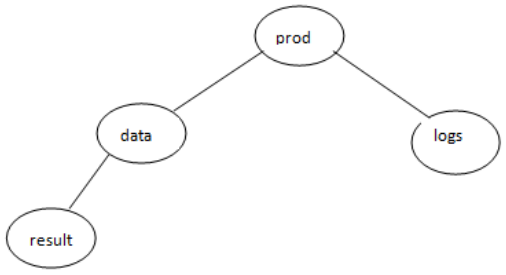

Figure 1: Directory Structure

The distributed structure parameters are given in Table 1.

Table 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 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 is depicted in Table 2.

Table 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 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 3:

Table 3. Distributed hash value Index for namenode

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.

3.2 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 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 2.

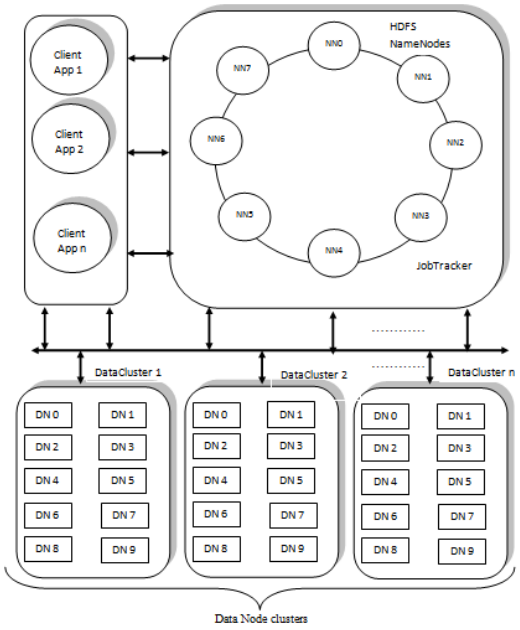

Figure 2: Proposed Architecture of Hadoop

3.2.1 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).

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.3 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 the owner of the resource that generated this key. This namenode is called the successor node of key k, denoted by successor (k).

3.4 Chord for Namespace Management

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.

4. 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 is used to construct the chord ring of name nodes and related data structures.

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 4 to 7. A 64 chord ring is constructed having namenode shown in Figure 3.

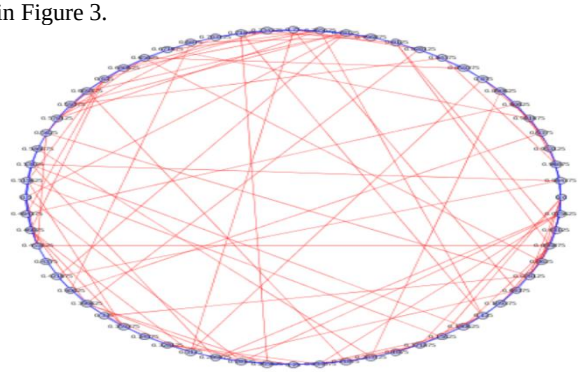

Figure 3: Chord Ring of 64 Name Node Servers

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 4. 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 4.

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 Table 5.

Table 5. Broadcast time on different number of name node servers

Number of name node 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, and 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 6. The smaller lookup time gives higher metadata lookup performance.

Table 6. Random key lookup time in a chord ring of different name node servers

Number of namenode server	No of steps in lookup	Time
8	288	0.015
16	310	0.007
32	355	0.010
64	448	0.012
128	623	0.012
256	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 7. The unicast time depends upon the number of name node server and directly improves the performance of key lookup operation.

Table 7. 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.

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 programs. To test the performance, random key searches are performed on a namespace of single and multiple name node servers. Results are collected and are shown 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. To evaluate the availability of the proposed system, the cache initialization has been implemented.

The time consumed to search 100 random metadata key queries from the cache is directly related to performance of the name node server. The larger search time affects the name node server performance and slows down the performance of

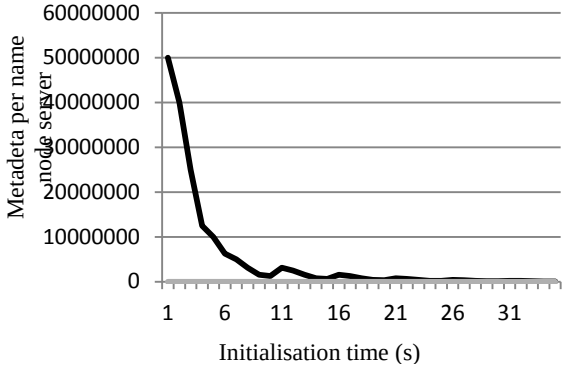

Figure 4: Initialization time vs Metadata objects per name node server

client application. Figure 4 shows that there is an exponential rise in cache initialization time for metadata objects number 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.

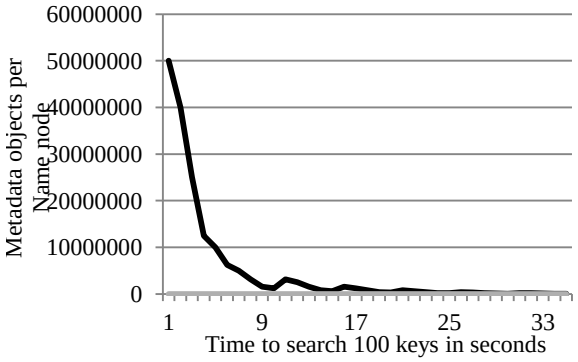

Figure 5: Time to search 100 random keys vs metadata objects per namenode server

From Figure 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 indicate the performance of whole system comprising of n name node server. This parameter indicates the number of client requests processed in a second. 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.

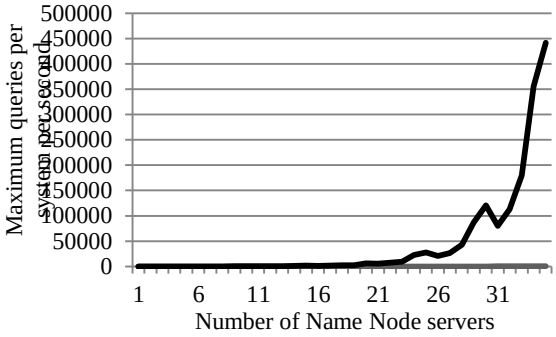

Figure 6: Number of name node server vs maximum number of queries per system

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 basis of number of files, average four metadata objects per file, total metadata objects and 200 bytes per objects as shown in Table 5.8.

Table 8 Size of Namespace for 1 billion and 2 billion files

NUMBER OF FILES	TOTAL METADATA OBJECTS	SIZE OF NAMESPACE IN GB
1000000000	4000000000	745.06
2000000000	8000000000	1490.12

The Table 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 9 is depicted to show the size of metadata per namenode server for different number of name node in chord ring.

Table 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 and is computed on the basis of average 3 blocks per file and block size 128 MB replicated thrice is shown in Table 10. 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 10 Number of Data Nodes and Number of files

NUMBER OF FILES	TOTAL STORAGE REQUIRED ON DATA NODES in TB	NUMBER OF DATA NODE WITH 8 TB OF STORAGE
1000000000	1098633	137329
2000000000	2197266	274658

The data given in tables from 8 to 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 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.

Table 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

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 12 and 13 respectively.

Table 12 Metadata load on successor node due to two consecutive name node failures

SIZE OF METADATA PER NAMENODE SERVER	METADATA LOAD	SIZE OF SUCCESSOR METADATA
23.28	46.56	69.84
11.64	23.28	34.92
5.82	11.64	17.46

Table 13 Data node load on successor node due to two consecutive failures

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

5. CONCLUSIONS

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.

5.1 Future Work

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.

6. REFERENCES

[1] Ghemawat, S., Gobioff, H. and Leung, S.T., 2003, The Google File System, Google.

[2] Dean, J. and Ghemawat, S., 2004, MapReduce: Simplified Data Processing on Large Clusters, Google.

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

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

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

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

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

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

[9] Wang, F., Qiu, J., Yang, J., Dong, B., Li, X. and Li, Y., November 2009, Hadoop High Availability through Metadata Replication, IBM China Research Laboratory, ACM.

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

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

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

[13] Shvachko, K., Kuang, H., Radia, S. and Chansler, R., 2010, The Hadoop Distributed File System, Mass Storage Systems and Technologies (MSST), IEEE 26th Symposium.

[14] Attebury, G. and Baranovski, A., 2009, Hadoop Distributed File System for the Grid, Nuclear Science Symposium Conference Record (NSS/MIC), IEEE.

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

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

[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] Shvachko, K.V., june 2010, 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.

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

[21] Huang, H. and Zheng, Y., 2010, PChord: a distributed hash table for P2P network, Frontiers Of Electrical and Electronic Engineering In China Volume 5, Number 1.

[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] Vu, Q.H., Lupu, M. and Ooi, B.C., Peer to Peer Computing principles and Applications, Springer

[26] Antony Chazapis, Georgios Tsoukalas , 2007, Global-scale peer-to-peer file services with DFS, IEEE 8th Grid Computing Conference

[27] Manghui Tu, Peng Li, I-Ling Yen, Bhavani Thuraisingham, Latifur Khan, JANUARY-MARCH 2010, Secure Data Objects Replication in Data Grid, IEEE Transactions on Dependable and Secure computing, Vol. 7, No. 1