Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Dr. Hadoop: an infinite scalable metadata management for Hadoop—How the baby elephant becomes immortal

Dipayan DEV[‡], Ripon PATGIRI

(*Department of Computer Science and Engineering, NIT Silchar, India*)

E-mail: dev.dipayan16@gmail.com; ripon@cse.nits.ac.in

Received Jan. 12, 2015; Revision accepted June 11, 2015; Crosschecked Dec. 25, 2015

Abstract: In this Exa byte scale era, data increases at an exponential rate. This is in turn generating a massive amount of metadata in the file system. Hadoop is the most widely used framework to deal with big data. Due to this growth of huge amount of metadata, however, the efficiency of Hadoop is questioned numerous times by many researchers. Therefore, it is essential to create an efficient and scalable metadata management for Hadoop. Hash-based mapping and subtree partitioning are suitable in distributed metadata management schemes. Subtree partitioning does not uniformly distribute workload among the metadata servers, and metadata needs to be migrated to keep the load roughly balanced. Hash-based mapping suffers from a constraint on the locality of metadata, though it uniformly distributes the load among NameNodes, which are the metadata servers of Hadoop. In this paper, we present a circular metadata management mechanism named dynamic circular metadata splitting (DCMS). DCMS preserves metadata locality using consistent hashing and locality-preserving hashing, keeps replicated metadata for excellent reliability, and dynamically distributes metadata among the NameNodes to keep load balancing. NameNode is a centralized heart of the Hadoop. Keeping the directory tree of all files, failure of which causes the single point of failure (SPOF). DCMS removes Hadoop's SPOF and provides an efficient and scalable metadata management. The new framework is named 'Dr. Hadoop' after the name of the authors.

Key words: Hadoop, NameNode, Metadata, Locality-preserving hashing, Consistent hashing
http://dx.doi.org/10.1631/FITEE.1500015 **CLC number:** TP311

1 Introduction

Big data, a recent buzz in the Internet world, is growing louder with every passing day. Facebook has almost 21 PB data in 200 million objects (Beaver *et al.*, 2010) whereas Jaguar ORNL has more than 5 PB data. The stored data is growing so rapidly that EB-scale storage systems are likely to be used by 2018–2020. By that time there should be more than one thousand 10 PB deployments. Hadoop has its own file system, the Hadoop Distributed File System (HDFS) (Shvachko *et al.*, 2010; Dev and Patgiri, 2014), which is one of the most widely used large-scale distributed file systems like GFS (Ghemawat *et al.*, 2003) and Ceph (Weil *et al.*, 2006) and which processes the 'big data' efficiently.

HDFS separates file system data access and metadata transactions to achieve better performance and scalability. Application data is stored in various storage servers called DataNodes whereas metadata ie stored in some dedicated server(s) called NameNode(s). The NameNode stores the global namespaces and directory hierarchy of the HDFS and other inode information. Clients access the data from DataNodes through NameNode(s) via network. In HDFS, all the metadata has only one copy stored in the NameNode. When the metadata server designated as NameNode in Hadoop fails, the whole Hadoop cluster goes

[‡] Corresponding author

ORCID: Dipayan DEV, http://orcid.org/0000-0002-0285-9551

offline. This is termed 'single point of failure' (SPOF) of Hadoop (Wiki, 2012).

The NameNode allows data transfers between a large number of clients and DataNodes, itself not being responsible for storage or retrieval of data. Moreover, NameNode keeps all the namespaces in its main memory, which is very likely to run out of memory with the increase in the metadata size. Considering all these factors, centralized architecture of NameNode in the Hadoop framework seems to be a serious bottleneck and looks quite impractical. A practical system needs an efficient and scalable NameNode cluster to provide an infinite scalability to the whole Hadoop framework.

The main problem in designing a NameNode cluster is how to partition the metadata efficiently among the cluster of NameNodes to provide high-performance metadata services (Brandt *et al.*, 2003; Weil *et al.*, 2004; Zhu *et al.*, 2008). A typical metadata server cluster splits metadata among itself and tries to keep a proper load balancing. To achieve this and to preserve better namespace locality, some servers are overloaded and some become a little bit under loaded.

The size of metadata ranges from 0.1 to 1 percent in size as compared to the whole data size stored (Miller *et al.*, 2008). This value seems negligible, but when measured in EB-scale data (Raicu *et al.*, 2011), the metadata becomes huge for storage in the main memory, e.g., 1 to 10 PB for 1 EB data. On the other hand, more than half of the file system accesses are used for metadata (Ousterhout *et al.*, 1985). Therefore, an efficient and high performance file system implies efficient and systemized metadata management (Dev and Patgiri, 2015). A better NameNode cluster management should be designed and implemented to resolve all the serious bottlenecks of a file system. The workload of metadata can be solved by uniformly distributing the metadata to all the NameNodes in the cluster. Moreover, with the ever growing metadata size, an infinite scalability should be achieved. Metadata of each NameNode server can be replicated to other NameNodes to provide better reliability and for excellent failure tolerance.

We examine these issues and propose an efficient solution. As Hadoop is a 'write once, read many' architecture, metadata consistency of the write operation (atomic operations) is beyond the scope of this paper.

We provide a modified model of Hadoop, termed Dr. Hadoop. 'Dr.' is an acronym of the authors (Dipayan, Ripon). Dr. Hadoop uses a novel NameNode server cluster architecture named 'dynamic circular metadata splitting' (DCMS), which removes the SPOF of Hadoop. The cluster is highly scalable and it uses a key-value store mechanism (DeCandia *et al.*, 2007; Escriva *et al.*, 2007; Lim *et al.*, 2007; Biplob *et al.*, 2010) that provides a simple interface: lookup (key) under write and read operations. In DCMS, locality-preserving hashing (LpH) is implemented for excellent namespace locality. Consistent hashing is used in addition to LpH, which provides a uniform distribution of metadata across the NameNode cluster. In DCMS, with the increase in the number of NameNodes in the cluster, the throughput of metadata operations does not decrease, and hence any increase in the metadata scale does not affect the throughput of the file system. To enhance the reliability of the cluster, DCMS also provides replication of each NameNode's metadata to different NameNodes.

We evaluate DCMS and its competing metadata management policies in terms of namespace locality. We compare Dr. Hadoop's performance with that of traditional Hadoop from multiple perspectives like throughput, fault tolerance, and NameNode's load. We demonstrate that DCMS is more productive than traditional state-of-the-art metadata management approaches for the large-scale file system.

2 Background

This section discusses the different types of metadata server of distributed file systems (Haddad, 2000; White *et al.*, 2001; Braam, 2007) and the major techniques used in distributed metadata management in large-scale systems.

2.1 Metadata server cluster scale

2.1.1 Single metadata server

A framework having a single metadata server (MDS) simplifies the overall architecture to a vast extent and enables the server to make data placement and replica management relatively easy. Such a structure, however, faces a serious bottleneck, which engenders the SPOF. Many distributed file systems (DFSs), such as Coda (Satyanarayanan *et al.*, 1990), divide their namespace among multiple storage servers, thereby making all of the metadata operations decentralized. Other DFSs, e.g., GFS (Ghemawat *et al.*, 2003), also use one MDS, with a failover server, which works on the failure of primary MDS. The application data and file system metadata are stored in different places in GFS. The metadata is stored in a dedicated server called master, while data servers called chunkservers are used for storage of application data. Using a single MDS at one time is a serious bottleneck in such an architecture, as the numbers of clients and/or files/directories are increasing day by day.

2.1.2 Multiple metadata servers

A file system's metadata can be dynamically distributed to several MDSs. The expansion of the MDS cluster provides high performance and avoids heavy loads on a particular server within the cluster. Many DFSs are now working to provide a distributed approach to their MDSs instead of a centralized namespace. Ceph (Weil *et al.*, 2006) uses clusters of MDS and has a dynamic subtree partitioning algorithm (Weil *et al.*, 2004) to evenly map the namespace tree to metadata servers. GFS is also moving into a distributed namespace approach (McKusick and Quinlan, 2009). The upcoming GFS will have more than hundreds of MDS with 100 million files per master server. Lustre (Braam, 2007) in its Lustre 2.2 release uses a clustered namespace. The purpose of this is to map a directory over multiple MDSs, where each MDS will contain a disjoint portion of the namespace.

2.2 Metadata organization techniques

A file system having cloud-scale data management (Cao *et al.*, 2011) should have multiple MDSs to dynamically split metadata across them. Subtree partitioning and hash-based mapping are two major techniques used for MDS clusters in large-scale file systems.

2.2.1 Hash-based mapping

Hash-based mapping (Corbett and Feitelson, 1996; Miller and Katz, 1997; Rodeh and Teperman, 2003) uses a hash-function, which is applied on a pathname or filename of a track file's metadata. This helps the clients to locate and discover the right MDS. Clients' requests are distributed evenly among the MDS to reduce the load of a single MDS. Vesta (Corbett and Feitelson, 1996), Rama (Miller and Katz, 1997), and zFs (Rodeh and Teperman, 2003) use hashing of the pathname to retrieve the metadata. The hash-based technique explores a better load balancing concept and removes the hot-spots, e.g., popular directories.

2.2.2 Subtree partitioning

Static subtree partitioning (Nagle *et al.*, 2004) explores a simple approach for distribution of metadata among MDS clusters. With this approach, the directory hierarchy is statically partitioned and each subtree is assigned to a particular MDS. It provides much better locality of reference and improved MDS independence compared to hash-based mapping. The major drawback of this approach is improper load balancing, which causes a system performance bottleneck. To cope with this, the subtree might be migrated in some cases, e.g., PanFS (Nagle *et al.*, 2004).

3 Dynamic circular metadata splitting (DCMS) protocol

The dynamic circular metadata splitting of the Dr. Hadoop framework is a circular arrangement of NameNodes. The internal architecture of a NameNode (metadata server) is shown in Fig. 1 and it is a part of the circular arrangement of DCMS. The hashtable storage manager keeps a hashtable to store the key-value of metadata. It also administers an SHA1 (U.S. Department of Commerce/NIST, 1995) based storage system under two components, i.e., a replication engine and failover management. The replication engine features the redundancy mechanisms and determines the throughput of read or write metadata in the DCMS.

DCMS implements metadata balancing across the servers using consistent hashing and excellent namespace locality using LpH that provides the most efficient metadata cluster management. A better LpH based cluster exploits the minimum number of remote procedure calls (RPCs) for read and write lookups.

All these features and approaches are described in the subsequent portions of this paper.

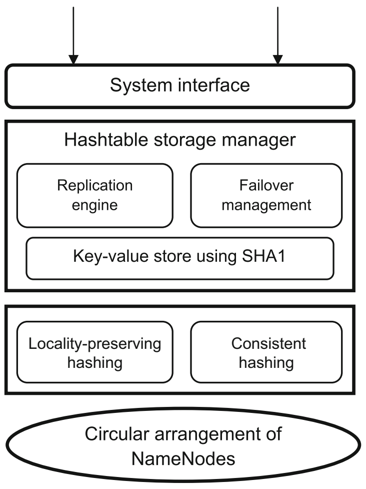

System interface
Hashtable storage manager
Replication engine
Failover management
Key-value store using SHA1
Locality-preserving hashing
Consistent hashing
Circular arrangement of NameNodes

Fig. 1 The skeleton of a NameNode server in DCMS

3.1 DCMS features

DCMS simplifies the design of Hadoop and NameNode cluster architecture and removes various bottlenecks by addressing all difficult problems:

1. Load balance: DCMS uses a consistent hashing approach, spreading keys uniformly all over the NameNodes; this contributes a high degree of typical load balance.

2. Decentralization: DCMS is fully distributed: all the nodes in the cluster have equal importance. This improves robustness and makes DCMS appropriate for loosely organized metadata lookup.

3. Scalability: DCMS provides infinite scalability to the Dr. Hadoop framework. With the increase in the size of metadata, only the nodes (NameNode) in the cluster need to be added. No extra parameter modification is required to achieve this scaling.

4. Two-for-one: The left and right NameNodes of a NameNode keep the metadata of the middle one and behave as the 'hot-standby' (Apache Software Foundation, 2012) to it.

5. Availability: DCMS uses a Resource-Manager to keep track of all the NameNodes which are newly added, as well as the failure ones. During failure, the left and right sides of the failure NameNode of DCMS can serve for the lookup of the failure one. The failure node will be replaced by electing a DataNode as NameNode using a suitable election algorithm. That is why there is no downtime on DCMS. The failure NameNode will serve as the DataNode, if recovered later.

6. Uniform key size: DCMS puts no restriction on the structure of the keys in the hashtable; the DCMS key-space is flat. It uses SHA1 on each file or directory path to generate a unique identifier of k bytes. So, even a huge file/directory pathname does not take more than k bytes of key.

7. Self-healing, self-managing, and self-balancing: In the DCMS, the administrator need not worry about the increase in the number of NameNodes or replacement of NameNodes. The DCMS has the feature of self-healing, self-balancing, and self-managing of metadata and the MetaData server.

3.2 Overview

DCMS is an in-memory hashtable-based system, in which a key-value pair is shown in Fig. 2. It is designed to meet the following five general goals: (1) high scalability of the NameNode cluster, (2) high availability with a suitable replication management, (3) excellent namespace locality, (4) fault tolerance capacity, and (5) dynamic load balancing.

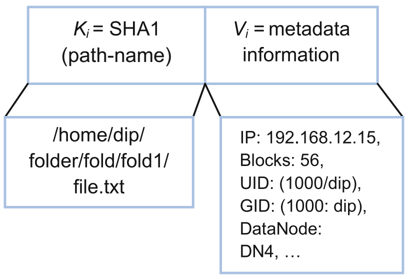

K_i = SHA1 (path-name)
V_i = metadata information
/home/dip/ folder/fold/fold1/ file.txt
IP: 192.168.12.15, Blocks: 56, UID: (1000/dip), GID: (1000: dip), DataNode: DN4, ...

Fig. 2 Key-value system. Key is SHA1 (pathname), while the value is its corresponding metadata

In DCMS design, each NameNode stores one unique hashtable. Like other hash-based mapping, it uses a hashing concept to distribute the metadata across multiple NameNodes in the cluster. It maintains a directory hierarchical structure such that metadata of all the files in a common directory gets stored in the same NameNode. For distribution of metadata, it uses locality-preserving hashing (LpH) which is based on the pathname of each file or directory. Due to the use of LpH in DCMS, it eliminates the overhead of directory hierarchy traversal. To access data, a client hashes the pathname of the file with the same locality-preserving hash function to locate which metadata server contains the metadata of the file, and then contacts the appropriate metadata server. The process is extremely efficient, typically involving a single message to a single NameNode.

DCMS applies the SHA1() hash function on the pathname of each file or directory's full path. SHA1 stands for 'secure hash algorithm', which generates a unique 20-byte identifier for every different input.

In HDFS, all the files are divided into predefined sizes of multiple chunks, called blocks. The SHA1() is applied to each block to generate a unique 20-byte identifier, which is stored in the hashtable as the key. For example, the identifiers of the first and third blocks of file '/home/dip/folder/fold/fold1/file.txt' are SHA1(home/dip/folder/fold/fold1/file.txt, 0) and SHA1(home/dip/folder/fold/fold1/file.txt, 2), respectively.

3.3 System architecture

In DCMS, each NameNode possesses equal priority and hence the cluster shows pure decentralized behavior. The typical system architecture is shown in Fig. 3. The DCMS is a cluster of NameNodes where each of them is organized in a circular fashion. Each NameNode's hostname is denoted by NameNode_X which has neighbors, viz., NameNode_$(X-1)$ and NameNode_$(X+1)$. The hash function is sufficiently random. This is SHA1 in our case. Many keys are inserted, due to the nature of consistent hashing; these keys will be evenly distributed across the various NameNode servers. DCMS improves the scalability of consistent hashing by avoiding the requirement that every node should know about every other node in the cluster. However, in DCMS, each node needs routing information of two nodes, which are its left and right nodes in topological order. This is because each NameNode will put a replica of its hashtable to its two neighbor servers. The replication management portion is discussed in the following section.

The DCMS holds three primary data structures: namenode, fileinfo, and clusterinfo. These data structures are the building blocks of the DCMS metadata. The namenode and clusterinfo are stored on the typical Resource-Manager (JobTracker) of typical Hadoop architecture, and fileinfo is stored on each namenode which handles the mapping to file to its paths in the DataNodes.

1. namenode: The namenode data structure stores the mapping of the NameNode's hostname to NameNode-URL. It is used to obtain the list of all the servers to keep track of all the NameNodes in the DCMS cluster. The Resource-Manager holds this data structure to track all the live NameNodes in the cluster by a heartbeat mechanism. It must update this mapping table information when a new NameNode joins or leaves the DCMS cluster. The Dr. Hadoop framework can be altered by reforming this data structure.

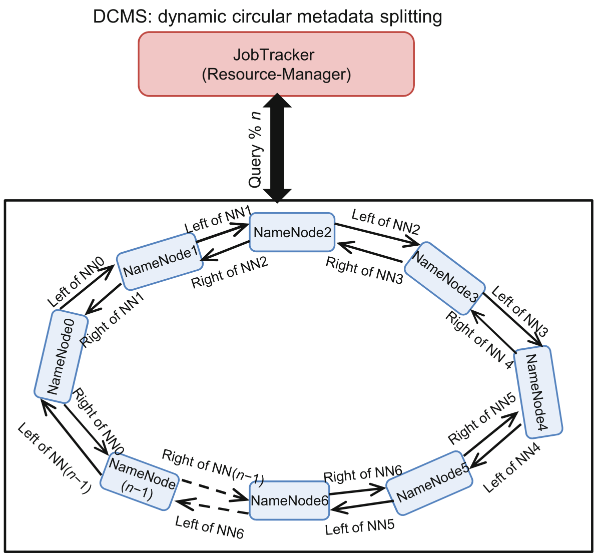


Fig. 3 System architecture. Physical NameNode servers compose a metadata cluster to form a DCMS overlay network

```
/* File Information mapping of: file or directory
   path - nodeURL */
public static Hashtable<FilePath, MetaData>
fileInfo = new Hashtable<FilePath, MetaData>();

/* Cluster information mapping of: nodeURL
   and ClusterInfo object */
public static Hashtable<String, ClusterInfo>
clusterinfo = new Hashtable<String, ClusterInfo>();

/* Data structure for storing all the NameNode servers */
public Hashtable<NameNode_hostname, Namenode-URL>
namenode = new Hashtable<NameNode_hostname,
                         Namenode-URL>();
```

2. fileinfo: The fileinfo data structure, stored in each NameNode of the DCMS cluster, keeps the mapping of the SHA1() of the file or directory path to the corresponding location where they are stored in the DataNodes. The mapping is basically the key-value stored in the hashtable. The mapping locates the DataNodes on the second layer. The mapped fileinfo metadata is used to access DataNodes on the cluster where the files are actually stored.

3. clusterinfo: The clusterinfo data structure, stored in the Resource-Manager, stores the basic information about the cluster such as storage size, available space, and the number of DataNodes.

3.4 Replication management

In Dr. Hadoop we replicate each hashtable of NameNodes to the different NameNodes. The idea of replication management of Dr. Hadoop in DCMS is that each replica starts from a particular state, receives some input (key-value), goes to the same state, and outputs the result. In DCMS, all the replicas possess the identical metadata management principle. The input here means only the metadata write requests, because only file/directory write requests can change the state of a NameNode server. The read request needs only a lookup.

We designate the main hashtable as primary, and the secondary hashtable replicas are placed in the left and right NameNodes of the primary. The primary is given the charge of the replication of all metadata updates and to preserve the consistency with the other two. As shown in Fig. 4, NameNode_$(X+1)$ stores the primary hashtable HT_$(X+1)$ and its secondary replicas are stored in NameNode_X and NameNode_$(X+2)$ as its left and right NameNodes, respectively. These two nodes, hosting the in-memory replicas, will behave as hot-standby to NameNode_$(X+1)$.

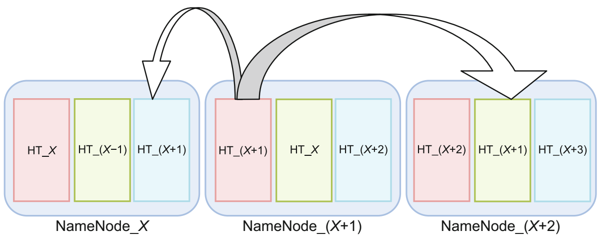


Fig. 4 Replication management of Dr. Hadoop in DCMS

The metadata requests are not concurrent in Dr. Hadoop in order to preserve metadata consistency. When a primary NameNode processes the metadata write requests, it first puts the replicas to its secondary before sending the acknowledgement back to the client. Details about these operations are explained later.

The secondary replica hashtable does not need to process any write metadata request. The clients send only the metadata write requests to the primary hashtable, i.e., HT_$(X+1)$ in the figure. However, the read requests are processed by any replica, either primary or secondary, i.e., HT_X or HT_$(X+2)$ in the figure, to increase the throughput of read.

4 Hashing approaches of DCMS

4.1 Consistent hashing

The consistent hash function assigns each NameNode and key (hashtable) an n-bit identifier using a base hash function such as SHA1 (U.S. Department of Commerce/NIST, 1995). A NameNode's identifier is chosen by hashing its IP address, while a key identifier for the hashtable is produced by hashing the key.

As explained in an earlier section, DCMS consists of a circular arrangement of NameNodes. Assume n bits are needed to specify an identifier. We applied consistent hashing in DCMS as follows: identifiers are ordered in a HashCode module 2^n. Key k is assigned to NameNode_0 if this result is 0, and so on. This node is called the home node of key k, denoted by home(k). If identifiers are denoted by 0 to 2^n-1, then home(k) is the first NameNode of the circle having hostname NameNode_0.

The following results are proven in the papers that introduced consistent hashing (Karger *et al.*, 1997; Lewin, 1998) and that are the foundation of our architecture.

Theorem 1 For any set of N nodes and K keys, with high probability, we have

1. Each node is responsible for at most $(1+\varepsilon)K/N$ keys.

2. When an $(N+1)$th node joins or leaves the network, responsibility for $O(K/N)$ keys changes hands (and only to or from the joining or leaving node).

With SHA1, this ε can be reduced to a negligible value providing a uniform K/N key distribution to each NameNode in the DCMS.

We need to discuss the term 'with high probability' mentioned in the theorem. As the NameNodes and keys in our model are randomly chosen, the probability distribution over the choices of keys is very likely to produce a balanced distribution. However, if an adversary chooses all the keys to hash to an identical identifier (NameNode), the situation would produce a highly unbalanced distribution, leading to destruction of the load balancing property. The consistent hashing paper uses 'k universal hash functions', which guarantees pure load balancing even when someone uses non-random keys.

Instead of using a 'universal hash function', the DCMS uses the standard SHA1 function as our base hash function. This will make the hashing more deterministic, so that the phrase 'high probability' no longer makes sense. However, it should be noted that, collision in SHA1 is seen, but with a negligible probability.

4.1.1 Collision probability of SHA1

Considering random hash values with a uniform distribution, a hash function that produces n bits and a collection of b different blocks, the probability p that one or more collisions will occur for different blocks is given by

$$p \leq \frac{b(b-1)}{2}\frac{1}{2^n}.$$

For a huge storage system that contains 1 PB (10^{15} bytes) of data or an even larger system that contains 1 EB (10^{18} bytes) stored as 8 KB blocks (10^{14} blocks), using the SHA1 hash function, the probability of a collision is less than 10^{-20}. Such a scenario seems sufficiently unlikely that we can ignore it and use the SHA1 hash as a unique identifier for a block.

4.2 Locality-preserving hashing

In EB-scale storage systems, we should implement near-optimal namespace locality by assigning keys that are consistent based on their full pathnames to the same NameNode.

To attain near-optimal locality, the entire directory tree nested under a point has to reside on the same NameNode. The clients distribute the metadata for the files and directories over DCMS by computing a hash of the parent path present in the file operation. Using the parent path implies that the metadata for all the files in a given directory is present at the same NameNode. Algorithms 1 and 2 describe the creation and reading operations of a file by a client over DCMS.

Algorithms 1 and 2 show the write and read operations respectively in Dr. Hadoop on DCMS to preserve the locality-preserving hashing.

In Algorithm 1, the client sends the query as a file path or directory path to Dr. Hadoop and it outputs the data in the DataNodes as well as metadata in the NameNodes of DCMS with proper replication and consistency.

As mentioned in Section 3.4, the hashtable of NameNode_X will be denoted as Hashtable_X.

Algorithm 1 Write operation of Dr. Hadoop in DCMS

```
Require: Write query(filepath) from the client
Ensure: File stored in DataNodes with proper replication and consistency
  for $i \leftarrow 0$ to $m$ do
    hashtable(NameNode_$i$) $\leftarrow$ hashtable_$i$;
  end for
  while true do
    Client issue write(/dir1/dir2/dir3/file.txt);
    // Perform hash of its parent path
    file.getAbsolutePath();
    if client path hashes to NameNode_$X$ then
      update(hashtable_$(X-1)$);
      update(hashtable_$(X+1)$);
    end if
    Reply back to the client from the NameNode_$X$ for acknowledgement;
  end while
```

Algorithm 2 Read operation of Dr. Hadoop in DCMS

```
Require: Read query(filepath) from the client
Ensure: Read data from files stored int the DataNodes
  for $i \leftarrow 0$ to $m$ do
    hashtable(NameNode_$i$)$\leftarrow$ hashtable_$i$;
  end for
  while true do
    Client issue read(/dir1/dir2/dir3/file.txt);
    // hash of its parent path
    file.getAbsolutePath();
    if client path hashes to NameNode_$X$ then
      parallel reading from
      hashtable_$X$
      hashtable_$(X-1)$
      hashtable_$(X+1)$;
    end if
    Perform getBlock()
    Perform getMoreElements()
    The client then communicates directly with the DataNodes to read the contents of the file;
  end while
```

When a client issues a write operation of '/dir1/dir2/dir3/file.txt', it hashes the parent path, i.e., hash of '/dir1/dir2/dir3', so that all the files that belong to that particular directory reside in the same NameNode. If the hash of the above query outputs 3, then the metadata of that file will get stored in the Hashtable_3. For synchronous one-copy replication purposes, that metadata will be replicated to Hashtable_2 and Hashtable_1. After the whole process is completed, Dr. Hadoop will send an acknowledgement to the client.

The read operation of Dr. Hadoop portrayed in Algorithm 2 to preserve LpH follows a similar trend to Algorithm 1. When the client issues a read operation on '/dir1/dir2/dir3/file.txt', it would again make a hash of its parent directory path. If during the write this value was 3, it will now output the same value. So, as per the algorithm, the client will automatically contact again NameNode_3 where the metadata actually resides. Now, as the metadata is stored in three consecutive hashtables, DCMS provides a parallel reading capacity, i.e., from Hashtable_2, Hashtable_3, and Hashtable_4. The application performs getBlock() in parallel each time to obtain the metadata of each block of the file. This hugely increases the throughput of the read operation of Dr. Hadoop.

Fig. 5 shows the step-by-step operation to access files by clients in Dr. Hadoop.

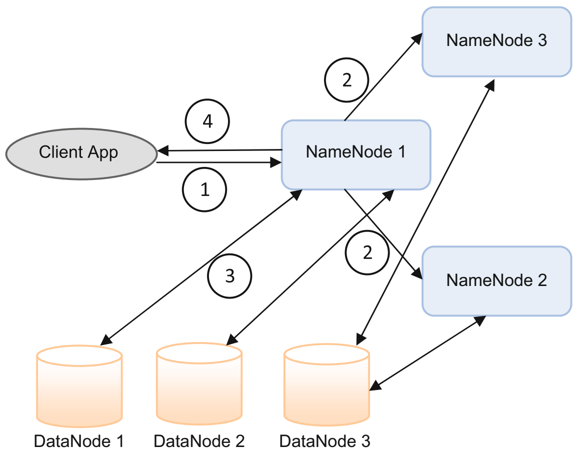


Fig. 5 Client accessing files on the Dr. Hadoop framework

Step 1: Client wishes to create a file named '/dir1/dir2/filename'. It computes a hash of the parent path, '/dir1/dir2', to determine which NameNode has to contact .

Step 2: Before returning the response back to the client, NameNode sends a replication request to its left and right topological NameNodes to perform the same operation.

Step 3: Look up or add a record to its hashtable and the file gets stored/retrieved in/from DataNodes.

Step 4: NameNode sends back the response to the client.

5 Scalability and failure management

5.1 Node join

In a DCMS-like dynamic network, new NameNodes join in the cluster whenever the metadata limit exceeds the combined main memory size (considering the replication factor) of all the NameNodes. While doing so, the main challenge for this is to preserve the locality of every key k in the cluster. To achieve this, DCMS needs to ensure two things:

1. Node's successor and predecessor are properly maintained.

2. home(k) is responsible for every key k of DCMS.

In this section we discuss how to conserve these two factors while adding a new NameNode to the DCMS cluster.

For the join operation, each NameNode maintains a successor and predecessor pointer. A NameNode's successor and predecessor pointer contains the DCMS identifier and IP address of the corresponding node. To preserve the above factors, DCMS must perform the following three operations when a node n, i.e., NameNode_N, joins the network:

1. Initialize the successor and predecessor of NameNode_N.

2. Update these two pointers of NameNode_$(N-1)$ and NameNode_0.

3. Notify the Resource-Manager to update its namenode data structure about the addition of NameNode_N.

5.2 Transferring replicas

The last operation to be executed when a NameNode_N joins DCMS is to shift the reasonability of holding the replicas (hashtable) of its new successor and predecessor. Typically, this involves moving the data associated with the keys to keep the balance of the hashtable in DCMS for Dr. Hadoop. Algorithm 3 and Fig. 6 provide the detailed overview of this operation.

5.3 Failure and replication

The number of NameNodes in DCMS of Dr. Hadoop would not be very large. So, we use heartbeat (Aguilera *et al.*, 1997) to do failure detection. Each NameNode sends a heartbeat to the Resource-Manager every 3 s. The Resource-Manager

Algorithm 3 Node join operation of Dr. Hadoop: addnew_NameNode_N()

```
Require: Random node from the cluster of DataNodes
Ensure: Add the $N$th NameNode of DCMS
  Randomly choose one DataNode from the cluster as the $N$th NameNode of DCMS
  while true do
    update_metadata(NameNode_$N$);
    update_metadata(NameNode_$(N-1)$);
    update_metadata(NameNode_0);
  end while
  while update_metadata(NameNode_$N$) do
    place(hashtable_$(N-1)$);
    place(hashtable_$N$);
    place(hashtable_0);
  end while
  while update_metadata(NameNode_$(N-1)$) do
    hashtable_0.clear();
    place(hashtable_$N$);
  end while
  while update_metadata(NameNode_$X$) do
    hashtable_$(N-1)$.clear();
    place(hashtable_$N$);
  end while
```

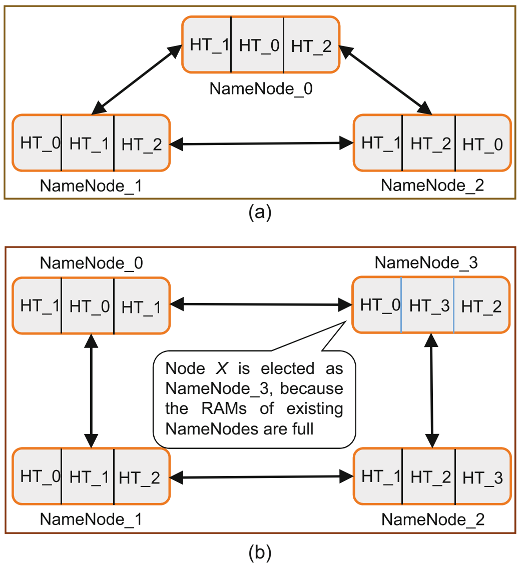


Fig. 6 Dynamic nature of Dr. Hadoop: (a) initial setup of DCMS with half filled RAM; (b) addition of NameNode in DCMS when RAM gets filled up

detects a NameNode failure by checking every 200 s if any node has not sent the heartbeat for at least 600 s. If a NameNode_N is declared a failure, other NameNodes whose successor and predecessor pointers contain NameNode_N must adjust their pointer accordingly. In addition, the failure of the node should not be allowed to distort the metadata operation in progress, as the cluster will be in a re-stabilized situation.

After a node fails, but before the stabilization takes place, a client might send a request to this NameNode for a lookup (key). Generally, these lookups would be processed after a timeout, but from another NameNode (its left or right) which will be acting as a hot standby for it, despite that failure. This case is possible because the hashtable has its replica stored in its left and right nodes.

6 Analysis

6.1 Design analysis

Let the total number of nodes in the cluster be n. In traditional Hadoop, there is only 1 primary NameNode, 1 Resource-Manager, and $(n-2)$ DataNodes to store the actual data. The single NameNode stores all the metadata of the framework. Hence, it becomes a centralized NameNode which cannot tolerate the crash of a single server.

In Dr. Hadoop, we use m NameNodes that distribute the metadata r times. So, Dr. Hadoop can handle the crash of $(r-1)$ NameNode servers. In traditional Hadoop, 1 remote procedure call (RPC) is enough for lookup using a hash of the file/directory path needed to create a file. Using Algorithm 1, however, Dr. Hadoop needs 3 RPCs to create a file and 1 RPC to read a file.

Dr. Hadoop uses m NameNodes to handle the read operation. The throughput of read metadata is thus m times that of the traditional Hadoop. Similarly, the throughput of write metadata operation is m/r times that of traditional Hadoop (m/r is calculated because the metadata is distributed over m NameNodes but with an overheard of replication factor r). The whole analysis and comparison are tabulated in Table 1.

6.2 Failure analysis

In the failure analysis of Dr. Hadoop, the failure of NameNode is assumed to be independent of one another and be equally distributed over time. We ignore the failures of DataNodes in this study because they have the same effect on Hadoop and Dr. Hadoop, which thus simplifies our failure analysis.

Table 1 Analytical comparison of traditional Hadoop and Dr. Hadoop

Parameter	Traditional Hadoop	Dr. Hadoop
Maximum number of NameNode crushes that can survive	0	$r-1$
Number of RPCs needed for read operation	1	1
Number of RPCs needed for write operation	1	r
Metadata storage per NameNode	X	$(X/r)m$
Throughput of metadata read	X	Xm
Throughput of metadata write	X	$X(m/r)$

6.2.1 Traditional Hadoop

Mean time between failure (MTBF) is defined by the mean time in which a system is supposed to fail. So, the probability that a NameNode server fails in a given time is denoted by

$$f = \frac{1}{\text{MTBF}}. \tag{1}$$

Let R_{Hadoop} be the time to recover from the failure. Therefore, the traditional Hadoop framework will be unavailable for fR_{Hadoop} of the time.

If NameNode fails once a month and takes 6 h to recover from the failure,

$$f = 1/(30 \times 24) = 1.38 \times 10^{-3}, \quad R_{\text{Hadoop}} = 6,$$

$$\text{Unavailability} = f \cdot R_{\text{Hadoop}}. \tag{2}$$

Thus, Unavailability $= 1.38\times10^{-3}\times6 = 8.28\times10^{-3}$. So, Availability $= (100 - 0.828) \times 100\%$ =99.172%.

6.2.2 Dr. Hadoop

In Dr. Hadoop, say there are m metadata servers used for DCMS and let r be the replication factor used. Let f denote the probability that a given server fails at a given time t and $R_{\text{Dr. Hadoop}}$ the time to recover from the failure.

Note that approximately we have

$$R_{\text{Dr.Hadoop}} = \frac{rR_{\text{Hadoop}}}{m}. \tag{3}$$

This is because the recovery time of metadata is directly proportional to the amount of metadata stored in its system. As per our assumption, the metadata is replicated r times.

Now, the probability of failure of any r consecutive NameNodes in DCMS can be calculated using the binomial distribution of the probability. Let P be the probability of this happening. We have

$$P = mf^r(1-f)^{m-r}. \tag{4}$$

Let there be 10 NameNodes and let each metadata be replicated three times, i.e., $r = 3$, and let the value of f be 0.1 in three days. Putting these values in the equation results in a P of 0.47%.

In Dr. Hadoop a portion of the system becomes offline if and only if the r consecutive NameNodes of DCMS fail within the recovery time of one another.

This situation occurs with a probability

$$F_{\text{Dr.Hadoop}} = mf\left(\frac{fR_{\text{Dr.Hadoop}}}{t}\right)^r(1-f)^{m-r}. \tag{5}$$

To compare the failure analysis with that of traditional Hadoop, we take the failure probability f much less than in earlier calculation. Let f be 0.1 in three days, m be 10, and for DCMS, $r = 3$.

$R_{\text{Dr. Hadoop}}$ is calculated as

$$R_{\text{Dr. Hadoop}} = \frac{rR_{\text{Hadoop}}}{m} = 3 \times 6/10 = 1.8.$$

So, the recovery time of Dr. Hadoop is 1.8 h, while in the case of Hadoop, it is 6 h. Now,

$$F_{\text{Dr.Hadoop}} = 10 \times 0.1 \times \left(\frac{0.1 \times 1.8}{3 \times 24}\right)^3 (1-0.1)^{10-3}.$$

The above gives

$$F_{\text{Dr. Hadoop}} = 7.46 \times 10^{-9}.$$

The file system of Dr. Hadoop is unavailable for $F_{\text{Dr.Hadoop}}R_{\text{Dr.Hadoop}}$ of the time.
$F_{\text{Dr.Hadoop}}R_{\text{Dr.Hadoop}} = 7.46 \times 10^{-9} \times 1.8 = 1.34 \times 10^{-8}$. Thus, Availability = 99.99%.

So, this shows the increase in the availability of Dr. Hadoop over traditional Hadoop, which is 99.172%. The improvement of recovery time is also shown and proved.

7 Performance evaluation

This section provides the performance evaluation of DCMS of Dr. Hadoop using trace-driven simulation. Locality of namespace is first carefully observed and then we perform the scalability measurement of DCMS. Performance evaluation of DCMS against locality preservation is compared with: (1)

FileHash in which files are randomly distributed based on their pathnames, each of them assigned to an MDS; (2) DirHash in which directories are randomly distributed just like in FileHash. Each NameNode identifier in the experiment is 160 bits in size, obtained from the SHA1 hash function. We use real traces as shown in Table 2. Yahoo means traces of NFS and email by the Yahoo finance group and its data size is 256.8 GB (including access pattern information). Microsoft means traces of Microsoft Windows production (Kavalanekar *et al.*, 2008) of build servers from BuildServer00 to BuildServer07 within 24 h, and its data size is 223.7 GB (access pattern information included). A metadata crawler is applied to the datasets that recursively extract file/directory metadata using the stat() function.

Table 2 Real data traces

Trace	Number of files	Data size (GB)	Metadata extracted (MB)
Yahoo	8 139 723	256.8	596.4
Microsoft	7 725 928	223.7	416.2

The cluster used for all the experiments has 40 DataNodes, 1 Resource-Manager, and 10 NameNodes. Each node is running at 3.10 GHz clock speed with 4 GB of RAM and a gigabit Ethernet NIC. All nodes are configured with a 500 GB hard disk. Ubuntu 14.04 is used as our operating system. Hadoop 2.5.0 version is configured for comparison between Hadoop and Dr. Hadoop keeping the HDFS block size as 512 MB.

For our experiments, we have developed an environment for different events of our simulator. The simulator is used for validation of different design choices and decisions.

To write the code for the simulator, the system (Dr. Hadoop) was studied in depth and different entities and parameters such as replication_factor, start_time, end_time, and split_path are identified. After this, the behaviors of the entities are specified and classes for each entity are defined.

The different classes for simulation are placed on an in-built event-driven Java simulator, SimJava (Fred and McNab, 1998). SimJava is a popular Java toolkit which uses multiple threads for the simulation. As our model is based on an object-oriented system, the multiple threaded simulator was preferred.

In terms of resource consumption, SimJava gives the best efficiency for long running simulations. For experiments with a massive amount of data, it might run out of system memory. This does not occur in our case, because we basically deal with metadata and the simulator delivers its best result when we verify it with a real-time calculation.

The accuracy of the designed simulator is measured based on the real-time calculation, using benchmarking based on the nature of the experiment. For failure analysis, the real-time calculation is done and the result is compared with that found from the simulator. For other experiments, benchmarks are executed and a mismatch ranging from milliseconds to tens of seconds is observed. The overall results show a mismatch of around 1.39% on average between the designed simulator and a real-time device.

To better understand the accuracy of the simulator, in Fig. 8a, a comparison between simulator results and real-time results is shown.

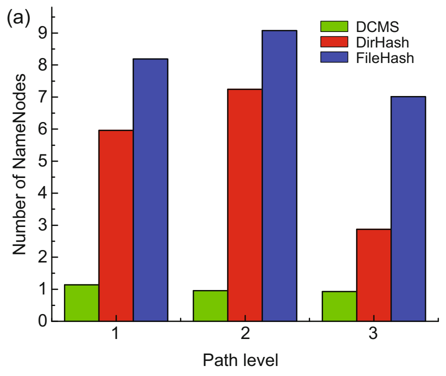
7.1 Experiment on locality-preserving

DCMS uses LpH which is the finest feature of Dr. Hadoop. Excellent namespace locality of Dr. Hadoop's MDS cluster is necessary to reduce the I/O request of a metadata lookup. In this experiment, we have used a parameter, locality $= \sum_{j=1}^{m} p_{ij}$, where p_{ij} (either 0 or 1) represents whether a subtree (directory) path p_i is stored in NameNode j or not. Basically, this metric shows the total number of times a subtree path is split across the NameNodes. Fig. 7 portraits the average namespace locality comparison of paths at three different levels on two given traces using three metadata distribution techniques.

Figs. 7a and 7b show that the performance of DCMS is significantly improved over FileHash and DirHash for the given two traces. This is because DCMS achieves optimal namespace locality using LpH; i.e., keys are assigned based on the order of the pathnames. In contrast, in the cases of FileHash and DirHash, the orders of pathnames are not maintained, so namespace locality is not preserved at all.

7.2 Scalability and storage capacity

The scalability of Dr. Hadoop is analyzed with the two real traces as shown in Table 2. The growth of metadata (namespace) is studied with the



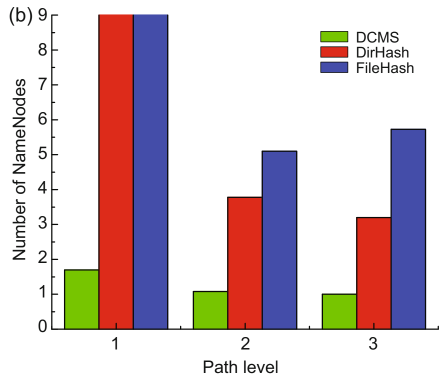


Fig. 7 Locality comparisons of paths at three levels over two traces in the cluster with 10 NameNodes: (a) Yahoo trace; (b) Microsoft Windows trace

increasing data uploaded to the cluster in HDFS. These observations are tabulated in Tables 3, 4, and 5 for Hadoop and Dr. Hadoop with 10 GB data uploaded on each attempt. The metadata size of Table 5 is 3 times its original size for Yahoo and Microsoft respectively because of the replication. Figs. 8a and 8b represent the scalability of Hadoop and Dr. Hadoop in terms of load (MB)/NameNode for both datasets. The graphs show a linear increment in the metadata size of Hadoop and Dr. Hadoop. In traditional Hadoop, with the increase in data size in the DataNodes, the metadata is likely to grow to the upper bound of the main memory of a single NameNode. So, the maximum limit of data size that DataNodes can afford is limited to the size of available memory on the single NameNode server. In Dr. Hadoop, DCMS provides a cluster of NameNodes, which reduces the metadata load rate per NameNode for the cluster. This results in enormous increase in the storage capacity of Dr. Hadoop.

Due to the scalable nature of DCMS in Dr. Hadoop, each NameNode's load is inconsequential in comparison to that of traditional Hadoop. In spite of three times replication of each metadata in DCMS, the load/NameNode of Dr. Hadoop shows a drastic decline compared to Hadoop's NameNode. Fewer loads on the main memory of a node implies that it is less prone to failure.

In Fig. 8a, we have also compared the value obtained by the simulator with that by real-time analysis. As the result suggests, the percentage of mismatch is very little and lines for Hadoop and Hadoop_RT are almost proximate. This validates the accuracy of the simulator.

Table 3 Incremental data storage vs. namespace size for the traditional Hadoop cluster (Yahoo trace)

Data (GB)	namespace (MB)	Data (GB)	namespace (MB)
10	29.80	140	318.33
20	53.13	150	337.22
30	77.92	160	356.71
40	96.18	170	373.18
50	119.02	180	394.22
60	142.11	190	426.13
70	159.17	200	454.76
80	181.67	210	481.01
90	203.09	220	512.16
100	229.37	230	536.92
110	251.04	240	558.23
120	279.30	250	570.12
130	299.82	256.8	594.40

Table 4 Incremental data storage vs. namespace size for the traditional Hadoop cluster (Microsoft trace)

Data (GB)	namespace (MB)	Data (GB)	namespace (MB)
10	20.50	130	254.12
20	41.83	140	271.89
30	59.03	150	297.12
40	77.08	160	312.71
50	103.07	170	329.11
60	128.18	180	352.12
70	141.90	190	369.77
80	157.19	200	384.76
90	174.34	210	393.04
100	190.18	220	401.86
110	214.20	223.7	416.02
120	237.43		

Table 5 Data storage vs. namespace size for the Dr. Hadoop cluster

Data	Namespace (MB)	
(GB)	Yahoo	Microsoft
100	684.11	572.65
200	1364.90	1152.89
256.8	1789.34	1248.02

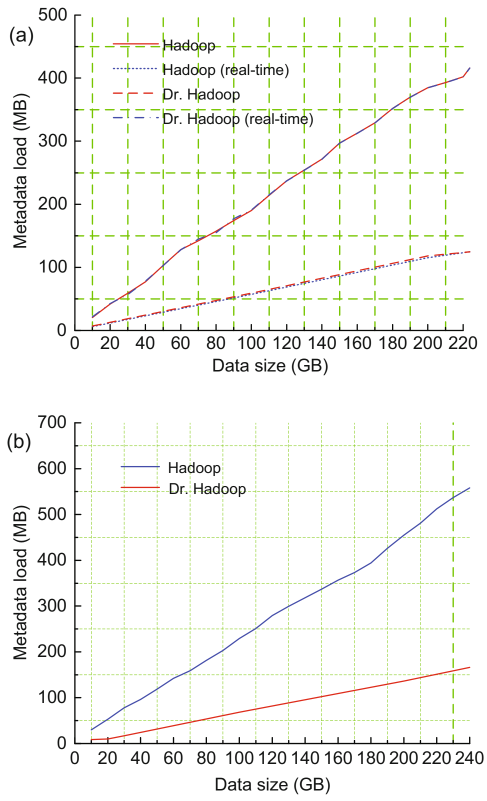


Fig. 8 Linear growth of metadata for Hadoop and Dr. Hadoop load per NameNode: (a) Microsoft Windows trace; (b) Yahoo trace

7.3 Experiment on throughput

In Dr. Hadoop, each NameNode of DCMS behaves as both a primary NameNode and a hot standby node for its left and right NameNodes. During failover, the hot standby takes over the responsibility of metadata lookup. That might affect the throughput of read and write operations. These two metrics are most imperative for our model as they dictate the load of work performed by the clients. We first performed the experiment of throughput for read/write when they are in no failure. Later, the throughput (operation/s) is obtained during the NameNode failure circumstances to evaluate the performance on fault tolerance.

7.3.1 Read/write throughput

This experiment demonstrates the throughput of metadata operations during no failover. A multi-threaded client is configured, which sends metadata operations (read and write) at an appropriate frequency and the corresponding successful operations are measured. We measure the successful completion of metadata operation per second to compute Dr. Hadoop's efficiency and capacity.

Figs. 9a and 9b show the average read and write throughput in terms of successful completion of operations for Hadoop and Dr. Hadoop for both data traces. Dr. Hadoop's DCMS throughput is significantly higher than that of Hadoop. This validates our claim in Table 1. The experiment is conducted using 10 NameNodes; after few seconds in Dr. Hadoop, the speed shows some reduction only because of the extra RPC involved.

7.3.2 Throughput against fault tolerance

This part demonstrates the fault tolerance of Dr. Hadoop using DCMS. A client thread is made to send 100 metadata operations (read and write) per second, and successful operations against these for Hadoop and Dr. Hadoop are displayed in Figs. 10a and 10b.

In Fig. 10a, the experiment is carried out on a Yahoo data trace, where Hadoop shows a steady state throughput initially. We kill the primary NameNode's daemon at t = 100 s and eventually the

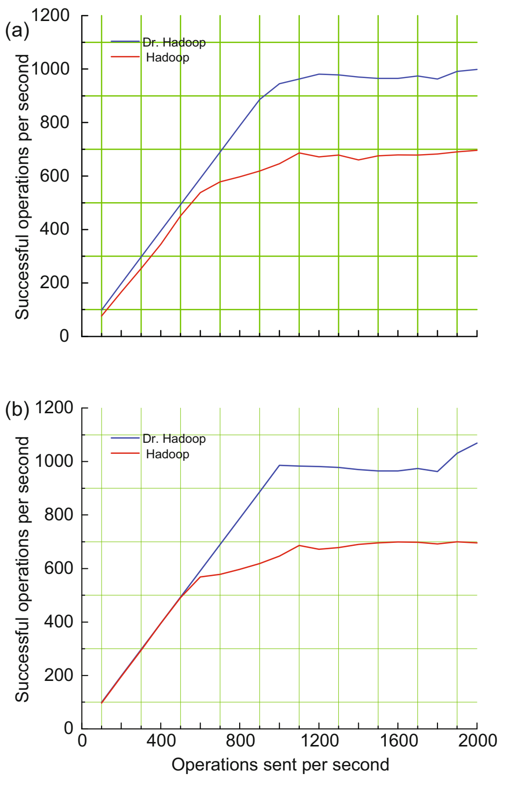


Fig. 9 Comparison of throughput under different load conditions: (a) Yahoo trace; (b) Microsoft Windows trace

whole HDFS goes offline. At around $t = 130$ s, the NameNode is again restarted and it recovers itself from the check-pointed state from the secondary NameNode and repeats the log operations that it failed to perform. During the recovery phase, there are few spikes because the NameNode buffers all the requests until it recovers and batches them all together.

To test the fault tolerance of DCMS in Dr. Hadoop, out of 10 NameNodes, we randomly kill the daemons of a few, viz., NameNode_0 and NameNode_6 at $t = 40$ s. This activity does not lead to any reduction of successful completion of metadata operations because the neighbor NameNodes act as hot standby to the killed ones. Again, at $t = 70$ s, we kill NameNode_3 and NameNode_8, leaving 6 NameNodes out of 10, which reflects an amount of declination in throughput. At $t = 110$ s, we restart the daemons of two NameNodes and this stabilizes the throughput of Dr. Hadoop.

An experiment with the Microsoft data trace is shown in Fig. 10b. It shows a similar trend to the Yahoo trace for Hadoop. During the test for Dr. Hadoop, we kill two sets of three consecutive NameNodes, NameNode_1 to NameNode_3 and NameNode_6 to NameNode_8, at $t = 90$ s to acquire the worst case scenario (with a probability of almost 10^{-8}). This made a portion of the file system unavailable, and throughput of Dr. Hadoop gives a poorest value. At $t = 110$ s we again start NameNode_1 to NameNode_3 to increase its throughput. Again, at $t = 130$ s we start NameNode_6 to NameNode_8, which eventually re-stabilizes the situation.

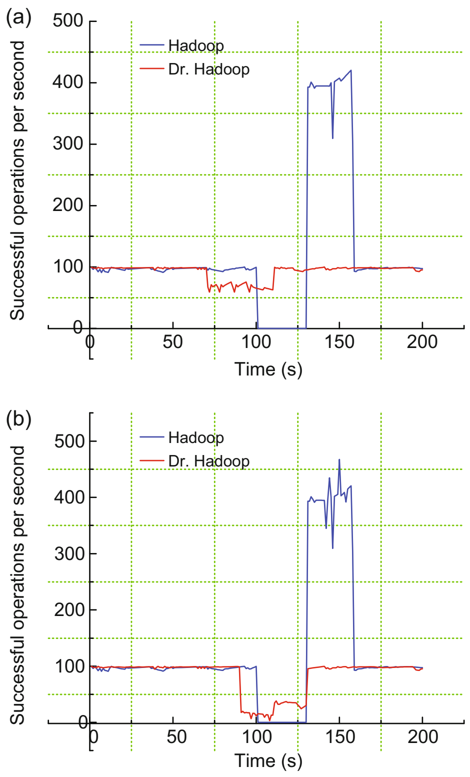


Fig. 10 Fault tolerance of Hadoop and Dr. Hadoop: (a) Yahoo trace; (b) Microsoft Windows trace

7.4 Metadata migration in a dynamic situation

In DCMS, addition and deletion of files or directories occur very frequently. NameNodes join and leave the system, the metadata distribution of Dr. Hadoop changes, and thus DCMS might have to perform migration of metadata to maintain consistent hashing. DCMS needs to justify two situations: the first is how DCMS behaves with the storage load/NameNode when the numbers of servers are constant. The second is how much key distribution or migration overhead is needed to maintain the proper metadata load balancing. The experiment on scalability and storage capacity has answered the first problem sufficiently.

Fig. 11 depicts the metadata migration overhead showing outstanding scalability. We perform the experiment as follows. All the NameNodes in the DCMS system are in an adequate load balancing state at the beginning, as discussed earlier in Section 4.1. In the experiment, two NameNodes are allowed to join the system randomly after a periodic time, and the system reaches a new balanced key distribution state. We examine how many metadata elements are migrated to the newly added servers.

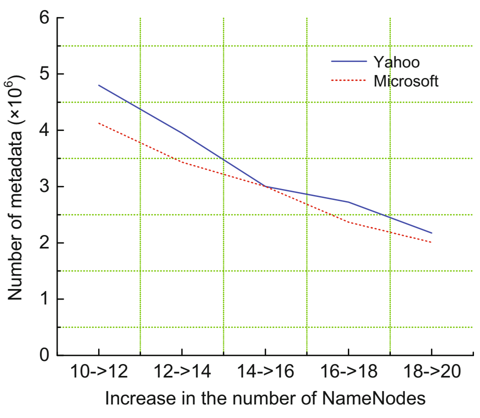


Fig. 11 Migration overhead

7.5 Hadoop vs. Dr. Hadoop with MapReduce job execution (WordCount)

In this experiment, a MapReduce job is executed to show the results for both frameworks. WordCount, a popular benchmark job, is executed on Hadoop and Dr. Hadoop (Fig. 12).

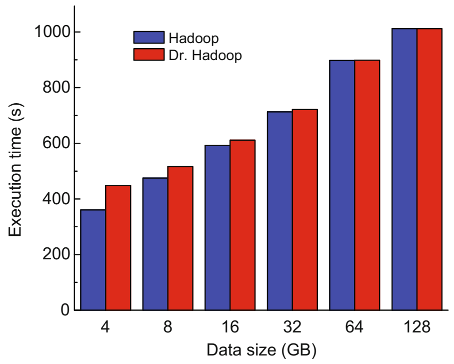


Fig. 12 Wordcount job execution time with different dataset sizes

The job is executed on a Wikipedia dataset, with varying input sizes from 4 to 128 GB. In Hadoop, the single NameNode is responsible for the job execution. In contrast, in Dr. Hadoop, the JobTracker (master for DCMS) submits the job to any particular NameNode after mapping the input filepath. So, an extra step is involved in the case of Dr. Hadoop, due to which a slower execution is shown in comparison to Hadoop.

The overhead of replicating metadata in Dr. Hadoop is also a factor for this little delay. As the locality of metadata is not 100% in Dr. Hadoop, the metadata of subdirectories might be stored in different NameNodes. So, this might be a case of increased RPC, which increases the job execution time initially.

However, as the size of the dataset increases beyond 32 GB, the running time is found almost to neutralize all these factors.

During the execution, the rest of the NameNodes are freely available for other jobs. This is not for the case of Hadoop, which has a single NameNode. So, it is quite practical to neglect the small extra overhead when considering a broader scenario.

8 Related work

Various recent studies for decentralization of the NameNode server are discussed in this section.

Providing Hadoop with a highly available metadata server is an urgent and topical problem in the Hadoop research community.

The Apache Foundation of Hadoop came out with a feature of secondary NameNode (White, 2009), which is purely optional. The secondary NameNode periodically checks NameNode's namespace status and merges the fsimage with edit logs. It decreases the restart time of the NameNode. Unfortunately, it is not a hot backup daemon of NameNode, thus not fully capable of hosting DataNodes in the absence of NameNode. So, it could not resolve the SPOF of Hadoop.

Apache Hadoop's core project team has been working on a backup NameNode that is capable of hosting DataNodes when NameNode fails. The NameNodes can have only one backup NameNode. It continually contacts the primary NameNode for synchronization and hence adds to the complexity of the architecture. According to the project team, they need a serious contribution from different studies for the backup NameNode contribution.

Wang *et al.* (2009) with a team from IBM China worked on NameNode's high availability by replicating the metadata. Their work is somewhat similar to our work, but the solution consists of three stages: initialization, replication, and failover. This increases the complexity of the whole system by having different systems for cluster management and for failure management. In our Dr. Hadoop, similar work is done with less overhead and with the same kind of machines.

A major project of Apache Hadoop (HDFS-976) was for the HA of Hadoop by providing a concept of the Avatar node (HDFS, 2010). The experiment was carried out on a cluster, which consists of over 1200 nodes. The fundamental approach of this concept was to switch between primary and standby NameNodes as if it were switched to an Avatar. For this, the standby Avatar node needs some support from the primary NameNode which creates an extra load and overhead to the primary NameNode because it is already exhausted by the huge client requests. This deteriorates the performance of the Hadoop cluster.

Zookeeper, a subproject of Apache Hadoop, uses replication among a cluster of servers and further provides a leader election algorithm for the coordination mechanism. However, the whole project focuses on coordination of distributed application rather than an HA solution.

ContextWeb performed an experiment on an HA solution of Cloudera Hadoop. DRBD was used from LINBIT and Heartbeat from a Linux-HA project, but the solution was not optimized for performance and availability of Hadoop. For example, it replicated numerous unnecessary data.

E. Sorensen added a hot standby feature to the Apache Derby (Bisciglia, 2009). Unfortunately, the solution does not assist a multi-threading approach to serve replicated messages. Therefore, it cannot be used in the parallel processing of large-scale data of Hadoop.

Okorafor and Patrick (2012) worked on HA of Hadoop, presenting a mathematical model. The model basically features zookeeper leader election. The solution covers only the HA of Resource-Manager, not the NameNode.

A systematic replica management was provided by Berkeley DB (Yadava, 2007), but it features only the database management system. If users want to use Berkeley DB replication for their application other than for a DBMS purpose, they have to spend a lot of time redesigning the replica framework.

IBM DB2 HADR (high availability disaster recovery) (Torodanhan, 2009) can adjust replication management for efficiency at runtime. The solution uses a simulator to estimate the previous performance of a replica. According to the actual scenario of Hadoop, however, this approach is not suitable for tuning the replication process.

9 Conclusions

This paper presents Dr. Hadoop, a modification of the Hadoop framework by providing a dynamic circular metadata splitting (DCMS), a dynamic and excellent scalable distributed metadata management system. DCMS keeps excellent namespace locality by exploiting locality-preserving hashing to distribute the metadata among multiple NameNodes. Attractive features of DCMS include its simplicity, self-healing, correctness, and good performance when a new NameNode joins the system.

When the size of the NameNode cluster changes, DCMS uses consistent hashing that maintains the balance among the servers. DCMS has an additional replication management approach, which makes it most reliable in case of a node failure. Dr. Hadoop continues to work correctly in such a case, albeit with degraded performance when consecutive replicated servers fail. Our theoretical analysis on availability of Dr. Hadoop shows that this framework stays alive for 99.99% of the time.

DCMS offers multiple advantages, such as high scalability, efficient balancing of metadata storage, and no bottleneck with negligible additional overhead. We believe that Dr. Hadoop using DCMS will be a valuable component for large-scale distributed applications when metadata management is the most essential part of the whole system.

Acknowledgements

The authors would like to thank Data Science and Analytic Lab of NIT Silchar for the research.

References

Aguilera, M.K., Chen, W., Toueg, S., 1997. Heartbeat: a timeoutfree failure detector for quiescent reliable communication. Proc. 11th Int. Workshop on Distributed Algorithms, p.126-140. http://dx.doi.org/10.1007/BFb0030680

Apache Software Foundation, 2012. Hot Standby for NameNode. Available from http://issues.apache.org/jira/browse/HDFS-976.

Beaver, D., Kumar, S., Li, H.C., *et al.*, 2010. Finding a needle in haystack: Facebookąŕs photo storage. OSDI, p.47-60.

Biplob, D., Sengupta, S., Li, J., 2010. FlashStore: high throughput persistent key-value store. Proc. VLDB Endowment, p.1414-1425.

Bisciglia, C., 2009. Hadoop HA Configuration. Available from http://www.cloudera.com/blog/2009/07/22/hadoop-haconfiguration/.

Braam, R. Z. PJ, 2007. Lustre: a Scalable, High Performance File System. Cluster File Systems, Inc.

Brandt, S.A., Miller, E.L, Long, D.D.E., *et al.*, 2003. Efficient metadata management in large distributed storage systems. IEEE Symp. on Mass Storage Systems, p.290-298.

Cao, Y., Chen, C., Guo, F., *et al.*, 2011. Es2: a cloud data storage system for supporting both OLTP and OLAP. Proc. IEEE ICDE, p.291-302.

Corbett, P.F., Feitelson, D.G., 1996. The Vesta parallel file system. *ACM Trans. Comput. Syst.*, **14**(3):225-264. http://dx.doi.org/10.1145/233557.233558

DeCandia, G., Hastorun, D., Jampani, M., *et al.*, 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Oper. Syst. Rev.*, **41**(6):205-220.

Dev, D., Patgiri, R., 2014. Performance evaluation of HDFS in big data management. Int. Conf. on High Performance Computing and Applications, p.1-7.

Dev, D., Patgiri, R., 2015. HAR+: archive and metadata distribution! Why not both? ICCCI, in press.

Escriva, R., Wong, B., Sirer, E.G., 2012. HyperDex: a distributed, searchable key-value store. *ACM SIGCOMM Comput. Commun. Rev.*, **42**(4):25-36. http://dx.doi.org/10.1145/2377677.2377681

Fred, H., McNab, R., 1998. SimJava: a discrete event simulation library for Java. *Simul. Ser.*, **30**:51-56.

Ghemawat, S., Gobioff, H., Leung, S.T., 2003. The Google file system. Proc. 19th ACM Symp. on Operating Systems Principles, p.29-43.

Haddad, I.F., 2000. Pvfs: a parallel virtual file system for Linux clusters. *Linux J.*, p.5.

Wiki, 2012. NameNode Failover, on Wiki Apache Hadoop. Available from http://wiki.apache.org/hadoop/NameNodeFailover.

HDFS, 2010. Hadoop AvatarNode High Availability. Available from http://hadoopblog.blogspot.com/2010/02/hadoop-namenode-high-availability.html.

Karger, D., Lehman, E., Leighton, F., *et al.*, 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. Proc. 29th Annual ACM Symp. on Theory of Computing, p.654-663.

Kavalanekar, S., Worthington, B.L., Zhang, Q., *et al.*, 2008. Characterization of storage workload traces from production Windows Servers. Proc. IEEE IISWC, p.119-128.

Lewin, D., 1998. Consistent hashing and random trees: algorithms for caching in distributed networks. Master Thesis, Department of EECS, MIT.

Lim, H., Fan, B., Andersen, D.G., *et al.*, 2011. SILT: a memory-efficient, high-performance key-value store. Proc. 23rd ACM Symp. on Operating Systems Principles.

McKusick, M.K., Quinlan, S., 2009. GFS: evolution on fast-forward. *ACM Queue*, **7**(7):10-20. http://dx.doi.org/10.1145/1594204.1594206

Miller, E.L., Katz, R.H., 1997. Rama: an easy-to-use, high-performance parallel file system. *Parall. Comput.*, **23**(4-5):419-446. http://dx.doi.org/10.1016/S0167-8191(97)00008-2

Miller, E.L., Greenan, K., Leung, A., *et al.*, 2008. Reliable and efficient metadata storage and indexing using nvram. Available from dcslab.hanyang.ac.kr/nvramos08/EthanMiller.pdf.

Nagle, D., Serenyi, D., Matthews, A., 2004. The Panasas activescale storage cluster-delivering scalable high bandwidth storage. Proc. ACM/IEEE SC, p.1-10.

Okorafor, E., Patrick, M.K., 2012. Availability of Jobtracker machine in hadoop/mapreduce zookeeper coordinated clusters. *Adv. Comput.*, **3**(3):19-30. http://dx.doi.org/10.5121/acij.2012.3302

Ousterhout, J.K., Costa, H.D., Harrison, D., *et al.*, 1985. A trace-driven analysis of the Unix 4.2 BSD file system. SOSP, p.15-24.

Raicu, I., Foster, I.T., Beckman, P., 2011. Making a case for distributed file systems at exascale. Proc. 3rd Int. Workshop on Large-Scale System and Application Performance, p.11-18. http://dx.doi.org/10.1145/1996029.1996034

Rodeh, O., Teperman, A., 2003. ZFS—a scalable distributed file system using object disks. IEEE Symp. on Mass Storage Systems, p.207-218.

Satyanarayanan, M., Kistler, J.J., Kumar, P., *et al.*, 1990. Coda: a highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, **39**(4):447-459. http://dx.doi.org/10.1109/12.54838

Shvachko, K., Kuang, H.R., Radia, S., *et al.*, 2010. The Hadoop Distributed File System. IEEE 26th Symp. on Mass Storage Systems and Technologies, p.1-10.

Torodanhan, 2009. Best Practice: DB2 High Availability Disaster Recovery. Available from http://www.ibm.com/developerworks/wikis/display/data/Best+Practice+-+DB2+High+Availability+Disaster+Recovery.

U.S. Department of Commerce/NIST, 1995. FIPS 180-1. Secure Hash Standard. National Technical Information Service, Springfield, VA.

Wang, F., Qiu, J., Yang, J., *et al.*, 2009. Hadoop high availability through metadata replication. Proc. 1st Int. Workshop on Cloud Data Management, p.37-44. http://dx.doi.org/10.1145/1651263.1651271

Weil, S.A., Pollack, K.T., Brandt, S.A., *et al.*, 2004. Dynamic metadata management for petabyte-scale file systems. SC, p.4.

Weil, S.A., Brandt, S.A., Miller, E.L., *et al.*, 2006. CEPH: a scalable, high-performance distributed file system. OSDI, p.307-320.

White, T., 2009. Hadoop: the Definitive Guide. O'Reilly Media, Inc.

White, B.S., Walker, M., Humphrey, M., *et al.*, 2001. Legionfs: a secure and scalable file system supporting cross-domain highperformance applications. Proc. ACM/IEEE Conf. on Supercomputing, p.59.

Yadava, H., 2007. The Berkeley DB Book. Apress.

Zhu, Y., Jiang, H., Wang, J., *et al.*, 2008. Hba: Distributed metadata management for large cluster-based storage systems. *IEEE Trans. Parall. Distrib. Syst.*, **19**(6):750-763. http://dx.doi.org/10.1109/TPDS.2007.70788