Spring 5-1-2014

DNN: A Distributed NameNode Filesystem for Hadoop

Ziling Huang
University of Nebraska-Lincoln, hzlgis@gmail.com

DNN: A DISTRIBUTED NAMENODE FILE SYSTEM FOR HADOOP

by

Ziling Huang

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Hong Jiang

Lincoln, Nebraska

April, 2014

DNN: A DISTRIBUTED NAMENODE FILE SYSTEM FOR HADOOP

Ziling Huang, M.S.

University of Nebraska, 2014

Adviser: Hong Jiang

The Hadoop Distributed File System (HDFS) is the distributed storage infrastructure for the Hadoop big-data analytics ecosystem. A single node, called the NameNode of HDFS stores the metadata of the entire file system and coordinates the file content placement and retrieval actions of the data storage subsystems, called DataNodes. However the single NameNode architecture has long been viewed as the Achilles' heel of the Hadoop Distributed file system, as it not only represents a single point of failure, but also limits the scalability of the storage tier in the system stack. Since Hadoop is now being deployed at increasing scale, this concern has become more prominent. Various solutions have been proposed to address this issue, but the current solutions are primarily focused on improving availability, ignoring or paying less attention to the important issue of scalability. In this thesis, we first present a brief study of the state-of-art solutions for the problem, assessing proposals from both industry and academia. Based on our unique observation of HDFS that most of the metadata operations in Hadoop workload tend to have direct access rather than exploiting locality, we argue that HDFS should have a flat namespace instead of the hierarchical one as used in traditional POSIX-based file system. We propose a novel distributed NameNode architecture based on the flat namespace that improves both the availability and scalability of HDFS, using the well-established hashing namespace partitioning approach that most existing solutions avoid to use because of the loss of hierarchical. We also evaluate the enhanced architecture using a Hadoop cluster, applying both a micro metadata benchmark and the standard Hadoop macro benchmark.

Acknowledgment

I would like to thank my advisor, Dr. Hong Jiang for his invaluable guidance and support over the past few years. He led me into this wonderful research area, taught me how to do research, and how important critical thinking is.

I would like to thank Dr. David Swanson and Dr. Ying Lu for serving as my committee member and reviewing my thesis.

I would like to thank all the members of the ADSL lab for their support and suggestions. I would like to express my special thanks to my friend Lei Xu, for his help, guidance, and friendship over the years.

I thank all my collaborators of the Advanced Development & Architecture team at NetApp Inc. as this work started when I was an summer intern in the team. I would like to thank Joseph Moore, who is the co-inventor of this idea, for his invaluable advices and encouragements. I would like to thank Stan Skelton, our Chief Cat Herder, for his generous support. I would also like to thank Jeff Stilger and Matthew Hannon for their contributions to the implementation and evaluation of this project.

I would like to thank the one who I love. This love goes beyond words, distance, and time.

Finally, I want to thank my parents for their love and support throughout my life. This thesis is dedicated to them.

This research is supported by NSF Grant #CNS-1116606 and NetApp Inc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The exponential growth and ubiquity of both structured and unstructured data have lead us into the Big Data era. The McKinsey Global Institute estimates that data volume is growing at 40% per year, and will grow 44x between 2009 and 2020 [18]. New types of data, such as log data, sensor data have also emerged. According to Gartner there will be nearly 26 billion devices on the Internet of Things by 2020, and each of these devices will generate data in real time. By applying analytics techniques such as machine learning algorithms and statistics analysis on the data, people are able to gain valuable business insights and also predict the future trends.

Big Data are often defined by four key characteristics: *Volume*, *Velocity*, *Variety* and *Value*. While people have been storing, managing and drawing insights from structured data using relational databases for decades, the four characteristics of Big Data have imposed challenges that have never existed before for the traditional data management infrastructure. The traditional relational database (e.g. OracleDB) are good at storing and analyzing structured data, while enterprise NAS (Network Attached Storage) and SAN (Storage Area Network) systems are good at storing unstructured data. However, in the Big Data era, the boundaries between these two types of data infrastructure is blurring, as none of the

two alone is capable of handling the challenges imposed by Big Data. A new type of data management infrastructure that can draw insights from large volumes of data in real time is urgently needed.

Hadoop is such an open-source Big Data infrastructure used for storing, managing and analyzing a large volume of data, it is designed to allow distributed processing and storage of data across thousands of machines. The Hadoop ecosystem consists of several projects. Two of these Hadoop projects, the distributed computational framework MapReduce [12] and the distributed storage layer Hadoop Distributed File System (HDFS), form the very foundation of the Hadoop ecosystem.

The Hadoop Distributed File System [5] is an open-source clone of the Google File System (GFS) [14] that is designed to provide high throughput and fault-tolerant storage on low-cost commodity hardware. In contrast to traditional POSIX distributed file systems (e.g., Lustre, PVFS, AFS, Ceph), HDFS is designed to support write-once-read-many (WORM) type of workloads, with optimizations for streaming access and large data sets (e.g., MapReduce [12]). While standard HDFS employs replication of data blocks to protect against hardware failures, there are also efforts to employ erasure coding techniques to provide fault tolerance [13, 25].

Large scale distributed file systems tend to separate the metadata management from file read/write operations so that complicated metadata accesses will not be in the I/O critical path to block common file I/O operations, which also enables parallel executions of metadata and file I/O operations. HDFS also decouples the metadata management from file I/O by means of two independent functional components: a single NameNode that manages the file system namespace, and multiple DataNodes that store the actual file block data and are responsible for serving read and write requests from Clients. The single NameNode architecture, however , has long been considered as the Achilles' heel of HDFS, as it not only represents a single-point-of-failure, but also is a major limiting factor for the scalability of

the entire HDFS cluster.

Researchers from academia and industry have proposed several solutions to improve the availability and scalability of metadata management in distributed file systems. Filesystem namespace partitioning can be used to break the originally unified namespace into several parts, so that the single metadata access point can be transformed into multiple such points in a metadata cluster, which provides better scalability and opportunity for high availability design. Sage Weil has categorized the namespace partitioning solutions into four types: (1) Static Subtree Partitioning, (2) Hashing, (3) Lazy Hybrid, and (4) Dynamic Subtree Partitioning [8, 28]. He also proposed a novel dynamic metadata management method for Ceph [27], a peta-scale POSIX distributed file system.

The HDFS community has also proposed solutions to address the availability problem. In an early solution, a secondary NameNode is used to recover the metadata when the primary NameNode fails. In order to shrink the window of data unavailability induced by the metadata recovery process, H.A. NameNode [6] is proposed so that the failed primary NameNode can instantly failover to a standby NameNode that is consistent with the primary NameNode in metadata content. On the scalability side, HDFS Federation is proposed so that multiple NameNodes can be combined into a single namespace. However, none of the solutions addresses the availability and scalability problems together, which results in significant deficiencies to be detailed in Chapter 2. Thus, we think that it is high time that a truly unified and scalable namespace be brought to HDFS.

In this thesis, we present the design of a Distributed NameNode, or a NameNode cluster, based on the well-established hashing method to partition and distribute the HDFS namespace and LSM-tree for the metadata storage. We aim to answer two research questions with this approach: *"why can the Hashing-based partitioning method, considered infeasible by conventional wisdom, work for HDFS?"* and *"how do we handle the split brain scenario for HDFS?"*

In this thesis, we make the following contributions:

- *The key observation that HDFS is different from traditional POSIX file systems due to the nature of Hadoop workloads*. Prior MapReduce workload studies [2,10,24] show that among the 4.1 billion metadata operations in OpenCloud, *open* operation itself accounts for 97% of all metadata operations. Among the remaining 3% metadata operations *create* and *list* accounts for about 2.4%. Operations like *mkdirs*, *rename*, *delete* and others account for the rest 0.6% operations. This clearly suggests that HDFS, unlike other POSIX-based distributed file systems, does not need to exploit the hierarchical locality for the purpose of metadata prefetching, and thus, *HDFS does not need to preserve the tree-like index structure, instead, a flat namespace is more appropriate for HDFS*.

- *Proving that the well-established Hashing method, considered unsuitable for traditional POSIX-based distributed file systems by conventional wisdom, is actually a perfect fit as a namespace partitioning method for HDFS.* Our study discovers that, while the hashing-based partitioning method flattens the namespace and thus completely destroys the hierarchical locality heavily relied on by POSIX-based distributed file systems, HDFS does not rely on such locality because of the unique workload characteristics of Hadoop we observed above.

- *A novel design of a Distributed NameNode cluster for HDFS (DNN).* Our DNN has resolved the NameNode single-point-of-failure problem that has troubled the HDFS community for years and unlocked the previous upper bound of maximum number of files a HDFS cluster can support.

- *A working prototype with the evaluation of real MapReduce workloads*. To the best of our knowledge, our HDFS with DNN is the first HDFS that dynamically distributes

metadata across multiple NameNodes and is being successfully integrated into the Hadoop ecosystem.

The rest of the thesis is organized as follows. In Chapter 2, we present the background on HDFS and distributed file system design and related works to motivate our DNN research. In Chapter 3, we present the rationale behind the choice of the hashing method, the design of the NameNode cluster, and the protocols governing the communication and cooperation among different HDFS components. We also present how the distributed NameNode will manage the DataNodes in HDFS, and provide an overview of the load-balancing and failure-recovery use cases. In Chapter 4, we evaluate the performance of an HDFS cluster, extended with our distributed NameNode, using both standard Hadoop macro benchmarks and metadata-intensive micro benchmarks. Chapter 5 and Chapter 6 summarize our work and discuss the future directions.

Chapter 2

Background and Related Work

2.1 Introduction to The Hadoop Ecosystem

Hadoop is an open-source ecosystem used for storing, managing and analyzing a large volume of data, designed to allow distributed processing of data sets across thousands of machines. The Hadoop idea originated from Google, as an effort to store and process the large-scale web index on thousands of x86 commodity servers, and has seen been adopted by other web giants such as Yahoo!, Facebook, Linkedin, Twitter, etc,. Hadoop offered a cost-efficient solution to storing, managing and more importantly, analyzing the vast volumes of data. For example, Facebook maintains a 30-petabyte Hadoop cluster that is used for advertising analytics, which is the core business for the company.

Hadoop ecosystem consists of several sub projects, and two of them form the very basic of Hadoop ecosystem, which are the distributed computation framework MapReduce [12] and the distributed storage layer Hadoop Distributed File System (HDFS).

2.1.1 Introduction to Hadoop Distributed File System

Three important goals are vital in designing a large-scale storage system for Big Data: 1) reliablity, 2) performance and 3) total cost of ownership (TCO). The Hadoop Distributed File System (HDFS) [5] is such a system that is designed to provide high sequential access throughput and fault tolerant storage on low-cost commodity hardware. In contrast to traditional POSIX distributed file systems (e.g., Lustre, PVFS, AFS, Ceph), HDFS is designed to support write-once-read-many (WORM) type of workloads with streaming access and large data sets (e.g., MapReduce [12]). Standard HDFS employs replication of data blocks to protect against hardware failures, and there are also efforts to employ erasure-coding techniques to provide fault tolerance [13, 25].

HDFS has a master/slave architecture and consisting of three key types pf components: *NameNode*, *DataNodes* and *Clients*, as shown in Figure 2.1.

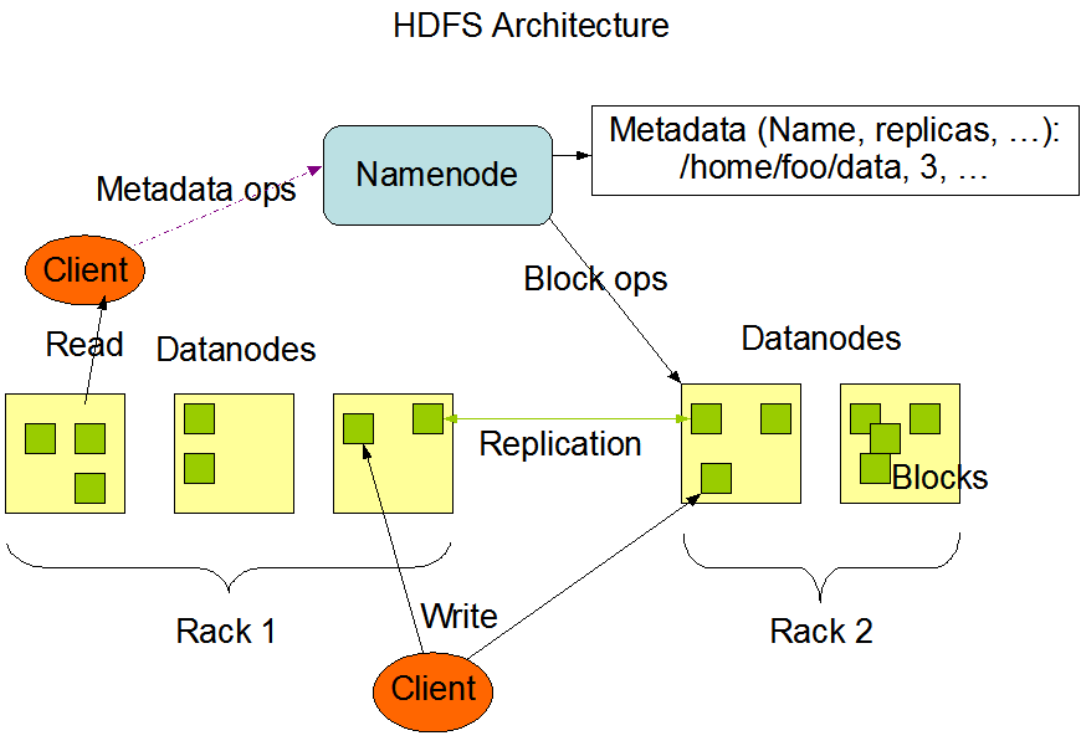

Figure 2.1: Architecture of the standard Hadoop Distributed File System.

A single *NameNode* manages the file system namespace and is in charge of data block placement and DataNode management. In HDFS, a file is split into blocks of data with a default size of 64MB and stored across the DataNodes. The NameNode will determine where to put each block according to a pluggable block placement policy. The standard HDFS NameNode stores the file system namespace in DRAM for fast access. The NameNode will also keep a persistent copy of the file system namespace called *FsImage* on disk for recovery. The changes to the file system namespace are recorded as *EditLog* and periodically merged with the *FsImage* file so that the persistent copy is always up-to-date.

The *DataNodes* store the actual file block data and are responsible for serving read and write requests from Clients. In standard HDFS, each machine that runs the DataNode is an inexpensive commodity x86 server with direct attached storage (DAS). HDFS employs replication for fault tolerance and protect the data blocks against node failures, each block will be replicated on three different DataNodes by default. DataNodes store the HDFS blocks as files in its local file system.

The HDFS *Clients* are used by higher level Hadoop applications to issue requests to HDFS. In order to take advantage of data locality, the standard HDFS co-locates compute with storage, so the HDFS client instance is also running in the node that DataNode is running.

2.1.2 Introduction to The MapReduce framework

MapReduce is proposed by Jeff Dean and Sanjay Ghemawat from Google as a parallel programming model for processing and generating large data sets [12]. MapReduce takes care of the parallelization and scheduling of tasks and can automatically handle failures. To use MapReduce, a user specifies a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values

associated with the same intermediate key. The MapReduce framework is both a parallel computation framework and a scheduling framework. The Apache MapReduce framework consists of two components: a single *JobTracker* and several *TaskTracker*. In a Hadoop cluster, the *JobTracker* process runs in co-location with the HDFS NameNode and coordinates the job scheduling by assigning Map and Reduce tasks to the HDFS DataNodes. The *TaskTracker* runs in co-location with the HDFS DataNodes and accepts tasks - Map, Reduce and Shuffle operations from the *JobTracker*.

2.2 Problems with the HDFS NameNode

The single NameNode architecture does however represent a single point of failure, and it is a major limiting factor for the scalability of an HDFS cluster. The original GFS chose this design because it simplifies the system design and enables the master to make placement and replication decisions using global knowledge [14, 19].

2.2.1 Availability

High availability is a major driving requirement behind large-scale distributed system design. Mission critical storage infrastructure used in scenarios such as telecom aims for perfect availability. In traditional HDFS, among all the components, the single NameNode is the most vulnerable part of the whole system. One or more DataNode failures may have a small performance impact on the user, however, the failure or crash of the NameNode machine will bring down the entire HDFS cluster as all traffic needs to go through the NameNode. Planned maintenance events such as software or hardware upgrades on the NameNode machine will also result in windows of cluster downtime. As an HDFS cluster may grow to as large as several thousands of machines, in either case , a highly available NameNode is needed.

To enhance availability, when the primary NameNode fails, HDFS will recover from the file system image stored on a secondary NameNode. This introduces a short service-unavailability window during the recovery process; however this window is generally considered tolerable as the NameNode rarely fails. Several solutions have been proposed to solve this availability problem by providing an automatic failover to a standby NameNode.

2.2.2 Scalability bottleneck

The single NameNode also limits the scalability of the file system. The NameNode stores the metadata in DRAM for fast access; however, as the size of DRAM is generally limited, this puts an upper bound on the number of files/blocks that HDFS can support. In the early days, this problem was not very serious: HDFS assumes a high data-to-metadata ratio. The deployments typically have large files and utilize a relatively large block size(64MB by default), so the total number of blocks and the DRAM required are tolerable. As a rule of thumb 1GB DRAM can support one million HDFS blocks. So, in a typical NameNode with 64GB DRAM, the DRAM of the NameNode will not likely become a bottleneck for the scalability of HDFS. However, the paradigm for Hadoop workloads (e.g., HBase, Spark [9, 30]) has shifted from the original sequential streaming batch-oriented pattern to a more small-files-based and interactive pattern, and the application model has extended beyond the traditional MapReduce. That is, the new workloads generate more small files and are more real-time oriented, which further places a huge burden on the single NameNode [15]. Thus, it is desirable to improve the scalability of NameNode by distributing the metadata management into multiple NameNodes.

The current HDFS community has also proposed several methods to solve this problem. In earlier versions, HDFS uses a secondary NameNode for recovery. In current versions, the solutions can be generally grouped into two categories:

- **High Availability Configuration for NameNode:** HDFS NameNode with high availability (H.A.) configuration uses two NameNodes, one as an active NameNode and the other as a standby NameNode. In an H.A. configuration, the two NameNodes are kept in a synchronized state all the time. When the active NameNode fails, it will fast failover to the standby NameNode [7] so that the standby NameNode can continue to service metadata requests. However this solutions fails to solve the scalability problem because redundancy does not adequately address the limit on scalability.

- **HDFS Federation:** combines multiple independent NameNodes together where each NameNode represents an independent namespace. But as a common drawback of the Static-Subtree-Partitioning approach, this method is not vertically scalable and, over time, different namespaces will become unbalanced. Furthermore, although the DataNodes will register with every NameNode, each NameNode is still a single point of failure.

2.3 Metadata Management in File System

A files ystem provides a file-level abstraction for the data stored inside the underlying storage media. User can store and retrieve data by the file name. Files can be organized under Directory, which together form the namespace of a file system. A file system typically organizes its namespace with a tree-structured hierarchical organization. A distributed file system is a file system that allows access to files from multiple hosts across a network.

2.3.1 Metadata Storage in File System

A traditional local file system such as Ext3, btrfs stores its namespace in a B-tree [1] like index structure on disk. Unfortunately, metadata workloads are dominated by random access, making it inefficient for traditional disk-based index [22]. Instead of storing the file system metadata in B-tree, TableFS [23] has been proposed to store the file system metadata in a Log-structured-merge-tree(LSM) [20] index structure. LSM tree is used in modern key-value stores such as BigTable and its open source clone HBase [4, 9] as it can provide fast random updates, inserts and deletes without sacrificing lookup performance. By storing metadata in LevelDB, a LSM tree based embedded database, TableFS outperforms other disk-based local file system by a minimum of 50% and as much as 1000%.

2.3.2 Namespace Partitioning Techniques

In order to improve the scalability, distributed file systems partition the single namespace across multiple servers. Namespace partitioning has been a research topic for a long time, and several methods have been proposed to solve this problem in academia. These can be generally categorized into four types: (1) Static Subtree Partitioning, (2) Hashing, (3) Lazy Hybrid, and (4) Dynamic Subtree Partitioning [8, 28].

The static subtree partitioning scheme statically partitions the file system directory hierarchy into different file servers. This approach has been taken by NFS, AFS, Coda, Sprite, etc. [11, 16, 26, 29]. Static subtree partitioning requires the system administrator to decide how the file system should be distributed and manually assign subtrees of the hierarchy to individual file servers in advance. This approach is simple and scalable horizontally: however, it cannot scale vertically, which means a portion of the hierarchy may become overloaded.

In contrast to the static subtree partitioning scheme, Hashing provides good load bal-

ancing across metadata servers. By hashing a unique file identifier, such as inode number or path name, the file system namespace can be evenly distributed to multiple metadata servers. However, the random distribution of file system namespace totally destroys all the hierarchical locality provided by the tree-like index structure, which is vital for POSIX-based distributed file systems as the POSIX semantics heavily relies on this locality for path traversal.

Lazy Hybrid (LH) is based on the Hashing scheme while trying to avoid the problem associated with path traversal by merging the net effect of the permission check into each file metadata record [8]. However, similar to the original Hashing approach, the loss of locality prevents it from adapted by traditional POSIX-based distributed file system [28].

Sage Weil proposed the dynamic subtree partitioning scheme [28] for the Ceph distributed file system. This approach is still based on the subtree-partitioning scheme, but instead of statically and manually partitioning the namespace, this approach is dynamic and automatic, capable of adapting to the changing workload. This approach effectively solves the deficiencies of the static subtree partitioing approach. However, it is highly complicated in implementation [27].

Chapter 3

Design And Implementation

In this chapter, we present the design and implementation of a Distributed NameNode (DNN) architecture for the Hadoop Distributed File System.

Figure 3.1 shows the DNN architecture in the Hadoop ecosystem, where the Hadoop application tier (e.g., MapReduce, HBase) sits on top of the HDFS storage tier. Inside the HDFS tier, the DNN clients serve as a bridge between the Hadoop application tier and the backend HDFS storage service. The Distributed NameNode replaces the NameNode component in standard HDFS, serving as a *scalable and high-availability metadata service* and also manages the HDFS DataNode. The HDFS DataNode services the I/O requests from DNN clients. Similar to the workflow in the standard HDFS, DNN clients will first issue metadata requests to the metadata service provided by DNN that in turn provides the location information of the requested file blocks to the clients, and then issue I/O requests to the corresponding DataNode.

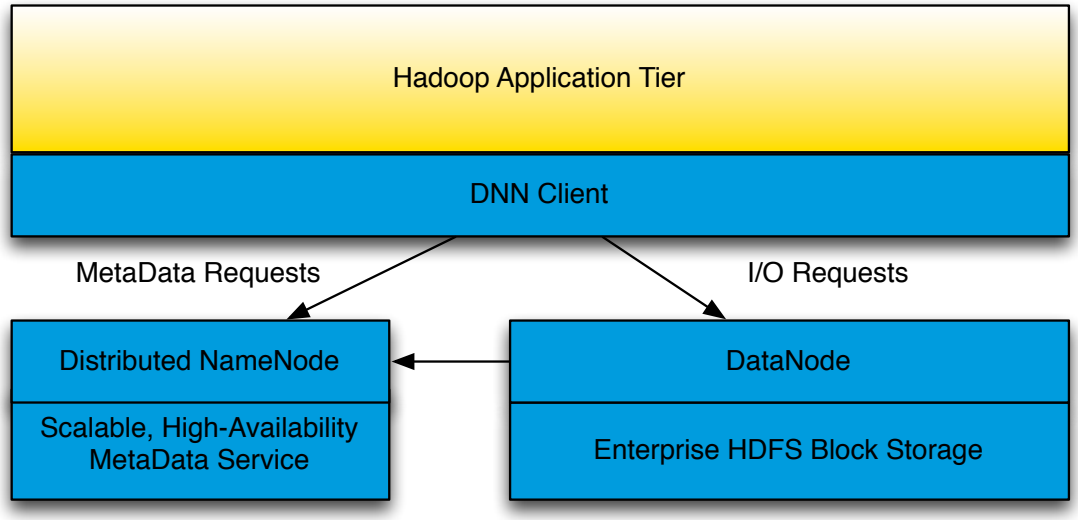

Figure 3.1: Overview of the Distributed NameNode Architecture in Hadoop Ecosystem. The box in yellow represents the Hadoop Application Tier (e.g. MapReduce, HBase). The boxes in blue represent the Hadoop Storage Stack: HDFS.

3.1 DNN Design Principles

In this section, we describe the design principles and rationales behind our distributed NameNode architecture.

3.1.1 HDFS Namespace: Hierarchical or Flat?

Traditional file systems exploit hierarchical locality by storing directly related information together whenever possible, i.e., in the same directory or subdirectory organized in a hierarchical tree structure, and prefetching potentially related information to more efficiently satisfy metadata operations. The hierarchical tree is linked together by the inodes, and adjacent inode structures are stored adjacent to their associated directory entries on disk. This

makes it efficient for metadata operations such as *readdir*, so that all inodes in this directory entry can be efficiently prefetched. The separation of filename and inode also has significant advantage for metadata operation that will change the file system namespace, such as *rename*. So for traditional POSIX-based file systems, which rely significantly on the hierarchical locality, sub-tree based partitioning approaches that also preserves the tree-like hierarchical locality are the best choice.

However, we argue that instead of using the traditional B-tree like hierarchical namespace, a flat namespace is more appropriate for HDFS and the Hadoop-type workloads. This argument is based on our analysis of the previous Hadoop workload studies, which reveals that nearly 97% of the metadata operations are *open* alone, while *create* and *list* account for 2% [10, 24]. What is more surprising is the fact that namespace-changing operations like *rename* only account for less than 1% of all metadata operations. Metadata operations like *Open* is extensively used by Hadoop applications to obtain the metadata of the file because, as in HDFS, the metadata of a file not only contains permissions and size, it also contain the locations of the HDFS blocks that belong to this file. The dominance of non-namespace changing metadata operations like *open* clearly suggests to us that Hadoop-like workloads do not have strong requirement for heavily rely on namespace hierarchical locality, but tend to be exhibit a *flattened namespace* with direct metadata access.

3.1.2 Rationales for Choosing The Hashing Method

The flat namespace has a huge advantage as it makes the hashing method a potentially extremely efficient design choice. We will review the advantages and disadvantages of the hashing method and explain why we choose this method for partitioning the HDFS namespace.

The well-established hashing method has several unique advantages for the HDFS namespace partitioning:

- The hashing method evenly distributes the file system namespace across metadata servers, so that no single directory will become a hotspot. This is potentially very useful for HDFS because MapReduce applications and users tend to put huge numbers of data files under one single data directory.
- The metadata requests are also distributed among metadata servers, which means the metadata server is scalable in terms of metadata performance. To handle more intensive metadata workloads, the system administrator only needs to add more NameNodes into the NameNode cluster.
- The hashing method unlocks the limitation on the number of files/blocks a file system can support. This is also extremely important for peta-scale distributed file systems such as HDFS.

On the downside, hashing methods also have some severe disadvantages, which prevents traditional POSIX-based file systems from adopting it:

- Due to its randomness nature, hashing eliminates all hierarchical locality, which is very harmful to POSIX file systems as they rely heavily on exploiting the locality in the file system hierarchical tree. For example, in a POSIX file system, a *readdir* operation is always followed by a *stat* operation to get the metadata for the file under the directory. Since in a B-tree like hierarchical structure, the metadata of the files are stored adjacent to the directory, POSIX-based file systems utilize this locality to prefetch the metadata of the files. However, hashing completely destroys this locality as it randomly distributes the metadata across multiple servers, making it likely for the metadata for the file and its parent directory to be located on two different servers and thus rendering metadata prefetching difficult if not impossible.
- The hashing method cannot prevent individual files from becoming a hotspot.

These disadvantages of hashing method, however, do not have significant negative impact on a flattened namespace such as HDFS. Therefore, we argue that hashing method

is a perfect fit for HDFS as HDFS's flattened namespace do not have a strict requirement on the hierarchical locality. As a result, we believe that, for HDFS' flattened namespace, the advantages of the hashing method far outweigh its disadvantages. In fact, we can take advantage of the hashing approach to solve the more severe problem, that of the scalability of the NameNode. As for the problem of individual hotspot files, we believe that it can be easily addressed by client-side caching.

3.1.3 Design Goals and Principles for a High Availability Metadata Service

Achieving high availability is one of the most important design goals of our Distributed NameNode (DNN) cluster as mission critical Hadoop cluster cannot tolerate long period of metadata service unavailability.

In HDFS, the root cause of NameNode unavailability can generally be grouped into two categories:

- Unavailability caused by the failure of the physical Storage Node that the HDFS NameNode process resides in.
- Unavailability caused by the planned maintenance or upgrade of the HDFS NameNode.

A recent study by Facebook suggests that planned software upgrade is the primary reason for cluster downtime [7], which will bring down the cluster but will not cause metadata loss. While a NameNode failure is rare, its unpredictable nature and potential consequences of critical metadata loss still make it one of the most important design issues for HDFS. A high availability metadata service should be able to tolerate both kinds of the unavailability.

There are three goals for our H.A. DNN design:

- The integrity of the critical file system metadata should be resilient from the NameNode failure.
- The Distributed NameNodes cluster should always be able to serve metadata requests.
- The degradation of DNN performance should be minimized during the failure recovery to provide a graceful and smooth transition from the clients' point of view.

To achieve these three design goals, our core design principle for the H.A. DNN cluster is to separate the *compute* critical path from the *metadata storage* critical path. In contrast to the NameNode in the standard HDFS, which stores the metadata directly inside the physical node where the NameNode process runs in and maintain a copy of the metadata in a remote physical node for recovery purposes, the NameNode in DNN cluster only services the metadata requests, but stores the metadata in a remote H.A. enterprise shared storage. By delegating the protection of metadata to the H.A. shared storage system, we ensure the safety of the critical metadata and achieved the first design goal.

To achieve the second and third design goals, we delegate the role of the failed NameNode evenly to other remaining healthy NameNodes in our DNN cluster. If one of the NameNodes has failed or needs to be shutdown for upgrade or maintenance, its responsibility of serving the metadata requests is logically and evenly shifted to all the other surviving NameNodes. No metadata transfer is involved in this process as the metadata is stored on the remote backend shared storage system. In theory, a DNN cluster with *N* NameNodes can tolerate the concurrent failures of *N - 1* NameNodes.

3.2 DNN Implementation

In this section, we describe the implementation details of DNN. Our implementation extends the standard HDFS architecture to support multiple active metadata servers (NameNodes). Our file system stack is implemented in C++ for performance consideration, as the Java garbage collection is a known performance overhead. A hashing scheme is used to assign responsibility for the partitioned subsets of the file system namespace to the individual NameNode servers. Clients and DataNode components of the cluster transparently leverage the partitioning mechanism to interact with the distributed metadata service. A separate protocol infrastructure is used to coordinate failover, load-balancing, etc. among the NameNode servers.

The Hadoop architecture utilizes a pluggable file system Java abstraction (interface) as the basis for interacting with the storage tier of a cluster. Though HDFS is the most widely known and used storage tier for Hadoop, it also supports other existing storage system such as qfs, Amazon S3, etc. [3, 21]. HDFS implements this interface by storing files in large blocks that are distributed across the execution nodes in the cluster. These nodes run a process, called the DataNode, that stores the blocks as files in the local file system.

Within the Hadoop application tier, we implement the file system Java interface using a Java® Native Interface (JNI) adaptation to our C++ DNN client library. The DNN client library adapts the HDFS file system interface to the partitioning scheme, and it enables a more performing I/O stack. We use a proprietary implementation of the DataNode that was initially designed to optimize HDFS using the RAID storage technology. This component has been specialized to accommodate the partitioning of the namespace at the file system metadata tier.

3.2.1 Hash Partitioning the HDFS Namespace

The client-side implementation of HDFS interacts with the distributed NameNodes to meet the contract of the file system interface. Each of these interactions is parameterized by the pathname of the targeted entity: either a filename or a directory name. Our distributed implementation leverages this parameter to direct metadata requests to the appropriate NameNode. The full pathname of a file/directory is hashed to a 32-bit integer that is then mapped onto one of the current partitions to determine the responsible NameNode. DataNodes report block updates to the metadata service for each new block received, and they periodically deliver a full inventory report. Our specialized DataNode leverages the hash partitioning to direct block-received postings to the correct NameNodes; the block inventory report is similarly partitioned using the hash values. However, the DataNode manages blocks via the block-ID and is not cognizant of the pathname of the file that is associated with a particular block. Hence the 64-bit block identifier encodes the hash of the owning file's pathname as the upper 32 bits, and the lower 32 bits as the block-ID assigned by the NameNode. When files are renamed, the *DataNode-to-NameNode protocol*, to be defined next, coordinates the reassignment of identifiers.

The primary differentiator for interactions with the metadata service, compared to the single-NameNode approach, is the application of hashing to direct operations to a particular partition. All client file system operations transparently apply the hash-partitioning function to the pathname parameter of the operation. DataNode reports to the distributed NameNode tier are similarly partitioned, with the exception of the heartbeat messages that are sent to all active NameNodes. The hash partitioning step is encapsulated in the protocol layer, so the core operations of the processes interacting with the NameNode are not changed.

Our approach has made it extremely efficient for operations such as *open*, *create*, *stat*

and *rm* as we avoided the path traversal by directly storing the filename in the *flattened namespace*. For the *readdir* operation, we store the filename record that belongs to the directory directly inside the *dentry* data structure for prefetching purpose, once the client gets all the filenames, it can *stat* all these files in one batch.

3.2.2 Distributed NameNode Cluster

An arbitrary number of NameNodes can be composed into a distributed NameNode Cluster. As shown in figure 3.2, one of the NameNodes will be designated as the *Master NameNode*, which coordinates and manages the other NameNodes. The *Master NameNode* will build the *Namespace Partition Table* (NPT) based on the number of NameNodes in the cluster at start and propagate the NPT to other NameNodes.If the *Master NameNode* fails, a leader election process takes place to designate an alternate master. Currently, we decide to utilize the leader election capability provided by Apache ZooKeeper.

Clients of the distributed NameNode, i.e., file system clients and DataNodes acquire the initial NPT upon registration, and are updated automatically if the partition is altered due to failover or load balancing.

Inter-system functionality is coordinated using the following three protocols:

- **Inter-NameNode Protocol** (INP), used by NameNodes to contact each other. INP is used in two scenarios: 1) Metadata operations that need the cooperation of more than one NameNode (e.g. mkdir), and 2) Inter-NameNode heartbeat and reports that are managed by the *Master NameNode*.

- **Client-to-NameNode Protocol** (CNP), used by Clients to contact NameNodes. Upon receiving a request, a Client will calculate the hash value based on the file/directory path and will decide, based on the NPT, which NameNode is responsible for the operation.

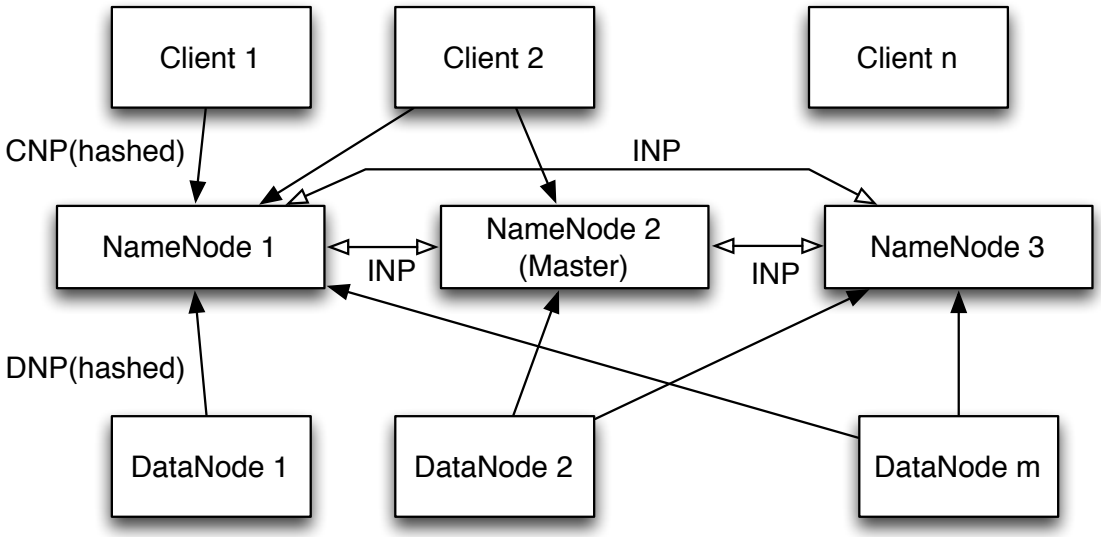

Figure 3.2: Detailed view of the DNN architecture.

- **DataNode-to-NameNode Protocol** (DNP), used by DataNodes to contact NameNodes. A DataNode will send block-received indications to the NameNode for the file to which the block belongs. As described in Section 3.2.1, the first 32 bits of the block ID and the NPT can be used by DataNode to idenfity the applicable NameNode for a block. Full block inventories are partitioned and communicated to the appropriate NameNodes on a periodic basis.

The most common interactions with the NameNode tier involve a single partition. These include file read operations and simple directory listings. File creation involves at most two NameNodes: one responsible for the file and another for the associated directory. The client (transparently) hashes the pathname of the file, and sends the *create* request to the NameNode with the applicable partition that the hash of the file belongs to. The receiving NameNode determines the partition for the containing directory for the file and, when required, leverages the Inter-NameNode Protocol to add this file record into the directory entry. Same as the standard NameNode, the receiving NameNode will choose a group of

DataNodes based on the block placement policy and return the addresses to the client.

There are some metadata operations that involve additional communication complexity for the distributed NameNode tier. A request for a full directory listing (one including size, ownership, etc. for each directory entry) could involve all of the NameNodes. This request is handled efficiently by the NameNode owning the directory. The directory entries are partitioned using the hash mechanism, and then up to $n-1$ messages are sent to the other NameNodes to collect the listing. The renaming of a file can involve at most four NameNodes, one for each pathname involved: source filename, source directory, destination filename, destination directory. The inclusion of the hash value in the corresponding block IDs for the file leads to additional communication complexity, which is bounded by the lesser of the total number of block replicas for the file, and the number of DataNodes. However, this use case involves only metadata updates and is completed in an efficient and consistent manner. For operations that involve the cooperation of more than one NameNode, the consistency and atomicity of the operation are guaranteed by the NameNode that initiates the request, and it will rollback if this distributed transaction fails.

As the brain of the HDFS cluster, HDFS NameNode not only services the metadata requests, it also plays a role in block placement and DataNode management. In our Distributed NameNode architecture, as each NameNode only manages a subset of the file system namespace and there is no overlapping among different NameNodes, each NameNode will make block placement decisions on a per-file basis by itself. The DataNodes and the Distributed NameNode cluster have an N-to-N relationship that is tied together by the first 32 bits of the block ID.

3.2.3 Metadata Storage

Internally, each NameNode maintains the metadata for its partition using the LevelDB key-value store based on the log-structured merge (LSM) tree data storage mechanism

[9, 20, 23].

Instead of storing the namespace in DRAM like the standard NameNode, the LevelDB key-value stores for multiple NameNodes are placed on a shared storage system, allowing the size of the metadata for a partition to grow beyond the limits of the system main memory. LevelDB maintains a substantial cache of metadata, resulting in efficiencies that rival the performance of an in-memory implementation. LevelDB ensures the consistency of persistent data through atomic operations. We support the overall consistency of file creation by using the synchronized write capabilities of LevelDB to commit all of the database records associated with a file to disk upon closure of the file.

We store each metadata record as a key-value pair in LevelDB, with the file or directory name as the key, and the metadata structure as the value. The inode structure for file contains the metadata for the file, including permissions, size, etc. The inode structure also contains the metadata of the blocks that belongs to this file, including block location, which is the mapping between the block and the DataNode where it resides. The directory entry is a bit different as it also contains all the file's name under this directory for *readdir* prefetching, as mentioned before.

3.2.4 Load Balancing

Our implementation for persistence of the metadata includes a novel mechanism for enabling metadata load balancing. Within each partition the namespace is further partitioned into sub-partitions, called *Storage Buckets*. Each *Storage Bucket* is persistent as an independent LevelDB key-value store instance. The instance for each storage bucket is deployed in a separately-mounted local file system. The underlying volume for each of these local file systems is provisioned from a shared RAID storage. Hence the coarse-grained partitioning exposed to the users of the metadata service can be rebalanced by redefining the partitioning on storage bucket boundaries. This mechanism is leveraged to address both

normally-occurring imbalances and to effectively rebalance in response to the failure of one of the NameNodes in the cluster.

Partitioning by hashing guarantees that files/directories are evenly distributed to each partition; however this does not lead to an evenly load-balanced metadata storage state among NameNodes. As NameNodes also maintain metadata of the HDFS Blocks, they may skew the metadata distribution as some files may contain more blocks than others. When the *Master NameNode* observes this sort of skew, it coordinates an orderly reassignment of the *Storage Buckets* among the NameNodes. The reallocation is accomplished using the the shared access enabled by the underlying RAID storage tier.

3.2.5 High Availability

To achieve the first design goal, we use an automatic failover mechanism in our distributed NameNode cluster. At least two NameNodes are required to form a distributed NameNode cluster so that even if one of them fails, other remaining NameNode(s) can take over its role.

The automatic failover process is similar to the load balancing process. Within the Inter-NameNode protocol, a periodical heartbeat mechanism is utilized to detect failures among NameNodes: all NameNodes in the Distributed NameNode cluster will send heartbeat message to the Master NameNode. The heartbeat message contains the current status of the NameNode, including current network load, capacity, etc. If the Master NameNode does not receive heartbeat message from a certain NameNode over a period, it will assume the NameNode has encountered a failure and will initiate the fail-over process.

When the Master NameNode has detected a failure, it will reassigns the control *Storage Buckets* of the failed NameNode to all remaining NameNodes so that no single NameNode will be overloaded. The Master NameNode will re-compute the hash ranges for the sub-

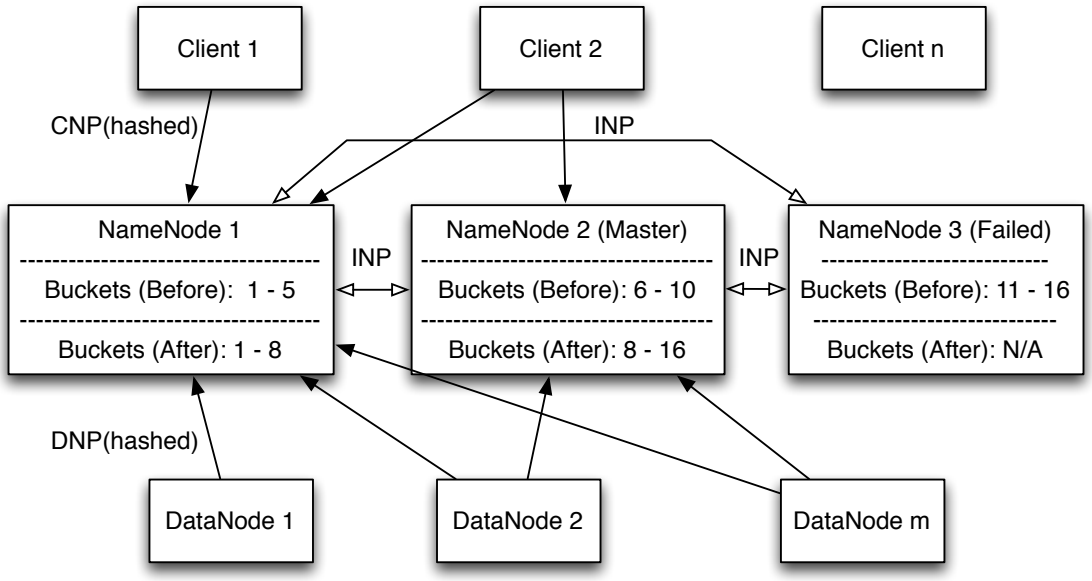

Figure 3.3: Architecture of DNN when one NameNode failed. In the figure, DNN is formed by three NameNodes, with NameNode 2 as Master. The figure shows the changes in DNN when NameNode 3 has failed.

partitions based on the status of the remaining NameNodes, and update the Namespace Partition Table. The new NPT will be propagated to the clients, the remaining NameNodes, and all the DataNodes following the protocols so that they will be able to route the requests to the correct NameNode according to the new NPT. The re-computation and propagation will take a very short time without user's notice. The outstanding metadata requests will sit in the client's queue and retry after it get the new NPT. It is worth noting that, in our design, the reassignment is only a logical control assignment and no data movement is involved as the metadata is stored on the backend shared storage system. This design simplifies the implementation and avoids the time for data transfer.

The following table shows the NPT before and after one of the NameNodes fails.

If the failed NameNode was the Master, a leader election process, provided by the Apache ZooKeeper framework [17] will be triggered prior to the rebalancing, so that the newly elected Master can initiate the failover process. The general flow of the leader elec-

NameNode	Hash Range Old	Buckets Old	Hash Range New	Buckets New
NameNode 1	0000 - 4FFF	1, 2, 3, 4, 5	0000 - 7FFFF	1, 2, 3, 4, 5, 6, 7, 8
NameNode 2	5000 - 9FFF	6, 7, 8, 9, 10	8000 - FFFF	9, 10, 11, 12, 13, 14, 15, 16
NameNode 3	AFFF - FFFF	11, 12, 13, 14, 15 ,16	N/A	N/A

Table 3.1: Namespace Partition Table before failure and after one NameNode failure. The Hash Range and Storage Buckets has been re-computed after the failure.

tion process within a group is as follows:

1. The surviving NameNodes nominate themselves to become a Master.

2. The ZooKeeper service chooses and notifies the new Master NameNode, which will then initiate the DNN failover process.

3. The ZooKeeper service monitors the liveness of the Master NameNode. When the service stops seeing the Master, election of a new leader is initiated.

4. At any point in time, other NameNodes can query the service for the current Master NameNode, or subscribe for notifications to receive leadership changes.

It is worth noting that, in our design, the Master NameNode has exactly the same responsibility as the other NameNodes, except that it is the initiator of the failover and load balancing process. Other than that, it stores exactly the same thing as other NameNodes. This ensures that the Master NameNode won't become the new single point of failure.

To achieve the second design goal, currently we choose to offload the data protection of metadata to the enterprise NAS or SAN RAID storage system which can deliver 99.999% data availability. Each sub-partition or *Storage Bucket* is stored as a LevelDB SSTable file in a logical unit number (LUN) on the remote shared storage system. After the failure of

one NameNode, other remaining NameNodes will mount the LUN that the new storage buckets belongs to. For example, as shown in Table 3.1, after NameNode 1 get the new NPT, it will mount the LUNs with *Storage Bucket* 6, 7, 8. Similarly, NameNode 2 will mount the LUNs with *Storage Bucket* 11 through 16, but also unmount the LUNs with *Storage Bucket* 6, 7 and 8 as they now are handled by NameNode 1.

Chapter 4

Performance Evaluation

To evaluate the performance of DNN we conduct two kinds of tests: a micro-benchmark is used to measure the performance of the NameNode tier in isolation, and a standard Hadoop benchmark is used to assess the performance of the overall deployment. For both tests, we measure the performance of HDFS with our distributed NameNode and compare the results obtained with the performance of HDFS using the standard NameNode.

Our test cluster consists of thirteen slave nodes (DataNode and TaskTracker collocated) and up to four NameNodes. Each node in the test cluster has 16 Intel Xeon® CPU cores and 96 GB of DRAM. All systems are connected via 10Gb Ethernet and are running Ubuntu Linux 12.04. Each slave node runs our proprietary DataNode implementation for the DNN tests, and the standard Hadoop DataNode for the Hadoop 1.1.1 NameNode. A 14-drive RAID-6 volume is used for data storage at each of the slave nodes.

4.1 Micro Benchmark Performance

We stress test our DNN and measure the performance and scalability through micro metadata-intensive benchmarks. In this test, the multi-threaded benchmark on each client node concurrently sends metadata requests to the NameNode. The metadata operations we measured

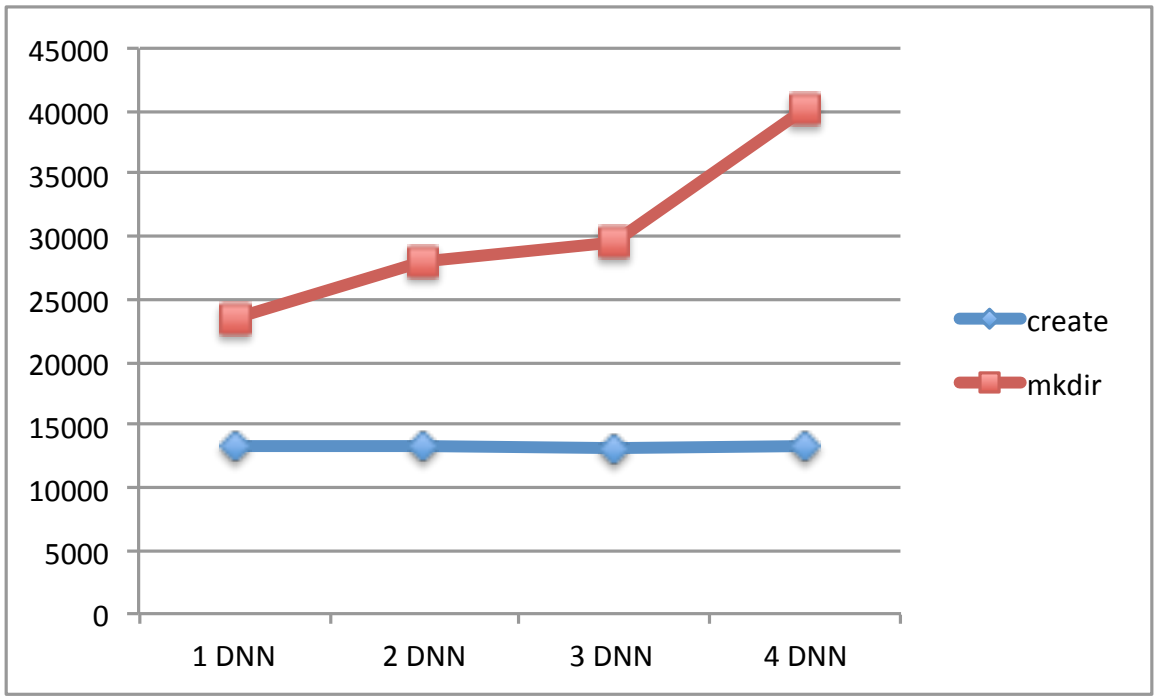

Figure 4.1: Result for metadata-intensive micro benchmark.

are *create*, *list* and *mkdir*. These results are preliminary, and we intend to increase this performance as our implementation reaches maturity. The table below shows the results of our test:

Test	**Std NN**	**1 DNN**	**2 DNN**	**3 DNN**	**4 DNN**
create	3,893 op/s	13,385 op/s	13,456 op/s	13,178 op/s	13,335 op/s
list	7,277 op/s	2,362 op/s	1,368 op/s	2,245 op/s	2,220 op/s
mkdir	N/A	23,521 op/s	27,979 op/s	29,479 op/s	40,108 op/s

Table 4.1: Result for Micro Benchmark

The "*create*" test is a modification of the standard NameNode Benchmark MapReduce job, which creates a new file in HDFS. For DNN, the amount of inter-NameNode traffic increases with the size of the NameNode cluster.

The "*list*" test is a Java client (4 total); the uniformity of the results indicates that the metadata tier is not saturated. Finally, the "mkdir" test is a C++ client that measures the number of directories that can be created by a fixed number of clients (64 total), reported in the number created per second. This test is of interest since the "*mkdir*" operation represents a distributed transaction: each operation will need to span across 2 NameNodes in this test, one for permission checking in the parent directory in one NameNode and the other to create the actual record in a different NameNode. The two RPC connections represent an extra overhead; however, the results in the table show that increasing the number of NameNodes, which increases the percentage of inter-NameNode communication, does not negatively affect the rate of execution. This is because loads are distributed and absorbed by the extra NameNodes, and the benefits far exceed the overhead generated by the distributed transaction. In summary, this micro metadata-intensive benchmark demonstrates the sub-linear scalability of the distributed NameNode cluster.

4.2 Macro MapReduce Benchmark Performance

To measure the I/O performance impact that DNN imposes on the entire HDFS infrastructure in a real Hadoop workload, we execute the standard MapReduce TeraGen and TeraSort benchmarks for 100GB and compare the results with those obtained from the Apache HDFS 1.1.1 distribution. TeraGen and TereSort are focused on the performance of the MapReduce execution framework; they are included to demonstrate that DNN supports standard MapReduce.

Test	**Std NN**	**2 DNN**	**3 DNN**	**4 DNN**
TeraGen	25s	1m14s	1m16s	1m12s
TeraSort	2m23s	6m15s	6m15s	6m28s

Table 4.2: Results for Macro Benchmark

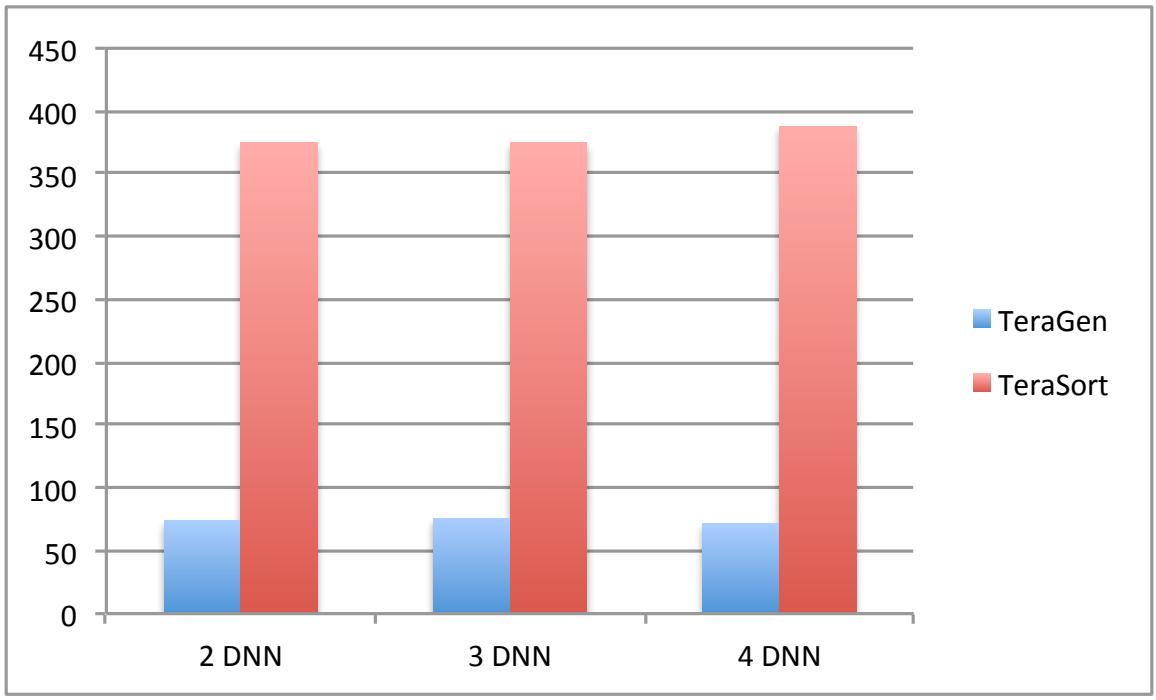

Figure 4.2: Result for the Hadoop TeraGen and Terasort Macro benchmark.

As shown in the table, HDFS with standard NameNode actually performs better than HDFS with Distributed NameNode. The reduced I/O performance for DNN is due to the immature implementation of our client protocol stack. However, the main purpose of this comprehensive macro benchmark test is to demonstrate a fully functional file system stack. The uniformity between NameNode cluster cardinalities displays the efficacy of the distributed approach for the metadata tier. We will address the client issue as future work.

Chapter 5

Discussion of Future Work

We are currently optimizing and tuning our file system, and we are trying to better integrate it into the Hadoop ecosystem. Our design goal is to make a reliable, scalable, high-throughput, and low-latency distributed storage back end for the real-time, big-data applications.

The Distributed NameNode, especially the client stack, is still in the prototyping phase. We need to increase the scale and intensity of the micro benchmark to better demonstrate the performance and scalability of the DNN NameNode layer. We also will use real world Hadoop workload to test the DNN.

In addition to addressing the client-side issues, we will also continue the development of robust H.A. capability and will evaluate its performance in a degraded state. We will also assess how fast the failover can complete.

We have also observed other unique challenges for the NameNode, brought by the new type of real time, low-latency workloads. An example of this is the load imbalance brought by the default random block-placement policy for assigning blocks to the DataNodes. We are designing an improved block-placement policy that maintains per-file balance across the DataNode cluster. Additionally, we are evaluating the possibility of optimizing data

placement for the small file problem. In the long term, we also plan to support random writes and NFS access, and we are planning to introduce block placement to flash storage as an additional performance storage tier.

Chapter 6

Conclusion

In this thesis, we proposed a novel Distributed NameNode architecture for the Hadoop Distributed File System and evaluated its performance using both micro metadata-intensive benchmark and macro Hadoop MapReduce benchmark. Specifically, we partition the flattened namespace of Hadoop Distributed File System using the well-established hashing approach based our observation of the uniqueness of HDFS and Hadoop workload. The hashing approach has been proposed before, but its major disadvantage: the loss of hierarchical locality, prevents it from being adapted by POSIX-based distributed file system as they rely heavily on the hierarchical locality for metadata prefetching. However we found that this method fit perfectly for HDFS and the Hadoop workload as previous workload studies suggest that most metadata operations (e.g. *open* accounts for 97% of all operations) in Hadoop does not exploit hierarchical locality. By adapting this approach into HDFS, our solution elegantly solved the single point of failure of NameNode that has troubled the HDFS community for years. We developed our HDFS prototype with Distributed NameNode in C++ and successfully integrated it into the standard Hadoop framework.

Bibliography

[1] B-tree. http://en.wikipedia.org/wiki/B-tree.

[2] C. L. Abad, N. Roberts, Y. Lu, and R. H. Campbell. A storage-centric analysis of MapReduce workloads: File popularity, temporal locality and arrival patterns. In *IISWC*, pages 100–109. IEEE Computer Society, 2012.

[3] Amazon.com. Amazon simple storage service(s3). http://aws.amazon.com/s3/.

[4] Apache.org. Apache hbase. http://hbase.apache.org.

[5] Apache.org. Hadoop distributed file system. http://hadoop.apache.org.

[6] Apache.org. Hdfs high availability. http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/HDFSHighAvailabilityWithNFS.html.

[7] D. Borthakur and et al. Apache hadoop goes realtime at facebook. SIGMOD '11. ACM.

[8] S. Brandt, E. Miller, D. D. E. Long, and L. Xue. Efficient metadata management in large distributed storage systems. In *Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 290–298, April 2003.

[9] F. Chang and et al. Bigtable: a distributed storage system for structured data. OSDI '06.

[10] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 390–399, Washington, DC, USA, 2011. IEEE Computer Society.

[11] H. Dave, L. James, and M. Michael. File system design for an nfs file server appliance. In *USENIX'94: Proceedings of the 1994 USENIX Annual Technical Conference*, 1994.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[13] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. Diskreduce: Raid for data-intensive scalable computing. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 6–10, New York, NY, USA, 2009. ACM.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, (5), 2003.

[15] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of HDFS Under HBase: A Facebook Messages Case Study. *USENIX Conference on File Storage Technologies (FAST)*, 2014.

[16] J. H. Howard. An overview of the andrew file system. In *USENIX Conference Proceedings '88*, 1988.

[17] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[18] M. G. Institute. Big data: The next frontier for innovation, competition, and productivity. 2011.

[19] M. K. McKusick and S. Quinlan. Gfs: Evolution on fast-forward. *Queue*, 7(7):10:10–10:20, Aug. 2009.

[20] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[21] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The quantcast file system. *Proc. VLDB Endow.*, 6(11):1092–1101, Aug. 2013.

[22] S. Patil and G. Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *FAST '11*, 2011.

[23] K. Ren and G. Gibson. Tablefs: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 145–156, Berkeley, CA, USA, 2013. USENIX Association.

[24] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. *Proc. VLDB Endow.*, 6(10):853–864, Aug. 2013.

[25] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: novel erasure codes for big data. In *Proceedings*

of the 39th international conference on Very Large Data Bases, PVLDB'13, pages 325–336. VLDB Endowment, 2013.

[26] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: a highly available file system for a distributed workstation environment. *Computers, IEEE Transactions on*, 39(4):447–459, Apr 1990.

[27] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06*, 2006.

[28] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 4–, Washington, DC, USA, 2004. IEEE Computer Society.

[29] B. B. Welch. *Naming, State Management, and User-level Extensions in the Sprite Distributed File System*. PhD thesis, 1990. AAI9103923.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. HotCloud'10.