The Hadoop Distributed File System

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler
Yahoo!
Sunnyvale, California USA
{Shv, Hairong, SRadia, Chansler}@Yahoo-Inc.com

***Abstract*—The Hadoop Distributed File System (HDFS) is designed to store very large data sets reliably, and to stream those data sets at high bandwidth to user applications. In a large cluster, thousands of servers both host directly attached storage and execute user application tasks. By distributing storage and computation across many servers, the resource can grow with demand while remaining economical at every size. We describe the architecture of HDFS and report on experience using HDFS to manage 25 petabytes of enterprise data at Yahoo!.**

Keywords: Hadoop, HDFS, distributed file system

I. INTRODUCTION AND RELATED WORK

Hadoop [1][16][19] provides a distributed file system and a framework for the analysis and transformation of very large data sets using the MapReduce [3] paradigm. An important characteristic of Hadoop is the partitioning of data and computation across many (thousands) of hosts, and executing application computations in parallel close to their data. A Hadoop cluster scales computation capacity, storage capacity and IO bandwidth by simply adding commodity servers. Hadoop clusters at Yahoo! span 25 000 servers, and store 25 petabytes of application data, with the largest cluster being 3500 servers. One hundred other organizations worldwide report using Hadoop.

HDFS	Distributed file system Subject of this paper!
MapReduce	Distributed computation framework
HBase	Column-oriented table service
Pig	Dataflow language and parallel execution framework
Hive	Data warehouse infrastructure
ZooKeeper	Distributed coordination service
Chukwa	System for collecting management data
Avro	Data serialization system

Table 1. Hadoop project components

Hadoop is an Apache project; all components are available via the Apache open source license. Yahoo! has developed and contributed to 80% of the core of Hadoop (HDFS and MapReduce). HBase was originally developed at Powerset, now a department at Microsoft. Hive [15] was originated and developed at Facebook. Pig [4], ZooKeeper [6], and Chukwa were originated and developed at Yahoo! Avro was originated at Yahoo! and is being co-developed with Cloudera.

HDFS is the file system component of Hadoop. While the interface to HDFS is patterned after the UNIX file system, faithfulness to standards was sacrificed in favor of improved performance for the applications at hand.

HDFS stores file system metadata and application data separately. As in other distributed file systems, like PVFS [2][14], Lustre [7] and GFS [5][8], HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called DataNodes. All servers are fully connected and communicate with each other using TCP-based protocols.

Unlike Lustre and PVFS, the DataNodes in HDFS do not use data protection mechanisms such as RAID to make the data durable. Instead, like GFS, the file content is replicated on multiple DataNodes for reliability. While ensuring data durability, this strategy has the added advantage that data transfer bandwidth is multiplied, and there are more opportunities for locating computation near the needed data.

Several distributed file systems have or are exploring truly distributed implementations of the namespace. Ceph [17] has a cluster of namespace servers (MDS) and uses a dynamic subtree partitioning algorithm in order to map the namespace tree to MDSs evenly. GFS is also evolving into a distributed namespace implementation [8]. The new GFS will have hundreds of namespace servers (masters) with 100 million files per master. Lustre [7] has an implementation of clustered namespace on its roadmap for Lustre 2.2 release. The intent is to stripe a directory over multiple metadata servers (MDS), each of which contains a disjoint portion of the namespace. A file is assigned to a particular MDS using a hash function on the file name.

II. ARCHITECTURE

A. NameNode

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by *inodes*, which record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (typically 128 megabytes, but user selectable file-by-file) and each block of the file is independently replicated at multiple DataNodes (typically three, but user selectable file-by-file). The NameNode maintains the namespace tree and the mapping of file blocks to DataNodes

978-1-4244-7153-9/10/$26.00 ©2010 IEEE

(the physical location of file data). An HDFS client wanting to read a file first contacts the NameNode for the locations of data blocks comprising the file and then reads block contents from the DataNode closest to the client. When writing data, the client requests the NameNode to nominate a suite of three DataNodes to host the block replicas. The client then writes data to the DataNodes in a pipeline fashion. The current design has a single NameNode for each cluster. The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster, as each DataNode may execute multiple application tasks concurrently.

HDFS keeps the entire namespace in RAM. The inode data and the list of blocks belonging to each file comprise the metadata of the name system called the *image*. The persistent record of the image stored in the local host's native files system is called a *checkpoint*. The NameNode also stores the modification log of the image called the *journal* in the local host's native file system. For improved durability, redundant copies of the checkpoint and journal can be made at other servers. During restarts the NameNode restores the namespace by reading the namespace and replaying the journal. The locations of block replicas may change over time and are not part of the persistent checkpoint.

B. DataNodes

Each block replica on a DataNode is represented by two files in the local host's native file system. The first file contains the data itself and the second file is block's metadata including checksums for the block data and the block's *generation stamp*. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional file systems. Thus, if a block is half full it needs only half of the space of the full block on the local drive.

During startup each DataNode connects to the NameNode and performs a *handshake*. The purpose of the handshake is to verify the *namespace ID* and the *software version* of the DataNode. If either does not match that of the NameNode the DataNode automatically shuts down.

The namespace ID is assigned to the file system instance when it is formatted. The namespace ID is persistently stored on all nodes of the cluster. Nodes with a different namespace ID will not be able to join the cluster, thus preserving the integrity of the file system.

The consistency of software versions is important because incompatible version may cause data corruption or loss, and on large clusters of thousands of machines it is easy to overlook nodes that did not shut down properly prior to the software upgrade or were not available during the upgrade.

A DataNode that is newly initialized and without any namespace ID is permitted to join the cluster and receive the cluster's namespace ID.

After the handshake the DataNode *registers* with the NameNode. DataNodes persistently store their unique *storage IDs*. The storage ID is an internal identifier of the DataNode, which makes it recognizable even if it is restarted with a different IP address or port. The storage ID is assigned to the DataNode when it registers with the NameNode for the first time and never changes after that.

A DataNode identifies block replicas in its possession to the NameNode by sending a *block report*. A block report contains the *block id*, the *generation stamp* and the length for each block replica the server hosts. The first block report is sent immediately after the DataNode registration. Subsequent block reports are sent every hour and provide the NameNode with an up-to-date view of where block replicas are located on the cluster.

During normal operation DataNodes send *heartbeats* to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available. The default heartbeat interval is three seconds. If the NameNode does not receive a heartbeat from a DataNode in ten minutes the NameNode considers the DataNode to be out of service and the block replicas hosted by that DataNode to be unavailable. The NameNode then schedules creation of new replicas of those blocks on other DataNodes.

Heartbeats from a DataNode also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. These statistics are used for the NameNode's space allocation and load balancing decisions.

The NameNode does not directly call DataNodes. It uses replies to heartbeats to send instructions to the DataNodes. The instructions include commands to:

- replicate blocks to other nodes;
- remove local block replicas;
- re-register or to shut down the node;
- send an immediate block report.

These commands are important for maintaining the overall system integrity and therefore it is critical to keep heartbeats frequent even on big clusters. The NameNode can process thousands of heartbeats per second without affecting other NameNode operations.

C. HDFS Client

User applications access the file system using the HDFS client, a code library that exports the HDFS file system interface.

Similar to most conventional file systems, HDFS supports operations to read, write and delete files, and operations to create and delete directories. The user references files and directories by paths in the namespace. The user application generally does not need to know that file system metadata and storage are on different servers, or that blocks have multiple replicas.

When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. It then contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the

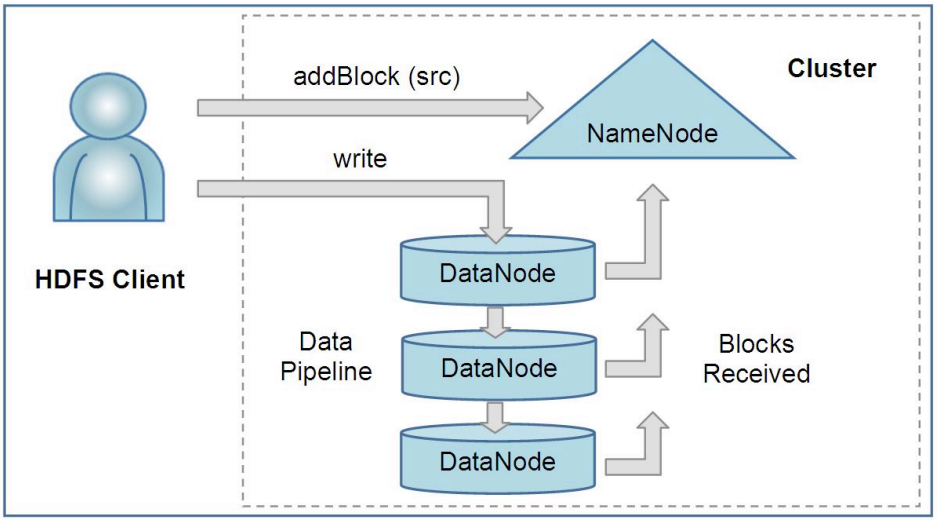

Figure 1. An HDFS client creates a new file by giving its path to the NameNode. For each block of the file, the NameNode returns a list of DataNodes to host its replicas. The client then pipelines data to the chosen DataNodes, which eventually confirm the creation of the block replicas to the NameNode.

client sends the further bytes of the file. Each choice of DataNodes is likely to be different. The interactions among the client, the NameNode and the DataNodes are illustrated in Fig. 1.

Unlike conventional file systems, HDFS provides an API that exposes the locations of a file blocks. This allows applications like the MapReduce framework to schedule a task to where the data are located, thus improving the read performance. It also allows an application to set the replication factor of a file. By default a file's replication factor is three. For critical files or files which are accessed very often, having a higher replication factor improves their tolerance against faults and increase their read bandwidth.

D. *Image and Journal*

The namespace image is the file system metadata that describes the organization of application data as directories and files. A persistent record of the image written to disk is called a *checkpoint*. The journal is a write-ahead commit log for changes to the file system that must be persistent. For each client-initiated transaction, the change is recorded in the journal, and the journal file is flushed and synched before the change is committed to the HDFS client. The checkpoint file is never changed by the NameNode; it is replaced in its entirety when a new checkpoint is created during restart, when requested by the administrator, or by the CheckpointNode described in the next section. During startup the NameNode initializes the namespace image from the checkpoint, and then replays changes from the journal until the image is up-to-date with the last state of the file system. A new checkpoint and empty journal are written back to the storage directories before the NameNode starts serving clients.

If either the checkpoint or the journal is missing, or becomes corrupt, the namespace information will be lost partly or entirely. In order to preserve this critical information HDFS can be configured to store the checkpoint and journal in multiple storage directories. Recommended practice is to place the directories on different volumes, and for one storage directory to be on a remote NFS server. The first choice prevents loss from single volume failures, and the second choice protects against failure of the entire node. If the NameNode encounters an error writing the journal to one of the storage directories it automatically excludes that directory from the list of storage directories. The NameNode automatically shuts itself down if no storage directory is available.

The NameNode is a multithreaded system and processes requests simultaneously from multiple clients. Saving a transaction to disk becomes a bottleneck since all other threads need to wait until the synchronous flush-and-sync procedure initiated by one of them is complete. In order to optimize this process the NameNode batches multiple transactions initiated by different clients. When one of the NameNode's threads initiates a flush-and-sync operation, all transactions batched at that time are committed together. Remaining threads only need to check that their transactions have been saved and do not need to initiate a flush-and-sync operation.

E. *CheckpointNode*

The NameNode in HDFS, in addition to its primary role serving client requests, can alternatively execute either of two other roles, either a *CheckpointNode* or a *BackupNode*. The role is specified at the node startup.

The CheckpointNode periodically combines the existing checkpoint and journal to create a new checkpoint and an empty journal. The CheckpointNode usually runs on a different host from the NameNode since it has the same memory requirements as the NameNode. It downloads the current checkpoint and journal files from the NameNode, merges them locally, and returns the new checkpoint back to the NameNode.

Creating periodic checkpoints is one way to protect the file system metadata. The system can start from the most recent checkpoint if all other persistent copies of the namespace image or journal are unavailable.

Creating a checkpoint lets the NameNode truncate the tail of the journal when the new checkpoint is uploaded to the NameNode. HDFS clusters run for prolonged periods of time without restarts during which the journal constantly grows. If the journal grows very large, the probability of loss or corruption of the journal file increases. Also, a very large journal extends the time required to restart the NameNode. For a large cluster, it takes an hour to process a week-long journal. Good practice is to create a daily checkpoint.

F. BackupNode

A recently introduced feature of HDFS is the *BackupNode*. Like a CheckpointNode, the BackupNode is capable of creating periodic checkpoints, but in addition it maintains an in-memory, up-to-date image of the file system namespace that is always synchronized with the state of the NameNode.

The BackupNode accepts the journal stream of namespace transactions from the active NameNode, saves them to its own storage directories, and applies these transactions to its own namespace image in memory. The NameNode treats the BackupNode as a journal store the same as it treats journal files in its storage directories. If the NameNode fails, the BackupNode's image in memory and the checkpoint on disk is a record of the latest namespace state.

The BackupNode can create a checkpoint without downloading checkpoint and journal files from the active NameNode, since it already has an up-to-date namespace image in its memory. This makes the checkpoint process on the BackupNode more efficient as it only needs to save the namespace into its local storage directories.

The BackupNode can be viewed as a read-only NameNode. It contains all file system metadata information except for block locations. It can perform all operations of the regular NameNode that do not involve modification of the namespace or knowledge of block locations. Use of a BackupNode provides the option of running the NameNode without persistent storage, delegating responsibility for the namespace state persisting to the BackupNode.

G. Upgrades, File System Snapshots

During software upgrades the possibility of corrupting the system due to software bugs or human mistakes increases. The purpose of creating snapshots in HDFS is to minimize potential damage to the data stored in the system during upgrades.

The snapshot mechanism lets administrators persistently save the current state of the file system, so that if the upgrade results in data loss or corruption it is possible to rollback the upgrade and return HDFS to the namespace and storage state as they were at the time of the snapshot.

The snapshot (only one can exist) is created at the cluster administrator's option whenever the system is started. If a snapshot is requested, the NameNode first reads the checkpoint and journal files and merges them in memory. Then it writes the new checkpoint and the empty journal to a new location, so that the old checkpoint and journal remain unchanged.

During handshake the NameNode instructs DataNodes whether to create a local snapshot. The local snapshot on the DataNode cannot be created by replicating the data files directories as this will require doubling the storage capacity of every DataNode on the cluster. Instead each DataNode creates a copy of the storage directory and hard links existing block files into it. When the DataNode removes a block it removes only the hard link, and block modifications during appends use the copy-on-write technique. Thus old block replicas remain untouched in their old directories.

The cluster administrator can choose to roll back HDFS to the snapshot state when restarting the system. The NameNode recovers the checkpoint saved when the snapshot was created. DataNodes restore the previously renamed directories and initiate a background process to delete block replicas created after the snapshot was made. Having chosen to roll back, there is no provision to roll forward. The cluster administrator can recover the storage occupied by the snapshot by commanding the system to abandon the snapshot, thus finalizing the software upgrade.

System evolution may lead to a change in the format of the NameNode's checkpoint and journal files, or in the data representation of block replica files on DataNodes. The *layout version* identifies the data representation formats, and is persistently stored in the NameNode's and the DataNodes' storage directories. During startup each node compares the layout version of the current software with the version stored in its storage directories and automatically converts data from older formats to the newer ones. The conversion requires the mandatory creation of a snapshot when the system restarts with the new software layout version.

HDFS does not separate layout versions for the NameNode and DataNodes because snapshot creation must be an all-cluster effort rather than a node-selective event. If an upgraded NameNode due to a software bug purges its image then backing up only the namespace state still results in total data loss, as the NameNode will not recognize the blocks reported by DataNodes, and will order their deletion. Rolling back in this case will recover the metadata, but the data itself will be lost. A coordinated snapshot is required to avoid a cataclysmic destruction.

III. File I/O Operations and Replica Mangement

A. File Read and Write

An application adds data to HDFS by creating a new file and writing the data to it. After the file is closed, the bytes written cannot be altered or removed except that new data can be added to the file by reopening the file for append. HDFS implements a single-writer, multiple-reader model.

The HDFS client that opens a file for writing is granted a lease for the file; no other client can write to the file. The writing client periodically renews the lease by sending a heartbeat to the NameNode. When the file is closed, the lease is revoked.

The lease duration is bound by a soft limit and a hard limit. Until the soft limit expires, the writer is certain of exclusive access to the file. If the soft limit expires and the client fails to close the file or renew the lease, another client can preempt the lease. If after the hard limit expires (one hour) and the client has failed to renew the lease, HDFS assumes that the client has quit and will automatically close the file on behalf of the writer, and recover the lease. The writer's lease does not prevent other clients from reading the file; a file may have many concurrent readers.

An HDFS file consists of blocks. When there is a need for a new block, the NameNode allocates a block with a unique block ID and determines a list of DataNodes to host replicas of the block. The DataNodes form a pipeline, the order of which minimizes the total network distance from the client to the last DataNode. Bytes are pushed to the pipeline as a sequence of packets. The bytes that an application writes first buffer at the client side. After a packet buffer is filled (typically 64 KB), the data are pushed to the pipeline. The next packet can be pushed to the pipeline before receiving the acknowledgement for the previous packets. The number of outstanding packets is limited by the outstanding packets window size of the client.

After data are written to an HDFS file, HDFS does not provide any guarantee that data are visible to a new reader until the file is closed. If a user application needs the visibility guarantee, it can explicitly call the *hflush* operation. Then the current packet is immediately pushed to the pipeline, and the hflush operation will wait until all DataNodes in the pipeline acknowledge the successful transmission of the packet. All data written before the hflush operation are then certain to be visible to readers.

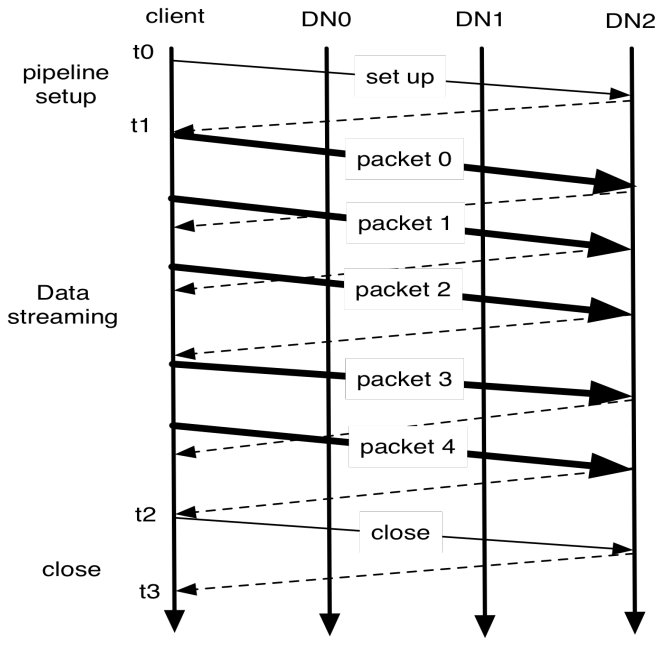

Figure 2. Data pipeline during block construction

If no error occurs, block construction goes through three stages as shown in Fig. 2 illustrating a pipeline of three DataNodes (DN) and a block of five packets. In the picture, bold lines represent data packets, dashed lines represent acknowledgment messages, and thin lines represent control messages to setup and close the pipeline. Vertical lines represent activity at the client and the three DataNodes where time proceeds from top to bottom. From t_0 to t_1 is the pipeline setup stage. The interval t_1 to t_2 is the data streaming stage, where t_1 is the time when the first data packet gets sent and t_2 is the time that the acknowledgment to the last packet gets received. Here an hflush operation transmits the second packet. The hflush indication travels with the packet data and is not a separate operation. The final interval t_2 to t_3 is the pipeline close stage for this block.

In a cluster of thousands of nodes, failures of a node (most commonly storage faults) are daily occurrences. A replica stored on a DataNode may become corrupted because of faults in memory, disk, or network. HDFS generates and stores checksums for each data block of an HDFS file. Checksums are verified by the HDFS client while reading to help detect any corruption caused either by client, DataNodes, or network. When a client creates an HDFS file, it computes the checksum sequence for each block and sends it to a DataNode along with the data. A DataNode stores checksums in a metadata file separate from the block's data file. When HDFS reads a file, each block's data and checksums are shipped to the client. The client computes the checksum for the received data and verifies that the newly computed checksums matches the checksums it received. If not, the client notifies the NameNode of the corrupt replica and then fetches a different replica of the block from another DataNode.

When a client opens a file to read, it fetches the list of blocks and the locations of each block replica from the NameNode. The locations of each block are ordered by their distance from the reader. When reading the content of a block, the client tries the closest replica first. If the read attempt fails, the client tries the next replica in sequence. A read may fail if the target DataNode is unavailable, the node no longer hosts a replica of the block, or the replica is found to be corrupt when checksums are tested.

HDFS permits a client to read a file that is open for writing. When reading a file open for writing, the length of the last block still being written is unknown to the NameNode. In this case, the client asks one of the replicas for the latest length before starting to read its content.

The design of HDFS I/O is particularly optimized for batch processing systems, like MapReduce, which require high throughput for sequential reads and writes. However, many efforts have been put to improve its read/write response time in order to support applications like Scribe that provide real-time data streaming to HDFS, or HBase that provides random, real-time access to large tables.

B. Block Placement

For a large cluster, it may not be practical to connect all nodes in a flat topology. A common practice is to spread the nodes across multiple racks. Nodes of a rack share a switch, and rack switches are connected by one or more core switches. Communication between two nodes in different racks has to go through multiple switches. In most cases, network bandwidth

between nodes in the same rack is greater than network bandwidth between nodes in different racks. Fig. 3 describes a cluster with two racks, each of which contains three nodes.

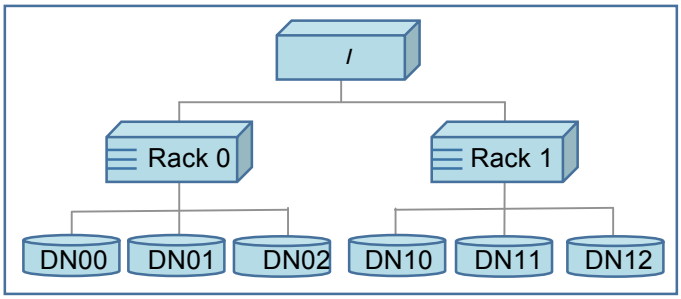

Figure 3. Cluster topology example

HDFS estimates the network bandwidth between two nodes by their distance. The distance from a node to its parent node is assumed to be one. A distance between two nodes can be calculated by summing up their distances to their closest common ancestor. A shorter distance between two nodes means that the greater bandwidth they can utilize to transfer data.

HDFS allows an administrator to configure a script that returns a node's rack identification given a node's address. The NameNode is the central place that resolves the rack location of each DataNode. When a DataNode registers with the NameNode, the NameNode runs a configured script to decide which rack the node belongs to. If no such a script is configured, the NameNode assumes that all the nodes belong to a default single rack.

The placement of replicas is critical to HDFS data reliability and read/write performance. A good replica placement policy should improve data reliability, availability, and network bandwidth utilization. Currently HDFS provides a configurable block placement policy interface so that the users and researchers can experiment and test any policy that's optimal for their applications.

The default HDFS block placement policy provides a tradeoff between minimizing the write cost, and maximizing data reliability, availability and aggregate read bandwidth. When a new block is created, HDFS places the first replica on the node where the writer is located, the second and the third replicas on two different nodes in a different rack, and the rest are placed on random nodes with restrictions that no more than one replica is placed at one node and no more than two replicas are placed in the same rack when the number of replicas is less than twice the number of racks. The choice to place the second and third replicas on a different rack better distributes the block replicas for a single file across the cluster. If the first two replicas were placed on the same rack, for any file, two-thirds of its block replicas would be on the same rack.

After all target nodes are selected, nodes are organized as a pipeline in the order of their proximity to the first replica. Data are pushed to nodes in this order. For reading, the NameNode first checks if the client's host is located in the cluster. If yes, block locations are returned to the client in the order of its closeness to the reader. The block is read from DataNodes in this preference order. (It is usual for MapReduce applications to run on cluster nodes, but as long as a host can connect to the NameNode and DataNodes, it can execute the HDFS client.)

This policy reduces the inter-rack and inter-node write traffic and generally improves write performance. Because the chance of a rack failure is far less than that of a node failure, this policy does not impact data reliability and availability guarantees. In the usual case of three replicas, it can reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three.

The default HDFS replica placement policy can be summarized as follows:

1. No Datanode contains more than one replica of any block.
2. No rack contains more than two replicas of the same block, provided there are sufficient racks on the cluster.

C. *Replication management*

The NameNode endeavors to ensure that each block always has the intended number of replicas. The NameNode detects that a block has become under- or over-replicated when a block report from a DataNode arrives. When a block becomes over replicated, the NameNode chooses a replica to remove. The NameNode will prefer not to reduce the number of racks that host replicas, and secondly prefer to remove a replica from the DataNode with the least amount of available disk space. The goal is to balance storage utilization across DataNodes without reducing the block's availability.

When a block becomes under-replicated, it is put in the replication priority queue. A block with only one replica has the highest priority, while a block with a number of replicas that is greater than two thirds of its replication factor has the lowest priority. A background thread periodically scans the head of the replication queue to decide where to place new replicas. Block replication follows a similar policy as that of the new block placement. If the number of existing replicas is one, HDFS places the next replica on a different rack. In case that the block has two existing replicas, if the two existing replicas are on the same rack, the third replica is placed on a different rack; otherwise, the third replica is placed on a different node in the same rack as an existing replica. Here the goal is to reduce the cost of creating new replicas.

The NameNode also makes sure that not all replicas of a block are located on one rack. If the NameNode detects that a block's replicas end up at one rack, the NameNode treats the block as under-replicated and replicates the block to a different rack using the same block placement policy described above. After the NameNode receives the notification that the replica is created, the block becomes over-replicated. The NameNode then will decides to remove an old replica because the over-replication policy prefers not to reduce the number of racks.

D. *Balancer*

HDFS block placement strategy does not take into account DataNode disk space utilization. This is to avoid placing new—more likely to be referenced—data at a small subset of

the DataNodes. Therefore data might not always be placed uniformly across DataNodes. Imbalance also occurs when new nodes are added to the cluster.

The balancer is a tool that balances disk space usage on an HDFS cluster. It takes a threshold value as an input parameter, which is a fraction in the range of (0, 1). A cluster is balanced if for each DataNode, the utilization of the node (ratio of used space at the node to total capacity of the node) differs from the utilization of the whole cluster (ratio of used space in the cluster to total capacity of the cluster) by no more than the threshold value.

The tool is deployed as an application program that can be run by the cluster administrator. It iteratively moves replicas from DataNodes with higher utilization to DataNodes with lower utilization. One key requirement for the balancer is to maintain data availability. When choosing a replica to move and deciding its destination, the balancer guarantees that the decision does not reduce either the number of replicas or the number of racks.

The balancer optimizes the balancing process by minimizing the inter-rack data copying. If the balancer decides that a replica A needs to be moved to a different rack and the destination rack happens to have a replica B of the same block, the data will be copied from replica B instead of replica A.

A second configuration parameter limits the bandwidth consumed by rebalancing operations. The higher the allowed bandwidth, the faster a cluster can reach the balanced state, but with greater competition with application processes.

E. *Block Scanner*

Each DataNode runs a block scanner that periodically scans its block replicas and verifies that stored checksums match the block data. In each scan period, the block scanner adjusts the read bandwidth in order to complete the verification in a configurable period. If a client reads a complete block and checksum verification succeeds, it informs the DataNode. The DataNode treats it as a verification of the replica.

The verification time of each block is stored in a human readable log file. At any time there are up to two files in top-level DataNode directory, current and prev logs. New verification times are appended to current file. Correspondingly each DataNode has an in-memory scanning list ordered by the replica's verification time.

Whenever a read client or a block scanner detects a corrupt block, it notifies the NameNode. The NameNode marks the replica as corrupt, but does not schedule deletion of the replica immediately. Instead, it starts to replicate a good copy of the block. Only when the good replica count reaches the replication factor of the block the corrupt replica is scheduled to be removed. This policy aims to preserve data as long as possible. So even if all replicas of a block are corrupt, the policy allows the user to retrieve its data from the corrupt replicas.

F. *Decommissioing*

The cluster administrator specifies which nodes can join the cluster by listing the host addresses of nodes that are permitted to register and the host addresses of nodes that are *not* permitted to register. The administrator can command the system to re-evaluate these include and exclude lists. A present member of the cluster that becomes excluded is marked for decommissioning. Once a DataNode is marked as decommissioning, it will not be selected as the target of replica placement, but it will continue to serve read requests. The NameNode starts to schedule replication of its blocks to other DataNodes. Once the NameNode detects that all blocks on the decommissioning DataNode are replicated, the node enters the decommissioned state. Then it can be safely removed from the cluster without jeopardizing any data availability.

G. *Inter-Cluster Data Copy*

When working with large datasets, copying data into and out of a HDFS cluster is daunting. HDFS provides a tool called DistCp for large inter/intra-cluster parallel copying. It is a MapReduce job; each of the map tasks copies a portion of the source data into the destination file system. The MapReduce framework automatically handles parallel task scheduling, error detection and recovery.

IV. PRACTICE AT YAHOO!

Large HDFS clusters at Yahoo! include about 3500 nodes. A typical cluster node has:

- 2 quad core Xeon processors @ 2.5ghz
- Red Hat Enterprise Linux Server Release 5.1
- Sun Java JDK 1.6.0_13-b03
- 4 directly attached SATA drives (one terabyte each)
- 16G RAM
- 1-gigabit Ethernet

Seventy percent of the disk space is allocated to HDFS. The remainder is reserved for the operating system (Red Hat Linux), logs, and space to spill the output of map tasks. (MapReduce intermediate data are not stored in HDFS.) Forty nodes in a single rack share an IP switch. The rack switches are connected to each of eight core switches. The core switches provide connectivity between racks and to out-of-cluster resources. For each cluster, the NameNode and the BackupNode hosts are specially provisioned with up to 64GB RAM; application tasks are never assigned to those hosts. In total, a cluster of 3500 nodes has 9.8 PB of storage available as blocks that are replicated three times yielding a net 3.3 PB of storage for user applications. As a convenient approximation, one thousand nodes represent one PB of application storage. Over the years that HDFS has been in use (and into the future), the hosts selected as cluster nodes benefit from improved technologies. New cluster nodes always have faster processors, bigger disks and larger RAM. Slower, smaller nodes are retired or relegated to clusters reserved for development and testing of Hadoop. The choice of how to provision a cluster node is largely an issue of economically purchasing computation and storage. HDFS does not compel a particular ratio of computation to storage, or set a limit on the amount of storage attached to a cluster node.

On an example large cluster (3500 nodes), there are about 60 million files. Those files have 63 million blocks. As each

block typically is replicated three times, every data node hosts 54 000 block replicas. Each day user applications will create two million new files on the cluster. The 25 000 nodes in Hadoop clusters at Yahoo! provide 25 PB of on-line data storage. At the start of 2010, this is a modest—but growing—fraction of the data processing infrastructure at Yahoo!. Yahoo! began to investigate MapReduce programming with a distributed file system in 2004. The Apache Hadoop project was founded in 2006. By the end of that year, Yahoo! had adopted Hadoop for internal use and had a 300-node cluster for development. Since then HDFS has become integral to the back office at Yahoo!. The flagship application for HDFS has been the production of the Web Map, an index of the World Wide Web that is a critical component of search (75 hours elapsed time, 500 terabytes of MapReduce intermediate data, 300 terabytes total output). More applications are moving to Hadoop, especially those that analyze and model user behavior.

Becoming a key component of Yahoo!'s technology suite meant tackling technical problems that are the difference between being a research project and being the custodian of many petabytes of corporate data. Foremost are issues of robustness and durability of data. But also important are economical performance, provisions for resource sharing among members of the user community, and ease of administration by the system operators.

A. Durability of Data

Replication of data three times is a robust guard against loss of data due to uncorrelated node failures. It is unlikely Yahoo! has ever lost a block in this way; for a large cluster, the probability of losing a block during one year is less than .005. The key understanding is that about 0.8 percent of nodes fail each month. (Even if the node is eventually recovered, no effort is taken to recover data it may have hosted.) So for the sample large cluster as described above, a node or two is lost each day. That same cluster will re-create the 54 000 block replicas hosted on a failed node in about two minutes. (Re-replication is fast because it is a parallel problem that scales with the size of the cluster.) The probability of several nodes failing within two minutes such that all replicas of some block are lost is indeed small.

Correlated failure of nodes is a different threat. The most commonly observed fault in this regard is the failure of a rack or core switch. HDFS can tolerate losing a rack switch (each block has a replica on some other rack). Some failures of a core switch can effectively disconnect a slice of the cluster from multiple racks, in which case it is probable that some blocks will become unavailable. In either case, repairing the switch restores unavailable replicas to the cluster. Another kind of correlated failure is the accidental or deliberate loss of electrical power to the cluster. If the loss of power spans racks, it is likely that some blocks will become unavailable. But restoring power may not be a remedy because one-half to one percent of the nodes will not survive a full power-on restart. Statistically, and in practice, a large cluster will lose a handful of blocks during a power-on restart. (The strategy of deliberately restarting one node at a time over a period of weeks to identify nodes that will not survive a restart has not been tested.)

In addition to total failures of nodes, stored data can be corrupted or lost. The block scanner scans all blocks in a large cluster each fortnight and finds about 20 bad replicas in the process.

B. Caring for the Commons

As the use of HDFS has grown, the file system itself has had to introduce means to share the resource within a large and diverse user community. The first such feature was a permissions framework closely modeled on the Unix permissions scheme for file and directories. In this framework, files and directories have separate access permissions for the owner, for other members of the user group associated with the file or directory, and for all other users. The principle differences between Unix (POSIX) and HDFS are that ordinary files in HDFS have neither "execute" permissions nor "sticky" bits.

In the present permissions framework, user identity is weak: you are who your host says you are. When accessing HDFS, the application client simply queries the local operating system for user identity and group membership. A stronger identity model is under development. In the new framework, the application client must present to the name system credentials obtained from a trusted source. Different credential administrations are possible; the initial implementation will use Kerberos. The user application can use the same framework to confirm that the name system also has a trustworthy identity. And the name system also can demand credentials from each of the data nodes participating in the cluster.

The total space available for data storage is set by the number of data nodes and the storage provisioned for each node. Early experience with HDFS demonstrated a need for some means to enforce the resource allocation policy across user communities. Not only must fairness of sharing be enforced, but when a user application might involve thousands of hosts writing data, protection against application inadvertently exhausting resources is also important. For HDFS, because the system metadata are always in RAM, the size of the namespace (number of files and directories) is also a finite resource. To manage storage and namespace resources, each directory may be assigned a quota for the total space occupied by files in the sub-tree of the namespace beginning at that directory. A separate quota may also be set for the total number of files and directories in the sub-tree.

While the architecture of HDFS presumes most applications will stream large data sets as input, the MapReduce programming framework can have a tendency to generate many small output files (one from each reduce task) further stressing the namespace resource. As a convenience, a directory sub-tree can be collapsed into a single Hadoop Archive file. A HAR file is similar to a familiar tar, JAR, or Zip file, but file system operation can address the individual files for the archive, and a HAR file can be used transparently as the input to a MapReduce job.

C. Benchmarks

A design goal of HDFS is to provide very high I/O bandwidth for large data sets. There are three kinds of measurements that test that goal.

- What is bandwidth observed from a contrived benchmark?
- What bandwidth is observed in a production cluster with a mix of user jobs?
- What bandwidth can be obtained by the most carefully constructed large-scale user application?

The statistics reported here were obtained from clusters of at least 3500 nodes. At this scale, total bandwidth is linear with the number of nodes, and so the interesting statistic is the bandwidth *per node*. These benchmarks are available as part of the Hadoop codebase.

The DFSIO benchmark measures average throughput for read, write and append operations. DFSIO is an application available as part of the Hadoop distribution. This MapReduce program reads/writes/appends random data from/to large files. Each map task within the job executes the same operation on a distinct file, transfers the same amount of data, and reports its transfer rate to the single reduce task. The reduce task then summarizes the measurements. The test is run without contention from other applications, and the number of map tasks is chosen to be proportional to the cluster size. It is designed to measure performance only during data transfer, and excludes the overheads of task scheduling, startup, and the reduce task.

- DFSIO Read: 66 MB /s per node
- DFSIO Write: 40 MB /s per node

For a production cluster, the number of bytes read and written is reported to a metrics collection system. These averages are taken over a few weeks and represent the utilization of the cluster by jobs from hundreds of individual users. On average each node was occupied by one or two application tasks at any moment (fewer than the number of processor cores available).

- Busy Cluster Read: 1.02 MB/s per node
- Busy Cluster Write: 1.09 MB/s per node

Bytes (TB)	**Nodes**	**Maps**	**Reduces**	**Time**	**HDFS I/0 Bytes/s**	
					Aggregate (GB)	**Per Node** (MB)
1	1460	8000	2700	62 s	32	22.1
1000	3658	80 000	20 000	58 500 s	34.2	9.35

Table 2. Sort benchmark for one terabyte and one petabyte of data. Each data record is 100 bytes with a 10-byte key. The test program is a general sorting procedure that is not specialized for the record size. In the terabyte sort, the block replication factor was set to one, a modest advantage for a short test. In the petabyte sort, the replication factor was set to two so that the test would confidently complete in case of a (not unexpected) node failure.

At the beginning of 2009, Yahoo! participated in the Gray Sort competition [9]. The nature of this task stresses the system's ability to move data from and to the file system (it really isn't about sorting). The competitive aspect means that the results in Table 2 are about the best a user application can achieve with the current design and hardware. The I/O rate in the last column is the combination of reading the input and writing the output from and to HDFS. In the second row, while the rate for HDFS is reduced, the total I/O per node will be about double because for the larger (petabyte!) data set, the MapReduce intermediates must also be written to and read from disk. In the smaller test, there is no need to spill the MapReduce intermediates to disk; they are buffered the memory of the tasks.

Large clusters require that the HDFS NameNode support the number of client operations expected in a large cluster. The NNThroughput benchmark is a single node process which starts the NameNode application and runs a series of client threads on the same node. Each client thread performs the same NameNode operation repeatedly by directly calling the NameNode method implementing this operation. The benchmark measures the number of operations per second performed by the NameNode. The benchmark is designed to avoid communication overhead caused by RPC connections and serialization, and therefore runs clients locally rather than remotely from different nodes. This provides the upper bound of pure NameNode performance.

Operation	**Throughput (ops/s)**
Open file for read	126 100
Create file	5600
Rename file	8300
Delete file	20 700
DataNode Heartbeat	300 000
Blocks report (blocks/s)	639 700

Table 3. NNThroughput benchmark

V. FUTURE WORK

This section presents some of the future work that the Hadoop team at Yahoo is considering; Hadoop being an open source project implies that new features and changes are decided by the Hadoop development community at large.

The Hadoop cluster is effectively unavailable when its NameNode is down. Given that Hadoop is used primarily as a batch system, restarting the NameNode has been a satisfactory recovery means. However, we have taken steps towards automated failover. Currently a BackupNode receives all transactions from the primary NameNode. This will allow a failover to a warm or even a hot BackupNode if we send block reports to both the primary NameNode and BackupNode. A few Hadoop users outside Yahoo! have experimented with manual failover. Our plan is to use Zookeeper, Yahoo's distributed consensus technology to build an automated failover solution.

Scalability of the NameNode [13] has been a key struggle. Because the NameNode keeps all the namespace and block locations in memory, the size of the NameNode heap has limited the number of files and also the number of blocks addressable. The main challenge with the NameNode has been that when its memory usage is close to the maximum the NameNode becomes unresponsive due to Java garbage collection and sometimes requires a restart. While we have encour-

aged our users to create larger files, this has not happened since it would require changes in application behavior. We have added quotas to manage the usage and have provided an archive tool. However these do not fundamentally address the scalability problem.

Our near-term solution to scalability is to allow multiple namespaces (and NameNodes) to share the physical storage within a cluster. We are extending our block IDs to be prefixed by *block pool* identifiers. Block pools are analogous to LUNs in a SAN storage system and a namespace with its pool of blocks is analogous as a file system volume.

This approach is fairly simple and requires minimal changes to the system. It offers a number of advantages besides scalability: it isolates namespaces of different sets of applications and improves the overall availability of the cluster. It also generalizes the block storage abstraction to allow other services to use the block storage service with perhaps a different namespace structure. We plan to explore other approaches to scaling such as storing only partial namespace in memory and truly distributed implementation of the NameNode in the future. In particular, our assumption that applications will create a small number of large files was flawed. As noted earlier, changing application behavior is hard. Furthermore, we are seeing new classes of applications for HDFS that need to store a large number of smaller files.

The main drawback of multiple independent namespaces is the cost of managing them, especially if the number of namespaces is large. We are also planning to use application or job centric namespaces rather than cluster centric namespaces—this is analogous to the per-process namespaces that are used to deal with remote execution in distributed systems in the late 80s and early 90s [10][11][12].

Currently our clusters are less than 4000 nodes. We believe we can scale to much larger clusters with the solutions outlined above. However, we believe it is prudent to have multiple clusters rather than a single large cluster (say three 6000-node clusters rather than a single 18 000-node cluster) as it allows much improved availability and isolation. To that end we are planning to provide greater cooperation between clusters. For example caching remotely accessed files or reducing the replication factor of blocks when files sets are replicated across clusters.

VI. Acknowledgment

We would like to thank all members of the HDFS team at Yahoo! present and past for their hard work building the file system. We would like to thank all Hadoop committers and collaborators for their valuable contributions. Corinne Chandel drew illustrations for this paper.

References

[1] Apache Hadoop. http://hadoop.apache.org/

[2] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. "PVFS: A parallel file system for Linux clusters," in Proc. of 4th Annual Linux Showcase and Conference, 2000, pp. 317–327.

[3] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," In Proc. of the 6th Symposium on Operating Systems Design and Implementation, San Francisco CA, Dec. 2004.

[4] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, U. Srivastava. "Building a High-Level Dataflow System on top of MapReduce: The Pig Experience," In Proc. of Very Large Data Bases, vol 2 no. 2, 2009, pp. 1414–1425

[5] S. Ghemawat, H. Gobioff, S. Leung. "The Google file system," In Proc. of ACM Symposium on Operating Systems Principles, Lake George, NY, Oct 2003, pp 29–43.

[6] F. P. Junqueira, B. C. Reed. "The life and times of a zookeeper," In Proc. of the 28th ACM Symposium on Principles of Distributed Computing, Calgary, AB, Canada, August 10–12, 2009.

[7] Lustre File System. http://www.lustre.org

[8] M. K. McKusick, S. Quinlan. "GFS: Evolution on Fast-forward," ACM Queue, vol. 7, no. 7, New York, NY. August 2009.

[9] O. O'Malley, A. C. Murthy. Hadoop Sorts a Petabyte in 16.25 Hours and a Terabyte in 62 Seconds. May 2009. http://developer.yahoo.net/blogs/hadoop/2009/05/hadoop_sorts_a_petabyte_in_162.html

[10] R. Pike, D. Presotto, K. Thompson, H. Trickey, P. Winterbottom, "Use of Name Spaces in Plan9," Operating Systems Review, 27(2), April 1993, pages 72–76.

[11] S. Radia, "Naming Policies in the spring system," In Proc. of 1st IEEE Workshop on Services in Distributed and Networked Environments, June 1994, pp. 164–171.

[12] S. Radia, J. Pachl, "The Per-Process View of Naming and Remote Execution," IEEE Parallel and Distributed Technology, vol. 1, no. 3, August 1993, pp. 71–80.

[13] K. V. Shvachko, "HDFS Scalability: The limits to growth," ;login:. April 2010, pp. 6–16.

[14] W. Tantisiriroj, S. Patil, G. Gibson. "Data-intensive file systems for Internet services: A rose by any other name ..." Technical Report CMU-PDL-08-114, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, October 2008.

[15] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, "Hive – A Warehousing Solution Over a Map-Reduce Framework," In Proc. of Very Large Data Bases, vol. 2 no. 2, August 2009, pp. 1626-1629.

[16] J. Venner, Pro Hadoop. Apress, June 22, 2009.

[17] S. Weil, S. Brandt, E. Miller, D. Long, C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," In Proc. of the 7th Symposium on Operating Systems Design and Implementation, Seattle, WA, November 2006.

[18] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, B. Zhou, "Scalable Performance of the Panasas Parallel file System", In Proc. of the 6th USENIX Conference on File and Storage Technologies, San Jose, CA, February 2008

[19] T. White, Hadoop: The Definitive Guide. O'Reilly Media, Yahoo! Press, June 5, 2009.