The Hadoop Distributed File System

The Architecture of
Open Source Applications
Elegance, Evolution, and a Few Fearless Hacks

Robert Chansler, Hairong Kuang, Sanjay Radia, Konstantin Shvachko, and Suresh Srinivas

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 40 petabytes of enterprise data at Yahoo!

8.1. Introduction

Hadoop[1] provides a distributed filesystem and a framework for the analysis and transformation of very large data sets using the MapReduce [DG04] paradigm. While the interface to HDFS is patterned after the Unix filesystem, faithfulness to standards was sacrificed in favor of improved performance for the applications at hand.

An important characteristic of Hadoop is the partitioning of data and computation across many (thousands) of hosts, and the execution of application computations in parallel close to their data. A Hadoop cluster scales computation capacity, storage capacity and I/O bandwidth by simply adding commodity servers. Hadoop clusters at Yahoo! span 40,000 servers, and store 40 petabytes of application data, with the largest cluster being 4000 servers. One hundred other organizations worldwide report using Hadoop.

HDFS stores filesystem metadata and application data separately. As in other distributed filesystems, like PVFS [CIRT00], Lustre[2], and GFS [GGL03], 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 rely on 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.

8.2. Architecture

8.2.1. NameNode

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by inodes. Inodes 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 blocks to DataNodes. 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.

8.2.2. Image and Journal

The inodes and the list of blocks that define the metadata of the name system are called the *image*. NameNode keeps the entire namespace image in RAM. The persistent record of the image stored in the NameNode's local native filesystem is called a checkpoint. The NameNode records changes to HDFS in a write-ahead log called the journal in its local native filesystem. The location of block replicas are not part of the persistent checkpoint.

Each client-initiated transaction is recorded in the journal, and the journal file is flushed and synced before the acknowledgment is sent to the client. The checkpoint file is never changed by the NameNode; a new file is written when a 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. A new checkpoint and an empty journal are written back to the storage directories before the NameNode starts serving clients.

For improved durability, redundant copies of the checkpoint and journal are typically stored on multiple independent local volumes and at remote NFS servers. The first choice prevents loss from a single volume failure, 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. When one of the NameNode's threads initiates a flush-and-sync operation, all the 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.

8.2.3. DataNodes

Each block replica on a DataNode is represented by two files in the local native filesystem. The first file contains the data itself and the second file records the block's metadata including checksums for the data and the 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 filesystems. 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 filesystem 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 protecting the integrity of the filesystem. 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 block allocation and load balancing decisions.

The NameNode does not directly send requests to 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 and send an immediate block report, and shut down the node.

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.

8.2.4. HDFS Client

User applications access the filesystem using the HDFS client, a library that exports the HDFS filesystem interface.

Like most conventional filesystems, 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 does not need to know that filesystem 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. The list is sorted by the network topology distance from the client. The client 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 client sends the further bytes of the file. Choice of DataNodes for each block is likely to be different. The interactions among the client, the NameNode and the DataNodes are illustrated in Figure 8.1.

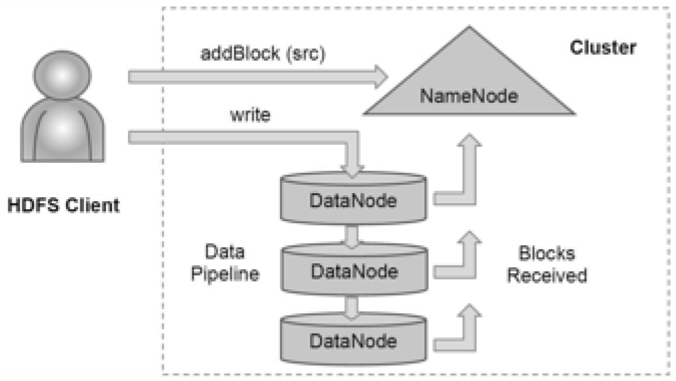

Figure 8.1: HDFS Client Creates a New File

Unlike conventional filesystems, 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 tolerance against faults and increases read bandwidth.

8.2.5. 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 filesystem 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 also lets the NameNode truncate 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.

8.2.6. 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 filesystem 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 in journal on 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 way 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 filesystem 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 of persisting the namespace state to the BackupNode.

8.2.7. Upgrades and Filesystem Snapshots

During software upgrades the possibility of corrupting the filesystem 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 filesystem, 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 directories containing the data files as this would 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; for snapshots created during upgrade, this finalizes 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.

8.3. File I/O Operations and Replica Management

Of course, the whole point of a filesystem is to store data in files. To understand how HDFS does this, we must look at how reading and writing works, and how blocks are managed.

8.3.1. 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 acknowledgment 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.

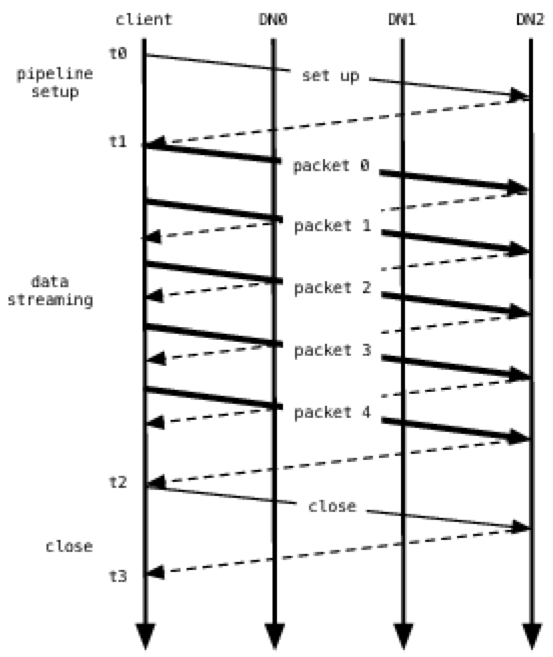

Figure 8.2: Data Pipeline While Writing a Block

If no error occurs, block construction goes through three stages as shown in Figure 8.2 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 `t0` to `t1` is the pipeline setup stage. The interval `t1` to `t2` is the data streaming stage, where `t1` is the time when the first data packet gets sent and `t2` is the time that the acknowledgment to the last packet gets received. Here an hflush operation transmits `packet 2`. The hflush indication travels with the packet data and is not a separate operation. The final interval `t2` to `t3` 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. Ongoing efforts will improve read/write response time for applications that require real-time data streaming or random access.

8.3.2. 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. Figure 8.3 describes a cluster with two racks, each of which contains three nodes.

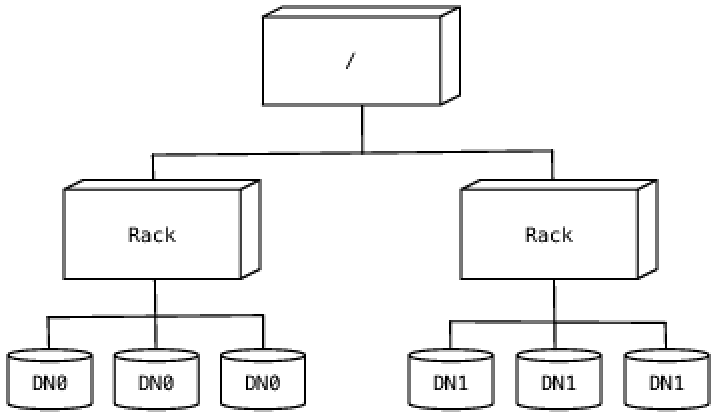

Figure 8.3: Cluster Topology

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 the distances to their closest common ancestor. A shorter distance between two nodes means greater bandwidth they can use 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 the 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 alternate policies that are 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 are placed on two different nodes in a different rack. The rest are placed on random nodes with restrictions that no more than one replica is placed at any one node and no more than two replicas are placed in the same rack, if possible. 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.

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.

8.3.3. 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 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 mis-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.

8.3.4. 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 with a lot of free storage. 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 between 0 and 1. A cluster is balanced if, for each DataNode, the utilization of the node[3] differs from the utilization of the whole cluster[4] 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 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.

8.3.5. 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 the top-level DataNode directory, the current and previous logs. New verification times are appended to the 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.

8.3.6. Decommissioning

The cluster administrator specifies list of nodes to be decommissioned. Once a DataNode is marked for 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.

8.3.7. 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 filesystem. The MapReduce framework automatically handles parallel task scheduling, error detection and recovery.

8.4. Practice at Yahoo!

Large HDFS clusters at Yahoo! include about 4000 nodes. A typical cluster node has two quad core Xeon processors running at 2.5 GHz, 4–12 directly attached SATA drives (holding two terabytes each), 24 Gbyte of RAM, and a 1-gigabit Ethernet connection. 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 64 GB RAM; application tasks are never assigned to those hosts. In total, a cluster of 4000 nodes has 11 PB (petabytes; 1000 terabytes) of storage available as blocks that are replicated three times yielding a net 3.7 PB of storage for user applications. Over the years that HDFS has been in use, the hosts selected as cluster nodes have benefited 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.

On an example large cluster (4000 nodes), there are about 65 million files and 80 million blocks. As each block typically is replicated three times, every data node hosts 60 000 block replicas. Each day, user applications will create two million new files on the cluster. The 40 000 nodes in Hadoop clusters at Yahoo! provide 40 PB of on-line data storage.

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.

8.4.1. 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 0.005. 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 60 000 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.

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. Bad replicas are replaced as they are discovered.

8.4.2. Features for Sharing HDFS

As the use of HDFS has grown, the filesystem itself has had to introduce means to share the resource among a large number of diverse users. 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 earlier version of HDFS, user identity was weak: you were who your host said you are. When accessing HDFS, the application client simply queries the local operating system for user identity and group membership. 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 uses 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 applications 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 filesystem operations can address the individual files within the archive, and a HAR file can be used transparently as the input to a MapReduce job.

8.4.3. Scaling and HDFS Federation

Scalability of the NameNode has been a key struggle [Shv10]. Because the NameNode keeps all the namespace and block locations in memory, the size of the NameNode heap limits the number of files and also the number of blocks addressable. This also limits the total cluster storage that can be supported by the NameNode. Users are encouraged to create larger files, but this has not happened since it would require changes in application behavior. Furthermore, we are seeing new classes of applications for HDFS that need to store a large number of small files. Quotas were added to manage the usage, and an archive tool has been provided, but these do not fundamentally address the scalability problem.

A new feature allows multiple independent namespaces (and NameNodes) to share the physical storage within a cluster. Namespaces use blocks grouped under a Block Pool. Block pools are analogous to logical units (LUNs) in a SAN storage system and a namespace with its pool of blocks is analogous to a filesystem volume.

This approach offers a number of advantages besides scalability: it can isolate namespaces of different applications improving the overall availability of the cluster. Block pool abstraction allows other services to use the block storage 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.

Applications prefer to continue using a single namespace. Namespaces can be mounted to create such a unified view. A client-side mount table provide an efficient way to do that, compared to a server-side mount table: it avoids an RPC to the central mount table and is also tolerant of its failure. The simplest approach is to have shared cluster-wide namespace; this can be achieved by giving the same client-side mount table to each client of the cluster. Client-side mount tables also allow applications to create a private namespace view. This is analogous to the per-process namespaces that are used to deal with remote execution in distributed systems [PPT+93, Rad94, RP93].

8.5. Lessons Learned

A very small team was able to build the Hadoop filesystem and make it stable and robust enough to use it in production. A large part of the success was due to the very simple architecture: replicated blocks, periodic block reports and central metadata server. Avoiding the full POSIX semantics also helped. Although keeping the entire metadata in memory limited the scalability of the namespace, it made the NameNode very simple: it avoids the complex locking of typical filesystems. The other reason for Hadoop's success was to quickly use the system for production at Yahoo!, as it was rapidly and incrementally improved. The filesystem is very robust and the NameNode rarely fails; indeed most of the down time is due to software upgrades. Only recently have failover solutions (albeit manual) emerged

Many have been surprised by the choice of Java in building a scalable filesystem. While Java posed challenges for scaling the NameNode due to its object memory overhead and garbage collection, Java has been responsible to the robustness of the system; it has avoided corruption due to pointer or memory management bugs.

8.6. Acknowledgment

We thank Yahoo! for investing in Hadoop and continuing to make it available as open source; 80% of the HDFS and MapReduce code was developed at Yahoo! We thank all Hadoop committers and collaborators for their valuable contributions.

Footnotes

1. `http://hadoop.apache.org`
2. `http://www.lustre.org`
3. Defined as the ratio of used space at the node to total capacity of the node.
4. Defined as the ratio of used space in the cluster to total capacity of the cluster.