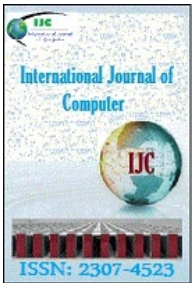
International Journal of Computer (IJC)

ISSN 2307-4531

http://gssrr.org/index.php?journal=InternationalJournalOfComputer&page=index



The Hadoop Distributed File System

[a*]Anmol Kumar Tank , [b]Amit Kumar Parmar , [c]Prof.Rajesh Srivastav

[a]M.TECH* Srgi,Jabalpur, Department of Computer science & engg.

[b]M.TECH* SSSCE,bhopal, Department of Computer science & engg.

[c]M.TECH SRGI & HOD CSE DEPT, Department of Computer science & engg.

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

1. 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.

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 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 Name Node. Application data are stored on other servers called DataNodes. All servers are fully connected and communicate with each other using TCP-based protocols.

---

* Corresponding author.
E-mail address: anmol14nov@gmail.com.

Table 1. Hadoop project components

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

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.

2. Architecture

A. Name Node

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the Name Node 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 (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 Data Nodes in a pipeline fashion. The current design has a single Name Node for each cluster. The cluster can have thousands of Data Nodes and tens of thousands of HDFS clients per cluster, as each Data Node 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. Data Nodes

Each block replica on a Data Node 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 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 Data Node *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-todate 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 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.

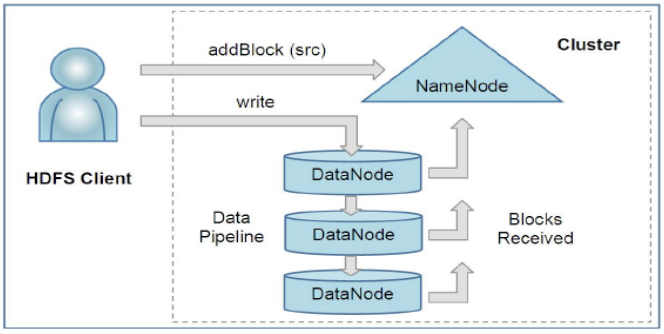


Figure 1. An HDFS client creates a new file by giving its path to the NameNode. For each block of the file, the NameNode returns

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.

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. 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 inmemory, 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 ownstorage 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.

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. HDFS does not separate layout versions for the NameNode and DataNodes because snapshot creation must be an allcluster 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

3. File I/O operations and replica management

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.

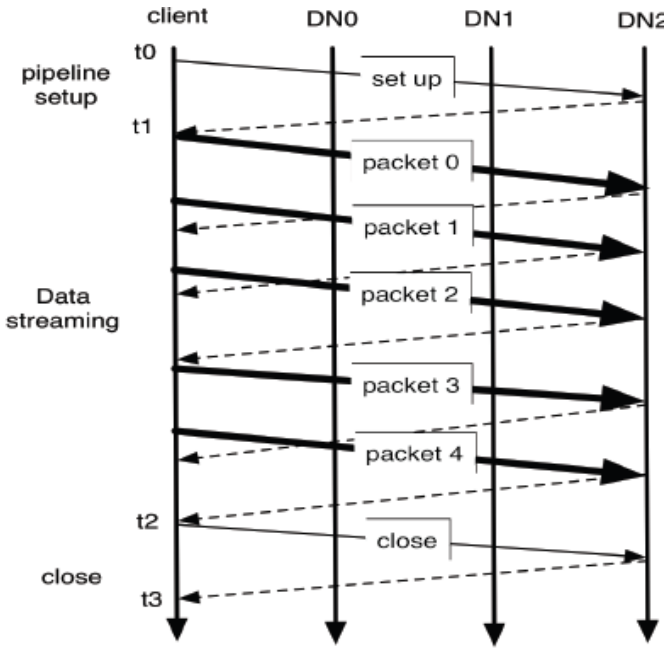


Figure 2. Data pipeline during block construction

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, realtime 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.

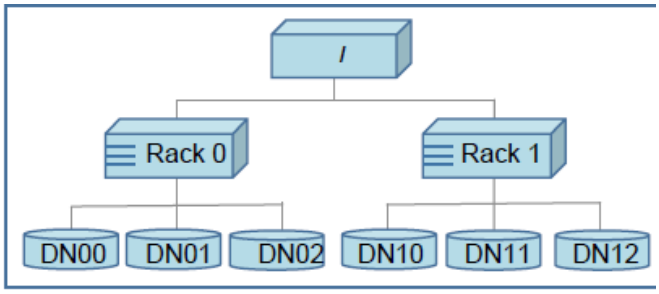


Figure 3. Cluster topology example

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.

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.

D. *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

4. Conclusion and Future Work

Nobody (yet!) ever got fired for using a Hadoop cluster! But memory is becoming cheap, by historical trends very cheap, and this is potentially very disruptive. Single big-memory servers may simply be more efficient than clusters, which can substantially change price points and complexity. In this paper we have described the trends, and conjectured that many jobs simply do not need large-scale clusters to run. We have demonstrated this with a machine learning algorithm, where complexity and cost can be lowered by using a single server solution.

We have shown that AdPredictor could scale using a server with big memory rather than clusters. We have also experimented extensively with a second algorithm: Frequent Item set Mining (FIM) [2]. This is used to determine sets of items that occur together frequently, e.g., in shopping baskets. This also has a version that is designed to support parallel or distributed implementation called SON [9], which also was suitably complex that the parallel version of the algorithm was published. The distributed version predates MapReduce, but is well suited to implementation on a MapReduce-like framework, and we have implemented on all the same platforms as Ad Predictor. We omit the results here for space reasons, but they support similar conclusions as for Ad Predictor. Are these results applicable more generally? We believe that DRAM sizes are at a tipping point for human generated data. Examples of such data are social (e.g. Twitter, Four Square) and shopping baskets. The size of such data is fundamentally limited by the number of people on the planet, which (fortunately) does not double every 18 months. Core counts and DRAM sizes per server by contrast, are still on a Moore's Law trajectory. Back of the envelope calculations convince us that a 512GB server could process all the items purchased at a major UK food retailer in the last year or could handle a GPS location per day per person in the UK, for an entire year.

ACKNOWLEDGMENT

We thanks to prof. rajesh srivastav for patiently teaching us about factor graphs, expectation propagation AdPredictor and providing data and for sharing a draft version of his paper with us.

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.

[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.

[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] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-scale bayesian click-through rate prediction for sponsored search advertising in microsofts bing search engine. In 27th ICML, June 2010.

[12] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In VLDB, pages 432–444, 1995.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI, 2004.

[14] K. Elmeleegy. Piranha: Optimizing short jobs in hadoop.