

HDFS Reliability

Tom White, Cloudera, 12 January 2008

The Hadoop Distributed Filesystem (HDFS) is a distributed storage system for reliably storing petabytes of data on clusters of commodity hardware. This short paper examines the reliability of HDFS and makes recommendations for best practices to follow when running an HDFS installation.

Overview of HDFS

HDFS has three classes of node:

- a single *name node*, responsible for managing the filesystem metadata
- a single *secondary name node*, responsible for checkpointing the name node's persistent state
- many *data nodes*, which are responsible for storing the file data

HDFS stores files as a series of blocks, each of which is by default 64MB in size. A block is the unit of storage for data nodes: they store and retrieve blocks, and have no concept of the files that are composed of these blocks. Blocks are stored on the underlying filesystem of the data node, as opposed to the data node managing their own storage, as a kernel-level filesystem would do. The name node holds that mapping from files to blocks, which it stores in memory as well as in a persistent metadata store on disk (in the image file and edit log). The mapping between blocks and the data nodes they reside on is *not* stored persistently. Instead, it is stored in the name node's memory, and is built up from the periodic block reports that data nodes send to the name node. One of the first things that a data node does on start up is send a block report to the name node, and this allows the name node to rapidly form a picture of the block distribution across the cluster.

Functioning data nodes send heartbeats to the name node every 3 seconds. This mechanism forms the communication channel between data node and name node: occasionally, the name node will piggyback a command to a data node on the heartbeat response. An example of a command might be "send a copy of block *b* to data node *d*".

Block replicas

Block replicas (referred to as "replicas" in this document) are written to distinct data nodes (to cope with data node failure), or distinct racks (to cope with rack failure). The *rack placement policy*¹ is managed by the name node, and replicas are placed as follows:

1. The first replica is placed on a random node in the cluster, unless the write originates from within the cluster, in which case it goes to the local node.
2. The second replica is written to a different rack from the first, chosen at random.
3. The third replica is written to the same rack as the second replica, but on a different node.

1. Currently the policy is fixed, however there is a proposal to make it pluggable. See <https://issues.apache.org/jira/browse/HADOOP-3799>

4. Fourth and subsequent replicas are placed on random nodes, although racks with many replicas are biased against, so replicas are spread out across the cluster.

When files are being written the data nodes form a pipeline to write the replicas in sequence. Data is sent through the pipeline in packets (smaller than a block), each of which must be acknowledged to count as a successful write. If a data node fails while the block is being written, it is removed from the pipeline. When the current block has been written, the name node will re-replicate it to make up for the missing replica due to the failed data node. Subsequent blocks will be written using a new pipeline with the required number of data nodes.

Clients

Filesystem clients communicate with the name node to retrieve metadata (such as information about the directory hierarchy, or the mapping from a file to its blocks), and with the data nodes to retrieve data. It is important that clients retrieve the file data direct from the data nodes rather than having it routed via the name node, as the latter would create a bottleneck at the name node and severely limit the aggregate bandwidth available to clients.

Secondary Name Node

The role of the secondary name node has caused considerable confusion for Hadoop users, since it has a misleading name. It is not a backup name node in the sense that it can take over the primary name node's function, but rather a checkpointing mechanism. During operation the name node maintains two on-disk data structures to represent the filesystem state: an image file and an edit log. The image file is a checkpoint of the filesystem metadata at a point in time, and the edit log is a transactional redo log of every filesystem metadata mutation since the image file was created. Incoming changes to the filesystem metadata (such as creating a new file) are written to the edit log². When the name node starts, it reconstructs the current state by replaying the edit log. To ensure that the log doesn't grow without bound, at periodic intervals the edit log is rolled, and a new checkpoint is created by applying the old edit log to the image. This process is performed by the secondary name node daemon, often on a separate machine to the primary since creating a checkpoint has similar memory requirements to the name node itself. A side effect of the checkpointing mechanism is that the secondary holds an out-of-date copy of the primary's persistent state, which, *in extremis*, can be used to recover the filesystem's state.

There is ongoing work to create a true backup name node, which would be a failover name node in the event of the primary failing³.

Safe mode

When the name node starts it enters a state where the filesystem is read only, and no blocks are replicated or deleted. This is called "safe mode". Safe mode is needed to allow the name node to do two things:

-
2. Edit log writes are flushed and synced to disk before a successful return code is returned to the application. Work is ongoing
 3. "Streaming Edits to a Standby Name-Node", <https://issues.apache.org/jira/browse/HADOOP-4539>

1. Reconstruct the state of the filesystem by loading the image file into memory and replaying the edit log.
2. Generate the mapping between blocks and data nodes by waiting for enough of the data nodes to check in.

If the name node didn't wait for the data nodes to check in, it would think that blocks were under-replicated and start re-replicating blocks across the cluster. Instead, the name node waits until enough data nodes check in to account for a configurable percentage of blocks (99.9% by default), which satisfy the minimum replication level (1 by default). The name node then waits a further fixed amount of time (30 seconds by default) to allow the cluster to settle down before exiting safe mode.

An administrator can make the name node enter or leave safe mode manually using Hadoop's `dfsadmin` command. This is useful when performing upgrades (see "Define backup and upgrade procedures" below), or diagnosing problems on the cluster. It is also possible to start the cluster in such a way that it will never leave safe mode automatically, by setting the percentage of blocks that meet the minimal replication requirement to over 100%.

Tools

distcp⁴ is a tool for performing a distributed copy of data within an HDFS cluster, or between HDFS clusters. It is implemented using MapReduce (so it needs a MapReduce cluster to be overlaid on the HDFS cluster), and therefore is very efficient since it can copy files in parallel. It is also possible to copy data between HDFS clusters running different versions of Hadoop, by using the *hftp* filesystem, which permits accessing HDFS over HTTP.

Over time the block distribution on an HDFS cluster can become unbalanced. The **balancer**⁵ tool can be run as a background process to re-distribute blocks across the cluster until the deviation from the average is below a certain threshold (default 10%). This tool is especially useful when adding new data nodes to a cluster, since HDFS does not automatically re-distribute blocks in this case.

Snapshots

Work is ongoing to support HDFS snapshots⁶. A snapshot is the state of the filesystem at a point in time. From the point of view of HDFS reliability, snapshots will enable incremental backup of a cluster.

Types of failure

We need to be precise about the types of failure that we are discussing. Data loss can occur for the following reasons:

1. Hardware failure or malfunction. A failure of one or more hardware components causes data to be lost.
2. Software error. A bug in the software causes data to be lost.
3. Human error. For example, a human operator inadvertently deletes the whole filesystem by typing:

4. <http://hadoop.apache.org/core/docs/current/distcp.html>

5. http://hadoop.apache.org/core/docs/current/hdfs_user_guide.html#Rebalancer

6. <https://issues.apache.org/jira/browse/HADOOP-3637>

```
hadoop fs -rmr /
```

This paper is mostly concerned with the first two types of failure, but we briefly consider the third separately, later on in the paper.

Hardware failures

Hardware failure can manifest itself in many different ways. Two manifestations that are pertinent to this analysis are

1. The simultaneous failure of multiple hardware components
2. The corruption of data on disk

Both of these types of failure are guarded against using replication. The rack placement policy described above can tolerate the failure of $R-1$ arbitrary data nodes in the cluster (where R is the replication factor in the cluster), or even a whole rack. Similarly, if a replica is corrupt due to a local failure, then the other replicas can be used instead.

How does Hadoop detect hardware failures?

For example, consider the case of a data node failing. The name node would notice that the data node is not sending heartbeats, then after a certain time period (10 minutes by default) it considers the node as dead, at which point it will re-replicate the blocks that were on the failed data node using replicas stored on other nodes of the cluster.

Detecting corrupt data requires a different approach. The principal technique is to use checksums to check for corruption. Corruption may occur during transmission of the block over the network, or when it is written to or read from disk. In Hadoop, the data nodes verify checksums on receipt of the block. If any checksum is invalid the data node will complain and the block will be resent. A block's checksums are stored along with the block data, to allow further integrity checks.

This is not sufficient to ensure that the data will be successfully read from disk in an uncorrupted state, so all reads from HDFS verify the block checksums too. Failures are reported to the name node, which organizes re-replication of the healthy replicas.

Because HDFS is often used to store data that isn't read very often, detecting corrupt data when it is read is undesirable: the failure may go undetected for a long period, during which other replicas may have failed. To remedy this, each data node runs a background thread to check block integrity. If it finds a corrupt block, it informs the name node which replicates the block from its uncorrupted replicas, and arranges for the corrupt block to be deleted. Blocks are re-verified every three weeks to protect against disk errors over time. New blocks are checked first, but there is still some lag in checking them. (See "Employ monitoring" below for details on how to monitor block scans.)

Software errors

Bugs in the software will always be present to some degree, but, again, having replicated data guards against a large class of these problems. Of course, it is possible to have software bugs that affect all replicas, but these are very rare.

The strongest defence against this scenario is maintaining good software development practices, including

- Unit testing
- Code review (human and with tools such as FindBugs)
- System testing at scale

Hadoop employs all these practices. However, there is always room for improvement, and as Hadoop matures and is adopted by more organizations, there will be a push to improve in all of these areas.

Block truncation errors

A recent case of software errors concerns inadvertent block truncation⁷. A bug in HDFS was occasionally causing blocks to be partially written, so their length was less than expected. To compound the problem the data node would report to the name node that the full length of the block was written correctly. Furthermore, the block scan would not detect the truncation since the block metadata was consistent with the truncated block. This case of "silent corruption" of a block ultimately led to data loss when replicas of the block were lost (either because the same bug hit, or for other reasons). This scenario has occurred an HDFS cluster at the University of Nebraska, where 200-300 files were lost⁸. The problem was also experienced at Yahoo!, however it is not known how widespread this problem was.

At the time of writing, this bug is still being fixed, but it illustrates how in this case safeguards at a different layer of the system were not sufficient to even detect the problem until it was too late and data was lost. When bugs like this are found the bug itself needs fixing of course, but perhaps even more importantly the checks in the system need to be improved to guard against the class of condition that caused the failure.

Best Practices

Use a common configuration

Hadoop is a highly configurable system, which makes the configuration space large. Operating a cluster using an unusual configuration is a good way to find bugs, so to maximize reliability, you should use a configuration that has been tested by others, preferably at the scale you operate at.

For example, Yahoo! and Facebook are two organizations that operate Hadoop at sufficiently large scale to experience regular hardware failures. During the course of development of HDFS, there have been many bugs that were detected (and subsequently fixed) due to operating at scale⁹, so these are good candidates to mimic in terms of configuration.

7. <https://issues.apache.org/jira/browse/HADOOP-4692>

8. See Hadoop User mailing list posting: <http://www.mail-archive.com/core-user@hadoop.apache.org/msg06462.html>

9. For example, <http://issues.apache.org/jira/browse/HADOOP-572>. See also Nigel Daley's blog post about testing at scale "Nigel's Law" (http://weblogs.java.net/blog/nidaley/archive/2007/07/nigels_law.html).

In general, this practice means using the default settings where possible, although there are some settings which are commonly changed from the default (for example HDFS block size is set to 128MB by Yahoo!¹⁰).

Use three or more replicas

In theory a replication level of two is sufficient to recover from the failure of a single node, or even a whole rack (since the second replica of every block is on a different rack). The problem in practice is that the backup replica may be corrupt, or absent due to a software error.

To some extent the file loss at the University of Nebraska was due to operating a large cluster with a non-standard configuration (two replicas), which made it more susceptible to bugs. The crux of the problem is that if corrupt blocks are not detected in a timely manner, or if a software bug masks invalid blocks, then the system thinks there are more valid blocks than there actually are. With only one backup replica (replication factor of two), it is only when the backup fails that the system detects that there are no block replicas, at which point it is too late to recover data.

With three replicas the system has a chance to detect the silent corruption after the second replica fails, and to re-replicate using the third replica.

The number of replicas can be set on a per file basis, either at the time of creation or after closing the file. For important or widely used files the replication should be increased above three.

Protect the name node

The name node is a single point of failure: if it fails, then the whole HDFS cluster is unusable. If it is unrecoverable, then all of the data in the cluster is unrecoverable. To avoid this catastrophic scenario the name node should have special treatment:

1. The name node should write its persistent metadata to multiple local disks. If one physical disk fails then there is a backup of the data on another disk. RAID can be used in this case too.
2. The name node should write its persistent metadata to a remote NFS mount. If the name node fails, then there is a backup of the data on NFS.
3. The secondary name node should run on a separate node to the primary. In the case of losing all of the primary's data (local disks and NFS), the secondary can provide a stale copy of the metadata. Since it is stale, there will be some data loss, but it will be a known amount of data loss, since the secondary makes periodic backups of the metadata on a configurable schedule.
4. Make backups of the name node's persistent metadata. You should keep multiple copies of different ages (1 day, 1 week, 1 month) to allow recovery in the case of corruption. A convenient way to do this is to use the checkpoints on the secondary as the source of the backup. These backups should be verified; at present the only way to do this is to start a new name node (on a separate, unreachable network to the production cluster) to visually check that it can reconstruct the filesystem metadata.

10. There was a recent case reported where setting the block size to 4GB (which is many times larger than the recommended size), caused file access issues. <http://www.mail-archive.com/core-user@hadoop.apache.org/msg06689.html>

5. Use directory quotas to set a maximum number of files that may live in the filesystem namespace. This measure prevents the destabilizing effect of the name node running out of memory due to too many files being created in the system.

There have been no reported software errors in the name node's persistent data storage that have caused unrecoverable data loss.

Employ monitoring

Where possible, Hadoop strives to detect failure and work around it. However, this is no substitute for a monitoring system that informs operators of the health of the cluster, so they can take early action when needed. Hadoop exposes monitoring hooks via JMX, which allows the cluster to be monitored using tools like Nagios, or Ganglia.

There are also routine administration activities that the cluster operations team should undertake, including:

- Running HDFS's `fsck` (filesystem check) tool daily to get reports of file and block health
- Viewing data node block scanner reports at <http://datanode:50075/blockScannerReport>¹¹

Define backup and upgrade procedures

To make performance enhancements or bug fixes it is occasionally necessary for HDFS to change its on-disk layout for filesystem metadata and/or data. In these cases, extra care is needed when performing an upgrade of Hadoop, since there is potential for data loss due to software errors. There are several precautions that are recommended:

- Do a dry run on a small cluster.
- Document the upgrade procedure for your cluster. There are upgrade instructions on the Hadoop Wiki¹², but having a custom set of instructions for your particular set up, incorporating lessons learned from a dry run, is invaluable when it needs to be repeated in the future.
- Always make multiple off-site backups of the name node's metadata.
- If the on-disk data layout has changed (stored on the data node), consider making a backup of the cluster, or at least of the most important files on the cluster. While all data layout upgrades have a facility to rollback to a previous format version (by keeping a copy of the data in the old layout), making backups is always recommended if possible. Using the `distcp` tool over `hftp` to backup data to a second HDFS cluster is a good way to make backups.

Human error

While it is impossible to completely remove the risk of data loss due to human factors (since there will always be a requirement to be able to delete data - for whatever reason, legal etc.), there are a number of safeguards and working practices that can help minimize data loss.

11. There is a Jira issue to allow the block scanner to be run manually. See <https://issues.apache.org/jira/browse/HADOOP-4865>

12. <http://wiki.apache.org/hadoop/Hadoop%20Upgrade>

Trash facility

If trash is enabled (and, it should be noted, by default it is not), files that are deleted using the Hadoop filesystem shell are moved into a special hidden trash directory, rather than being deleted immediately. The trash directory is deleted periodically by the system. Any files that are mistakenly deleted can be recovered manually by moving them out of the trash directory. The trash deletion period is configurable, we recommend 24 hours as a sensible default.

The trash facility is a user level feature, it is not used when programmatically deleting files. As a best practice we recommend considering moving data to be deleted to the trash directory (to be deleted by the system after the given time), rather than doing an immediate delete.

Permissions

HDFS has a permissions system, based on the unix user/group model, that it is currently un-authenticated (like NFS version 3). It is designed to prevent data loss on a shared cluster (as opposed to providing data privacy, for example).

As a best practice, it is recommended that different processes in the data pipeline are segregated using the permissions controls on files and directories. For example, the process that feeds raw data into the cluster would make the files read only for other users. Processes that need to analyze and transform the raw data would not have permission to delete the data. Furthermore, there may be multiple downstream processes: these should have different user names depending on their role, so that the permissions on the data they consume and produce is carefully controlled.

Summary of HDFS Reliability Best Practices

1. Use a common HDFS configuration.
2. Use replication level of 3 (as a minimum), or more for critical (or widely-used) data.
3. Configure the name node to write to multiple local disks and NFS. Run the secondary on a separate node. Make multiple, periodic backups of name node persistent state.
4. Actively monitor your HDFS cluster.
5. Define backup and upgrade procedures.
6. Enable HDFS trash, and avoid programmatic deletes - prefer the trash facility.
7. Devise a set of users and permissions for your workflow.

Thanks to Brian Bockelman, Dhruba Borthakur, Jeff Hammerbacher, Aaron Kimball, Mike Olson, Matei Zaharia, and Philip Zeyliger for reading drafts of this paper.