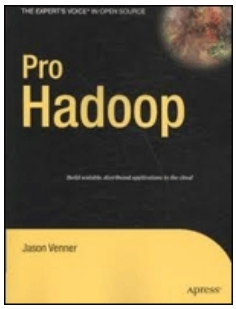

Pro Hadoop

by Jason Venner

Apress. (c) 2009. Copying Prohibited.

Reprinted for JORN P. KUHLENKAMP, IBM

jpkuhlen@us.ibm.com

Reprinted with permission as a subscription benefit of **Books24x7**,
http://www.books24x7.com/

All rights reserved. Reproduction and/or distribution in whole or in part in electronic,paper or other forms without written permission is prohibited.

Chapter 4: HDFS Details for Multimachine Clusters

As you learned in the previous chapter, the defaults provided for multimachine clusters will work well for very small clusters, but they are not suitable for large clusters (the clusters will fail in unexpected and difficult-to-understand ways). This chapter covers HDFS installation for multimachine clusters that are not very small, as well as HDFS tuning factors, recovery procedures, and troubleshooting tips. But first, let's look at some of the configuration trade-offs faced by IT departments.

Configuration Trade-Offs

There appears to be an ongoing conflict between the optimal machine and network configurations for Hadoop Core and the configurations required by IT departments.

IT departments are generally looking for low-overhead ways of maintaining high availability for all equipment. The IT department model commonly requires RAID 1 and RAID 5 for disks to minimize machine downtime from disk failures. IT departments also prefer managed network switches, as this allows for reporting and virtual local area network (VLAN) configurations. These strategies reduce the risk of machine failure and provide network diagnostics, flexibility, and simplified administration. Operations staff also prefer high-availability solutions for production applications.

Hadoop Core does not need highly reliable storage on the DataNode or TaskTracker nodes. Hadoop Core greatly benefits from increased network bandwidth.

The highest performance Hadoop Core installations will have separate and possibly multiple disks or arrays for each stream of I/O. The DataNode storage will be spread over multiple disks or arrays to allow interleaved I/O, and the TaskTracker intermediate output will also go to a separate disk or array. This configuration reduces the contention for I/O on any given array or device, thus increasing the maximum disk I/O performance of the machine substantially. If the switch ports are inexpensive, using bonded network interface cards (NICs) to increase per machine network bandwidth will greatly increase the I/O performance of the cluster.

Hadoop Core provides high availability of DataNode and TaskTracker services without requiring special hardware, software, or configuration. However, there is no simple solution for high availability for the NameNode or JobTracker. High availability for the NameNode is an active area of development within the Hadoop community. The techniques described in this chapter allow for rapid recovery from NameNode failures. All of these techniques require special configuration and have some performance cost. With Hadoop 0.19.0, there is built-in recovery for JobTracker failures, and generally, high availability for the JobTracker is not considered critical.

Affordable Disk Performance vs. Network Performance

Hadoop Core is designed to take advantage of commodity hardware, rather than more expensive special-purpose hardware.

In my experience, the bulk of custom-purchased Hadoop nodes appear to be 2U 8-way machines, with six internal drive bays, two Gigabit Ethernet (GigE) interfaces, and 8GB of RAM. Most of the disk drives that are being used for Hadoop are inexpensive SATA drives that generally have a sustained sequential transfer rate of about 70Mbps.

The RAID setup preferred by my IT department groups six of these drives in a RAID 5 array that provides roughly 250 Mbps sequential transfers to the application layer. Mixed read/write operations provide about 100 Mbps, as all I/O operations require seeks on the same set of drives. If the six drives were provided as individual drives or as three RAID 0 pairs, the mixed read/write I/O transfer rate would be higher, as seeks for each I/O operation could occur on different drives.

The common network infrastructure is GigE network cards on a GigE switch, providing roughly 100 Mbps I/O. I've used bonded pairs of GigE cards to provide 200 Mbps I/O for high-demand DataNodes to good effect.

IT departments seem to prefer large managed switches, resulting in a high per-port cost. For Hadoop nodes, providing dumb, unmanaged crossbar switches for the DataNodes is ideal.

HDFS Installation for Multimachine Clusters

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM

Apress, Jason Venner (c) 2009, Copying Prohibited

Setting up an HDFS installation for a multimachine cluster involves the following steps:

- Build the configuration.
- Distribute your installation data to all of the machines that will host HDFS servers.
- Format your HDFS.
- Start your HDFS installation.
- Verify HDFS is running.

The following sections detail these steps.

Building the HDFS Configuration

As discussed in the previous chapter, building the HDFS configuration requires generating the `conf/hadoop-site.xml`, `conf/slaves`, and `conf/masters` files. You also need to customize the `conf/hadoop-env.sh` file.

Generating the conf/hadoop-site.xml File

In the `conf/hadoop-site.xml` file, you tell Hadoop where the data files reside on the file system. At the simplest level, this requires setting a value for `hadoop.tmp.dir`, and providing a value for `fs.default.name` to indicate the master node of the HDFS cluster, as shown in Listing 4-1.

Listing 4-1: A Minimal hadoop-site.xml for an HFS Cluster (conf/hadoop-site.xml)

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://master:54310/</value>
  <description>The name of the default file system.  A URI whose
  scheme and authority determine the FileSystem implementation. The
  uri's scheme determines the config property (fs.SCHEME.impl) naming
  the FileSystem implementation class. The uri's authority is used to
  determine the host, port, etc. for a filesystem.</description>
</property>

<property>
  <name>hadoop.tmp.dir</name>
  <value>/hdfs</value>
  <description>A base for other all storage directories,
          temporary and persistent.</description>
</property>
```

This will configure a cluster with a NameNode on the host master, and all HDFS storage under the directory `/hdfs`.

Generating the conf/slaves and conf/masters Files

On the machine master, create the `conf/slaves` file, and populate it with the names of the hosts that will be DataNodes, one per line.

In the file `conf/masters`, add a single host to be the secondary NameNode. For safety, make it a separate machine, rather than `localhost`.

Customizing the conf/hadoop-env.sh File

The `conf/hadoop-env.sh` file provides system environment configuration information for all processes started by Hadoop, as well as all processes run by the user through the scripts in the `bin` directory of the installation. At a very minimum, this script must ensure that the correct `JAVA_HOME` environment variable is set. Table 4-1 provides a list of the required, commonly set, and optional environment variables.

Table 4-1: Environment Variables for Hadoop Processes

Variable	Description	Default
`JAVA_HOME`	This is required. It must be the root of the	System JDK

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

	JDK installation, such that `${JAVA_HOME}/bin/java` is the program to start a JVM.	
`HADOOP_NAMENODE_OPTS`	Additional command-line arguments for the NameNode server. The default enables local JMX access.	`"-Dcom.sun.management.jmxremote $HADOOP_NAMENODE_OPTS"`
`HADOOP_SECONDARYNAMENODE_OPTS`	Additional command-line arguments for the secondary NameNode server. The default enables local JMX access.	`"-Dcom.sun.management.jmxremote $HADOOP_SECONDARYNAMENODE_OPTS"`
`HADOOP_DATANODE_OPTS`	Additional command-line arguments for the DataNode servers. The default enables local JMX access.	`"-Dcom.sun.management.jmxremote $HADOOP_DATANODE_OPTS"`
`HADOOP_BALANCER_OPTS`	Additional command-line arguments for the Balancer service. The default enables local JMX access.	`"-Dcom.sun.management.jmxremote $HADOOP_BALANCER_OPTS"`
`HADOOP_JOBTRACKER_OPTS`	Additional command-line arguments for the JobTracker server. The default enables local JMX access.	`"-Dcom.sun.management.jmxremote $HADOOP_JOBTRACKER_OPTS"`
`HADOOP_TASKTRACKER_OPTS`	Additional command-line arguments for the TaskTracker servers.	
`HADOOP_CLIENT_OPTS`	Additional command-line arguments for all nonserver processes started by `bin/hadoop`. This is not applied to the server processes, such as the NameNode, JobTracker, TaskTracker, and DataNode.	
`HADOOP_SSH_OPTS`	Additional command-line arguments for any `ssh` process run by scripts in `bin`. This commented-out option in the `stock hadoop-env.sh` file sets the `ssh` connection timeout to 1 second and instructs `ssh` to forward the `HADOOP_CONF_DIR` environment variable to the remote shell.	`"-o ConnectTimeout=1 -o SendEnv=HADOOP_CONF_DIR"`
`HADOOP_LOG_DIR`	The root directory path that Hadoop logging files will be created under.	`${HADOOP_HOME}/logs`
`HADOOP_SLAVES`	The path of the file containing the list of hostnames to be used as DataNode and or TaskTracker servers.	`${HADOOP_HOME}/conf/slaves`
`HADOOP_SLAVE_SLEEP`	The amount of time to sleep between `ssh` commands when operating on all of the slave nodes.	`0.1`
`HADOOP_PID_DIR`	The directory that server process ID (PID) files are written to. Used by the service start and stop scripts to determine if a prior instance of a server is running. The default, `/tmp`, is not a good location, as the server PID files will be periodically removed by the system `temp` file cleaning service.	`/tmp`
`HADOOP_IDENT_STRING`	Used in constructing path names for cluster instance-specific file names, such as the file names of the log files for the server processes and the PID files for the server processes.	`$USER`
`HADOOP_NICENESS`	CPU scheduling nice factor to apply to server processes. `5` is recommended for DataNode servers and `10` for TaskTrackers. The suggested settings prioritize the DataNode over the TaskTracker to ensure that DataNode	

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

	requests are more rapidly serviced. They also help ensure that the NameNode or JobTracker servers have priority if a DataNode or TaskTracker is colocated on the same machine. These suggested settings facilitate smooth cluster operation and enable easier monitoring.	
`HADOOP_CLASSPATH`	Extra entries for the classpath for all Hadoop Java processes. If your jobs always use specific JARs, and these JARs are available on all systems in the same location, adding the JARs here ensures that they are available to all tasks and reduces the overhead in setting up a job on the cluster.	
`HADOOP_HEAPSIZE`	The maximum process heap size for running tasks. `2000`, indicating 2GB, is suggested. See this book's appendix for details on the `JobConf` object.	
`HADOOP_OPTS`	Additional command-line options for all processes started by `bin/hadoop`. This setting is normally present but commented out.	`-server`

Distributing Your Installation Data

Distribute your Hadoop installation to all the machines in `conf/masters` and `conf/slaves`, as well as the machine `master`. Ensure that the user that will own the servers can write to `/hdfs` and the directory set for `HADOOP_PID_DIR` in `conf/hadoop-env.sh`, on all of the machines in `conf/masters`, `conf/slaves`, and `master`.

Finally, ensure that passwordless SSH from `master` to all of the machines in `conf/masters` and `conf/slaves` works.

Setting JAVA_HOME and Permissions

The user is responsible for ensuring that the `JAVA_HOME` environment variable is configured. The framework will issue an `ssh` call to each slave machine:

```
execute ${HADOOP_HOME}/bin/hadoop-daemon.sh start SERVER
```

This sources `${HADOOP_HOME}/conf/hadoop-env.sh`.

The administrator is also responsible for ensuring that the `JAVA_HOME` environment variable is set correctly in the login scripts or in the `hadoop-env.sh` script.

Additionally, the administrator must ensure that the storage directories are writable by the Hadoop user. In general, this simply means constructing the `hadoop.tmp.dir` directory set in the `conf/hadoop-site.xml` file, and `chowning` the directory to the user that the Hadoop servers will run as.

Formatting Your HDFS

At this point, you are ready to actually format your HDFS installation. Run the following to format a NameNode for the first time:

```
hadoop namenode -format
```

```
namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = master/127.0.0.1
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 0.19.0
STARTUP_MSG:   build = ...
```


Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

```
************************************************************/
namenode.FSNamesystem: fsOwner=jason,jason,lp,wheel,matching
namenode.FSNamesystem: supergroup=supergroup
namenode.FSNamesystem: isPermissionEnabled=true
common.Storage: Image file of size 95 saved in 0 seconds.
common.Storage: Storage directory /hdfs/dfs/name has been successfully formatted.
namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at master/127.0.0.1
************************************************************/
```

Note The exact stack traces and line numbers will vary with your version of Hadoop Core.

If you have already formatted a NameNode with this data directory, you will see that the command will try to reformat the NameNode:

```
hadoop namenode -format
```

```
09/01/24 12:03:57 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = master/127.0.0.1
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 0.19.0
STARTUP_MSG:   build = ...
************************************************************/
Re-format filesystem in /hdfs/dfs/name ? (Y or N) y
Format aborted in /hdfs/dfs/name
09/01/24 12:04:01 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at master/127.0.0.1
************************************************************/
```

If the user doing the formatting does not have write permissions, the output will be as follows:

```
bin/hadoop namenode -format
```

```
INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = master/127.0.0.1
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 0.19.0
STARTUP_MSG:   build = ...
************************************************************/
INFO namenode.FSNamesystem: fsOwner=jason,jason,lp,wheel,matching
INFO namenode.FSNamesystem: supergroup=supergroup
INFO namenode.FSNamesystem: isPermissionEnabled=true
ERROR namenode.NameNode: java.io.IOException:
    Cannot create directory /tmp/test1/dir/dfs/name/current
    at org.apache.hadoop.hdfs.server.common.Storage$StorageDirectory. ➥
    clearDirectory(Storage.java:295)
    at org.apache.hadoop.hdfs.server.namenode.FSImage.format(FSImage.java:1067)
    at org.apache.hadoop.hdfs.server.namenode.FSImage.format(FSImage.java:1091)
    at org.apache.hadoop.hdfs.server.namenode.NameNode.format(NameNode.java:767)

    at org.apache.hadoop.hdfs.server.namenode.NameNode. ➥
    createNameNode(NameNode.java:851)
    at org.apache.hadoop.hdfs.server.namenode.NameNode.main(NameNode.java:868)

09/01/25 19:14:37 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at master/127.0.0.1
```

If you are reformatting an HDFS installation, it is recommended that you wipe the `hadoop.tmp.dir` directories on all of

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

the DataNode machines.

Starting Your HDFS Installation

After you've configured and formatted HDFS, it is time to actually start your multimachine HDFS cluster. You can use the `bin/start-dfs.sh` command for this, as follows:

```
bin/start-dfs.sh
```

```
starting namenode, logging to
     /home/jsn/src/hadoop-0.19.0/bin/../logs/hadoop-jsn-namenode-master.out
slave1: starting datanode, logging to
    /home/jsn/src/hadoop-0.19.0/bin/../logs/hadoop-jsn-datanode-at.out
slave1: starting secondarynamenode, logging to
    /home/jsn/src/hadoop-0.19.0/bin/../logs/hadoop-jsn-secondarynamenode-slave1.out
slave2: starting datanode, logging to
    /home/jsn/src/hadoop-0.19.0/bin/../logs/hadoop-jsn-datanode-at.out
slave2: starting secondarynamenode, logging to
    /home/jsn/src/hadoop-0.19.0/bin/../logs/hadoop-jsn-secondarynamenode-slave2.out
slave3: starting datanode, logging to
    /home/jsn/src/hadoop-0.19.0/bin/../logs/hadoop-jsn-datanode-at.out
slave3: starting secondarynamenode, logging to
    /home/jsn/src/hadoop-0.19.0/bin/../logs/hadoop-jsn-secondarynamenode-slave3.out
slave4: starting datanode, logging to
    /home/jsn/src/hadoop-0.19.0/bin/../logs/hadoop-jsn-datanode-at.out
slave4: starting secondarynamenode, logging to
    /home/jsn/src/hadoop-0.19.0/bin/../logs/hadoop-jsn-secondarynamenode-slave4.out
slave5: starting datanode, logging to
    /home/jsn/src/hadoop-0.19.0/bin/../logs/hadoop-jsn-datanode-at.out
slave5: starting secondarynamenode, logging to
    /home/jsn/src/hadoop-0.19.0/bin/../logs/hadoop-jsn-secondarynamenode-slave5.out
```

If you have any lines like the following in your output, then the script was unable to `ssh` to the slave node:

```
slave1: ssh: connect to host slave1 port 22: No route to host
```

If you have a block like the following, you have not distributed your Hadoop installation correctly to all of the slave nodes:

```
slave1: bash: line 0: cd: /home/jason/src/hadoop-0.19.0/bin/..: ➥
No such file or directory
slave1: bash: /home/jason/src/hadoop-0.19.0/bin/hadoop-daemon.sh: ➥
No such file or directory
slave1: bash: line 0: cd: /home/jason/src/hadoop-0.19.0/bin/..: ➥
No such file or directory
slave1: bash: /home/jason/src/hadoop-0.19.0/bin/hadoop-daemon.sh: ➥
No such file or directory
```

The following output indicates that the directory specified in `hadoop-env.sh` for the server PID files was not writable:

```
slave1: mkdir: cannot create directory `/var/bad-hadoop': Permission denied
slave1: starting datanode, logging to
 /home/jason/src/hadoop-0.19.0/bin/../logs/hadoop-jason-datanode-slave1.out
slave1: /home/jason/src/hadoop-0.19.0/bin/hadoop-daemon.sh: line 118:
  /var/bad-hadoop/pids/hadoop-jason-datanode.pid: No such file or directory
slave1: mkdir: cannot create directory `/var/bad-hadoop': Permission denied
slave1: starting secondarynamenode, logging to
 /home/jason/src/hadoop-0.19.0/bin/../ ➥
logs/hadoop-jason-secondarynamenode-slave1.out
slave1: /home/jason/src/hadoop-0.19.0/bin/hadoop-daemon.sh: line 118: ➥
/var/bad-hadoop/pids/hadoop-jason-secondarynamenode.pid: ➥
No such file or directory
```


Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

When Hadoop Core is starting services on the cluster, the required directories for the server operation are created if needed. These include the PID directory and the working and temporary storage directories for the server. If the framework is unable to create a directory, there will be error messages to that effect logged to the error stream of the script being used to start the services. Any messages in the startup output about files or directories that are not writable, or directories that could not be created, must be addressed. In some cases, the server will start, and the cluster may appear to run, but there will be stability and reliability issues.

At this point, the next step is to verify that the DataNode servers started and that they were able to establish service with the NameNode.

Verifying HDFS Is Running

To verify that the server processes are in fact running, wait roughly 1 minute after the finish of `start-dfs.sh`, and then check that the NameNode and DataNode exist.

Checking the NameNodes

On the master machine, run the `jps` command as follows:

```
${JAVA_HOME}/bin/jps
```

```
1887 Jps
19379 NameNode
```

The first value in the output is the PID of the `java` process, and it will be different in your output.

If there is no NameNode, the initialization failed, and you will need to examine the log file to determine the problem. The log data will be in the file `logs/hadoop-${USER}-namenode.log`.

Note The NameNode actually performs the formatting operation, and the formatting results end up in the NameNode log.

The examples in Listings 4-2 and 4-3 are excerpts from a NameNode log, demonstrating different failure cases.

Listing 4-2: Did the NameNode Format Fail Due to Insufficient Permissions?

```
bin/hadoop namenode -format
```

```
INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = master/127.0.0.1
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 0.19.0
STARTUP_MSG:   build = ...
************************************************************/
INFO namenode.FSNamesystem: fsOwner=jason,jason,lp,wheel,matching
INFO namenode.FSNamesystem: supergroup=supergroup
INFO namenode.FSNamesystem: isPermissionEnabled=true
ERROR namenode.NameNode: java.io.IOException: ➥
Cannot create directory /tmp/test1/dir/dfs/name/current
    at org.apache.hadoop.hdfs.server.common.Storage$StorageDirectory.➥
clearDirectory(Storage.java:295)
    at org.apache.hadoop.hdfs.server.namenode.FSImage.format(FSImage.java:1067)
    at org.apache.hadoop.hdfs.server.namenode.FSImage.format(FSImage.java:1091)

    at org.apache.hadoop.hdfs.server.namenode.NameNode.format(NameNode.java:767)
    at org.apache.hadoop.hdfs.server.namenode.NameNode. ➥
    createNameNode(NameNode.java:851)
    at org.apache.hadoop.hdfs.server.namenode.NameNode.main(NameNode.java:868)
```


Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

```
09/01/25 19:14:37 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at master/127.0.0.1
```

Listing 4-2 indicates that the NameNode process was unable to find a valid directory for HDFS metadata. When this occurs, the command `hadoop namenode -format` must be run, to determine the actual failure. If the `format` command completes successfully, the next `start-dfs.sh` run should complete successfully. The example in Listing 4-3 demonstrates the failed `format` command output. The actual directory listed will be different in actual usage, the directory path `/tmp/test1/dir/dfs/name/current` *was constructed just for this test*.

Listing 4-3: A Failed Format Due to Directory Permissions

```
bin/hadoop namenode -format
```

```
09/04/04 13:13:37 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = at/192.168.1.119
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 0.19.1-dev
STARTUP_MSG:   build =  -r ; compiled by 'jason' on Tue Mar 17 04:03:57 PDT 2009
************************************************************/
INFO namenode.FSNamesystem: fsOwner=jason,jason,lp
INFO namenode.FSNamesystem: supergroup=supergroup
INFO namenode.FSNamesystem: isPermissionEnabled=true
ERROR namenode.NameNode: java.io.IOException: ➥
Cannot create directory /tmp/test1/dir/dfs/name/current
    at org.apache.hadoop.hdfs.server.common.Storage$StorageDirectory. ➥
clearDirectory(Storage.java:295)
    at org.apache.hadoop.hdfs.server.namenode.FSImage.format(FSImage.java:1067)
    at org.apache.hadoop.hdfs.server.namenode.FSImage.format(FSImage.java:1091)
    at org.apache.hadoop.hdfs.server.namenode.NameNode.format(NameNode.java:767)
    at org.apache.hadoop.hdfs.server.namenode.NameNode. ➥
createNameNode(NameNode.java:851)
    at org.apache.hadoop.hdfs.server.namenode.NameNode.main(NameNode.java:868)

INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at at/192.168.1.119
************************************************************/
```

The web interface provided by the NameNode will show information about the status of the NameNode. By default, it will provide a service on `http://master:50070/`.

Checking the DataNodes

You also need to verify that there are DataNodes on each of the slave nodes. Use `jps` via the `bin/slaves.sh` command to look for DataNode processes:

```
bin/slaves.sh jps | grep Datanode | sort
```

```
slave1: 2564 DataNode
slave2: 2637 DataNode
slave3: 1532 DataNode
slave4: 7810 DataNode
slave5: 8730 DataNode
```

This example shows five DataNodes, one for each slave. If you do not have a DataNode on each of the slaves, something has failed. Each machine may have a different reason for failure, so you'll need to examine the log files on each machine.

The common reason for DataNode failure is that the `dfs.data.dir` was not writable, as shown in Listing 4-4.

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

Listing 4-4: Excerpt from a DataNode Log File on Failure to Start Due to Permissions Problems

```
2009-01-28 07:50:05,441 INFO org.apache.hadoop.hdfs.server.datanode. ➥
DataNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting DataNode
STARTUP_MSG:   host = slave1/127.0.0.1
STARTUP_MSG:   args = []
STARTUP_MSG:   version = 0.19.1-dev
STARTUP_MSG:   build =  -r ; compiled by 'jason' on Wed Jan 21 18:10:58 PST 2009
************************************************************/
2009-01-28 07:50:05,653 WARN org.apache.hadoop.hdfs.server.datanode. ➥
DataNode: Invalid directory in dfs.data.dir: can not create directory: ➥
 /tmp/test1/dir/dfs/data
2009-01-28 07:50:05,653 ERROR org.apache.hadoop.hdfs.server.datanode. ➥
DataNode: All directories in dfs.data.dir are invalid.
2009-01-28 07:50:05,654 INFO org.apache.hadoop.hdfs.server.datanode. ➥
DataNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down DataNode at at/127.0.0.1
************************************************************/
```

The DataNode may also be unable to contact the NameNode due to network connectivity or firewall issues. In fact, I had half of a new cluster fail to start, and it took some time to realize that the newly installed machines had a default firewall that blocked the HDFS port. Listing 4-5 shows an excerpt from a DataNode log for a DataNode that failed to start due to network connectivity problems.

Listing 4-5: DataNode Log Excerpt, Failure to Connect to the NameNode

```
INFO org.apache.hadoop.hdfs.server.datanode.DataNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting DataNode
STARTUP_MSG:   host = slave1/127.0.0.1
STARTUP_MSG:   args = []
STARTUP_MSG:   version = 0.19.1-dev
STARTUP_MSG:   build =...
************************************************************/
INFO org.apache.hadoop.ipc.Client: Retrying connect to server:
    master/192.168.1.2:8020. Already tried 0 time(s).
INFO org.apache.hadoop.ipc.Client: Retrying connect to server:
    master/192.168.1.2:8020. Already tried 1 time(s).
INFO org.apache.hadoop.ipc.Client: Retrying connect to server:
    master/192.168.1.2:8020. Already tried 2 time(s).
INFO org.apache.hadoop.ipc.Client: Retrying connect to server:
    master/192.168.1.2:8020. Already tried 3 time(s).
INFO org.apache.hadoop.ipc.Client: Retrying connect to server:
    master/192.168.1.2:8020. Already tried 4 time(s).
INFO org.apache.hadoop.ipc.Client: Retrying connect to server:
    master/192.168.1.2:8020. Already tried 5 time(s).
INFO org.apache.hadoop.ipc.Client: Retrying connect to server:
    master/192.168.1.2:8020. Already tried 6 time(s).
INFO org.apache.hadoop.ipc.Client: Retrying connect to server:
    master/192.168.1.2:8020. Already tried 7 time(s).
INFO org.apache.hadoop.ipc.Client: Retrying connect to server:
    master/192.168.1.2:8020. Already tried 8 time(s).
INFO org.apache.hadoop.ipc.Client: Retrying connect to server:
    master/192.168.1.2:8020. Already tried 9 time(s).
ERROR org.apache.hadoop.hdfs.server.datanode.DataNode:
  java.io.IOException: Call to master/192.168.1.2:8020 failed on local exception:
    No route to host
    at org.apache.hadoop.ipc.Client.call(Client.java:699)
    at org.apache.hadoop.ipc.RPC$Invoker.invoke(RPC.java:216)
    at $Proxy4.getProtocolVersion(Unknown Source)
    at org.apache.hadoop.ipc.RPC.getProxy(RPC.java:319)
    at org.apache.hadoop.ipc.RPC.getProxy(RPC.java:306)
    at org.apache.hadoop.ipc.RPC.getProxy(RPC.java:343)
    at org.apache.hadoop.ipc.RPC.waitForProxy(RPC.java:288)
```


Reprinted for ibm\jpkuhlen@us.ibm.com, IBM

Apress, Jason Venner (c) 2009, Copying Prohibited

```
    at org.apache.hadoop.hdfs.server.datanode.DataNode. ➥
    startDataNode(DataNode.java:258)

    at org.apache.hadoop.hdfs.server.datanode.DataNode.<init>(DataNode.java:205)
    at org.apache.hadoop.hdfs.server.datanode.DataNode. ➥
    makeInstance(DataNode.java:1199)
    at org.apache.hadoop.hdfs.server.datanode.DataNode. ➥
    instantiateDataNode(DataNode.java:1154)
    at org.apache.hadoop.hdfs.server.datanode.DataNode. ➥
    createDataNode(DataNode.java:1162)
    at org.apache.hadoop.hdfs.server.datanode.DataNode.main(DataNode.java:1284)
Caused by: java.net.NoRouteToHostException: No route to host
    at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
    at sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:574)
    at sun.nio.ch.SocketAdaptor.connect(SocketAdaptor.java:100)
    at org.apache.hadoop.ipc.Client$Connection.setupIOstreams(Client.java:299)
    at org.apache.hadoop.ipc.Client$Connection.access$1700(Client.java:176)
    at org.apache.hadoop.ipc.Client.getConnection(Client.java:772)
    at org.apache.hadoop.ipc.Client.call(Client.java:685)
    ... 12 more

INFO org.apache.hadoop.hdfs.server.datanode.DataNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down DataNode at at/127.0.0.1
************************************************************/
```

Note It is important to note the IP address to which the DataNode is trying to connect. If the IP address is correct, verify that the NameNode is accepting connections on that port.

Hadoop also provides the `dfsadmin -report` command-line tool, which will provide a somewhat verbose listing of the DataNodes in a service. You can run this useful script from your system monitoring tools, so that alerts can be generated if DataNodes go offline. The following is an example run of this tool:

```
bin/hadoop dfsadmin -report
```

```
Configured Capacity: 190277042176 (177.21 GB)
Present Capacity: 128317767680 (119.51 GB)
DFS Remaining: 128317718528 (119.51 GB)
DFS Used: 49152 (48 KB)
DFS Used%: 0%

-------------------------------------------------
Datanodes available: 5 (5 total, 0 dead)

Name: 192.168.1.3:50010
Decommission Status : Normal
Configured Capacity: 177809035264 (165.6 GB)
DFS Used: 24576 (24 KB)
Non DFS Used: 50479497216 (47.01 GB)
DFS Remaining: 127329513472(118.58 GB)
DFS Used%: 0%
DFS Remaining%: 71.61%
Last contact: Wed Jan 28 08:27:20 GMT-08:00 2009

...
```

Tuning Factors

Here, we will look at tuning the cluster system and the HDFS parameters for performance and reliability.

Commonly, the two most important factors are network bandwidth and disk throughput. Memory use and CPU overhead for thread handling may also be issues.

HDFS uses a basic file system block size of 64MB, and the JobTracker also chunks task input into 64MB segments. Using large blocks helps reduce the cost of a disk seek compared with the read/write time, thereby increasing the aggregate I/O

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

rate when multiple requests are active. The large input-split size reduces the ratio of task setup time to task run time, as there is work to be done to set up a task before the TaskTracker can start the mapper or reducer on the input split.

The various tuning factors available control the maximum number of requests in progress. In general, the more requests in progress, the more contention there is for storage operations and network bandwidth, with a corresponding increase in memory requirements and CPU overhead for handling all of the outstanding requests. If the number of requests allowed is too low, the cluster may not fully utilize the disk or network bandwidth, or cause requests to timeout.

Most of the tuning parameters for HDFS do not yet have exact science behind the settings. Currently, tuning is ad hoc and per cluster. In general, the selections are a compromise between various factors. Finding the sweet spot requires knowledge of the local system and experience.

Let's examine each of the tuning factors in turn.

File Descriptors

Hadoop Core uses large numbers of file descriptors for MapReduce, and the `DFSClient` uses a large number of file descriptors for communicating with the HDFS NameNode and DataNode server processes. The `DFSClient` code also presents a misleading error message when there has been a failure to allocate a file descriptor: `No live nodes contain current block`.

> **Note** The class `org.apache.hadoop.hdfs.DFSClient` provides the actual file services for applications interacting with the HDFS file system. All file system operations and file operations performed by the application will be translated into method calls on the `DFSClient` object, which will in turn issue the appropriate Remote Procedure Call (RPC) calls to the NameNode and the DataNodes relevant for the operations. As of Hadoop 0.18, these RPC calls use the Java NIO services. Prior to Hadoop 0.18, blocking operations and fixed timeouts were used for the RPC calls.

Most sites immediately bump up the number of file descriptors to 64,000, and large sites, or sites that have MapReduce jobs that open many files, might go higher.

For a Linux machine, the simple change is to add a line to the `/etc/security/limits.conf` file of the following form:

```
* hard nofile 64000
```

This changes the per-user file descriptor limit to 64,000 file descriptors. If you will run a much larger number of file descriptors, you may need to alter the per-system limits via changes to `fs.file-max` in `/etc/sysctl.conf`. A line of the following form would set the system limit to 640,000 file descriptors:

```
fs.file-max = 640000
```

At this point, you may alter the `limits.conf` file line to this:

```
* hard nofile 640000
```

Changes to `limits.conf` take effect on the next login, and changes to `sysctl.conf` take place on the next reboot. You may run `sysctl` by hand (`sysctl -p`) to cause the `sysctl.conf` file to be reread and applied.

The web page at `http://support.zeus.com/zws/faqs/2005/09/19/filedescriptors` provides some instructions for several Unix operating systems. For Windows XP instructions, see `http://weblogs.asp.net/mikedopp/archive/2008/05/16/increasing- user-handle-and-gdi-handle-limits.aspx`.

Any user that runs processes that access HDFS should have a large limit on file descriptor access, and all applications that open files need careful checking to make sure that the files are explicitly closed. Trusting the JVM garbage collection to close your open files is a critical mistake.

Block Service Threads

Each DataNode maintains a pool of threads that handle block requests. The parameter `dfs.datanode.handler.count` controls the number of threads the DataNode will use for handling IPC requests. These are the threads that accept connections on `dfs.datanode.ipc.address`, the configuration value described in Chapter 3.

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

Currently, there does not appear to be significant research or tools on the tuning of this parameter. The overall concept is to balance JVM overhead due to the number of threads with disk and network I/O. The more requests active at a time, the more overlap for disk and network I/O there is. At some point, the overlap results in contention rather than increased performance.

The default value for the `dfs.datanode.handler.count` parameter is `3`, which seems to be fine for small clusters. Medium-size clusters may use a value of `30`, set as follows:

```
<property>
  <name>dfs.datanode.handler.count</name>
  <value>30</value>
  <description>The number of server threads for the datanode.</description>
</property>
```

NameNode Threads

Each HDFS file operation—create, open, close, read, write, stat, and unlink—requires a NameNode transaction. The NameNode maintains the metadata of the file system in memory. Any operation that changes the metadata, such as open, write, or unlink, results in the NameNode writing a transaction to the disks, and an asynchronous operation to the secondary NameNodes.

Note Some installations add an NFS directory to the list of local locations to which the file system is written. This adds a substantial latency to any metadata altering transaction. Through Hadoop 0.19.0, the NameNode edit logs are forcibly flushed to disk storage, but space for the updates is preallocated before the update to reduce the overall latency.

The parameter `dfs.namenode.handler.count` controls the number of threads that will service NameNode requests. The default value is `10`. Increasing this value will substantially increase the memory utilization of the NameNode and may result in reduced performance due to I/O contention for the metadata updates. However, if your map and reduce tasks create or remove large numbers of files, or execute many sync operations after writes, this number will need to be higher, or you will experience file system timeouts in your tasks.

In my setup, I have a cluster of 20 DataNodes that are very active. For this cluster, the `dfs.namenode.handler.count` parameter is set to `512`:

```
<property>
  <name>dfs.namenode.handler.count</name>
  <value>512</value>
  <description>The number of server threads for the namenode.</description>
</property>
```

Note The NameNode serializes the entire directory metadata and then sends it when a request for the status of files in a directory is made. For large directories, this can be a considerable amount of data. With many threads servicing requests, the amount of memory used by in transit serialized requests may be very large. I have worked with a cluster where this transitory memory requirement exceeded 1GB.

Server Pending Connections

Hadoop Core's `ipc.server.listen.queue.size` parameter sets the listen depth for accepting connections. This value is the number of outstanding, unserviced connection requests allowed by the operating system, before connections are refused. Its default value is `128`.

Many operating systems have hard and small limits for this value. If your thread counts are lower, this value will need to be higher, to prevent refused connections. If you find connection-refused errors in your log files, you may want to increase the value of this parameter.

Reserved Disk Space

Hadoop Core historically has not handled out-of-disk-space exceptions gracefully. As a method for preventing out-of-disk conditions, Hadoop Core provides four parameters: two for HDFS and two for MapReduce.

It is common, especially in smaller clusters, for a DataNode and a TaskTracker to reside on each computer node in the cluster. It is also common for the TaskTracker's temporary storage to be stored in the same file system as the HDFS data

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

blocks. With HDFS, blocks written by an application that is running on the same machine as a DataNode will have one replica placed on that DataNode. It is very easy for a task to fill up the disk space on a partition and cause HDFS failures, or for HDFS to fill up a partition and result in task failures. Tuning the disk space reservation parameters will minimize these failures.

You can adjust the following parameters:

`mapred.local.dir.minspacestart`: This parameter controls how much space must be available in the temporary area used for map and reduce task output, before a task may be assigned to a TaskTracker. This is checked only before task assignment. If you have more than one TaskTracker on the node, you may need to multiply this minimum by the TaskTracker count to ensure sufficient space. This parameter has a default value of `0`, disabling the check.

`mapred.local.dir.minspacekill`: This parameter will cause the tasks in progress to be killed if the available MapReduce local space falls below this threshold. This parameter has a default value of `0`, disabling the check.

`dfs.datanode.du.reserved`: This parameter specifies how much of the available disk space to exclude from use for HDFS. If your file system has 10GB available and `dfs.datanode.du.reserved` is set to 10GB (`10737418240`), HDFS will not store new blocks on this DataNode. This parameter has a default value of `0`, disabling the check.

`dfs.datanode.du.pct`: This value controls what percentage of the physically available space is available for use by HDFS for new data blocks. For example, if there are 10GB available in the host file system partition, and the default value of `0.98` is in effect, only 10GB * 0.98 bytes (10,522,669,875 bytes) will actually be considered available. This parameter has been dropped from Hadoop 0.19.

Caution The `dfs.datanode.du.pct` parameter is undocumented and may not have the meaning described here in your version of Hadoop.

Storage Allocations

The Balancer service will slowly move data blocks between DataNodes to even out the storage allocations. This is very helpful when DataNodes have different storage capacities. The Balancer will also slowly re-replicate underreplicated blocks

The parameter `dfs.balance.bandwidthPerSec` controls the amount of bandwidth that may be used for balancing between DataNodes. The default value is 1MB (`1048576`).

The Balancer is run by the command `start-balancer.sh [-threshold VALUE]`. The argument `-threshold VALUE`, is optional, and the default threshold is 10%. The Balancer task will run until the free space percentage for block storage of each DataNode is approximately equal, with variances in free space percentages of up to the defined threshold allowed.

The `start-balancer.sh` script prints the name of a file to the standard output that will contain the progress reports for the Balancer. It is common for the Balancer task to be run automatically and periodically. The `start-balancer.sh` script will not allow multiple instances to be run, and it is safe to run this script repeatedly.

In general, the Balancer should be used if some DataNodes are close to their free space limits while other DataNodes have plenty of available space. This usually becomes an issue only when the DataNode storage capacity varies significantly or large datasets are written into HDFS from a machine that is also a DataNode.

The Balancer may not be able to complete successfully if the cluster is under heavy load, the threshold percentage is very small, or there is insufficient free space available. If you need to stop the Balancer at any time, you can use the command `stop-balancer.sh`.

Disk I/O

Hadoop Core is designed for jobs that have large input datasets and large output datasets. This I/O will be spread across multiple machines and will have different patterns depending on the purpose of the I/O, as follows:

- The NameNode handles storing the metadata for the file system. This includes the file paths, the blocks that make up the files, and the DataNodes that hold the blocks. The NameNode writes a journal entry to the edit log when a change is made. The NameNode keeps the entire dataset in memory to enable faster response time for requests that don't

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

involve modification.

- The DataNodes store, retrieve, and delete data blocks. The basic block size is 64MB per block. Unless an archive (such as a `.zip`, `.tar`, `tgz`, `.tar.gz`, or `.har` file) is used, blocks will not be shared among multiple files.
- The TaskTracker will hold a copy of all of the unpacked elements of the distributed cache for a job, as well as store the partitioned and sorted map output, potentially the merge sort output, and the reduce input. There are also temporary data files and buffered HDFS data blocks.

The default for Hadoop is to place all of these storage areas under the directory defined by the parameter `hadoop.tmp.dir`. In installations where multiple partitions, each on a separate physical drive or RAID assembly, are available, you may specify directories and directory sets that are stored on different physical devices to minimize seek or transfer contention.

All of the HDFS data stores are hosted on native operating system file systems. These file systems have many tuning parameters. For the NameNode data stores, RAID 1 with hot spares is preferred for the low-level storage. For the DataNodes, no RAID is preferred, but RAID 5 is acceptable.

The file systems should be constructed with awareness of the stripe and stride of any RAID arrays and, where possible, should be mounted in such a way that access time information is not updated. Linux supports disabling the access time updates for several file systems with the `noatime` and `nodiratime` file system mount time options. I have found that this change alone has provided a 5% improvement in performance on DataNodes.

Journaled file systems are not needed for the NameNode, as the critical data is written synchronously. Journaled file systems are not recommended for DataNodes due to the increase in write loading. The downside of not having a journal is that crash recovery time becomes much larger.

Secondary NameNode Disk I/O Tuning

The secondary NameNode provides a replica of the file system metadata that is used to recover a failed NameNode. It is critically important that its storage directories be on separate physical devices from the NameNode.

There is some debate about locating the secondary NameNode itself on a separate physical machine. The merging of the edit logs into the file system image may be faster and have lower overhead when the retrieval of the edit logs and writing of the file system image are local. However, having the secondary NameNode on a separate machine provides rapid recovery to the previous checkpoint time in the event of a catastrophic failure of the NameNode server machine.

Note For maximum safety, the secondary NameNode and NameNode should run on separate physical machines, with physically separate storage.

The secondary NameNode uses the parameter `fs.checkpoint.dir` to determine which directory to use to maintain the file system image. The default value is `${hadoop.tmp.dir}/dfs/namesecondary`. This parameter may be given a comma-separated list of directories. The image is replicated across the set of comma-separated values.

The secondary NameNode uses the parameter `fs.checkpoint.edits.dir` to hold the edit log, essentially the journal for the file system. It, like `fs.checkpoint.dir`, may be a commaseparated list of items, and the data will be replicated across the set of values. The default value is the value of `fs.checkpoint.dir`. The data written to the `fs.checkpoint.edits.dir` tends to be many synchronous small writes. The update operations are allowed to run behind the NameNode's updates.

The secondary NameNode server will take a snapshot from the NameNode at defined time intervals. The interval is defined by the parameter `fs.checkpoint.period`, which is a time in seconds, with a default value of `3600`. If the NameNode edit log grows by more than `fs.checkpoint.size` bytes (the default value is `67108864`), a checkpoint is also triggered.

The secondary NameNode periodically (`fs.checkpoint.period`) requests a checkpoint from the NameNode. At that point, the NameNode will close the current edit log and start a new edit log. The file system image and the just closed edit log will be copied to the secondary NameNode. The secondary NameNode will apply the edit log to the file system image, and then transfer the up-to-date file system image back to the NameNode, which replaces the prior file system image with the merged copy.

The secondary NameNode configuration is not commonly altered, except in very high utilization situations. In these cases,

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

multiple file systems on separate physical disks are used for the set of locations configured in the `fs.checkpoint.edits.dir` and `fs.checkpoint.dir` parameters.

NameNode Disk I/O Tuning

The NameNode is the most critical piece of equipment in your Hadoop Core cluster. The NameNode, like the secondary NameNode, maintains a journal and a file system image.

The parameter `dfs.name.dir` provides the directories in which the NameNode will store the file system image and is a comma-separated list (again, the image will be replicated across the set of values). The file system image is read and updated only at NameNode start time, as of Hadoop version 0.19. The default value for this parameter is `${hadoop.tmp.dir}/dfs/name`.

The parameter `dfs.name.edits.dir` contains the comma-separated list of directories to which the edit log or journal will be written. This is updated for every file system metadata-altering operation, synchronously. Your entire HDFS cluster will back up waiting for these updates to flush to the disk. If your cluster is experiencing a high rate of file create, rename, delete, or high-volume writes, there will be a high volume of writes to this set of directories. Any other I/O operations to file system partitions holding these directories will perform badly. The default value of this parameter is `${dfs.name.dir}`.

Some installations will place directories on remote mounted file systems in this list, to ensure that an exact copy of the data is available in the event of a NameNode disk failure.

RAID 1 or RAID 10 is recommended for the `dfs.name.edits.dir` set, rather than RAID 5, due to the increased write latency for small block writes.

Any data loss in the directories specified by `dfs.name.dir` and `dfs.name.edits.dir` will result in the loss of data in your HDFS. Caution, redundancy, and reliability are critical.

> **Caution** The NameNode is a critical single point of failure. Any loss of data can wipe out your entire HDFS datastore.

DataNode Disk I/O Tuning

The DataNode provides two services: block storage, and retrieval of HDFS data and storage accounting information for the NameNode.

Through at least Hadoop 0.19.0, the storage accounting has significant problems if there are more than a few hundred thousand blocks per DataNode. This is because a linear scan of the blocks is performed on a frequent basis to provide accounting information.

The parameter `dfs.datanode.handler.count` (covered earlier in the "Block Service Threads" section) is the number of worker threads for storage requests. The following are some other DataNode parameters that may be adjusted:

`dfs.data.dir`: This parameter sets the location(s) for block storage. It is a commaseparated list of directories that will be used for block storage by the DataNode. Blocks will be distributed in a round-robin manner among the directories. If multiple directories are to be used, it is best if each directory resides on an independent storage device. This will allow concurrent I/O operations to be active on all of the devices. The default value for this parameter is `${hadoop.tmp.dir}/dfs/data`.

`dfs.replication`: This parameter controls the number of copies of each data block the cluster attempts to maintain. HDFS is resilient in the face of individual DataNode failures, because individual blocks are replicated to more than one machine. The HDFS cluster can withstand the failure of one less than the value of `dfs.replication` before there will be service degradation, in that some files may not be served as all of the blocks are not available. In small to medium-sized clusters, 3 is a good value. For large clusters, it should be a number that is at least two larger than the expected number of machine failures per day. If you have a 1,000-machine cluster and expect to have no more than 5 machines fail at a time, setting the replication factor to 7 is reasonable. The disk storage requirements and the write network bandwidth used are multiplied by this number.

> **Note** When a client writes data to HDFS, each block is written to `dfs.replication` count DataNodes. Each node writes to the next node, to mitigate the network load on the individual machines. The nodes are selected more or less on a random basis, with some simple rules. If the origination machine is a DataNode for the HDFS cluster being written, one replica will go to that DataNode. HDFS has some concept of network topology, but this does not appear to be generally used as yet.

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

`dfs.block.size`: This parameter controls the basic file system block size for HDFS. HDFS is highly optimized for a medium number of very large files. The default value for an HDFS block is 64MB (67,108,864 bytes). Each file data block in the file system will be in a single file in the `dfs.data.dir` directory. A file in HDFS is composed of one or more data blocks. The last data block of a file may have less than the `dfs.block.size` bytes of data present. Any prior blocks will have `dfs.block.size` bytes of data. The block size may be specified on a per-file basis when the file is created. The individual blocks of the file are replicated onto `dfs.replication` DataNodes. The default input split size for a file being used as input in a MapReduce job is the `dfs.block.size` for the file. HDFS currently does not deal with very large numbers of files or very large numbers of blocks per DataNode.

Note Application developers are encouraged to use archives for storing multiple small files instead of creating them directly in HDFS. In one of my applications, there was a three-order-of-magnitude speed increase in an application when a set of zip files were constructed once per reduce task, instead of large numbers of small files in the reduce task.

Network I/O Tuning

Very little can be done to tune network I/O at the Hadoop Core level. The performance tuning must take place at the application layer and the hardware layer.

Hadoop Core does provide the ability to select the network to bind to for data services and the ability to specify an IP address to use for hostname resolution. This is ideal for clusters that have an internal network for data traffic and an external network for client communications. The following are the two relevant parameters:

`dfs.datanode.dns.interface`: This parameter may be set to an interface name, such as `eth0` or `en1`, and that interface will be used by the DataNode for HDFS communications.

`dfs.datanode.dns.nameserver`: This parameter may be set to provide an IP address to use for DNS-based hostname resolution, instead of the system default hostname resolution strategy.

If your installation has switch ports to spare and the switches support it, bonding your individual network connections can greatly increase the network bandwidth available to individual machines.

Caution The large-block HDFS traffic will dominate a network. It is best to isolate this traffic from your other traffic.

At the application layer, if your applications are data-intensive and your data is readily compressible, using block-or record-level compression may drastically reduce the I/O that the job requires. You can set this compression as follows:

```
FileOutput.setCompressOutput(jobConf,"true");
jobConf.setBoolean("mapred.output.compress",true);
```

The default compression codec will be used. Compressed input files that are not `SequenceFiles`, a Hadoop Core binary file format, will not be split, and a single task will handle a single file.

Recovery from Failure

Once you have configured your cluster and started running jobs, life will happen, and you may need to recover from a failure. Here, we will look at how HDFS protects from many individual failures and how to recover from other failures.

HDFS has two types of data, with different reliability requirements and recovery patterns.

- The NameNode stores the file system metadata and provides direct replication of the data and run-behind remote copies of the data.
- The DataNodes provide redundant storage by replicating data blocks to multiple DataNodes.

NameNode Recovery

The default Hadoop installation provides no protection from catastrophic NameNode server failures. The only default protection is that provided by a RAID 1 or RAID 5 storage device for the file system image and edit logs. You can avoid data loss and unrecoverable machine failures by running the secondary NameNode on an alternate machine. Storing the file system image and file system edit logs on multiple physical devices, or even multiple physical machines, also provides protection.

When a NameNode server fails, best practices require that all the JobTracker and Task-Trackers be restarted after the

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

NameNode is restarted. All incomplete HDFS blocks will be lost, but there should be no file or data loss for existing files or for completed blocks in actively written files.

> **Note** Hadoop 0.19.0 has initial support for a sync operator that allows the flushing of HDFS blocks that are not a full `dfs.block.size` value, but support for this is early and known to be unreliable. This feature has been disabled in Hadoop 0.19.1, but it may return in Hadoop 0.20.0.

The NameNode may be configured to write the metadata log to multiple locations on the host server's file system. In the event of data loss or corruption to one of these locations, the NameNode may be recovered by repairing or removing the failed location from the configuration, removing the data from that location, and restarting the NameNode. For rapid recovery, you may simply remove the failed location from the configuration and restart the NameNode.

If the NameNode needs to be recovered from a secondary NameNode, the procedure is somewhat more complex. Here are the steps:

1. Shut down the secondary NameNode.
2. Copy the contents of the `Secondary:fs.checkpoint.dir` to the `Namenode:dfs.name.dir`.
3. Copy the contents of the `Secondary: fs.checkpoint.edits.dir` to the `Namenode:dfs.name.edits.dir`.
4. When the copy completes, you may start the NameNode and restart the secondary NameNode.

All data written to HDFS after the last secondary NameNode checkpoint was taken will be removed and lost. The default frequency of the checkpoints is specified by the `fs.checkpoint.period` parameter

At present, there are no public forensic tools that will recover data from blocks on the DataNodes.

DataNode Recovery and Addition

The procedure for adding a new DataNode to a cluster and restarting a failed DataNode are identical and simple. The server process just needs to be started, assuming the configuration is correct, and the NameNode will integrate the new server or reintegrate a restarted server into the cluster.

> **Tip** As long as your cluster does not have underreplicated files and no file's replication count is less than 3, it is generally safe to forcibly remove a DataNode from the cluster by killing the DataNode server process. The next section covers how to decommission a DataNode gracefully.

The command `hadoop-daemon.sh start datanode` will start a DataNode server on a machine, if one is not already running. The configuration in the `conf` directory associated with the script will be used to determine the NameNode address and other configuration parameters.

If more DataNodes than the `dfs.replication` value fail, some file blocks will be unavailable. Your cluster will still be able to write files and access the blocks of the files that remain available. It is advisable to stop your MapReduce jobs by invoking the `stop-mapred.sh` script, as most applications do not deal well with partial dataset availability. When sufficient DataNodes have been returned to service, you may resume MapReduce job processing by invoking the `start-mapred.sh` script.

When you add new nodes, or return a node to service after substantial data has been written to HDFS, the added node may start up with substantially less utilization than the rest of the DataNodes in the cluster. Running the Balancer via `start-balancer.sh` will rebalance the blocks.

DataNode Decommissioning

A running DataNode sometimes needs to be decommissioned. While you may just shut down the DataNode, and the cluster will recover, there is a procedure for gracefully decommissioning a running DataNode. This procedure becomes particularly important if your cluster has underreplicated blocks or you need to decommission more nodes than your `dfs.replication` value.

> **Caution** You must not stop the NameNode during this process, or start this process while the NameNode is not running. The file specified by `dfs.hosts.exclude` has two purposes. One is to exclude the hosts from connecting to the NameNode, which takes effect if the parameter is set when the NameNode starts. The other starts the decommission process for the hosts, which takes place if the value is first seen after a Hadoop

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

```
dfsadmin -refreshNodes.
```

The procedure is as follows:

1. Create a file on the NameNode machine with the hostnames or IP addresses of the DataNodes you wish to decommission, say `/tmp/nodes_to_decommission`. This file should contain one hostname or IP address per line, with standard Unix line endings.

2. Modify the `hadoop-site.xml` file by adding, or updating the following block:

```
<property>
  <name>dfs.hosts.exclude</name>
  <value>/tmp/nodes_to_decommission</value>
  <description>Names a file that contains a list of hosts that are
  not permitted to connect to the namenode. The full pathname of the
  file must be specified.  If the value is empty, no hosts are
  excluded.</description>
</property>
```

3. Run the following command to start the decommissioning process:

```
hadoop dfsadmin -refreshNodes
```

4. When the process is complete, you will see a line in the NameNode log file like the following for each entry in the file: `tmp/nodes_to_decommission.`

```
Decommission complete for node IP:PORT
```

Deleted File Recovery

It is not uncommon for a user to accidentally delete large portions of the HDFS file system due to a program error or a command-line error. Unless your configuration has the delete-to-trash function enabled, via setting the parameter `fs.trash.interval` to a nonzero value, deletes are essentially immediate and forever.

If an erroneous large delete is in progress, your best bet is to terminate the NameNode and secondary NameNodes immediately, and then shut down the DataNodes. This will preserve as much data as possible. Use the procedures described earlier to recover from the secondary NameNode that has the edit log modification time closest to the time the deletion was started.

The `fs.trash.interval` determines how often the currently deleted files are moved to a date-stamped subdirectory of the deleting user's `.Trash` directory. The value is a time in minutes. Files that have not had a trash checkpoint will be under the `.Trash/current` directory in a path that is identical to their original path. Only one prior checkpoint is kept.

Troubleshooting HDFS Failures

The previous section dealt with the common and highly visible failure cases of a server process crashing or the machine hosting a server process failing. This section will cover how you can determine what has happened when the failure is less visible or why a server process is crashing.

There are a number of failures that can trip up an HDFS administrator or a MapReduce programmer. In the current state of development, it is not always clear from Hadoop's behavior or log messages what the failure or solution is. The usual first indication that something is in need of maintenance is a complaint from users that their jobs are performing poorly or failing, or a page from your installation monitoring tool such as Nagios.

NameNode Failures

The NameNode is the weak point in the highly available HDFS cluster. As noted earlier, currently there are no high-availability solutions for the NameNode. The NameNode has been designed to keep multiple copies of critical data on the local machine and close in time replicas on auxiliary machines. Let's examine how it can fail.

Out of Memory

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

The NameNode keeps all of the metadata for the file system in memory to speed request services. The NameNode also serializes directory listings before sending the result to requesting applications. The memory requirements grow with the number of files and the total number of blocks in the file system. If your file system has directories with many entries and applications are scanning the directory, this can cause a large transient increase in the memory requirements for the name server.

I once had a cluster that was using the Filesystem in Userspace (FUSE) `contrib` package to export HDFS as a read-only file system on a machine, which re-exported that file system via the Common Internet File System (CIFS) to a Windows server machine. The access patterns triggered repeated out-of-memory exceptions on the NameNode. (Using FUSE is discussed in Chapter 8.)

If the DataNodes are unreliable, and they are dropping out of service and then returning to service after a gap, the NameNode will build a large queue of blocks with invalid states. This may consume very large amounts of memory if large numbers of blocks become transitorily unavailable. For this problem, addressing the DataNode reliability is the only real solution, but increasing the memory size on the NameNode can help.

There are three solutions for NameNode out-of-memory problems:

- Increase the memory available to the NameNode, and ensure the machine has sufficient real memory to support this increase.
- Ensure that no directory has a large number of entries.
- Alter application access patterns so that there are not large numbers of directory listings.

For a 32-bit JVM, the maximum memory size is about 2.5GB, which will support a small HDFS cluster with a under a million files and blocks.

As a general rule, pin the full memory allocation by setting the starting heap size for the JVM to the maximum heap size.

Data Loss or Corruption

Ideally, the underlying disk storage used for the NameNode's file system image (`dfs.name.dir`) and edit log (`dfs.name.edits.dir`) should be a highly reliable storage system such as a RAID 1 or RAID 5 disk array with hot spares. Even so, catastrophic failures or user errors can result in data loss.

The NameNode configuration will accept a comma-separated list of directories and will maintain copies of the data in the full set of directories. This additional level of redundancy provides a current time backup of the file system metadata. The secondary NameNode provides a few-minutes-behind replica of the NameNode data.

If your configuration has multiple directories that contain the file system image or the edit log, and one of them is damaged, delete that directory's content and restart the NameNode. If the directory is unavailable, remove it from the list in the configuration and restart the NameNode.

If all of the `dfs.name.dir` directories are unavailable or suspect, do the following:

1. Archive the data if required.
2. Wipe all of the directories listed in `dfs.name.dir`.
3. Copy the contents of the `fs.checkpoint.dir` from the secondary NameNode to the `fs.checkpoint.dir` on the primary NameNode machine.
4. Run the following NameNode command:
   ```
   hadoop namenode -importCheckpoint
   ```

If there is no good copy of the NameNode data, the secondary NameNode image may be imported. The secondary NameNode takes periodic snapshots, at `fs.checkpoint.period` intervals, so it is not as current as the NameNode data.

You may simply copy the file system image from the secondary NameNode to a file system image directory on the NameNode, and then restart.

As the imported data is older than the current state of the HDFS file system, the NameNode may spend significant time in

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM Apress, Jason Venner (c) 2009, Copying Prohibited

safe mode as it brings the HDFS block store into consistency with restored snapshot.

No Live Node Contains Block Errors

Usually, if you see the `no live node contains block` error, it will be in the log files for your applications. It means that the client code in your application that interfaces with HDFS was unable to find a block of a requested file. For this error to occur, the client code received a list of DataNode and block ID pairs from the NameNode, and was unable to retrieve the block from any of the DataNodes.

This error commonly occurs when the application is unable to open a connection to any of the DataNodes. This may be because there are no more file descriptors available, there is a DNS resolution failure, there is a network problem, or all of the DataNodes in question are actually unavailable. The most common case is the out-of-file-descriptor situation. This may be corrected by increasing the number of file descriptors available, as described earlier in this chapter. An alternative is to minimize unclosed file descriptors in the applications.

I've seen DNS resolution failures transiently appear, and as a general rule, I now use IP addresses instead of hostnames in the configuration files. It is very helpful if the DNS reverse lookup returns the same name as the hostname of the machine.

Write Failed

If there are insufficient DataNodes available to allow full replication of a newly written block, the write will not be allowed to complete. This may result in a zero-length or incomplete file, which will need to be manually removed.

DataNode or NameNode Pauses

Through at least Hadoop 0.19.0, the DataNode has two periodic tasks that do a linear scan of all of the data blocks stored by the DataNode. If this process starts taking longer than a small number of minutes, the DataNode will be marked as disconnected by the NameNode.

When a DataNode is marked as disconnected, the NameNode queues all of the blocks that had a replica on that DataNode for replication. If the number of blocks is large, the NameNode may pause for a noticeable period while queuing the blocks.

The only solutions for this at present are to add enough DataNodes, so that no DataNode has more than a few hundred thousand data blocks, or to alter your application's I/O patterns to use Hadoop archives or zip files to pack many small HDFS subblock-size files into single HDFS files. The latter approach results in a reduction in the number of blocks stored in HDFS and the number of blocks per DataNode.

A simple way to work out how many blocks is too many is to run the following on a DataNode:

```
time find dfs.data.dir -ls > /dev/null
```

If it takes longer than a few hundred seconds, you are in the danger zone. If it takes longer than a few minutes, you are in the pain zone. Replace `dfs.data.dir` with an expanded value from your Hadoop configuration. The timeout interval is 600 seconds, and hard-coded. If your `ls` takes anywhere close to that, your cluster will be unstable.

> **Note** The NameNode may pause if the one of the directories used for the `dfs.name.edits.dir` or `dfs.name.dir` is taking time to complete writes.

Summary

HDFS is a wonderful global file system for a medium number of very large files. With reasonable care and an understanding of HDFS's limitations it will serve you well.

HDFS is not a general-purpose file system to be used for large numbers of small files or for rapid creation and deletion of files.

Through HDFS 0.19.0, the HDFS NameNode is a single point of failure and needs careful handling to minimize the risk of data loss. Using a separate machine for your secondary NameNode, and having multiple devices for the file system image and edit logs, will go a long way toward providing a fail-safe, rapid recovery solution.

Monitoring your DataNode's block totals will go a long way toward avoiding congestion collapse issues in your HDFS. You can do this by simply running a `find` on the `dfs.data.dir` and making sure that it takes less than a couple of minutes.

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM
Apress, Jason Venner (c) 2009, Copying Prohibited

Ensuring that your HDFS data traffic is segregated from your normal application traffic and crosses as few interswitch backhauls as possible will help to avoid network congestion and application misbehavior.

Ultimately, remember that HDFS is designed for a small to medium number of very large files, and not for transitory storage of large numbers of small files.

Reprinted for ibm\jpkuhlen@us.ibm.com, IBM

Apress, Jason Venner (c) 2009, Copying Prohibited