FUTURE VISION BIE

By K B Hemanth Raj

Visit : https://hemanthrajhemu.github.io

A Small Contribution Would Support Us.

Dear Viewer,

Future Vision BIE is a free service and so that any Student/Research Personal **Can Access Free of Cost**.

If you would like to say **thanks**, you can make a **small contribution** to the author of this site.

Contribute whatever you feel this is worth to you. This gives **us support** & to bring **Latest Study Material** to you. After the Contribution Fill out this Form (https://forms.gle/tw3T3bUVpLXL8omX7). To Receive a **Paid E-Course for Free**, from our End within 7 Working Days.

Regards

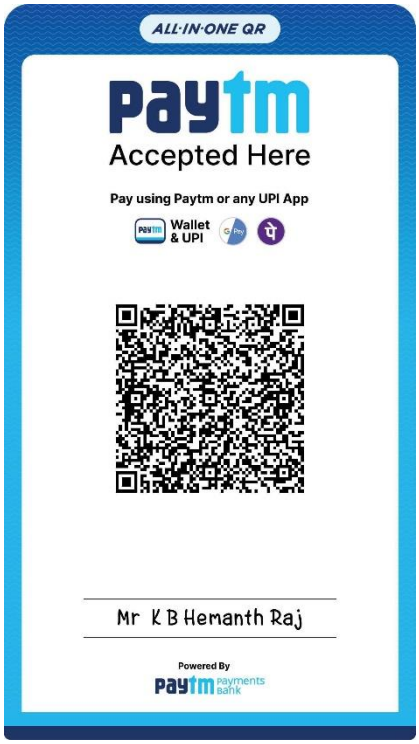
- K B Hemanth Raj (Admin)

Contribution Methods

UPI ID

1. futurevisionbie@oksbi
2. futurevisionbie@paytm

Scan & Pay



Account Transfer

Account Holder's Name: K B Hemanth Raj

Account Number: 39979402438

IFSC Code: SBIN0003982

MICR Code: 560002017

More Info: https://hemanthrajhemu.github.io/Contribution/

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering & MORE...

Stay Connected... get Updated... ask your queries...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Hadoop® 2 Quick-Start Guide

Learn the Essentials of Big Data Computing in the Apache Hadoop® 2 Ecosystem

Douglas Eadline

3. Hadoop Distributed File System Basics

In This Chapter:

- The design and operation of the Hadoop Distributed File System (HDFS) are presented.
- Important HDFS topics such as block replication, Safe Mode, rack awareness, High Availability, Federation, backup, snapshots, NFS mounting, and the HDFS web GUI are discussed.
- Examples of basic HDFS user commands are provided.
- HDFS programming examples using Java and C are provided.

The Hadoop Distributed File System is the backbone of Hadoop MapReduce processing. New users and administrators often find HDFS different than most other UNIX/Linux file systems. This chapter highlights the design goals and capabilities of HDFS that make it useful for Big Data processing.

HADOOP DISTRIBUTED FILE SYSTEM DESIGN FEATURES

The Hadoop Distributed File System (HDFS) was designed for Big Data processing. Although capable of supporting many users simultaneously, HDFS is not designed as a true parallel file system. Rather, the design assumes a large file write-once/read-many model that enables other optimizations and relaxes many of the concurrency and coherence overhead requirements of a true parallel file system. For instance, HDFS rigorously restricts data writing to one user at a time. All additional writes are "append-only," and there is no random writing to HDFS files. Bytes are always appended to the end of a stream, and byte streams are guaranteed to be stored in the order written.

The design of HDFS is based on the design of the Google File System (GFS). A paper published by Google provides further background on GFS (http://research.google.com/archive/gfs.html).

HDFS is designed for data streaming where large amounts of data are read from disk in bulk. The HDFS block size is typically 64MB or 128MB. Thus, this approach is entirely unsuitable for standard POSIX file system use. In addition, due to the sequential nature of the data, there is no local caching mechanism. The large block and file sizes make it more efficient to reread data from HDFS than to try to cache the data.

Perhaps the most interesting aspect of HDFS—and the one that separates it from other file systems—is its data locality. A principal design aspect of Hadoop MapReduce is the emphasis on moving the computation to the data rather than moving the data to the computation. This distinction is reflected in how Hadoop clusters are implemented. In other high-performance systems, a parallel file system will exist on hardware separate from the compute hardware. Data is then moved to and from the computer components via high-speed interfaces to the parallel file system array. HDFS, in contrast, is designed to work on the same hardware as the compute portion of the cluster. That is, a single server node in the cluster is often both a computation engine and a storage engine for the application.

Finally, Hadoop clusters assume node (and even rack) failure will occur at some point. To deal with this situation, HDFS has a redundant design that can tolerate system failure and still provide the data needed by the compute part of the program.

The following points summarize the important aspects of HDFS:

- The write-once/read-many design is intended to facilitate streaming reads.
- Files may be appended, but random seeks are not permitted. There is no caching of data.
- Converged data storage and processing happen on the same server nodes.
- "Moving computation is cheaper than moving data."
- A reliable file system maintains multiple copies of data across the cluster. Consequently, failure of a single node (or even a rack in a large cluster) will not bring down the file system.
- A specialized file system is used, which is not designed for general use.

HDFS COMPONENTS

The design of HDFS is based on two types of nodes: a NameNode and multiple DataNodes. In a basic design, a single NameNode manages all the metadata needed to store and retrieve the actual data from the DataNodes. No data is actually stored on the NameNode, however. For a minimal Hadoop installation, there needs to be a single NameNode daemon and a single DataNode daemon running on at least one machine (see the section "Installing Hadoop from Apache Sources" in Chapter 2, "Installation Recipes").

The design is a master/slave architecture in which the master (NameNode) manages the file system namespace and regulates access to files by clients. File system namespace operations such as opening, closing, and renaming files and directories are all managed by the NameNode. The NameNode also determines the mapping of blocks to DataNodes and handles DataNode failures.

The slaves (DataNodes) are responsible for serving read and write requests from the file system to the clients. The NameNode manages block creation, deletion, and replication.

An example of the client/NameNode/DataNode interaction is provided in Figure 3.1. When a client writes data, it first communicates with the NameNode and requests to create a file. The NameNode determines how many blocks are needed and provides the client with the DataNodes that will store the data. As part of the storage process, the data blocks are replicated after they are written to the assigned node. Depending on how many nodes are in the cluster, the NameNode will attempt to write replicas of the data blocks on nodes that are in other separate racks (if possible). If there is only one rack, then the replicated blocks are written to other servers in the same rack. After the DataNode acknowledges that the file block replication is complete, the client closes the file and informs the NameNode that the operation is complete. Note that the NameNode does not write any data directly to the DataNodes. It does, however, give the client a limited amount of time to complete the operation. If it does not complete in the time period, the operation is canceled.

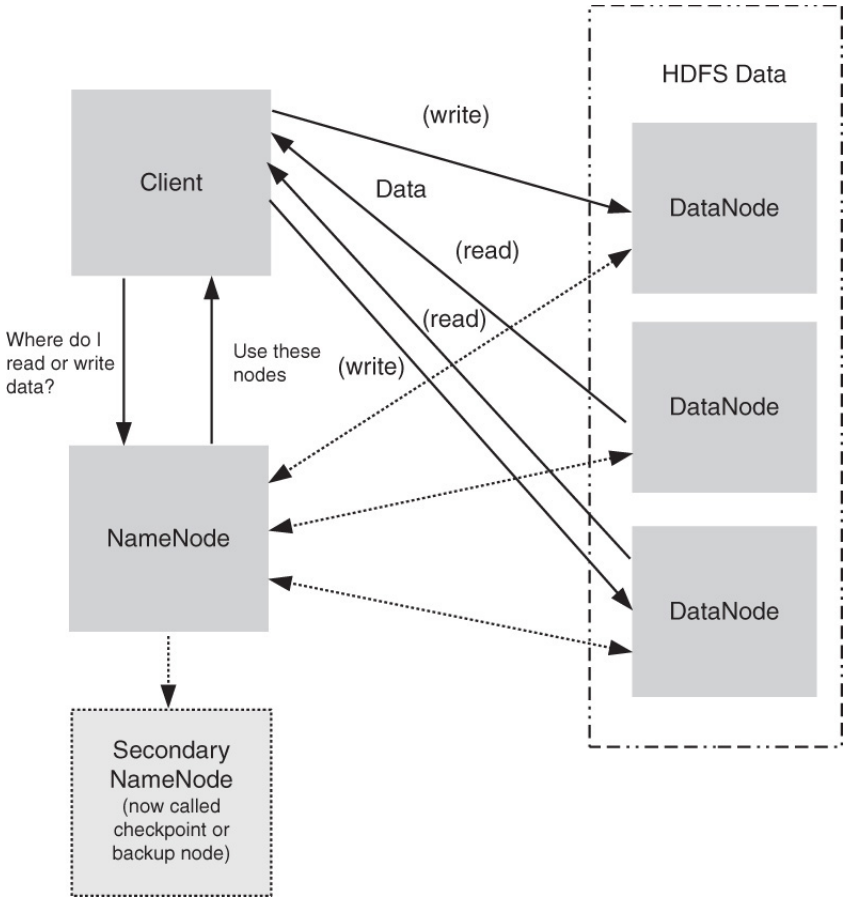


Figure 3.1 Various system roles in an HDFS deployment

Reading data happens in a similar fashion. The client requests a file from the NameNode, which returns the best DataNodes from which to read the data. The client then accesses the data directly from the DataNodes.

Thus, once the metadata has been delivered to the client, the NameNode steps back and lets the conversation between the client and the DataNodes proceed. While data transfer is progressing, the NameNode also monitors the DataNodes by listening for heartbeats sent from DataNodes. The lack of a heartbeat signal indicates a potential node failure. In such a case, the NameNode will route around the failed DataNode and begin re-replicating the now-missing blocks. Because the file system is redundant, DataNodes can be taken offline (decommissioned) for maintenance by informing the NameNode of the DataNodes to exclude from the HDFS pool.

The mappings between data blocks and the physical DataNodes are not kept in persistent storage on the NameNode. For performance reasons, the NameNode stores all metadata in memory. Upon startup, each DataNode provides a block report (which it keeps in persistent storage) to the NameNode. The block reports are sent every 10 heartbeats. (The interval between reports is a configurable property.) The reports enable the NameNode to keep an up-to-date account of all data blocks in the cluster.

In almost all Hadoop deployments, there is a SecondaryNameNode. While not explicitly required by a NameNode, it is highly recommended. The term "SecondaryNameNode" (now called CheckPointNode) is somewhat misleading. It is not an active failover node and cannot replace

the primary NameNode in case of its failure. (See the section "NameNode High Availability" later in this chapter for more explanation.)

The purpose of the SecondaryNameNode is to perform periodic checkpoints that evaluate the status of the NameNode. Recall that the NameNode keeps all system metadata memory for fast access. It also has two disk files that track changes to the metadata:

- An image of the file system state when the NameNode was started. This file begins with `fsimage_*`and is used only at startup by the NameNode.
- A series of modifications done to the file system after starting the NameNode. These files begin with `edit_*` and reflect the changes made after the `fsimage_*` file was read.

The location of these files is set by the `dfs.namenode.name.dir` property in the `hdfs-site.xml`file.

The SecondaryNameNode periodically downloads `fsimage` and edits files, joins them into a new `fsimage`, and uploads the new `fsimage` file to the NameNode. Thus, when the NameNode restarts, the `fsimage` file is reasonably up-to-date and requires only the edit logs to be applied since the last checkpoint. If the SecondaryNameNode were not running, a restart of the NameNode could take a prohibitively long time due to the number of changes to the file system.

Thus, the various roles in HDFS can be summarized as follows:

- HDFS uses a master/slave model designed for large file reading/streaming.
- The NameNode is a metadata server or "data traffic cop."
- HDFS provides a single namespace that is managed by the NameNode.
- Data is redundantly stored on DataNodes; there is no data on the NameNode.
- The SecondaryNameNode performs checkpoints of NameNode file system's state but is not a failover node.

HDFS Block Replication

As mentioned, when HDFS writes a file, it is replicated across the cluster. The amount of replication is based on the value of `dfs.replication` in the `hdfs-site.xml` file. This default value can be overruled with the `hdfs dfs-setrep` command. For Hadoop clusters containing more than eight DataNodes, the replication value is usually set to 3. In a Hadoop cluster of eight or fewer DataNodes but more than one DataNode, a replication factor of 2 is adequate. For a single machine, like the pseudo-distributed install in Chapter 2, the replication factor is set to 1.

If several machines must be involved in the serving of a file, then a file could be rendered unavailable by the loss of any one of those machines. HDFS combats this problem by replicating each block across a number of machines (three is the default).

In addition, the HDFS default block size is often 64MB. In a typical operating system, the block size is 4KB or 8KB. The HDFS default block size is not the minimum block size, however. If a 20KB file is written to HDFS, it will create a block that is approximately 20KB in size. (The underlying file system may have a minimal block size that increases the actual file size.) If a file of size 80MB is written to HDFS, a 64MB block and a 16MB block will be created.

As mentioned in Chapter 1, "Background and Concepts," HDFS blocks are not exactly the same as the data splits used by the MapReduce process. The HDFS blocks are based on size, while the splits are based on a logical partitioning of the data. For instance, if a file contains discrete

records, the logical split ensures that a record is not split physically across two separate servers during processing. Each HDFS block may consist of one or more splits.

Figure 3.2 provides an example of how a file is broken into blocks and replicated across the cluster. In this case, a replication factor of 3 ensures that any one DataNode can fail and the replicated blocks will be available on other nodes—and then subsequently re-replicated on other DataNodes.

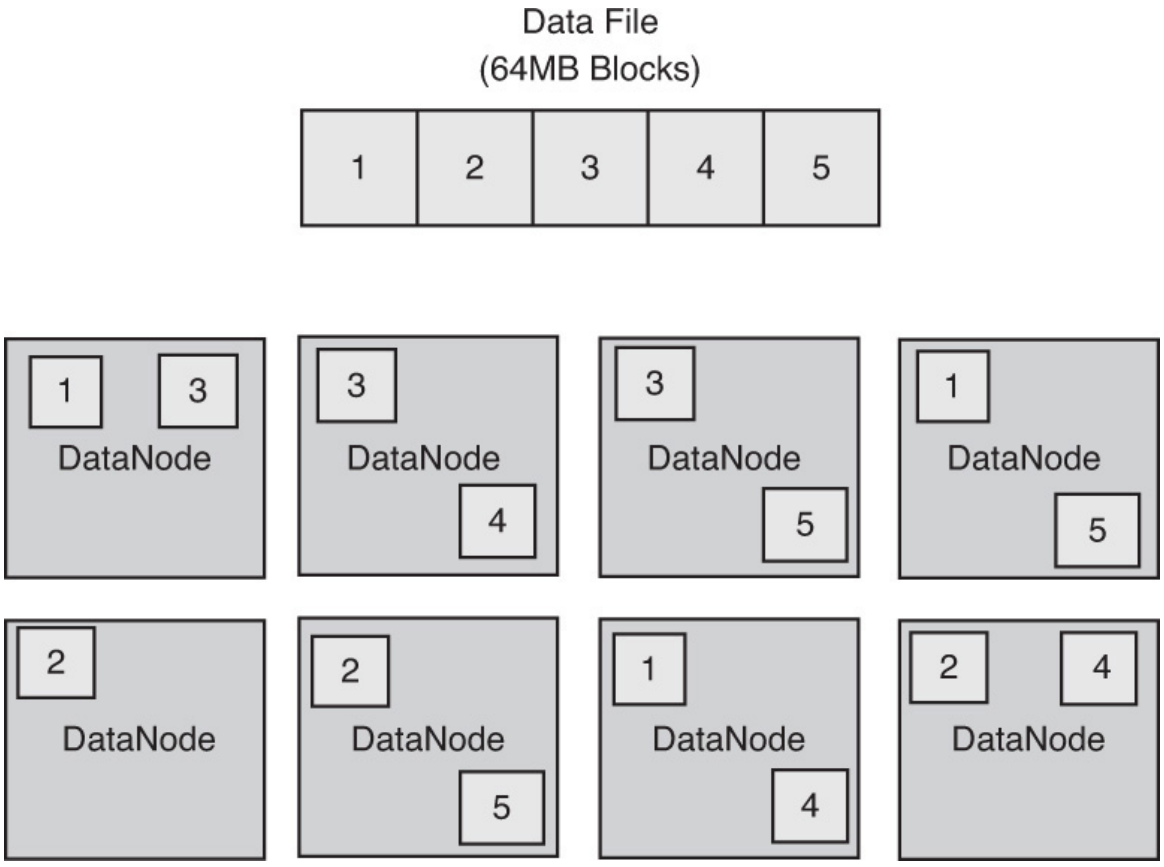


Figure 3.2 HDFS block replication example

HDFS Safe Mode

When the NameNode starts, it enters a read-only *safe mode* where blocks cannot be replicated or deleted. Safe Mode enables the NameNode to perform two important processes:

1. The previous file system state is reconstructed by loading the `fsimage` file into memory and replaying the edit log.

2. The mapping between blocks and data nodes is created by waiting for enough of the DataNodes to register so that at least one copy of the data is available. Not all DataNodes are required to register before HDFS exits from Safe Mode. The registration process may continue for some time.

HDFS may also enter Safe Mode for maintenance using the `hdfs dfsadmin-safemode` command or when there is a file system issue that must be addressed by the administrator.

Rack Awareness

Rack awareness deals with data locality. Recall that one of the main design goals of Hadoop MapReduce is to move the computation to the data. Assuming that most data center networks do not offer full bisection bandwidth, a typical Hadoop cluster will exhibit three levels of data locality:

1. Data resides on the local machine (best).

2. Data resides in the same rack (better).

3. Data resides in a different rack (good).

When the YARN scheduler is assigning MapReduce containers to work as mappers, it will try to place the container first on the local machine, then on the same rack, and finally on another rack.

In addition, the NameNode tries to place replicated data blocks on multiple racks for improved fault tolerance. In such a case, an entire rack failure will not cause data loss or stop HDFS from working. Performance may be degraded, however.

HDFS can be made rack-aware by using a user-derived script that enables the master node to map the network topology of the cluster. A default Hadoop installation assumes all the nodes belong to the same (large) rack. In that case, there is no option 3.

NameNode High Availability

With early Hadoop installations, the NameNode was a single point of failure that could bring down the entire Hadoop cluster. NameNode hardware often employed redundant power supplies and storage to guard against such problems, but it was still susceptible to other failures. The solution was to implement NameNode High Availability (HA) as a means to provide true failover service.

As shown in Figure 3.3, an HA Hadoop cluster has two (or more) separate NameNode machines. Each machine is configured with exactly the same software. One of the NameNode machines is in the Active state, and the other is in the Standby state. Like a single NameNode cluster, the Active NameNode is responsible for all client HDFS operations in the cluster. The Standby NameNode maintains enough state to provide a fast failover (if required).

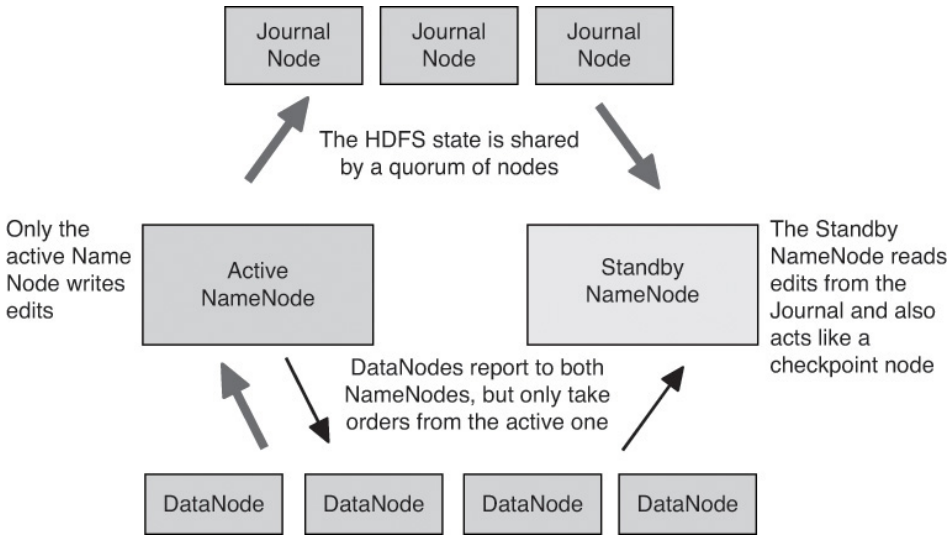


Figure 3.3 HDFS High Availability design

To guarantee the file system state is preserved, both the Active and Standby NameNodes receive block reports from the DataNodes. The Active node also sends all file system edits to a quorum of Journal nodes. At least three physically separate JournalNode daemons are required, because edit log modifications must be written to a majority of the JournalNodes. This design will enable the system to tolerate the failure of a single JournalNode machine. The Standby node continuously reads the edits from the JournalNodes to ensure its namespace is synchronized with that of the Active node. In the event of an Active NameNode failure, the Standby node reads all remaining edits from the JournalNodes before promoting itself to the Active state.

To prevent confusion between NameNodes, the JournalNodes allow only one NameNode to be a writer at a time. During failover, the NameNode that is chosen to become active takes over the role of writing to the JournalNodes. A SecondaryNameNode is not required in the HA configuration because the Standby node also performs the tasks of the Secondary NameNode.

Apache ZooKeeper is used to monitor the NameNode health. Zookeeper is a highly available service for maintaining small amounts of coordination data, notifying clients of changes in that data, and monitoring clients for failures. HDFS failover relies on ZooKeeper for failure detection and for Standby to Active NameNode election. The Zookeeper components are not depicted in Figure 3.3.

HDFS NameNode Federation

Another important feature of HDFS is NameNode Federation. Older versions of HDFS provided a single namespace for the entire cluster managed by a single NameNode. Thus, the resources of a single NameNode determined the size of the namespace. Federation addresses this limitation by adding support for multiple NameNodes/namespaces to the HDFS file system. The key benefits are as follows:

- *Namespace scalability.* HDFS cluster storage scales horizontally without placing a burden on the NameNode.
- *Better performance.* Adding more NameNodes to the cluster scales the file system read/write operations throughput by separating the total namespace.
- *System isolation.* Multiple NameNodes enable different categories of applications to be distinguished, and users can be isolated to different namespaces.

Figure 3.4 illustrates how HDFS NameNode Federation is accomplished. NameNode1 manages the `/research` and `/marketing` namespaces, and NameNode2 manages the `/data` and `/project`namespaces. The NameNodes do not communicate with each other and the DataNodes "just store data block" as directed by either NameNode.

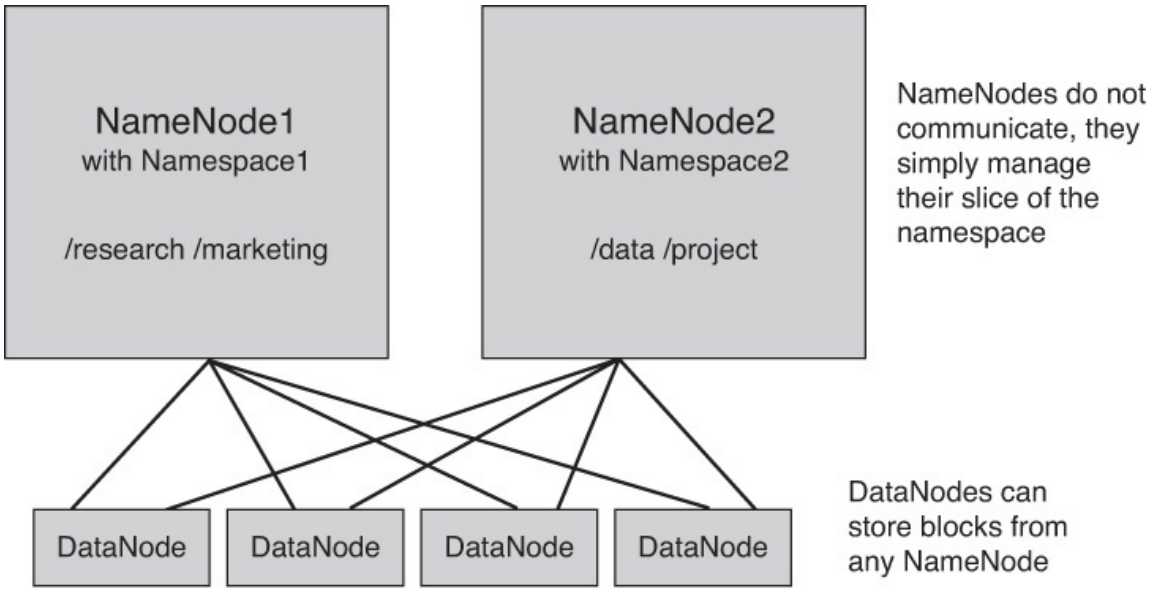


Figure 3.4 HDFS NameNode Federation example

HDFS Checkpoints and Backups

As mentioned earlier, the NameNode stores the metadata of the HDFS file system in a file called `fsimage`. File systems modifications are written to an edits log file, and at startup the NameNode merges the edits into a new `fsimage`. The SecondaryNameNode or CheckpointNode periodically fetches edits from the NameNode, merges them, and returns an updated `fsimage` to the NameNode.

An HDFS BackupNode is similar, but also maintains an up-to-date copy of the file system namespace both in memory and on disk. Unlike a CheckpointNode, the BackupNode does not need to download the `fsimage` and edits files from the active NameNode because it already has an up-to-date namespace state in memory. A NameNode supports one BackupNode at a time. No CheckpointNodes may be registered if a Backup node is in use.

HDFS Snapshots

HDFS snapshots are similar to backups, but are created by administrators using the `hdfs dfs snapshot` command. HDFS snapshots are read-only point-in-time copies of the file system. They offer the following features:

- Snapshots can be taken of a sub-tree of the file system or the entire file system.
- Snapshots can be used for data backup, protection against user errors, and disaster recovery.
- Snapshot creation is instantaneous.
- Blocks on the DataNodes are not copied, because the snapshot files record the block list and the file size. There is no data copying, although it appears to the user that there are duplicate files.
- Snapshots do not adversely affect regular HDFS operations.

See Chapter 10, "Basic Hadoop Administration Procedures," for information on creating HDFS snapshots.

HDFS NFS Gateway

The HDFS NFS Gateway supports NFSv3 and enables HDFS to be mounted as part of the client's local file system. Users can browse the HDFS file system through their local file systems that provide an NFSv3 client compatible operating system. This feature offers users the following capabilities:

- Users can easily download/upload files from/to the HDFS file system to/from their local file system.
- Users can stream data directly to HDFS through the mount point. Appending to a file is supported, but random write capability is not supported.

Mounting a HDFS over NFS is explained in Chapter 10, "Basic Hadoop Administration Procedures."

HDFS USER COMMANDS

The following is a brief command reference that will facilitate navigation within HDFS. Be aware that there are alternative options for each command and that the examples given here are simple use-cases. What follows is by no means a full description of HDFS functionality. For more information, see the section "Summary and Additional Resources" at the end of the chapter.

Brief HDFS Command Reference

The preferred way to interact with HDFS in Hadoop version 2 is through the `hdfs` command. Previously, in version 1 and subsequently in many Hadoop examples, the `hadoop dfs` command was used to manage files in HDFS. The `hadoop dfs` command will still work in version 2, but its use will cause a message to be displayed indicating that the use of `hadoop dfs` is deprecated.

The following listing presents the full range of options that are available for the `hdfs` command. In the next section, only portions of the `dfs` and `hdfsadmin` options are explored. Chapter 10, "Basic Hadoop Administration Procedures," provides examples of administration with the `hdfs` command.

```
Usage: hdfs [--config confdir] COMMAND
      where COMMAND is one of:
  dfs                  run a file system command on the file systems
                       supported in Hadoop.
  namenode -format     format the DFS file system
  secondarynamenode    run the DFS secondary namenode
  namenode             run the DFS namenode
  journalnode          run the DFS journalnode
  zkfc                 run the ZK Failover Controller daemon
  datanode             run a DFS datanode
  dfsadmin             run a DFS admin client
  haadmin              run a DFS HA admin client
  fsck                 run a DFS file system checking utility
  balancer             run a cluster balancing utility
  jmxget               get JMX exported values from NameNode or DataNode.
```

```
mover               run a utility to move block replicas across
                    storage types
oiv                 apply the offline fsimage viewer to an fsimage
oiv_legacy          apply the offline fsimage viewer to an legacy
                    fsimage
oev                 apply the offline edits viewer to an edits file
fetchdt             fetch a delegation token from the NameNode
getconf             get config values from configuration
groups              get the groups which users belong to
snapshotDiff        diff two snapshots of a directory or diff the
                    current directory contents with a snapshot
lsSnapshottableDir  list all snapshottable dirs owned by the current
                    user Use -help to see options
portmap             run a portmap service
nfs3                run an NFS version 3 gateway
cacheadmin          configure the HDFS cache
crypto              configure HDFS encryption zones
storagepolicies     get all the existing block storage policies
version             print the version
```

Most commands print help when invoked w/o parameters.

General HDFS Commands

The version of HDFS can be found from the version option. Examples in this section are run on the HDFS version shown here:

```
$ hdfs version
Hadoop 2.6.0.2.2.4.2-2
Subversion git@github.com:hortonworks/hadoop.git -r
22a563ebe448969d07902aed869ac13c652b2872
Compiled by jenkins on 2015-03-31T19:49Z
Compiled with protoc 2.5.0
From source with checksum b3481c2cdbe2d181f2621331926e267
This command was run using /usr/hdp/2.2.4.2-2/hadoop/hadoop-
common-2.6.0.2.2.4.2-2.jar
```

HDFS provides a series of commands similar to those found in a standard POSIX file system. A list of those commands can be obtained by issuing the following command. Several of these commands will be highlighted here under the user account `hdfs`.

```
$ hdfs dfs
Usage: hadoop fs [generic options]
        [-appendToFile <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
        [-checksum <src> ...]
        [-chgrp [-R] GROUP PATH...]
```

```
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-copyFromLocal [-f] [-p] [-l] <localsrc> ... <dst>]
        [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-count [-q] [-h] <path> ...]
        [-cp [-f] [-p | -p[topax]] <src> ... <dst>]
        [-createSnapshot <snapshotDir> [<snapshotName>]]
        [-deleteSnapshot <snapshotDir> <snapshotName>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] <path> ...]
        [-expunge]
        [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-getfacl [-R] <path>]
        [-getfattr [-R] {-n name | -d} [-e en] <path>]
        [-getmerge [-nl] <src> <localdst>]
        [-help [cmd ...]]
        [-ls [-d] [-h] [-R] [<path> ...]]
        [-mkdir [-p] <path> ...]
        [-moveFromLocal <localsrc> ... <dst>]
        [-moveToLocal <src> <localdst>]
        [-mv <src> ... <dst>]
        [-put [-f] [-p] [-l] <localsrc> ... <dst>]
        [-renameSnapshot <snapshotDir> <oldName> <newName>]
        [-rm [-f] [-r|-R] [-skipTrash] <src> ...]
        [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
        [-setfacl [-R] [{-b|-k} {-m|-x <acl_spec>} <path>]|[--set
          <acl_spec> <path>]]
        [-setfattr {-n name [-v value] | -x name} <path>]
        [-setrep [-R] [-w] <rep> <path> ...]
        [-stat [format] <path> ...]
        [-tail [-f] <file>]
        [-test -[defsz] <path>]
        [-text [-ignoreCrc] <src> ...]
        [-touchz <path> ...]
        [-truncate [-w] <length> <path> ...]
        [-usage [cmd ...]]

Generic options supported are
-conf <configuration file>     specify an application configuration file
-D <property=value>            use value for given property
-fs <local|namenode:port>      specify a namenode
-jt <local|resourcemanager:port>    specify a ResourceManager
-files <comma separated list of files>    specify comma separated files to
                                be copied to the map reduce cluster
-libjars <comma separated list of jars>    specify comma separated jar
                                files to include in the classpath.
```

-archives <comma separated list of archives> specify comma separated
archives to be unarchived
on the compute machines.

The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]

List Files in HDFS

To list the files in the root HDFS directory, enter the following command:

```
$ hdfs dfs -ls /
```

```
Found 10 items
drwxrwxrwx   - yarn   hadoop         0 2015-04-29 16:52 /app-logs
drwxr-xr-x   - hdfs   hdfs           0 2015-04-21 14:28 /apps
drwxr-xr-x   - hdfs   hdfs           0 2015-05-14 10:53 /benchmarks
drwxr-xr-x   - hdfs   hdfs           0 2015-04-21 15:18 /hdp
drwxr-xr-x   - mapred hdfs           0 2015-04-21 14:26 /mapred
drwxr-xr-x   - hdfs   hdfs           0 2015-04-21 14:26 /mr-history
drwxr-xr-x   - hdfs   hdfs           0 2015-04-21 14:27 /system
drwxrwxrwx   - hdfs   hdfs           0 2015-05-07 13:29 /tmp
drwxr-xr-x   - hdfs   hdfs           0 2015-04-27 16:00 /user
drwx-wx-wx   - hdfs   hdfs           0 2015-05-27 09:01 /var
```

To list files in your home directory, enter the following command:

```
$ hdfs dfs -ls
```

```
Found 13 items
drwx------   - hdfs hdfs          0 2015-05-27 20:00 .Trash
drwx------   - hdfs hdfs          0 2015-05-26 15:43 .staging
drwxr-xr-x   - hdfs hdfs          0 2015-05-28 13:03 DistributedShell
drwxr-xr-x   - hdfs hdfs          0 2015-05-14 09:19 TeraGen-50GB
drwxr-xr-x   - hdfs hdfs          0 2015-05-14 10:11 TeraSort-50GB
drwxr-xr-x   - hdfs hdfs          0 2015-05-24 20:06 bin
drwxr-xr-x   - hdfs hdfs          0 2015-04-29 16:52 examples
drwxr-xr-x   - hdfs hdfs          0 2015-04-27 16:00 flume-channel
drwxr-xr-x   - hdfs hdfs          0 2015-04-29 14:33 oozie-4.1.0
drwxr-xr-x   - hdfs hdfs          0 2015-04-30 10:35 oozie-examples
drwxr-xr-x   - hdfs hdfs          0 2015-04-29 20:35 oozie-oozi
drwxr-xr-x   - hdfs hdfs          0 2015-05-24 18:11 war-and-peace-input
drwxr-xr-x   - hdfs hdfs          0 2015-05-25 15:22 war-and-peace-output
```

The same result can be obtained by issuing the following command:

```
$ hdfs dfs -ls /user/hdfs
```

Make a Directory in HDFS

To make a directory in HDFS, use the following command. As with the `-ls` command, when no path is supplied, the user's home directory is used (e.g., `/users/hdfs`).

```
$ hdfs dfs -mkdir stuff
```

Copy Files to HDFS

To copy a file from your current local directory into HDFS, use the following command. If a full path is not supplied, your home directory is assumed. In this case, the file `test` is placed in the directory `stuff` that was created previously.

```
$ hdfs dfs -put test stuff
```

The file transfer can be confirmed by using the `-ls` command:

```
$ hdfs dfs -ls stuff

Found 1 items
-rw-r--r--   2 hdfs hdfs      12857 2015-05-29 13:12 stuff/test
```

Copy Files from HDFS

Files can be copied back to your local file system using the following command. In this case, the file we copied into HDFS, `test`, will be copied back to the current local directory with the name `test-local`.

Click here to view code image

```
$ hdfs dfs -get stuff/test test-local
```

Copy Files within HDFS

The following command will copy a file in HDFS:

```
$ hdfs dfs -cp stuff/test test.hdfs
```

Delete a File within HDFS

The following command will delete the HDFS file `test.dhfs` that was created previously:

```
$ hdfs dfs -rm test.hdfs

Moved: 'hdfs://limulus:8020/user/hdfs/stuff/test' to trash at: hdfs://
limulus:8020/user/hdfs/.Trash/Current
```

Note that when the `fs.trash.interval` option is set to a non-zero value in `core-site.xml`, all deleted files are moved to the user's `.Trash` directory. This can be avoided by including the `-skipTrash` option.

```
$ hdfs dfs -rm -skipTrash stuff/test

Deleted stuff/test
```

Delete a Directory in HDFS

The following command will delete the HDFS directory `stuff` and all its contents:

```
$ hdfs dfs -rm -r -skipTrash stuff
Deleted stuff
```

Get an HDFS Status Report

Regular users can get an abbreviated HDFS status report using the following command. Those with HDFS administrator privileges will get a full (and potentially long) report. Also, this command uses `dfsadmin` instead of `dfs` to invoke administrative commands. The status report is similar to the data presented in the HDFS web GUI (see the section "HDFS Web GUI").

```
$ hdfs dfsadmin -report

Configured Capacity: 1503409881088 (1.37 TB)
Present Capacity: 1407945981952 (1.28 TB)
DFS Remaining: 1255510564864 (1.14 TB)
DFS Used: 152435417088 (141.97 GB)
DFS Used%: 10.83%
Under replicated blocks: 54
Blocks with corrupt replicas: 0
Missing blocks: 0

-------------------------------------------------
report: Access denied for user deadline. Superuser privilege is required
```

HDFS WEB GUI

HDFS provides an informational web interface. Starting the interface is described in Chapter 2, "Installation Recipes," in the section "Install Hadoop from Apache Sources." The interface can also be started from inside the Ambari management interface (see Chapter 9, "Managing Hadoop with Apache Ambari"). HDFS must be started and running on the cluster before the GUI can be used. It may be helpful to examine the information reported by the GUI as a way to explore some of the HDFS concepts presented in this chapter. The HDFS web GUI is described further in Chapter 10, "Basic Hadoop Administration Procedures."

USING HDFS IN PROGRAMS

This section describes two methods for writing user applications that can use HDFS from within programs.

HDFS Java Application Example

When using Java, reading from and writing to Hadoop DFS is no different from the corresponding operations with other file systems. The code in Listing 3.1 is an example of reading, writing, and deleting files from HDFS, as well as making directories. The example is available from the book download page (see Appendix A, "Book Webpage and Code Download") or from http://wiki.apache.org/hadoop/HadoopDfsReadWriteExample.

To be able to read from or write to HDFS, you need to create a Configuration object and pass configuration parameters to it using Hadoop configuration files. The example in Listing 3.1 assumes the Hadoop configuration files are in `/etc/hadoop/conf`. If you do not assign the configuration objects to the local Hadoop XML files, your HDFS operation will be performed on the local file system and not on the HDFS.

Listing 3.1 **HadoopDFSFileReadWrite.java**

```
package org.myorg;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class HDFSClient {
    public HDFSClient() {
    }
    public void addFile(String source, String dest) throws
IOException {
        Configuration conf = new Configuration();
        // Conf object will read the HDFS configuration
parameters from
        // these XML files.
        conf.addResource(new Path("/etc/hadoop/conf/core-
site.xml"));
        conf.addResource(new Path("/etc/hadoop/conf/hdfs-
site.xml"));
        FileSystem fileSystem = FileSystem.get(conf);
        // Get the filename out of the file path
        String filename =
source.substring(source.lastIndexOf('/') + 1,
            source.length());
```

```
        // Create the destination path including the filename.
        if (dest.charAt(dest.length() - 1) != '/') {
            dest = dest + "/" + filename;
        } else {
            dest = dest + filename;
        }
        // System.out.println("Adding file to " + destination);
        // Check if the file already exists
        Path path = new Path(dest);
        if (fileSystem.exists(path)) {
            System.out.println("File " + dest + " already
exists");
            return;
        }
        // Create a new file and write data to it.
        FSDataOutputStream out = fileSystem.create(path);
        InputStream in = new BufferedInputStream(new
FileInputStream(
            new File(source)));
        byte[] b = new byte[1024];
        int numBytes = 0;
        while ((numBytes = in.read(b)) > 0) {
            out.write(b, 0, numBytes);
        }
        // Close all the file descripters
        in.close();
        out.close();
        fileSystem.close();
    }

    public void readFile(String file) throws IOException {
        Configuration conf = new Configuration();
        conf.addResource(new Path("/etc/hadoop/conf/core-
site.xml"));
        FileSystem fileSystem = FileSystem.get(conf);
        Path path = new Path(file);
        if (!fileSystem.exists(path)) {
            System.out.println("File " + file + " does not
exists");
            return;
        }
        FSDataInputStream in = fileSystem.open(path);
        String filename = file.substring(file.lastIndexOf('/') +
1,
            file.length());
        OutputStream out = new BufferedOutputStream(new
FileOutputStream(
```

```
            new File(filename)));
        byte[] b = new byte[1024];
        int numBytes = 0;
        while ((numBytes = in.read(b)) > 0) {
            out.write(b, 0, numBytes);
        }
        in.close();
        out.close();
        fileSystem.close();
    }

    public void deleteFile(String file) throws IOException {
        Configuration conf = new Configuration();
        conf.addResource(new Path("/etc/hadoop/conf/core-
site.xml"));
        FileSystem fileSystem = FileSystem.get(conf);
        Path path = new Path(file);
        if (!fileSystem.exists(path)) {
            System.out.println("File " + file + " does not
exists");
            return;
        }
        fileSystem.delete(new Path(file), true);
        fileSystem.close();
    }

    public void mkdir(String dir) throws IOException {
        Configuration conf = new Configuration();
        conf.addResource(new Path("/etc/hadoop/conf/core-
site.xml"));
        FileSystem fileSystem = FileSystem.get(conf);
        Path path = new Path(dir);
        if (fileSystem.exists(path)) {
            System.out.println("Dir " + dir + " already not
exists");
            return;
        }
        fileSystem.mkdirs(path);
        fileSystem.close();
    }

    public static void main(String[] args) throws IOException {
       if (args.length < 1) {
            System.out.println("Usage: hdfsclient
add/read/delete/mkdir" +
                " [<local_path> <hdfs_path>]");
            System.exit(1);
```

```
        }
        HDFSClient client = new HDFSClient();
        if (args[0].equals("add")) {
            if (args.length < 3) {
                System.out.println("Usage: hdfsclient add
<local_path> " +
                "<hdfs_path>");
                System.exit(1);
            }
            client.addFile(args[1], args[2]);
        } else if (args[0].equals("read")) {
            if (args.length < 2) {
                System.out.println("Usage: hdfsclient read
<hdfs_path>");
                System.exit(1);
            }
            client.readFile(args[1]);
        } else if (args[0].equals("delete")) {
            if (args.length < 2) {
               System.out.println("Usage: hdfsclient delete
<hdfs_path>");
                System.exit(1);
            }
            client.deleteFile(args[1]);
        } else if (args[0].equals("mkdir")) {
            if (args.length < 2) {
                System.out.println("Usage: hdfsclient mkdir
<hdfs_path>");
                System.exit(1);
            }
            client.mkdir(args[1]);
        } else {
            System.out.println("Usage: hdfsclient
add/read/delete/mkdir" +
                " [<local_path> <hdfs_path>]");
            System.exit(1);
        }
        System.out.println("Done!");
    }
}
```

The `HadoopDFSFileReadWrite.java` example in Listing 3.1 can be compiled on Linux systems using the following steps. First, create a directory to hold the classes:

```
$ mkdir  HDFSClient-classes
```

Next, compile the program using `'hadoop classpath'` to ensure all the class paths are available:

$ javac -cp 'hadoop classpath' -d HDFSClient-classes HDFSClient.java

Finally, create a Java archive file:

$ jar -cvfe HDFSClient.jar org/myorg.HDFSClient -C HDFSClient-classes/ .

The program can be run to check for available options as follows:

$ hadoop jar ./HDFSClient.jar
Usage: hdfsclient add/read/delete/mkdir [<local_path> <hdfs_path>]

A simple file copy from the local system to HDFS can be accomplished using the following command:

$ hadoop jar ./HDFSClient.jar add ./NOTES.txt /user/hdfs

The file can be seen in HDFS by using the `hdfs dfs -ls` command:

$ hdfs dfs -ls NOTES.txt
-rw-r--r-- 2 hdfs hdfs 502 2015-06-03 15:43 NOTES.txt

HDFS C Application Example

HDFS can be used in C programs by incorporating the Java Native Interface (JNI)–based C application programming interface (API) for Hadoop HDFS. The library, `libhdfs`, provides a simple C API to manipulate HDFS files and the file system. `libhdfs` is normally available as part of the Hadoop installation. More information about the API can be found on the Apache webpage, http://wiki.apache.org/hadoop/LibHDFS.

Listing 3.2 is small example program that illustrates use of the API. This example is available from the book download page; see Appendix A, "Book Webpage and Code Download."

Listing 3.2 **hdfs-simple-test.c**

```
#include  <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hdfs.h"

int main(int argc, char **argv) {

    hdfsFS fs = hdfsConnect("default", 0);
    const char* writePath = "/tmp/testfile.txt";
    hdfsFile writeFile = hdfsOpenFile(fs, writePath,
WRONGLY|O_CREAT, 0, 0, 0);
```

```
    if(!writeFile) {
          fprintf(stderr, "Failed to open %s for writing!\n",
writePath);
          exit(-1);
    }
    char* buffer = "Hello, World!\n";
    tSize num_written_bytes = hdfsWrite(fs, writeFile,
(void*)buffer, strlen(buffer)+1);
    if (hdfsFlush(fs, writeFile)) {
           fprintf(stderr, "Failed to 'flush' %s\n", writePath);
          exit(-1);
    }
   hdfsCloseFile(fs, writeFile);
}
```

The example can be built using the following steps. The following software environment is assumed:

- Operating system: Linux
- Platform: RHEL 6.6
- Hortonworks HDP 2.2 with Hadoop Version: 2.6

The first step loads the Hadoop environment paths. In particular, the `$HADOOP_LIB` path is needed for the compiler.

$. /etc/hadoop/conf/hadoop-env.sh

The program is compiled using `gcc` and the following command line. In addition to `$HADOOP_LIB`, the `$JAVA_HOME` path is assumed to be in the local environment. If the compiler issues errors or warnings, confirm that all paths are correct for the Hadoop and Java environment.

$ gcc hdfs-simple-test.c -I$HADOOP_LIB/include -I$JAVA_HOME/include -L$HADOOP_LIB/lib -L$JAVA_HOME/jre/lib/amd64/server -ljvm -lhdfs -o hdfs-simple-test

The location of the run-time library path needs to be set with the following command:

$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$JAVA_HOME/jre/lib/amd64/server:$HADOOP_LIB/lib

The Hadoop class path needs to be set with the following command. The `-glob` option is required because Hadoop version 2 uses a wildcard syntax in the output of the `hadoop classpath` command. Hadoop version 1 used the full path to every jar file without wildcards. Unfortunately, Java does not expand the wildcards automatically when launching an embedded JVM via JNI, so older scripts may not work. The `-glob` option expands the wildcards.

$ export CLASSPATH='hadoop classpath -glob'

The program can be run using the following. There may be some warnings that can be ignored.

$ /hdfs-simple-test

The new file contents can be inspected using the `hdfs dfs -cat` command:

$ hdfs dfs -cat /tmp/testfile.txt
Hello, World!

SUMMARY AND ADDITIONAL RESOURCES

Apache Hadoop HDFS has become a mature, high-performance, Big Data file system. Its design is fundamentally different from many conventional file systems and provides many important capabilities needed by Hadoop applications—including data locality. The two key components, the NameNode and DataNodes, combine to offer a scalable and robust distributed storage solution that operates on commodity servers. The NameNode can be augmented by secondary or checkpoint nodes to improve performance and robustness. Advanced features like High Availability, NameNode Federation, snapshots, and NFSv3 mounts are also available to HDFS administrators.

Users interact with HDFS using a command interface that is similar to traditional POSIX-style file systems. A small subset of HDFS user commands is all that is needed to gain immediate use of HDFS. In addition, HDFS presents a web interface for easy navigation of file system information. Finally, HDFS can be easily used in end-user applications written using the Java or C languages.

Additional information and background on HDFS can be obtained from the following resources:

- **HDFS background**
- http://hadoop.apache.org/docs/stable1/hdfs_design.html
- http://developer.yahoo.com/hadoop/tutorial/module2.html
- http://hadoop.apache.org/docs/stable/hdfs_user_guide.html
- **HDFS user commands**
- http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html
- **HDFS Java programming**
- http://wiki.apache.org/hadoop/HadoopDfsReadWriteExample
- **HDFS libhdfs programming in C**
- http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/LibHdfs.html

4. Running Example Programs and Benchmarks

In This Chapter:

- The steps needed to run the Hadoop MapReduce examples are provided.
- An overview of the YARN ResourceManager web GUI is presented.
- The steps needed to run two important benchmarks are provided.
- The `mapred` command is introduced as a way to list and kill MapReduce jobs.

When using new or updated hardware or software, simple examples and benchmarks help confirm proper operation. Apache Hadoop includes many examples and benchmarks to aid in this task. This chapter provides instructions on how to run, monitor, and manage some basic MapReduce examples and benchmarks.

RUNNING MAPREDUCE EXAMPLES

All Hadoop releases come with MapReduce example applications. Running the existing MapReduce examples is a simple process—once the example files are located, that is. For example, if you installed Hadoop version 2.6.0 from the Apache sources under `/opt`, the examples will be in the following directory:

/opt/hadoop-2.6.0/share/hadoop/mapreduce/

In other versions, the examples may be in `/usr/lib/hadoop-mapreduce/` or some other location. The exact location of the example jar file can be found using the `find` command:

$ find / -name "hadoop-mapreduce-examples*.jar" -print

For this chapter the following software environment will be used:

- OS: Linux
- Platform: RHEL 6.6
- Hortonworks HDP 2.2 with Hadoop Version: 2.6

In this environment, the location of the examples is `/usr/hdp/2.2.4.2-2/hadoop-mapreduce`. For the purposes of this example, an environment variable called `HADOOP_EXAMPLES` can be defined as follows:

$ export HADOOP_EXAMPLES=/usr/hdp/2.2.4.2-2/hadoop-mapreduce

Once you define the examples path, you can run the Hadoop examples using the commands discussed in the following sections.

Listing Available Examples

A list of the available examples can be found by running the following command. In some cases, the version number may be part of the jar file (e.g., in the version 2.6 Apache sources, the file is named `hadoop-mapreduce-examples-2.6.0.jar`).

$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar

Note

In previous versions of Hadoop, the command `hadoop jar . . .` was used to run MapReduce programs. Newer versions provide the `yarn` command, which offers more capabilities. Both commands will work for these examples.

The possible examples are as follows:

```
An example program must be given as the first argument.
Valid program names are:
  aggregatewordcount: An Aggregate based map/reduce program that counts
  the words in the input files.
  aggregatewordhist: An Aggregate based map/reduce program that computes
  the histogram of the words in the input files.
  bbp: A map/reduce program that uses Bailey-Borwein-Plouffe to compute
  exact digits of Pi.
  dbcount: An example job that count the pageview counts from a database.
  distbbp: A map/reduce program that uses a BBP-type formula to compute
  exact bits of Pi.
  grep: A map/reduce program that counts the matches of a regex in the
  input.
  join: A job that effects a join over sorted, equally partitioned
  datasets
  multifilewc: A job that counts words from several files.
  pentomino: A map/reduce tile laying program to find solutions to
  pentomino problems.
  pi: A map/reduce program that estimates Pi using a quasi-Monte
  Carlo method.
  randomtextwriter: A map/reduce program that writes 10GB of random
  textual data per node.
  randomwriter: A map/reduce program that writes 10GB of random data
  per node.
  secondarysort: An example defining a secondary sort to the reduce.
  sort: A map/reduce program that sorts the data written by the
  random writer.
  sudoku: A sudoku solver.
  teragen: Generate data for the terasort
  terasort: Run the terasort
  teravalidate: Checking results of terasort
  wordcount: A map/reduce program that counts the words in the
  input files.
  wordmean: A map/reduce program that counts the average length of
```

the words in the input files.
wordmedian: A map/reduce program that counts the median length of the words in the input files.
wordstandarddeviation: A map/reduce program that counts the standard deviation of the length of the words in the input files.

To illustrate several features of Hadoop and the YARN ResourceManager service GUI, the `pi` and `terasort` examples are presented next. To find help for running the other examples, enter the example name without any arguments. Chapter 6, "MapReduce Programming," covers one of the other popular examples called `wordcount`.

Running the Pi Example

The `pi` example calculates the digits of π using a quasi-Monte Carlo method. If you have not added users to HDFS (see Chapter 10, "Basic Hadoop Administration Procedures"), run these tests as user `hdfs`. To run the `pi` example with 16 maps and 1,000,000 samples per map, enter the following command:

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar pi 16 1000000
```

If the program runs correctly, you should see output similar to the following. (Some of the Hadoop INFO messages have been removed for clarity.)

```
Number of Maps  = 16
Samples per Map = 1000000
Wrote input for Map #0
Wrote input for Map #1
Wrote input for Map #2
Wrote input for Map #3
Wrote input for Map #4
Wrote input for Map #5
Wrote input for Map #6
Wrote input for Map #7
Wrote input for Map #8
Wrote input for Map #9
Wrote input for Map #10
Wrote input for Map #11
Wrote input for Map #12
Wrote input for Map #13
Wrote input for Map #14
Wrote input for Map #15
Starting Job
...
15/05/13 20:10:30 INFO mapreduce.Job:  map 0% reduce 0%
15/05/13 20:10:37 INFO mapreduce.Job:  map 19% reduce 0%
15/05/13 20:10:39 INFO mapreduce.Job:  map 50% reduce 0%
15/05/13 20:10:46 INFO mapreduce.Job:  map 56% reduce 0%
```

```
15/05/13 20:10:47 INFO mapreduce.Job:  map 94% reduce 0%
15/05/13 20:10:48 INFO mapreduce.Job:  map 100% reduce 100%
15/05/13 20:10:48 INFO mapreduce.Job: Job job_1429912013449_0047 completed
successfully
15/05/13 20:10:48 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=358
                FILE: Number of bytes written=1949395
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=4198
                HDFS: Number of bytes written=215
                HDFS: Number of read operations=67
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=3
        Job Counters
                Launched map tasks=16
                Launched reduce tasks=1
                Data-local map tasks=16
                Total time spent by all maps in occupied slots (ms)=158378
                Total time spent by all reduces in occupied slots (ms)=8462
                Total time spent by all map tasks (ms)=158378
                Total time spent by all reduce tasks (ms)=8462
                Total vcore-seconds taken by all map tasks=158378
                Total vcore-seconds taken by all reduce tasks=8462
                Total megabyte-seconds taken by all map tasks=243268608
                Total megabyte-seconds taken by all reduce tasks=12997632
        Map-Reduce Framework
                Map input records=16
                Map output records=32
                Map output bytes=288
                Map output materialized bytes=448
                Input split bytes=2310
                Combine input records=0
                Combine output records=0
                Reduce input groups=2
                Reduce shuffle bytes=448
                Reduce input records=32
                Reduce output records=0
                Spilled Records=64
                Shuffled Maps=16
                Failed Shuffles=0
                Merged Map outputs=16
                GC time elapsed (ms)=1842
                CPU time spent (ms)=11420
```

```
        Physical memory (bytes) snapshot=13405769728
        Virtual memory (bytes) snapshot=33911930880
        Total committed heap usage (bytes)=17026777088
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    File Input Format Counters
        Bytes Read=1888
    File Output Format Counters
        Bytes Written=97
Job Finished in 23.718 seconds
Estimated value of Pi is 3.14159125000000000000
```

Notice that the MapReduce progress is shown in the same way as Hadoop version 1, but the application statistics are different. Most of the statistics are self-explanatory. The one important item to note is that the YARN MapReduce framework is used to run the program. (See Chapter 1, "Background and Concepts," and Chapter 8, "Hadoop YARN Applications," for more information about YARN frameworks.)

Using the Web GUI to Monitor Examples

This section provides an illustration of using the YARN ResourceManager web GUI to monitor and find information about YARN jobs. The Hadoop version 2 YARN ResourceManager web GUI differs significantly from the MapReduce web GUI found in Hadoop version 1. Figure 4.1 shows the main YARN web interface. The cluster metrics are displayed in the top row, while the running applications are displayed in the main table. A menu on the left provides navigation to the nodes table, various job categories (e.g., New, Accepted, Running, Finished, Failed), and the Capacity Scheduler (covered in Chapter 10, "Basic Hadoop Administration Procedures"). This interface can be opened directly from the Ambari YARN service Quick Links menu or by directly entering http://hostname:8088 into a local web browser. For this example, the `pi` application is used. Note that the application can run quickly and may finish before you have fully explored the GUI. A longer-running application, such as `terasort`, may be helpful when exploring all the various links in the GUI.

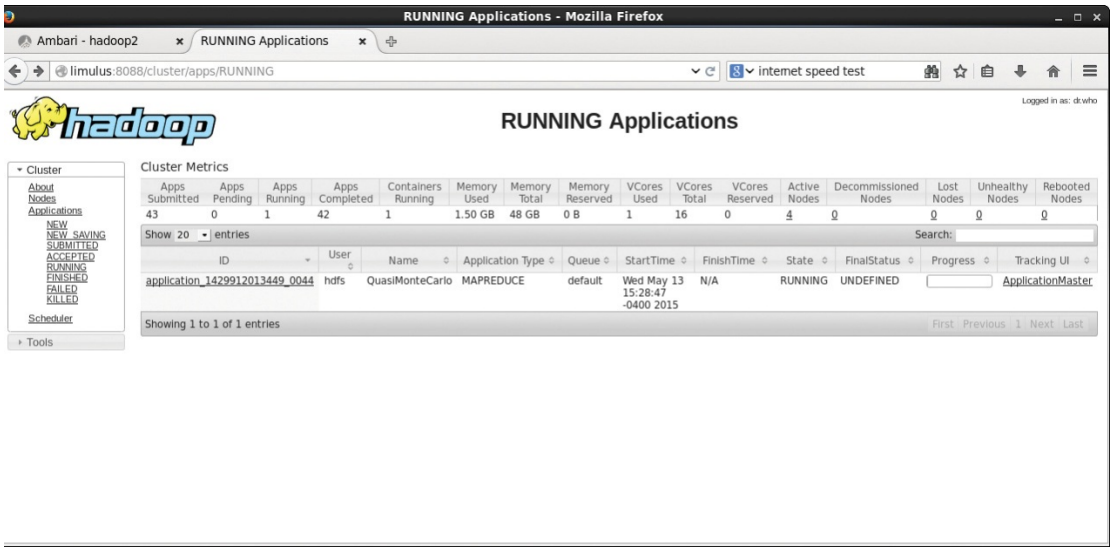


Figure 4.1 Hadoop RUNNING Applications web GUI for the pi example

For those readers who have used or read about Hadoop version 1, if you look at the Cluster Metrics table, you will see some new information. First, you will notice that the "Map/Reduce Task Capacity" has been replaced by the number of running containers. If YARN is running a MapReduce job, these containers can be used for both map and reduce tasks. Unlike in Hadoop version 1, the number of mappers and reducers is not fixed. There are also memory metrics and links to node status. If you click on the Nodes link (left menu under About), you can get a summary of the node activity and state. For example, Figure 4.2 is a snapshot of the node activity while the `pi` application is running. Notice the number of containers, which are used by the MapReduce framework as either mappers or reducers.

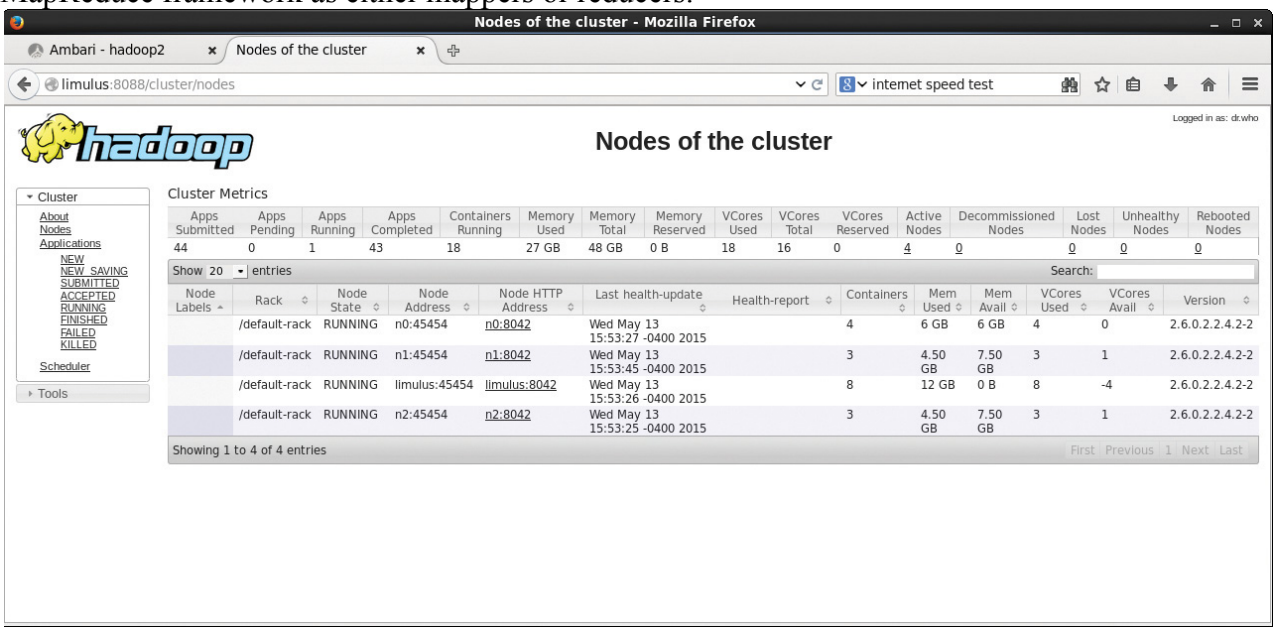


Figure 4.2 Hadoop YARN ResourceManager nodes status window

Going back to the main Applications/Running window (Figure 4.1), if you click on the `application_14299...` link, the Application status window in Figure 4.3 will appear. This window provides an application overview and metrics, including the cluster node on which the ApplicationMaster container is running.

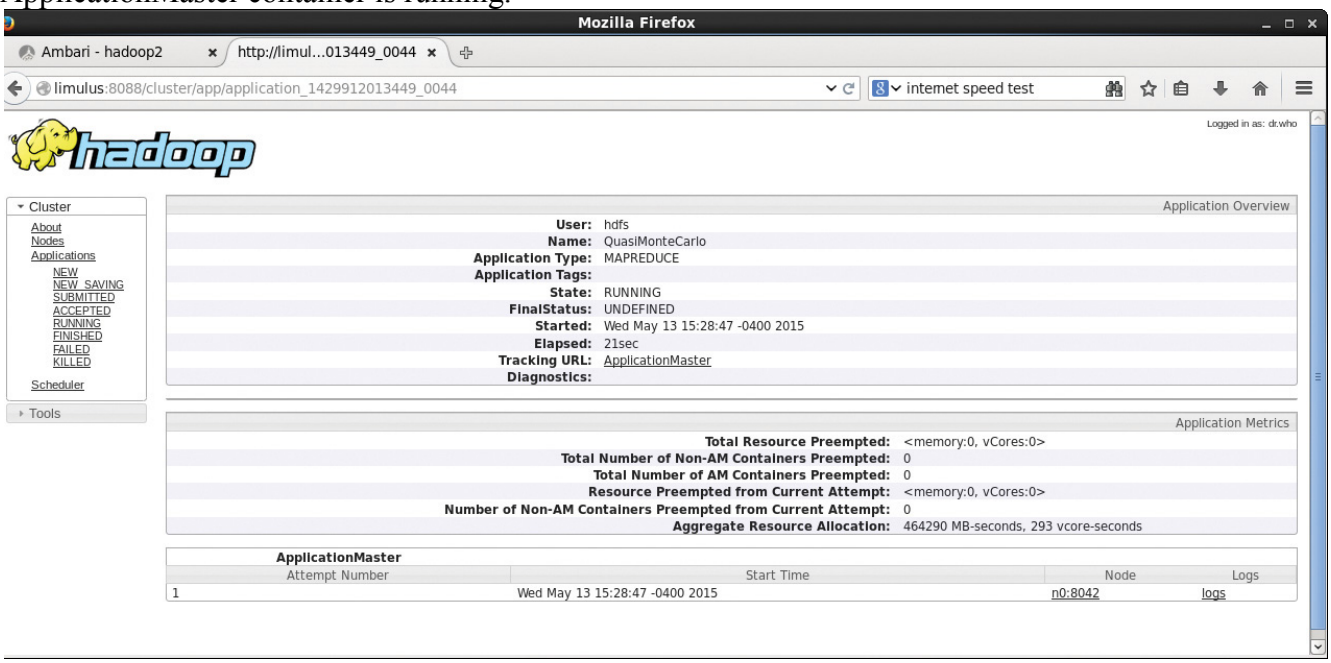


Figure 4.3 Hadoop YARN application status for the `pi` example

Clicking the ApplicationMaster link next to "Tracking URL:" in Figure 4.3 leads to the window shown in Figure 4.4. Note that the link to the application's ApplicationMaster is also found in the last column on the main Running Applications screen shown in Figure 4.1.

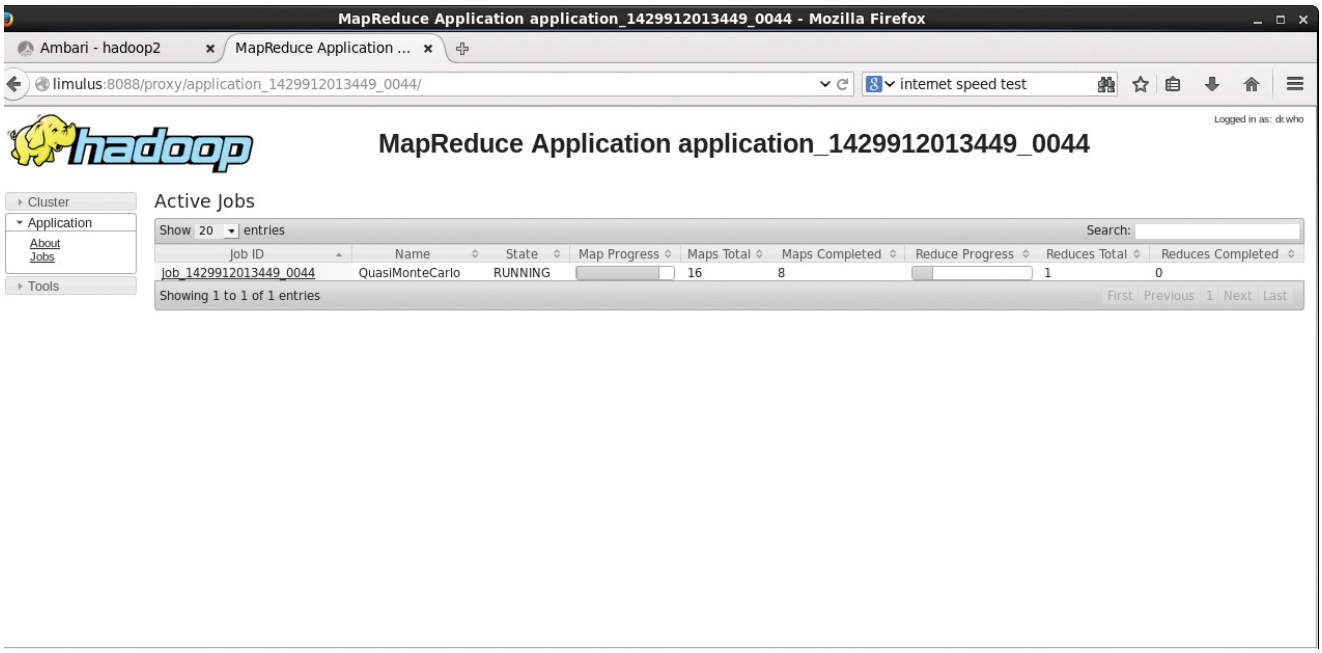


Figure 4.4 Hadoop YARN ApplicationMaster for MapReduce application

In the MapReduce Application window, you can see the details of the MapReduce application and the overall progress of mappers and reducers. Instead of containers, the MapReduce application now refers to maps and reducers. Clicking `job_14299...` brings up the window shown in Figure 4.5. This window displays more detail about the number of pending, running, completed, and failed mappers and reducers, including the elapsed time since the job started.

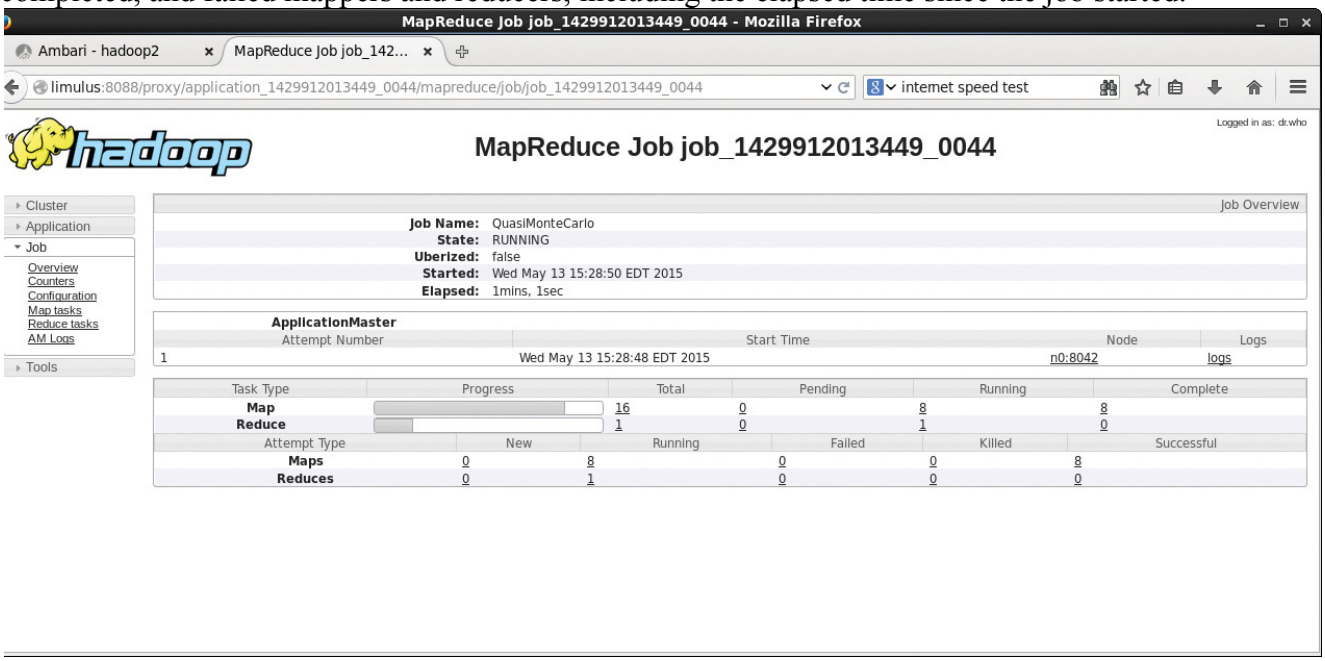


Figure 4.5 Hadoop YARN MapReduce job progress

The status of the job in Figure 4.5 will be updated as the job progresses (the window needs to be refreshed manually). The ApplicationMaster collects and reports the progress of each mapper and reducer task. When the job is finished, the window is updated to that shown in Figure 4.6. It reports the overall run time and provides a breakdown of the timing of the key phases of the MapReduce job (map, shuffle, merge, reduce).

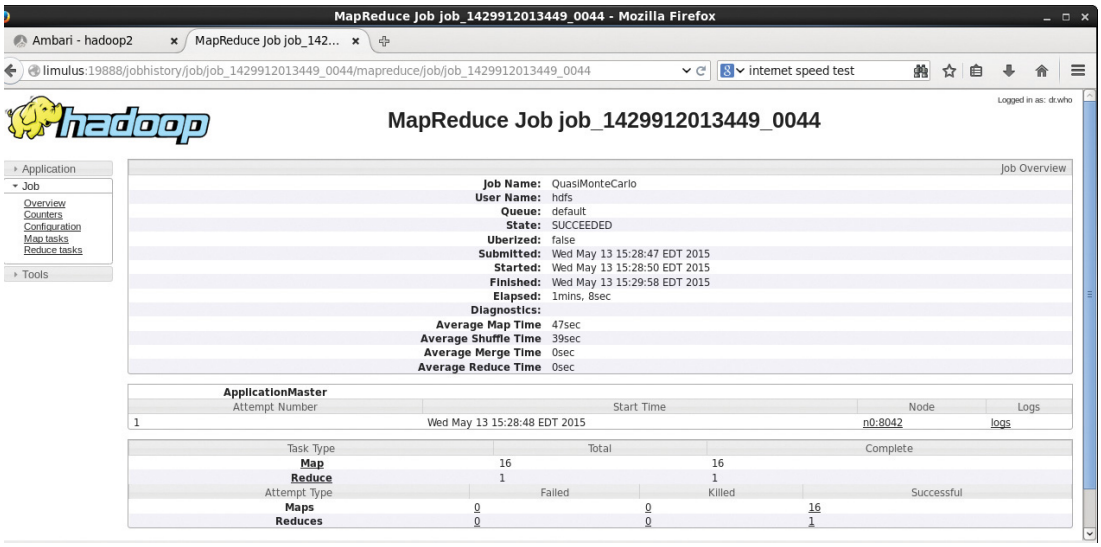


Figure 4.6 Hadoop YARN completed MapReduce job summary

If you click the node used to run the ApplicationMaster (`n0:8042` in Figure 4.6), the window in Figure 4.7 opens and provides a summary from the NodeManager on node `n0`. Again, the NodeManager tracks only containers; the actual tasks running in the containers are determined by the ApplicationMaster.

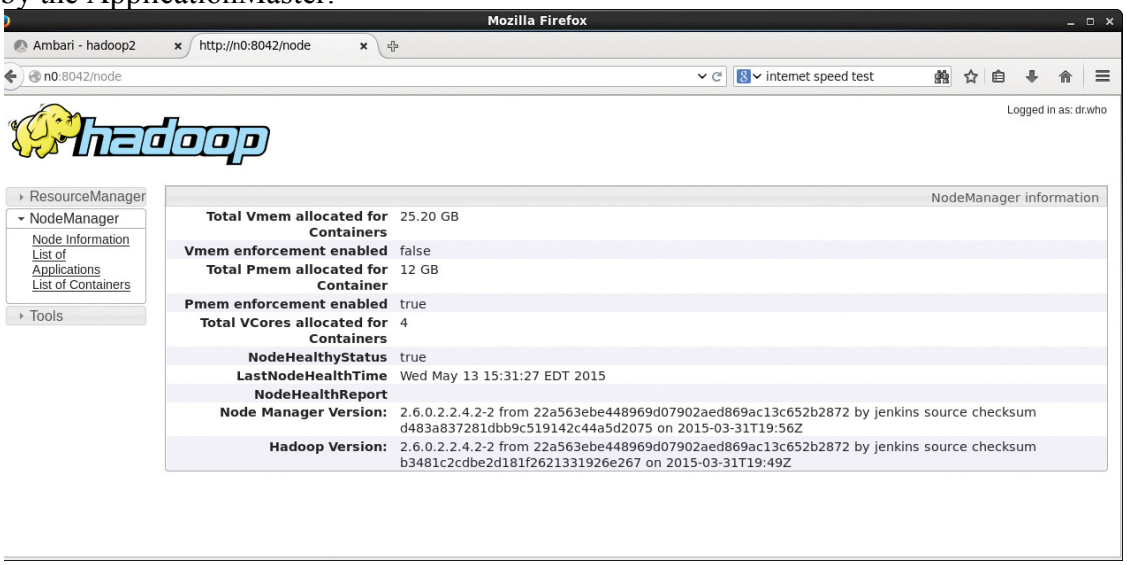


Figure 4.7 Hadoop YARN NodeManager for n0 job summary

Going back to the job summary page (Figure 4.6), you can also examine the logs for the ApplicationMaster by clicking the "logs" link. To find information about the mappers and reducers, click the numbers under the Failed, Killed, and Successful columns. In this example, there were 16 successful mappers and one successful reducer. All the numbers in these columns lead to more information about individual map or reduce process. For instance, clicking the "16" under "-Successful" in Figure 4.6 displays the table of map tasks in Figure 4.8. The metrics for the Application Master container are displayed in table form. There is also a link to the log file

for each process (in this case, a map process). Viewing the logs requires that the `yarn.log.aggregation-enable` variable in the `yarn-site.xml` file be set. For more on changing Hadoop settings, see Chapter 9, "Managing Hadoop with Apache Ambari."

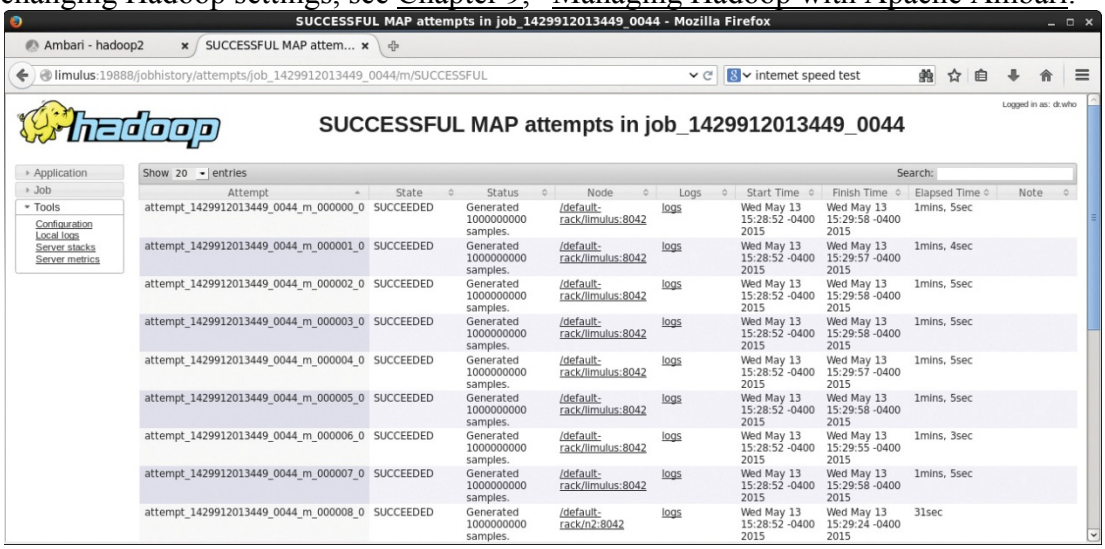


Figure 4.8 Hadoop YARN MapReduce logs available for browsing

If you return to the main cluster window (Figure 4.1), choose Applications/Finished, and then select our application, you will see the summary page shown in Figure 4.9.

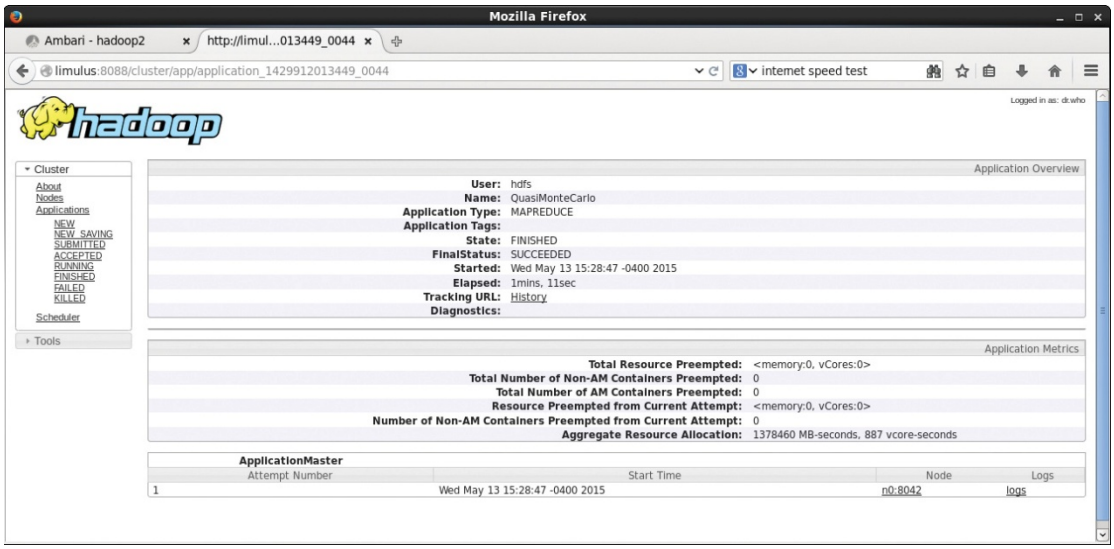


Figure 4.9 Hadoop YARN application summary page

There are a few things to notice in the previous windows. First, because YARN manages applications, all information reported by the ResourceManager concerns the resources provided and the application type (in this case, `MAPREDUCE`). In Figure 4.1 and Figure 4.4, the `YARN ResourceManager` refers to the `pi` example by its application-id (`application_1429912013449_0044`). YARN has no data about the actual application other than the fact that it is a MapReduce job. Data from the actual MapReduce job are provided by

the MapReduce framework and referenced by a job-id (`job_1429912013449_0044`) in Figure 4.6. Thus, two clearly different data streams are combined in the web GUI: YARN *application*s and MapReduce framework *jobs*. If the framework does not provide job information, then certain parts of the web GUI will not have anything to display.

Another interesting aspect of the previous windows is the dynamic nature of the mapper and reducer tasks. These tasks are executed as YARN containers, and their number will change as the application runs. Users may request specific numbers of mappers and reducers, but the ApplicationMaster uses them in a dynamic fashion. As mappers complete, the ApplicationMaster will return the containers to the ResourceManager and request a smaller number of reducer containers. This feature provides for much better cluster utilization because mappers and reducers are dynamic—rather than fixed—resources.

RUNNING BASIC HADOOP BENCHMARKS

Many Hadoop benchmarks can provide insight into cluster performance. The best benchmarks are always those that reflect real application performance. The two benchmarks discussed in this section, `terasort` and `TestDFSIO`, provide a good sense of how well your Hadoop installation is operating and can be compared with public data published for other Hadoop systems. The results, however, should not be taken as a single indicator for system-wide performance on all applications.

The following benchmarks are designed for full Hadoop cluster installations. These tests assume a multi-disk HDFS environment. Running these benchmarks in the Hortonworks Sandbox or in the pseudo-distributed single-node install from Chapter 2 is not recommended because all input and output (I/O) are done using a single system disk drive.

Running the Terasort Test

The `terasort` benchmark sorts a specified amount of randomly generated data. This benchmark provides combined testing of the HDFS and MapReduce layers of a Hadoop cluster. A full `terasort`benchmark run consists of the following three steps:

1. Generating the input data via `teragen` program.
2. Running the actual `terasort` benchmark on the input data.
3. Validating the sorted output data via the `teravalidate` program.

In general, each row is 100 bytes long; thus the total amount of data written is 100 times the number of rows specified as part of the benchmark (i.e., to write 100GB of data, use 1 billion rows). The input and output directories need to be specified in HDFS. The following sequence of commands will run the benchmark for 50GB of data as user `hdfs`. Make sure the `/user/hdfs` directory exists in HDFS before running the benchmarks.

1. Run `teragen` to generate rows of random data to sort.

Click here to view code image

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar
teragen 500000000 /user/hdfs/TeraGen-50GB
```

2. Run `terasort` to sort the database.

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar
terasort /user/hdfs/TeraGen-50GB /user/hdfs/TeraSort-50GB
```

3. Run `teravalidate` to validate the sort.

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar
teravalidate /user/hdfs/TeraSort-50GB /user/hdfs/TeraValid-50GB
```

To report results, the time for the actual sort (`terasort`) is measured and the benchmark rate in megabytes/second (MB/s) is calculated. For best performance, the actual `terasort` benchmark should be run with a replication factor of 1. In addition, the default number of `terasort` reducer tasks is set to 1. Increasing the number of reducers often helps with benchmark performance. For example, the following command will instruct `terasort` to use four reducer tasks:

$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar terasort -Dmapred.reduce.tasks=4 /user/hdfs/TeraGen-50GB /user/hdfs/TeraSort-50GB

Also, do not forget to clean up the `terasort` data between runs (and after testing is finished). The following command will perform the cleanup for the previous example:

$ hdfs dfs -rm -r -skipTrash Tera*

Running the TestDFSIO Benchmark

Hadoop also includes an HDFS benchmark application called `TestDFSIO`.
The `TestDFSIO`benchmark is a read and write test for HDFS. That is, it will write or read a number of files to and from HDFS and is designed in such a way that it will use one map task per file. The file size and number of files are specified as command-line arguments. Similar to the `terasort` benchmark, you should run this test as user `hdfs`.

Similar to `terasort`, `TestDFSIO` has several steps. In the following example, 16 files of size 1GB are specified. Note that the `TestDFSIO` benchmark is part of the `hadoop-mapreduce-client-jobclient.jar`. Other benchmarks are also available as part of this jar file. Running it with no arguments will yield a list. In addition to `TestDFSIO`, `NNBench` (load testing the NameNode) and `MRBench` (load testing the MapReduce framework) are commonly used Hadoop benchmarks. Nevertheless, `TestDFSIO` is perhaps the most widely reported of these benchmarks. The steps to run `TestDFSIO` are as follows:
1. Run `TestDFSIO` in write mode and create data.

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-client-jobclient-
tests.jar TestDFSIO -write  -nrFiles 16 -fileSize 1000
```

Example results are as follows (date and time prefix removed).

```
fs.TestDFSIO: ----- TestDFSIO ----- : write
fs.TestDFSIO:            Date & time: Thu May 14 10:39:33 EDT
2015
fs.TestDFSIO:        Number of files: 16
fs.TestDFSIO: Total MBytes processed: 16000.0
```

```
fs.TestDFSIO:      Throughput mb/sec: 14.890106361891005
fs.TestDFSIO: Average IO rate mb/sec: 15.690713882446289
fs.TestDFSIO:  IO rate std deviation: 4.0227035201665595
fs.TestDFSIO:     Test exec time sec: 105.631
```

2. Run `TestDFSIO` in read mode.

```
$ yarn jar  $HADOOP_EXAMPLES/hadoop-mapreduce-client-jobclient-
tests.jar TestDFSIO -read  -nrFiles 16 -fileSize 1000
```

Example results are as follows (date and time prefix removed). The large standard deviation is due to the placement of tasks in the cluster on a small four-node cluster.

```
fs.TestDFSIO: ----- TestDFSIO ----- : read
fs.TestDFSIO:            Date & time: Thu May 14 10:44:09 EDT
2015
fs.TestDFSIO:        Number of files: 16
fs.TestDFSIO: Total MBytes processed: 16000.0
fs.TestDFSIO:      Throughput mb/sec: 32.38643494172466
fs.TestDFSIO: Average IO rate mb/sec: 58.72880554199219
fs.TestDFSIO:  IO rate std deviation: 64.60017624360337
fs.TestDFSIO:     Test exec time sec: 62.798
```

3. Clean up the TestDFSIO data.

```
$ yarn jar  $HADOOP_EXAMPLES/hadoop-mapreduce-client-jobclient-
tests.jar TestDFSIO -clean
```

Running the `TestDFSIO` and `terasort` benchmarks help you gain confidence in a Hadoop installation and detect any potential problems. It is also instructive to view the Ambari dashboard and the YARN web GUI (as described previously) as the tests run.

Managing Hadoop MapReduce Jobs

Hadoop MapReduce jobs can be managed using the `mapred job` command. The most important options for this command in terms of the examples and benchmarks are `-list`, `-kill`, and `-status`. In particular, if you need to kill one of the examples or benchmarks, you can use the `mapred job -list` command to find the `job-id` and then use `mapred job -kill <job-id>` to kill the job across the cluster. MapReduce jobs can also be controlled at the application level with the `yarn application` command (see Chapter 10, "Basic Hadoop Administration Procedures"). The possible options for `mapred job` are as follows:

```
$ mapred job
Usage: CLI <command> <args>
    [-submit <job-file>]
    [-status <job-id>]
    [-counter <job-id> <group-name> <counter-name>]
    [-kill <job-id>]
    [-set-priority <job-id> <priority>]. Valid values for priorities
```

are: VERY_HIGH HIGH NORMAL LOW VERY_LOW
[-events <job-id> <from-event-#> <#-of-events>]
[-history <jobHistoryFile>]
[-list [all]]
[-list-active-trackers]
[-list-blacklisted-trackers]
[-list-attempt-ids <job-id> <task-type> <task-state>]. Valid values
for <task-type> are REDUCE MAP. Valid values for <task-state> are
running, completed
[-kill-task <task-attempt-id>]
[-fail-task <task-attempt-id>]
[-logs <job-id> <task-attempt-id>]

Generic options supported are
-conf <configuration file> specify an application configuration file
-D <property=value> use value for given property
-fs <local|namenode:port> specify a namenode
-jt <local|resourcemanager:port> specify a ResourceManager
-files <comma separated list of files> specify comma separated files to
be copied to the map reduce cluster
-libjars <comma separated list of jars> specify comma separated jar
files to include in the classpath.
-archives <comma separated list of archives> specify comma separated
archives to be unarchived on the compute machines.

The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]

SUMMARY AND ADDITIONAL RESOURCES

No matter what the size of the Hadoop cluster, confirming and measuring the MapReduce performance of that cluster is an important first step. Hadoop includes some simple applications and benchmarks that can be used for this purpose. The YARN ResourceManager web GUI is a good way to monitor the progress of any application. Jobs that run under the MapReduce framework report a large number of run-time metrics directly (including logs) back to the GUI; these metrics are then presented to the user in a clear and coherent fashion. Should issues arise when running the examples and benchmarks, the `mapred job` command can be used to kill a MapReduce job.

Additional information and background on each of the examples and benchmarks can be found from the following resources:

- **Pi Benchmark**
- https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/pi/package-summary.html
- **Terasort Benchmark**

- https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html
- **Benchmarking and Stress Testing an Hadoop Cluster**
- http://www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench (uses Hadoop V1, will work with V2)

5. Hadoop MapReduce Framework

In This Chapter:

- The MapReduce computation model is presented using simple examples.
- The Apache Hadoop MapReduce framework data flow is explained.
- MapReduce fault tolerance, speculative execution, and hardware are discussed.

The MapReduce programming model is conceptually simple. Based on two simple steps—applying a mapping process and then reducing (condensing/collecting) the results—it can be applied to many real-world problems. In this chapter, we examine the MapReduce process using basic command-line tools. We then expand this concept into a parallel MapReduce model.

THE MAPREDUCE MODEL

Apache Hadoop is often associated with MapReduce computing. Prior to Hadoop version 2, this assumption was certainly true. Hadoop version 2 maintained the MapReduce capability and also made other processing models available to users. Virtually all the tools developed for Hadoop, such as Pig and Hive, will work seamlessly on top of the Hadoop version 2 MapReduce.

The MapReduce computation model provides a very powerful tool for many applications and is more common than most users realize. Its underlying idea is very simple. There are two stages: a mapping stage and a reducing stage. In the mapping stage, a *mapping procedure* is applied to input data. The map is usually some kind of filter or sorting process.

For instance, assume you need to count how many times the name "Kutuzov" appears in the novel *War and Peace*. One solution is to gather 20 friends and give them each a section of the book to search. This step is the map stage. The reduce phase happens when everyone is done counting and you sum the total as your friends tell you their counts.

Now consider how this same process could be accomplished using simple *nix command-line tools. The following `grep` command applies a specific map to a text file:

```
$ grep " Kutuzov " war-and-peace.txt
```

This command searches for the word `Kutuzov` (with leading and trailing spaces) in a text file called `war-and-peace.txt`. Each match is reported as a single line of text that contains the search term. The actual text file is a 3.2MB text dump of the novel *War and Peace* and is available from the book download page (see Appendix A, "Book Webpage and Code Download"). The search term, `Kutuzov`, is a character in the book. If we ignore the `grep` count (`-c`) option for the moment, we can reduce the number of instances to a single number (257) by sending (piping) the results of `grep` into `wc -l`. (`wc -l` or "word count" reports the number of lines it receives.)

```
$ grep " Kutuzov " war-and-peace.txt|wc -l
257
```

Though not strictly a MapReduce process, this idea is quite similar to and much faster than the manual process of counting the instances of Kutuzov in the printed book. The analogy can be taken a bit further by using the two simple (and naive) shell scripts shown in Listing 5.1 and Listing 5.2. The shell scripts are available from the book download page (see Appendix A). We can perform the same operation (much more slowly) and tokenize both the `Kutuzov` and `Petersburg` strings in the text:

```
$ cat war-and-peace.txt |./mapper.sh |./reducer.sh
Kutuzov,315
Petersburg,128
```

Notice that more instances of Kutuzov have been found (the first `grep` command ignored instances like "`Kutuzov.`" or "`Kutuzov,`"). The mapper inputs a text file and then outputs data in a (key, value) pair (token-name, count) format. Strictly speaking, the input to the script is the file and the keys are `Kutuzov` and `Petersburg`. The reducer script takes these key–value pairs and combines the similar tokens and counts the total number of instances. The result is a new key–value pair (token-name, sum).

Listing 5.1 **Simple Mapper Script**

```
#!/bin/bash
while read line ; do
  for token in $line; do
  if [ "$token" = "Kutuzov" ] ; then
    echo "Kutuzov,1"
  elif [ "$token" = "Petersburg" ] ; then
    echo "Petersburg,1"
  fi
  done
done
```

Listing 5.2 **Simple Reducer Script**

```
#!/bin/bash
kcount=0
pcount=0
while read line ; do
  if [ "$line" = "Kutuzov,1" ] ; then
   let kcount=kcount+1
  elif [ "$line" = "Petersburg,1" ] ; then
   let pcount=pcount+1
  fi
done
echo "Kutuzov,$kcount"
echo "Petersburg,$pcount"
```

Formally, the MapReduce process can be described as follows. The mapper and reducer functions are both defined with respect to data structured in (key, value) pairs. The mapper takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

Map(key1,value1) → list(key2,value2)

The reducer function is then applied to each key–value pair, which in turn produces a collection of values in the same domain:

Reduce(key2, list (value2)) → list(value3)

Each reducer call typically produces either one value (`value3`) or an empty response. Thus, the MapReduce framework transforms a list of (key, value) pairs into a list of values.

The MapReduce model is inspired by the map and reduce functions commonly used in many functional programming languages. The functional nature of MapReduce has some important properties:

- Data flow is in one direction (map to reduce). It is possible to use the output of a reduce step as the input to another MapReduce process.
- As with functional programing, the input data are not changed. By applying the mapping and reduction functions to the input data, new data are produced. In effect, the original state of the Hadoop data lake is always preserved (see Chapter 1, "Background and Concepts").
- Because there is no dependency on how the mapping and reducing functions are applied to the data, the mapper and reducer data flow can be implemented in any number of ways to provide better performance.

Distributed (parallel) implementations of MapReduce enable large amounts of data to be analyzed quickly. In general, the mapper process is fully scalable and can be applied to any subset of the input data. Results from multiple parallel mapping functions are then combined in the reducer phase.

As mentioned in Chapter 1, Hadoop accomplishes parallelism by using a distributed file system (HDFS) to slice and spread data over multiple servers. Apache Hadoop MapReduce will try to move the mapping tasks to the server that contains the data slice. Results from each data slice are then combined in the reducer step. This process is explained in more detail in the next section.

HDFS is not required for Hadoop MapReduce, however. A sufficiently fast parallel file system can be used in its place. In these designs, each server in the cluster has access to a high-performance parallel file system that can rapidly provide any data slice. These designs are typically more expensive than the commodity servers used for many Hadoop clusters.

MAPREDUCE PARALLEL DATA FLOW

From a programmer's perspective, the MapReduce algorithm is fairly simple. The programmer must provide a mapping function and a reducing function. Operationally, however, the Apache Hadoop parallel MapReduce data flow can be quite complex. Parallel execution of MapReduce requires other steps in addition to the mapper and reducer processes. The basic steps are as follows:

1. Input Splits. As mentioned, HDFS distributes and replicates data over multiple servers. The default data chunk or block size is 64MB. Thus, a 500MB file would be broken into 8 blocks and written to different machines in the cluster. The data are also replicated on multiple machines (typically three machines). These data slices are physical boundaries determined by HDFS and have nothing to do with the data in the file. Also, while not considered part of the MapReduce process, the time required to load and distribute data throughout HDFS servers can be considered part of the total processing time.

The input splits used by MapReduce are logical boundaries based on the input data. For example, the split size can be based on the number of records in a file (if the data exist as records) or an actual size in bytes. Splits are almost always smaller than the HDFS block size. The number of splits corresponds to the number of mapping processes used in the map stage.

2. Map Step. The mapping process is where the parallel nature of Hadoop comes into play. For large amounts of data, many mappers can be operating at the same time. The user provides the specific mapping process. MapReduce will try to execute the mapper on the machines where the block resides. Because the file is replicated in HDFS, the least busy node with the data will be chosen. If all nodes holding the data are too busy, MapReduce will try to pick a node that is closest to the node that hosts the data block (a characteristic called rack-awareness). The last choice is any node in the cluster that has access to HDFS.

3. Combiner Step. It is possible to provide an optimization or pre-reduction as part of the map stage where key–value pairs are combined prior to the next stage. The combiner stage is optional.

4. Shuffle Step. Before the parallel reduction stage can complete, all similar keys must be combined and counted by the same reducer process. Therefore, results of the map stage must be collected by key–value pairs and shuffled to the same reducer process. If only a single reducer process is used, the shuffle stage is not needed.

5. Reduce Step. The final step is the actual reduction. In this stage, the data reduction is performed as per the programmer's design. The reduce step is also optional. The results are written to HDFS. Each reducer will write an output file. For example, a MapReduce job running four reducers will create files called `part-0000`, `part-0001`, `part-0002`, and `part-0003`.

Figure 5.1 is an example of a simple Hadoop MapReduce data flow for a word count program. The map process counts the words in the split, and the reduce process calculates the total for each word. As mentioned earlier, the actual computation of the map and reduce stages are up to the programmer. The MapReduce data flow shown in Figure 5.1 is the same regardless of the specific map and reduce tasks.

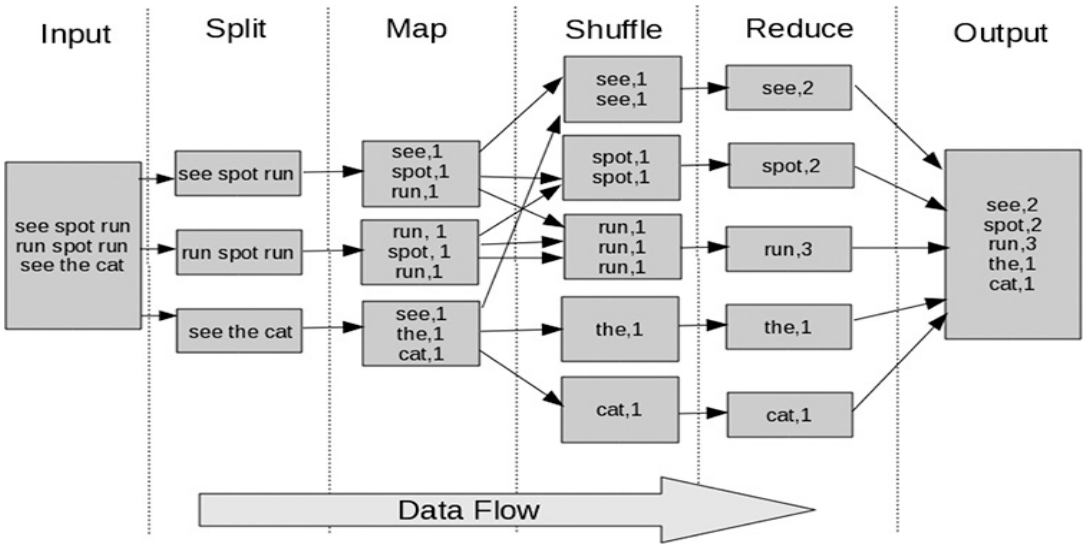


Figure 5.1 Apache Hadoop parallel MapReduce data flow

The input to the MapReduce application is the following file in HDFS with three lines of text. The goal is to count the number of times each word is used.

see spot run
run spot run
see the cat

The first thing MapReduce will do is create the data splits. For simplicity, each line will be one split. Since each split will require a map task, there are three mapper processes that count the number of words in the split. On a cluster, the results of each map task are written to local disk and not to HDFS. Next, similar keys need to be collected and sent to a reducer process. The shuffle step requires data movement and can be expensive in terms of processing time. Depending on the nature of the application, the amount of data that must be shuffled throughout the cluster can vary from small to large.

Once the data have been collected and sorted by key, the reduction step can begin (even if only partial results are available). It is not necessary—and not normally recommended—to have a reducer for each key–value pair as shown in Figure 5.1. In some cases, a single reducer will provide adequate performance; in other cases, multiple reducers may be required to speed up the reduce phase. The number of reducers is a tunable option for many applications. The final step is to write the output to HDFS.

As mentioned, a combiner step enables some pre-reduction of the map output data. For instance, in the previous example, one map produced the following counts:

(run,1)
(spot,1)
(run,1)

As shown in Figure 5.2, the count for `run` can be combined into `(run,2)` before the shuffle. This optimization can help minimize the amount of data transfer needed for the shuffle phase.

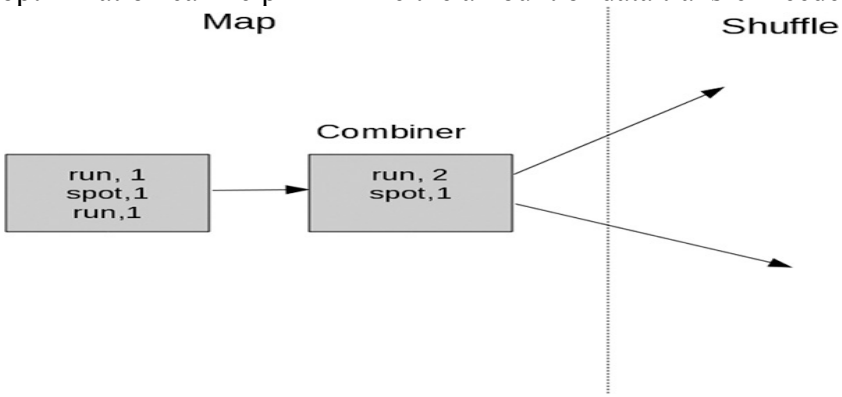


Figure 5.2 Adding a combiner process to the map step in MapReduce

The Hadoop YARN resource manager and the MapReduce framework determine the actual placement of mappers and reducers. As mentioned earlier, the MapReduce framework will try to place the map task as close to the data as possible. It will request the placement from the YARN scheduler but may not get the best placement due to the load on the cluster. In general, nodes can run both mapper and reducer tasks. Indeed, the dynamic nature of YARN enables the work containers used by completed map tasks to be returned to the pool of available resources.

Figure 5.3 shows a simple three-node MapReduce process. Once the mapping is complete, the same nodes begin the reduce process. The shuffle stage makes sure the necessary data are sent to each mapper. Also note that there is no requirement that all the mappers complete at the same time or that the mapper on a specific node be complete before a reducer is started. Reducers can be set to start shuffling based on a threshold of percentage of mappers that have finished.

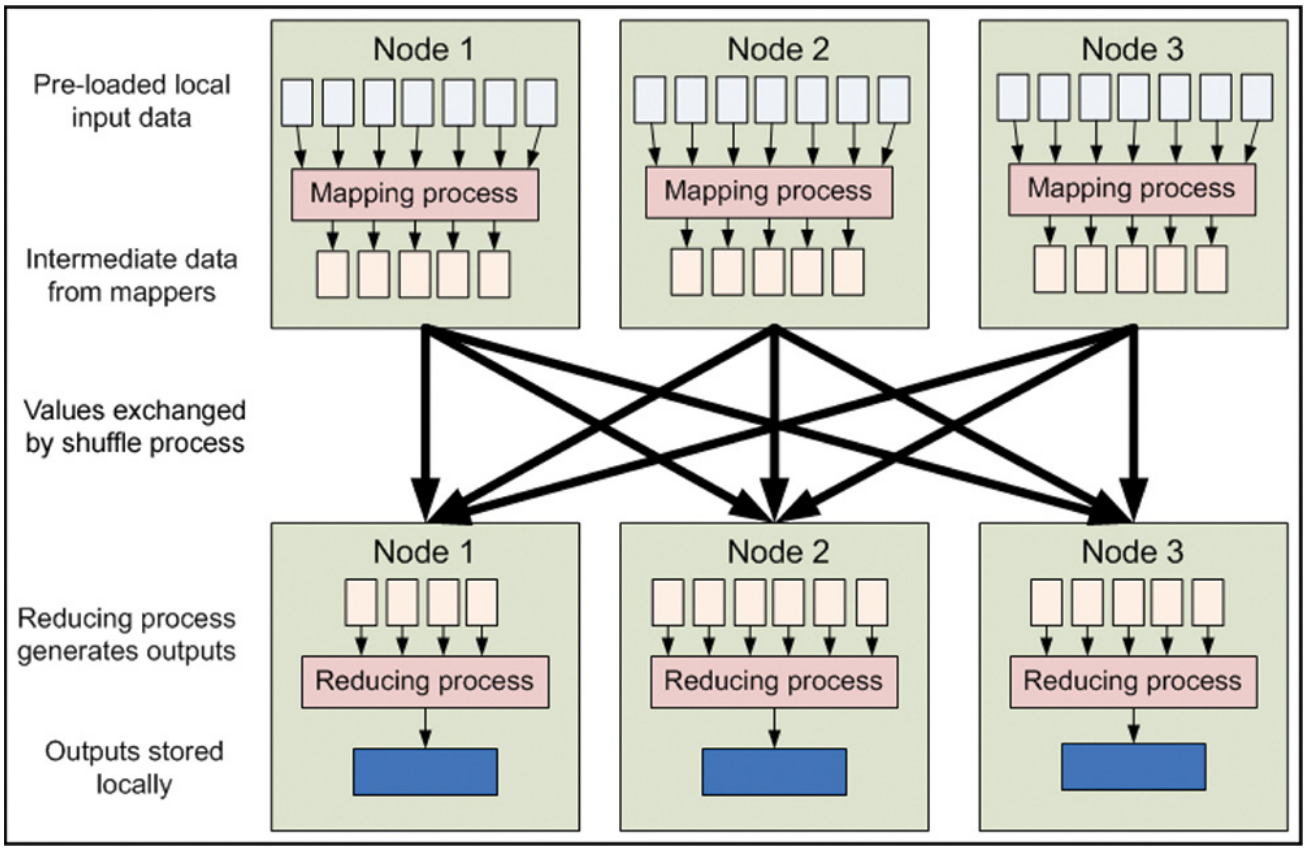


Figure 5.3 Process placement during MapReduce (Adapted from Yahoo Hadoop Documentation)

Finally, although the examples are simple in nature, the parallel MapReduce algorithm can be scaled up to extremely large data sizes. For instance, the Hadoop word count sample application (see Chapter 6, "MapReduce Programming") can be run on the three lines given earlier or on a 3TB file. The application requires no changes to account for the scale of the problem—a feature that is one of the remarkable advantages of MapReduce processing.

FAULT TOLERANCE AND SPECULATIVE EXECUTION

One of the most interesting aspects of parallel MapReduce operation is the strict control of data flow throughout the execution of the program. For example, mapper processes do not exchange data with other mapper processes, and data can only go from mappers to reducers—not the other direction. The confined data flow enables MapReduce to operate in a fault-tolerant fashion.

The design of MapReduce makes it possible to easily recover from the failure of one or many map processes. For example, should a server fail, the map tasks that were running on that machine could easily be restarted on another working server because there is no dependence on any other map task. In functional language terms, the map tasks "do not share state" with other mappers. Of course, the application will run more slowly because work needs to be redone, but it will complete.

In a similar fashion, failed reducers can be restarted. However, there may be additional work that has to be redone in such a case. Recall that a completed reduce task writes results to HDFS. If a node fails after this point, the data should still be available due to the redundancy in HDFS. If

reduce tasks remain to be completed on a down node, the MapReduce ApplicationMaster will need to restart the reducer tasks. If the mapper output is not available for the newly restarted reducer, then these map tasks will need to be restarted. This process is totally transparent to the user and provides a fault-tolerant system to run applications.

Speculative Execution

One of the challenges with many large clusters is the inability to predict or manage unexpected system bottlenecks or failures. In theory, it is possible to control and monitor resources so that network traffic and processor load can be evenly balanced; in practice, however, this problem represents a difficult challenge for large systems. Thus, it is possible that a congested network, slow disk controller, failing disk, high processor load, or some other similar problem might lead to slow performance without anyone noticing.

When one part of a MapReduce process runs slowly, it ultimately slows down everything else because the application cannot complete until all processes are finished. The nature of the parallel MapReduce model provides an interesting solution to this problem. Recall that input data are immutable in the MapReduce process. Therefore, it is possible to start a copy of a running map process without disturbing any other running mapper processes. For example, suppose that as most of the map tasks are coming to a close, the ApplicationMaster notices that some are still running and schedules redundant copies of the remaining jobs on less busy or free servers. Should the secondary processes finish first, the other first processes are then terminated (or vice versa). This process is known as *speculative execution*. The same approach can be applied to reducer processes that seem to be taking a long time. Speculative execution can reduce cluster efficiency because redundant resources are assigned to applications that seem to have a slow spot. It can also be turned off and on in the `mapred-site.xml` configuration file (see Chapter 9, "Managing Hadoop with Apache Ambari").

Hadoop MapReduce Hardware

The capability of Hadoop MapReduce and HDFS to tolerate server—or even whole rack—failures can influence hardware designs. The use of commodity (typically x86_64) servers for Hadoop clusters has made low-cost, high-availability implementations of Hadoop possible for many data centers. Indeed, the Apache Hadoop philosophy seems to assume servers will always fail and takes steps to keep failure from stopping application progress on a cluster.

The use of server nodes for both storage (HDFS) and processing (mappers, reducers) is somewhat different from the traditional separation of these two tasks in the data center. It is possible to build Hadoop systems and separate the roles (discrete storage and processing nodes). However, a majority of Hadoop systems use the general approach where servers enact both roles. Another interesting feature of dynamic MapReduce execution is the capability to tolerate dissimilar servers. That is, old and new hardware can be used together. Of course, large disparities in performance will limit the faster systems, but the dynamic nature of MapReduce execution will still work effectively on such systems.

SUMMARY AND ADDITIONAL RESOURCES

The Apache Hadoop MapReduce framework is a powerful yet simple computation model that can be scaled from one to thousands of processors. The functional nature of MapReduce enables

scalable operation without the need to modify the user's application. In essence, the programmer can focus on the application requirements and not the parallel execution methodology.

Parallel MapReduce data flow is easily understood by examining the various component steps and identifying how key–value pairs traverse the cluster. The Hadoop MapReduce design also makes possible transparent fault tolerance and possible optimizations through speculative execution. Further information on Apache Hadoop MapReduce can be found from the following sources:

- https://developer.yahoo.com/hadoop/tutorial/module4.html (based on Hadoop version 1, but still a good MapReduce background)
- http://en.wikipedia.org/wiki/MapReduce
- http://research.google.com/pubs/pub36249.html

6. MapReduce Programming

In This Chapter:

- The classic Java WordCount program for Hadoop is compiled and run.
- A Python WordCount application using the Hadoop streaming interface is introduced.
- The Hadoop Pipes interface is used to run a C++ version of WordCount.
- An example of MapReduce chaining is presented using the Hadoop Grep example.
- Strategies for MapReduce debugging are presented.

At the base level, Hadoop provides a platform for Java-based MapReduce programming. These applications run natively on most Hadoop installations. To offer more variability, a streaming interface is provided that enables almost any programming language to take advantage of the Hadoop MapReduce engine. In addition, a pipes C++ interface is provided that can work directly with the MapReduce components. This chapter provides programming examples of these interfaces and presents some debugging strategies.

COMPILING AND RUNNING THE HADOOP WORDCOUNT EXAMPLE

The Apache Hadoop `WordCount.java` program for Hadoop version 2, shown in Listing 6.1, is the equivalent of the C programing language `hello-world.c` example. It should be noted that two versions of this program can be found on the Internet. The Hadoop version 1 example uses the older `org.apache.hadoop.mapred` API, while the Hadoop version 2 example, shown here in Listing 6.1, uses the newer `org.apache.hadoop.mapreduce` API. If you experience errors compiling `WordCount.java`, double-check the source code and Hadoop versions.

Listing 6.1 **WordCount.java**

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
```

```
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new
StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException,
InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

WordCount is a simple application that counts the number of occurrences of each word in a given input set. The example will work with all installation methods presented in Chapter 2, "Installation Recipes" (i.e., HDP Sandbox, pseudo-distributed, full cluster, or cloud).

As discussed in Chapter 5, the MapReduce framework operates exclusively on key–value pairs; that is, the framework views the input to the job as a set of key–value pairs and produces a set of key–value pairs of different types. The MapReduce job proceeds as follows:

(input) <k1, v1> -> **`map`** -> <k2, v2> -> **`combine`** -> <k2, v2> -> **`reduce`** -> <k3, v3> (output)

The mapper implementation, via the `map` method, processes one line at a time as provided by the specified `TextInputFormat` class. It then splits the line into tokens separated by whitespaces using the `StringTokenizer` and emits a key–value pair of `<word, 1>`. The relevant code section is as follows:

```
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
```

Given two input files with contents `Hello World Bye World` and `Hello Hadoop Goodbye Hadoop`, the WordCount mapper will produce two maps:

```
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>

< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>
```

As can be seen in Listing 6.1, WordCount sets a mapper

Click here to view code image

```
job.setMapperClass(TokenizerMapper.class);
```

a combiner

```
job.setCombinerClass(IntSumReducer.class);
```

and a reducer

job.setReducerClass(IntSumReducer.class);

Hence, the output of each map is passed through the local combiner (which sums the values in the same way as the reducer) for local aggregation and then sends the data on to the final reducer. Thus, each map above the combiner performs the following pre-reductions:

< Bye, 1>
< Hello, 1>
< World, 2>

< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>

The reducer implementation, via the reduce method, simply sums the values, which are the occurrence counts for each key. The relevant code section is as follows:

```
public void reduce(Text key, Iterable<IntWritable> values,
                   Context context
                   ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
```

The final output of the reducer is the following:

< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>`

The source code for `WordCount.java` is available from the book download page (see Appendix A, "Book Webpage and Code Download"). To compile and run the program from the command line, perform the following steps:

1. Make a local `wordcount_classes` directory.

```
$ mkdir wordcount_classes
```

2. Compile the `WordCount.java`a program using the `'hadoop classpath'` command to include all the available Hadoop class paths.

```
$ javac -cp `hadoop classpath` -d wordcount_classes
WordCount.java
```

3. The jar file can be created using the following command:

```
$ jar -cvf wordcount.jar -C wordcount_classes/
```

4. To run the example, create an input directory in HDFS and place a text file in the new directory. For this example, we will use the `war-and-peace.txt` file (available from the book download page; see Appendix A):

```
$ hdfs dfs -mkdir war-and-peace-input
$ hdfs dfs -put war-and-peace.txt war-and-peace-input
```

5. Run the WordCount application using the following command:

```
$ hadoop jar wordcount.jar WordCount war-and-peace-input war-
and-peace-output
```

If everything is working correctly, Hadoop messages for the job should look like the following (abbreviated version):

```
15/05/24 18:13:26 INFO impl.TimelineClientImpl: Timeline service address:
http://limulus:8188/ws/v1/timeline/
15/05/24 18:13:26 INFO client.RMProxy: Connecting to ResourceManager at
limulus/10.0.0.1:8050
15/05/24 18:13:26 WARN mapreduce.JobSubmitter: Hadoop command-line option parsing
not performed. Implement the Tool interface and execute your application with
ToolRunner to remedy this.
15/05/24 18:13:26 INFO input.FileInputFormat: Total input paths to process : 1
15/05/24 18:13:27 INFO mapreduce.JobSubmitter: number of splits:1
[...]
File Input Format Counters
        Bytes Read=3288746
File Output Format Counters
        Bytes Written=467839
```

In addition, the following files should be in the `war-and-peace-output` directory. The actual file name may be slightly different depending on your Hadoop version.

```
$ hdfs dfs -ls war-and-peace-output
Found 2 items
-rw-r--r--   2 hdfs hdfs          0 2015-05-24 11:14 war-and-peace-output/_SUCCESS
-rw-r--r--   2 hdfs hdfs     467839 2015-05-24 11:14 war-and-peace-output/part-r-00000
```

The complete list of word counts can be copied from HDFS to the working directory with the following command:

$ hdfs dfs -get war-and-peace-output/part-r-00000.

If the WordCount program is run again using the same outputs, it will fail when it tries to overwrite the `war-and-peace-output` directory. The output directory and all contents can be removed with the following command:

$ hdfs dfs -rm -r -skipTrash war-and-peace-output

USING THE STREAMING INTERFACE

The Apache Hadoop steaming interface enables almost any program to use the MapReduce engine. The streams interface will work with any program that can read and write to `stdin` and `stdout`.

When working in the Hadoop streaming mode, only the mapper and the reducer are created by the user. This approach does have the advantage that the mapper and the reducer can be easily tested from the command line. In this example, a Python mapper and reducer, shown in Listings 6.2 and 6.3, will be used. The source code can be found on the book download page (see Appendix A) or at http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python.

Listing 6.2 **Python Mapper Script (mapper.py)**

```
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

Listing 6.3 **Python Reducer Script (reducer.py)**

```
#!/usr/bin/env python

from operator import itemgetter
import sys
```

```
current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)
    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

The operation of the `mapper.py` script can be observed by running the command as shown in the following:

```
$ echo "foo foo quux labs foo bar quux" | ./mapper.py
Foo    1
Foo    1
Quux   1
Labs   1
Foo    1
Bar    1
Quux   1
```

Piping the results of the map into the `sort` command can create a simulated shuffle phase:

```
$ echo "foo foo quux labs foo bar quux" | ./mapper.py|sort -k1,1
Bar     1
Foo     1
Foo     1
Foo     1
Labs    1
Quux    1
Quux    1
```

Finally, the full MapReduce process can be simulated by adding the `reducer.py` script to the following command pipeline:

```
$ echo "foo foo quux labs foo bar quux" | ./mapper.py|sort -k1,1|./reducer.py
Bar     1
Foo     3
Labs    1
Quux    2
```

To run this application using a Hadoop installation, create, if needed, a directory and move the `war-and-peace.txt` input file into HDFS:

```
$ hdfs dfs -mkdir war-and-peace-input
$ hdfs dfs -put war-and-peace.txt war-and-peace-input
```

Make sure the output directory is removed from any previous test runs:

```
$ hdfs dfs -rm -r -skipTrash war-and-peace-output
```

Locate the `hadoop-streaming.jar` file in your distribution. The location may vary, and it may contain a version tag. In this example, the Hortonworks HDP 2.2 distribution was used. The following command line will use the `mapper.py` and `reducer.py` to do a word count on the input file.

```
$ hadoop jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar \
-file ./mapper.py \
-mapper ./mapper.py \
-file ./reducer.py -reducer ./reducer.py \
-input war-and-peace-input/war-and-peace.txt \
-output war-and-peace-output
```

The output will be the familiar (`_SUCCESS` and `part-00000`) in the `war-and-peace-output`directory. The actual file name may be slightly different depending on your Hadoop version. Also note that the Python scripts used in this example could be Bash, Perl, Tcl, Awk, compiled C code, or any language that can read and write from `stdin` and `stdout`.
Although the streaming interface is rather simple, it does have some disadvantages over using Java directly. In particular, not all applications are string and character based, and it would be

awkward to try to use `stdin` and `stdout` as a way to transmit binary data. Another disadvantage is that many tuning parameters available through the full Java Hadoop API are not available in streaming.

USING THE PIPES INTERFACE

Pipes is a library that allows C++ source code to be used for mapper and reducer code. Applications that require high performance when crunching numbers may achieve better throughput if written in C++ and used through the Pipes interface.

Both key and value inputs to pipes programs are provided as STL strings (`std::string`). As shown in Listing 6.4, the program must define an instance of a mapper and an instance of a reducer. A program to use with Pipes is defined by writing classes extending `Mapper` and `Reducer`. Hadoop must then be informed as to which classes to use to run the job.

The Pipes framework on each machine assigned to your job will start an instance of your C++ program. Therefore, the executable must be placed in HDFS prior to use.

Listing 6.4 **wordcount.cpp and Example of Hadoop Pipes Interface Using C++**

```
#include <algorithm>
#include <limits>
#include <string>
#include  "stdint.h"  // <--- to prevent uint64_t errors!
#include "Pipes.hh"
#include "TemplateFactory.hh"
#include "StringUtils.hh"

using namespace std;
class WordCountMapper : public HadoopPipes::Mapper {
public:
  // constructor: does nothing
  WordCountMapper( HadoopPipes::TaskContext& context ) {
  }
  // map function: receives a line, outputs (word,"1")
  // to reducer.
  void map( HadoopPipes::MapContext& context ) {
    //--- get line of text ---
    string line = context.getInputValue();
    //--- split it into words ---
    vector< string > words =
      HadoopUtils::splitString( line, " " );
    //--- emit each word tuple (word, "1" ) ---
    for ( unsigned int i=0; i < words.size(); i++ ) {
      context.emit( words[i], HadoopUtils::toString( 1 ) );
    }
  }
};
```

```
class WordCountReducer : public HadoopPipes::Reducer {
public:
  // constructor: does nothing
  WordCountReducer(HadoopPipes::TaskContext& context) {
  }
  // reduce function
  void reduce( HadoopPipes::ReduceContext& context ) {
    int count = 0;
    //--- get all tuples with the same key, and count their
numbers ---
    while ( context.nextValue() ) {
      count += HadoopUtils::toInt( context.getInputValue() );
    }
//--- emit (word, count) ---
    context.emit(context.getInputKey(), HadoopUtils::toString(
count ));
  }
};
int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(HadoopPipes::TemplateFactory<
                              WordCountMapper,
                              WordCountReducer >() );
}
```

The `wordcount.cpp` source is available from the book download page (see Appendix A) or from http://wiki.apache.org/hadoop/C++WordCount. The location of the Hadoop `include` files and libraries may need to be specified when compiling the code. If `$HADOOP_HOME` is defined, the following options should provide the correct path. Check to make sure the paths are correct for your installation.

-L$HADOOP_HOME/lib/native/ -I$HADOOP_HOME/include

Additionally, the original source code may need to be changed depending on where the `include` files are located (i.e., some distributions may not use the `hadoop` prefix). In Listing 6.4, the following lines (from the original program) had the `hadoop` prefix removed:

#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"

The program can be compiled with the following line (adjusted for `include` file and library locations). In this example, use of Hortonworks HDP 2.2 is assumed.

$ g++ wordcount.cpp -o wordcount -L$HADOOP_HOME/lib/native/ -I$HADOOP_HOME/../usr/include -lhadooppipes -lhadooputils -lpthread -lcrypto

If needed, create the `war-and-peace-input` directory and move the file into HDFS:

```
$ hdfs dfs -mkdir war-and-peace-input
$ hdfs dfs -put war-and-peace.txt war-and-peace-input
```

As mentioned, the executable must be placed into HDFS so YARN can find the program. Also, the output directory must be removed before running the program:

```
$ hdfs dfs -put wordcount bin
$ hdfs dfs -rm -r -skipTrash war-and-peace-output
```

To run the program, enter the following line (shown in multiple lines for clarity). The lines specifying the `recordreader` and `recordwriter` indicate that the default Java text versions should be used. Also note that the location of the program in HDFS must be specified.

```
$ mapred pipes \
-D hadoop.pipes.java.recordreader=true  \
-D hadoop.pipes.java.recordwriter=true \
-input war-and-peace-input   \
-output war-and-peace-output  \
-program bin/wordcount
```

When run, the program will produce the familiar output (`_SUCCESS` and `part-00000`) in the `war-and-peace-output` directory. The `part-00000` file should be identical to the Java WordCount version.

COMPILING AND RUNNING THE HADOOP GREP CHAINING EXAMPLE

The Hadoop `Grep.java` example extracts matching strings from text files and counts how many times they occurred. The command works differently from the *nix `grep` command in that it does not display the complete matching line, only the matching string. If matching lines are needed for the string `foo`, use `.*foo.*` as a regular expression.

The program runs two map/reduce jobs in sequence and is an example of *MapReduce chaining*. The first job counts how many times a matching string occurs in the input, and the second job sorts matching strings by their frequency and stores the output in a single file. Listing 6.5 displays the source code for `Grep.java`. It is also available from the book download page (see Appendix A) or directly from the Hadoop examples source jar file.
Note that all the Hadoop example source files can be extracted by locating the `hadoop-mapreduce-examples-*-sources.jar` either from a Hadoop distribution or from the Apache Hadoop website (as part of a full Hadoop package) and then extracting the files using the following command (your version tag may be different):

Click here to view code image

```
$ jar xf hadoop-mapreduce-examples-2.6.0-sources.jar
```

Listing 6.5 **Hadoop Grep.java Example**

```
package org.apache.hadoop.examples;

import java.util.Random;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import
org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.map.InverseMapper;
import org.apache.hadoop.mapreduce.lib.map.RegexMapper;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import
org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.hadoop.mapreduce.lib.reduce.LongSumReducer;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

/* Extracts matching regexs from input files and counts them. */
public class Grep extends Configured implements Tool {
  private Grep() {}                                // singleton
  public int run(String[] args) throws Exception {
    if (args.length < 3) {
      System.out.println("Grep <inDir> <outDir> <regex>
[<group>]");
      ToolRunner.printGenericCommandUsage(System.out);
      return 2;
    }
    Path tempDir =
      new Path("grep-temp-"+
          Integer.toString(new
Random().nextInt(Integer.MAX_VALUE)));
    Configuration conf = getConf();
    conf.set(RegexMapper.PATTERN, args[2]);
    if (args.length == 4)
      conf.set(RegexMapper.GROUP, args[3]);

    Job grepJob = new Job(conf);
    try {
      grepJob.setJobName("grep-search");
      FileInputFormat.setInputPaths(grepJob, args[0]);
      grepJob.setMapperClass(RegexMapper.class);
      grepJob.setCombinerClass(LongSumReducer.class);
```

```
      grepJob.setReducerClass(LongSumReducer.class);
      FileOutputFormat.setOutputPath(grepJob, tempDir);
      grepJob.setOutputFormatClass(SequenceFileOutputFormat.clas
s);
      grepJob.setOutputKeyClass(Text.class);
      grepJob.setOutputValueClass(LongWritable.class);
      grepJob.waitForCompletion(true);

      Job sortJob = new Job(conf);
      sortJob.setJobName("grep-sort");
      FileInputFormat.setInputPaths(sortJob, tempDir);
      sortJob.setInputFormatClass(SequenceFileInputFormat.class)
;
      sortJob.setMapperClass(InverseMapper.class);
      sortJob.setNumReduceTasks(1);                 // write a
single file
      FileOutputFormat.setOutputPath(sortJob, new
Path(args[1]));
      sortJob.setSortComparatorClass(          // sort by
decreasing freq
        LongWritable.DecreasingComparator.class);
      sortJob.waitForCompletion(true);
    }
    finally {
      FileSystem.get(conf).delete(tempDir, true);
    }
    return 0;
  }
  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new Grep(),
args);
    System.exit(res);
  }
}
```

In the preceding code, each mapper of the first job takes a line as input and matches the user-provided regular expression against the line. The `RegexMapper` class is used to perform this task and extracts text matching using the given regular expression. The matching strings are output as <matching string, 1> pairs. As in the previous WordCount example, each reducer sums up the number of matching strings and employs a combiner to do local sums. The actual reducer uses the `LongSumReducer` class that outputs the sum of long values per reducer input key.
The second job takes the output of the first job as its input. The mapper is an inverse map that reverses (or swaps) its input <key, value> pairs into <value, key>. There is no reduction step, so the `IdentityReducer` class is used by default. All input is simply passed to the output. (Note: There is also an `IdentityMapper` class.) The number of reducers is set to 1, so the output is stored in one file and it is sorted by count in descending order. The output text file contains a count and a string per line.

The example also demonstrates how to pass a command-line parameter to a mapper or a reducer.

The following discussion describes how to compile and run the `Grep.java` example. The steps are similar to the previous WordCount example:

1. Create a directory for the application classes as follows:

```
$ mkdir Grep_classes
```

2. Compile the `WordCount.java` program using the following line:

```
$ javac -cp `hadoop classpath`  -d Grep_classes Grep.java
```

3. Create a Java archive using the following command:

```
$ jar -cvf Grep.jar -C Grep_classes/ .
```

If needed, create a directory and move the `war-and-peace.txt` file into HDFS:

```
$ hdfs dfs -mkdir war-and-peace-input
$ hdfs dfs -put war-and-peace.txt war-and-peace-input
```

As always, make sure the output directory has been removed by issuing the following command:

```
$ hdfs dfs -rm -r -skipTrash war-and-peace-output
```

Entering the following command will run the Grep program:

```
$ hadoop jar Grep.jar org.apache.hadoop.examples.Grep war-and-peace-input war-and-peace-output Kutuzov
```

As the example runs, two stages will be evident. Each stage is easily recognizable in the program output. The results can be found by examining the resultant output file.

```
$ hdfs dfs -cat war-and-peace-output/part-r-00000
530    Kutuzov
```

DEBUGGING MAPREDUCE

The best advice for debugging *parallel* MapReduce applications is this: Don't. The key word here is *parallel*. Debugging on a distributed system is hard and should be avoided at all costs.

The best approach is to make sure applications run on a simpler system (i.e., the HDP Sandbox or the pseudo-distributed single-machine install) with smaller data sets. Errors on these systems are much easier to locate and track. In addition, unit testing applications before running at scale is important. If applications can run successfully on a single system with a subset of real data, then running in parallel should be a simple task because the MapReduce algorithm is transparently scalable. Note that many higher-level tools (e.g., Pig and Hive) enable local mode development for this reason. Should errors occur at scale, the issue can be tracked from the log

file (see the section "Hadoop Log Management") and may stem from a systems issue rather than a program artifact.

When investigating program behavior at scale, the best approach is to use the application logs to inspect the actual MapReduce progress. The time-tested debug print statements are also visible in the logs.

Listing, Killing, and Job Status

As mentioned in Chapter 4, "Running Example Programs and Benchmarks," jobs can be managed using the `mapred job` command. The most import options are `-list`, `-kill`, and `-status`. In addition, the `yarn application` command can be used to control all applications running on the cluster (See Chapter 10, "Basic Hadoop Administration Procedures").

Hadoop Log Management

The MapReduce logs provide a comprehensive listing of both mappers and reducers. The actual log output consists of three files—`stdout`, `stderr`, and `syslog` (Hadoop system messages)—for the application. There are two modes for log storage. The first (and best) method is to use log aggregation. In this mode, logs are aggregated in HDFS and can be displayed in the YARN ResourceManager user interface (see Figure 6.1) or examined with the `yarn logs` command (see the section "Command-Line Log Viewing").

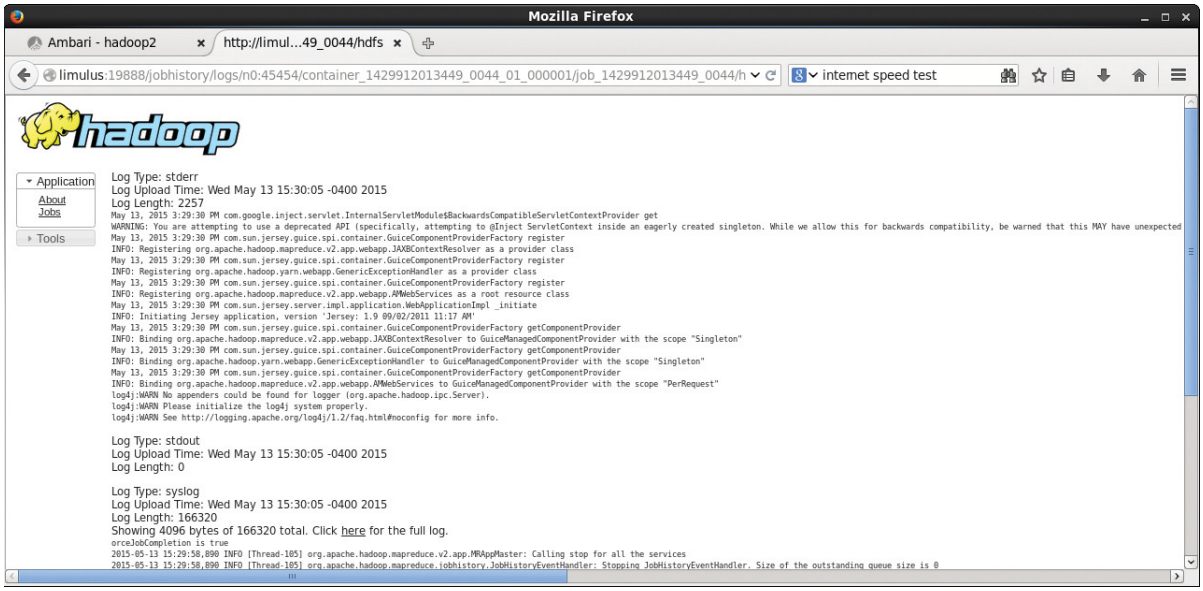


Figure 6.1 Log information for map process (stdout, stderr, and syslog)

If log aggregation is not enabled, the logs will be placed locally on the cluster nodes on which the mapper or reducer ran. The location of the unaggregated local logs is given by the `yarn.nodemanager.log-dirs` property in the `yarn-site.xml` file. Without log aggregation, the cluster nodes used by the job must be noted, and then the log files must be obtained directly from the nodes. Log aggregation is *highly recommended*.

Enabling YARN Log Aggregation

If Apache Hadoop was installed from the official Apache Hadoop sources, the following settings will ensure log aggregation is turned on for the system.

If you are using Ambari or some other management tool, change the setting using that tool (see Chapter 9, "Managing Hadoop with Apache Ambari"). Do not handmodify the configuration files. For example, if you are using Apache Ambari, check under the YARN Service Configs tab for `yarn.log-aggregation-enable`. The default setting is enabled.

Note

Log aggregation is disabled in the pseudo-distributed installation presented in Chapter 2.

To manually enable log aggregation, follows these steps:

1. As the HDFS superuser administrator (usually user `hdfs`), create the following directory in HDFS:

```
$ hdfs dfs -mkdir -p /yarn/logs
$ hdfs dfs -chown -R yarn:hadoop /yarn/logs
$ hdfs dfs -chmod -R g+rw /yarn/logs
```

2. Add the following properties in the `yarn-site.xml` (on all nodes) and restart all YARN services on all nodes (the ResourceManager, NodeManagers, and JobHistoryServer).

```
<property>
  <name>yarn.nodemanager.remote-app-log-dir</name>
  <value>/yarn/logs</value>
</property>
<property>
  <name>yarn.log-aggregation-enable</name>
  <value>true</value>
</property>
```

Web Interface Log View

The most convenient way to view logs is to use the YARN ResourceManager web user interface. In Chapter 4, a list of mapper tasks is shown in Figure 4.8. Each task has a link to the logs for that task. If log aggregation is enabled, clicking on the log link will show a window similar to Figure 6.1.

In the figure, the contents of `stdout`, `stderr`, and `syslog` are displayed on a single page. If log aggregation is not enabled, a message stating that the logs are not available will be displayed.

Command-Line Log Viewing

MapReduce logs can also be viewed from the command line. The `yarn logs` command enables the logs to be easily viewed together without having to hunt for individual log files on the cluster nodes. As before, log aggregation is required for use. The options to `yarn logs` are as follows:

```
$ yarn logs
Retrieve logs for completed YARN applications.
usage: yarn logs -applicationId <application ID> [OPTIONS]
```

```
general options are:
 -appOwner <Application Owner>   AppOwner (assumed to be current user if
                                 not specified)
 -containerId <Container ID>     ContainerId (must be specified if node
                                 address is specified)
 -nodeAddress <Node Address>     NodeAddress in the format nodename:port
                                 (must be specified if container id is
                                 specified)
```

For example, after running the pi example program (discussed in Chapter 4), the logs can be examined as follows:

```
$ hadoop jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar pi 16 100000
```

After the pi example completes, note the applicationId, which can be found either from the application output or by using the `yarn application` command. The applicationId will start with `application_` and appear under the Application-Id column.

```
$ yarn application -list -appStates FINISHED
```

Next, run the following command to produce a dump of all the logs for that application. Note that the output can be long and is best saved to a file.

```
$ yarn logs -applicationId application_1432667013445_0001 > AppOut
```

The `AppOut` file can be inspected using a text editor. Note that for each container, `stdout`, `stderr`, and `syslog` are provided (the same as the GUI version in Figure 6.1). The list of actual containers can be found by using the following command:

```
$ grep -B 1 ===== AppOut
```

For example (output truncated):

```
 [...]
Container: container_1432667013445_0001_01_000008 on limulus_45454
==========================================================================
--
Container: container_1432667013445_0001_01_000010 on limulus_45454
==========================================================================
--
Container: container_1432667013445_0001_01_000001 on n0_45454
==========================================================
--
Container: container_1432667013445_0001_01_000023 on n1_45454
==========================================================
```

[...]

A specific container can be examined by using the containerId and the nodeAddress from the preceding output. For example, `container_1432667013445_0001_01_000023` can be examined by entering the command following this paragraph. Note that the node name (`n1`) and port number are written as `n1_45454` in the command output. To get the nodeAddress, simply replace the _ with a `:` (i.e., `-nodeAddress n1:45454`). Thus, the results for a single container can be found by entering this line:

```
$ yarn logs -applicationId application_1432667013445_0001 -containerId
container_1432667013445_0001_01_000023 -nodeAddress n1:45454|more
```

SUMMARY AND ADDITIONAL RESOURCES

Writing MapReduce programs for Hadoop can be done in a variety of ways. The most direct method uses Java and the current MapReduce API. The `WordCount.java` example is a good starting point from which to explore this process.

Apache Hadoop also offers a streaming interface that enables the user to write mappers and reducers in any language that supports the `stdin` and `stdout` interfaces. These text-based applications can be written in almost any programming language.
The Hadoop Pipes interface enables MapReduce applications to be written in C++ and run directly on the cluster. The Hadoop `Grep.java` application provides an example of cascading MapReduce and can be used as a starting point for further exploration.

Finally, the need to debug MapReduce applications can be minimized by careful testing and staging of those applications before they are run at scale (over many servers in the cluster). Hadoop log analysis provides plenty of information to assist with debugging and is available both in the web YARN ResourceManager GUI and from the command line.

Additional information and background on the MapReduce programming methods can be found from the following resources:

- **Apache Hadoop Java MapReduce example**
- http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v1.0
- **Apache Hadoop streaming example**
- http://hadoop.apache.org/docs/r1.2.1/streaming.html
- http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python
- **Apache Hadoop pipes example**
- http://wiki.apache.org/hadoop/C++WordCount
- https://developer.yahoo.com/hadoop/tutorial/module4.html#pipes
- **Apache Hadoop grep example**
- http://wiki.apache.org/hadoop/Grep
- https://developer.yahoo.com/hadoop/tutorial/module4.html#chaining
- **Debugging MapReduce**

- http://wiki.apache.org/hadoop/HowToDebugMapReducePrograms
- http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Debugging

FUTURE VISION BIE

By K B Hemanth Raj

Visit : https://hemanthrajhemu.github.io

Quick Links for Faster Access.

CSE 8th Semester - https://hemanthrajhemu.github.io/CSE8/

ISE 8th Semester - https://hemanthrajhemu.github.io/ISE8/

ECE 8th Semester - https://hemanthrajhemu.github.io/ECE8/

8th Semester CSE - TEXTBOOK - NOTES - QP - SCANNER & MORE

17CS81 IOT - https://hemanthrajhemu.github.io/CSE8/17SCHEME/17CS81/

17CS82 BDA - https://hemanthrajhemu.github.io/CSE8/17SCHEME/17CS82/

17CS832 UID - https://hemanthrajhemu.github.io/CSE8/17SCHEME/17CS832/

17CS834 SMS - https://hemanthrajhemu.github.io/CSE8/17SCHEME/17CS834/

8th Semester Computer Science & Engineering (CSE)

8th Semester CSE Text Books: https://hemanthrajhemu.github.io/CSE8/17SCHEME/Text-Book.html

8th Semester CSE Notes: https://hemanthrajhemu.github.io/CSE8/17SCHEME/Notes.html

8th Semester CSE Question Paper: https://hemanthrajhemu.github.io/CSE8/17SCHEME/Question-Paper.html

8th Semester CSE Scanner: https://hemanthrajhemu.github.io/CSE8/17SCHEME/Scanner.html

8th Semester CSE Question Bank: https://hemanthrajhemu.github.io/CSE8/17SCHEME/Question-Bank.html

8th Semester CSE Answer Script: https://hemanthrajhemu.github.io/CSE8/17SCHEME/Answer-Script.html

Contribution Link:

https://hemanthrajhemu.github.io/Contribution/

Stay Connected... get Updated... ask your queries...

Join Telegram to get Instant Updates:
https://telegram.me/joinchat/AAAAAFTtp8kuvCHALxuMaQ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/futurevisionbie/