CHAPTER 1

Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is a Java-based distributed, scalable, and portable filesystem designed to span large clusters of commodity servers. The design of HDFS is based on GFS, the Google File System, which is described in a paper published by Google. Like many other distributed filesystems, HDFS holds a large amount of data and provides transparent access to many clients distributed across a network. Where HDFS excels is in its ability to store very large files in a reliable and scalable manner.

HDFS is designed to store a lot of information, typically petabytes (for very large files), gigabytes, and terabytes. This is accomplished by using a block-structured filesystem. Individual files are split into fixed-size blocks that are stored on machines across the cluster. Files made of several blocks generally do not have all of their blocks stored on a single machine.

HDFS ensures reliability by replicating blocks and distributing the replicas across the cluster. The default replication factor is three, meaning that each block exists three times on the cluster. Block-level replication enables data availability even when machines fail.

This chapter begins by introducing the core concepts of HDFS and explains how to interact with the filesystem using the native built-in commands. After a few examples, a Python client library is introduced that enables HDFS to be accessed programmatically from within Python applications.

Overview of HDFS

The architectural design of HDFS is composed of two processes: a process known as the NameNode holds the metadata for the filesystem, and one or more DataNode processes store the blocks that make up the files. The NameNode and DataNode processes can run on a single machine, but HDFS clusters commonly consist of a dedicated server running the NameNode process and possibly thousands of machines running the DataNode process.

The NameNode is the most important machine in HDFS. It stores metadata for the entire filesystem: filenames, file permissions, and the location of each block of each file. To allow fast access to this information, the NameNode stores the entire metadata structure in memory. The NameNode also tracks the replication factor of blocks, ensuring that machine failures do not result in data loss. Because the NameNode is a single point of failure, a secondary NameNode can be used to generate snapshots of the primary NameNode's memory structures, thereby reducing the risk of data loss if the NameNode fails.

The machines that store the blocks within HDFS are referred to as DataNodes. DataNodes are typically commodity machines with large storage capacities. Unlike the NameNode, HDFS will continue to operate normally if a DataNode fails. When a DataNode fails, the NameNode will replicate the lost blocks to ensure each block meets the minimum replication factor.

The example in Figure 1-1 illustrates the mapping of files to blocks in the NameNode, and the storage of blocks and their replicas within the DataNodes.

The following section describes how to interact with HDFS using the built-in commands.

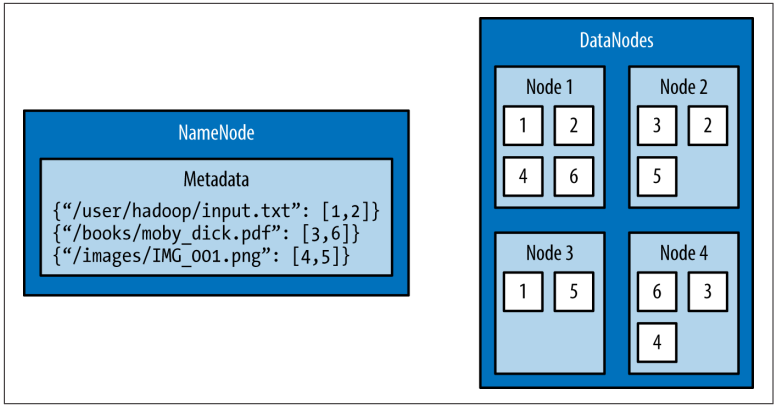

Figure 1-1. An HDFS cluster with a replication factor of two; the NameNode contains the mapping of files to blocks, and the DataNodes store the blocks and their replicas

Interacting with HDFS

Interacting with HDFS is primarily performed from the command line using the script named `hdfs`. The `hdfs` script has the following usage:

```
$ hdfs COMMAND [-option <arg>]
```

The `COMMAND` argument instructs which functionality of HDFS will be used. The `-option` argument is the name of a specific option for the specified command, and `<arg>` is one or more arguments that that are specified for this option.

Common File Operations

To perform basic file manipulation operations on HDFS, use the `dfs` command with the `hdfs` script. The `dfs` command supports many of the same file operations found in the Linux shell.

It is important to note that the `hdfs` command runs with the permissions of the system user running the command. The following examples are run from a user named "hduser."

List Directory Contents

To list the contents of a directory in HDFS, use the `-ls` command:

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

Running the `-ls` command on a new cluster will not return any results. This is because the `-ls` command, without any arguments, will attempt to display the contents of the user's home directory on HDFS. This is not the same home directory on the host machine (e.g., */home/$USER*), but is a directory within HDFS.

Providing `-ls` with the forward slash (`/`) as an argument displays the contents of the root of HDFS:

```
$ hdfs dfs -ls /
Found 2 items
drwxr-xr-x   - hadoop supergroup    0 2015-09-20 14:36 /hadoop
drwx------   - hadoop supergroup    0 2015-09-20 14:36 /tmp
```

The output provided by the `hdfs dfs` command is similar to the output on a Unix filesystem. By default, `-ls` displays the file and folder permissions, owners, and groups. The two folders displayed in this example are automatically created when HDFS is formatted. The `hadoop` user is the name of the user under which the Hadoop daemons were started (e.g., NameNode and DataNode), and the `supergroup` is the name of the group of superusers in HDFS (e.g., hadoop).

Creating a Directory

Home directories within HDFS are stored in */user/$HOME*. From the previous example with `-ls`, it can be seen that the */user* directory does not currently exist. To create the */user* directory within HDFS, use the `-mkdir` command:

```
$ hdfs dfs -mkdir /user
```

To make a home directory for the current user, `hduser`, use the `-mkdir` command again:

```
$ hdfs dfs -mkdir /user/hduser
```

Use the `-ls` command to verify that the previous directories were created:

```
$ hdfs dfs -ls -R /user
drwxr-xr-x   - hduser supergroup     0 2015-09-22 18:01 /user/
hduser
```

Copy Data onto HDFS

After a directory has been created for the current user, data can be uploaded to the user's HDFS home directory with the `-put` command:

```
$ hdfs dfs -put /home/hduser/input.txt /user/hduser
```

This command copies the file */home/hduser/input.txt* from the local filesystem to */user/hduser/input.txt* on HDFS.

Use the `-ls` command to verify that *input.txt* was moved to HDFS:

```
$ hdfs dfs -ls
Found 1 items
-rw-r--r--   1 hduser supergroup          52 2015-09-20 13:20
input.txt
```

Retrieving Data from HDFS

Multiple commands allow data to be retrieved from HDFS. To simply view the contents of a file, use the `-cat` command. `-cat` reads a file on HDFS and displays its contents to stdout. The following command uses `-cat` to display the contents of */user/hduser/input.txt*:

```
$ hdfs dfs -cat input.txt
jack be nimble
jack be quick
jack jumped over the candlestick
```

Data can also be copied from HDFS to the local filesystem using the `-get` command. The `-get` command is the opposite of the `-put` command:

```
$ hdfs dfs -get input.txt /home/hduser
```

This command copies *input.txt* from */user/hduser* on HDFS to */home/hduser* on the local filesystem.

HDFS Command Reference

The commands demonstrated in this section are the basic file operations needed to begin using HDFS. Below is a full listing of file manipulation commands possible with `hdfs dfs`. This listing can also be displayed from the command line by specifying `hdfs dfs` without any arguments. To get help with a specific option, use either `hdfs dfs -usage <option>` or `hdfs dfs -help <option>`.

```
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]
    [-find <path> ... <expression> ...]
    [-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 configu-
ration 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 separa-
```

```
ted files to be copied to the map reduce cluster
-libjars <comma separated list of jars>    specify comma sepa-
rated 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]
```

The next section introduces a Python library that allows HDFS to be accessed from within Python applications.

Snakebite

Snakebite is a Python package, created by Spotify, that provides a Python client library, allowing HDFS to be accessed programmatically from Python applications. The client library uses protobuf messages to communicate directly with the NameNode. The Snakebite package also includes a command-line interface for HDFS that is based on the client library.

This section describes how to install and configure the Snakebite package. Snakebite's client library is explained in detail with multiple examples, and Snakebite's built-in CLI is introduced as a Python alternative to the `hdfs dfs` command.

Installation

Snakebite requires Python 2 and python-protobuf 2.4.1 or higher. Python 3 is currently not supported.

Snakebite is distributed through PyPI and can be installed using `pip`:

```
$ pip install snakebite
```

Client Library

The client library is written in Python, uses protobuf messages, and implements the Hadoop RPC protocol for talking to the NameNode. This enables Python applications to communicate directly with HDFS and not have to make a system call to `hdfs dfs`.

List Directory Contents

Example 1-1 uses the Snakebite client library to list the contents of the root directory in HDFS.

Example 1-1. python/HDFS/list_directory.py

```
from snakebite.client import Client

client = Client('localhost', 9000)
for x in client.ls(['/']):
   print x
```

The most important line of this program, and every program that uses the client library, is the line that creates a client connection to the HDFS NameNode:

```
client = Client('localhost', 9000)
```

The `Client()` method accepts the following parameters:

`host` (*`string`*)
: Hostname or IP address of the NameNode

`port` (*`int`*)
: RPC port of the NameNode

`hadoop_version` (*`int`*)
: The Hadoop protocol version to be used (default: 9)

`use_trash` (*`boolean`*)
: Use trash when removing files

`effective_use` (*`string`*)
: Effective user for the HDFS operations (default: None or current user)

The `host` and `port` parameters are required and their values are dependent upon the HDFS configuration. The values for these parameters can be found in the *hadoop/conf/core-site.xml* configuration file under the property `fs.defaultFS`:

```
<property>
   <name>fs.defaultFS</name>
   <value>hdfs://localhost:9000</value>
</property>
```

For the examples in this section, the values used for `host` and `port` are `localhost` and `9000`, respectively.

After the client connection is created, the HDFS filesystem can be accessed. The remainder of the previous application used the `ls` command to list the contents of the root directory in HDFS:

```
for x in client.ls(['/']):
   print x
```

It is important to note that many of methods in Snakebite return generators. Therefore they must be consumed to execute. The `ls` method takes a list of paths and returns a list of maps that contain the file information.

Executing the *list_directory.py* application yields the following results:

```
$ python list_directory.py
{'group': u'supergroup', 'permission': 448, 'file_type': 'd',
'access_time':    0L,    'block_replication':    0,    'modifica-
tion_time':  1442752574936L,  'length':  0L,  'blocksize':  0L,
'owner': u'hduser', 'path': '/tmp'}
{'group': u'supergroup', 'permission': 493, 'file_type': 'd',
'access_time':    0L,    'block_replication':    0,    'modifica-
tion_time':  1442742056276L,  'length':  0L,  'blocksize':  0L,
'owner': u'hduser', 'path': '/user'}
```

Create a Directory

Use the `mkdir()` method to create directories on HDFS. Example 1-2 creates the directories */foo/bar* and */input* on HDFS.

Example 1-2. python/HDFS/mkdir.py

```
from snakebite.client import Client

client = Client('localhost', 9000)
for p in client.mkdir(['/foo/bar', '/input'], create_parent=True):
   print p
```

Executing the *mkdir.py* application produces the following results:

```
$ python mkdir.py
{'path': '/foo/bar', 'result': True}
{'path': '/input', 'result': True}
```

The `mkdir()` method takes a list of paths and creates the specified paths in HDFS. This example used the `create_parent` parameter to ensure that parent directories were created if they did not already exist. Setting `create_parent` to `True` is analogous to the `mkdir -p` Unix command.

Deleting Files and Directories

Deleting files and directories from HDFS can be accomplished with the `delete()` method. Example 1-3 recursively deletes the */foo* and */bar* directories, created in the previous example.

Example 1-3. python/HDFS/delete.py

```
from snakebite.client import Client

client = Client('localhost', 9000)
for p in client.delete(['/foo', '/input'], recurse=True):
   print p
```

Executing the *delete.py* application produces the following results:

```
$ python delete.py
{'path': '/foo', 'result': True}
{'path': '/input', 'result': True}
```

Performing a recursive delete will delete any subdirectories and files that a directory contains. If a specified path cannot be found, the delete method throws a `FileNotFoundException`. If recurse is not specified and a subdirectory or file exists, `DirectoryException` is thrown.

The `recurse` parameter is equivalent to `rm -rf` and should be used with care.

Retrieving Data from HDFS

Like the `hdfs dfs` command, the client library contains multiple methods that allow data to be retrieved from HDFS. To copy files from HDFS to the local filesystem, use the `copyToLocal()` method. Example 1-4 copies the file */input/input.txt* from HDFS and places it under the */tmp* directory on the local filesystem.

Example 1-4. python/HDFS/copy_to_local.py

```
from snakebite.client import Client

client = Client('localhost', 9000)
for f in client.copyToLocal(['/input/input.txt'], '/tmp'):
   print f
```

Executing the *copy_to_local.py* application produces the following result:

```
$ python copy_to_local.py
{'path': '/tmp/input.txt', 'source_path': '/input/input.txt',
'result': True, 'error': ''}
```

To simply read the contents of a file that resides on HDFS, the `text()` method can be used. Example 1-5 displays the content of */input/input.txt*.

Example 1-5. python/HDFS/text.py

```
from snakebite.client import Client

client = Client('localhost', 9000)
for l in client.text(['/input/input.txt']):
   print l
```

Executing the *text.py* application produces the following results:

```
$ python text.py
jack be nimble
jack be quick
jack jumped over the candlestick
```

The `text()` method will automatically uncompress and display gzip and bzip2 files.

CLI Client

The CLI client included with Snakebite is a Python command-line HDFS client based on the client library. To execute the Snakebite CLI, the hostname or IP address of the NameNode and RPC port of the NameNode must be specified. While there are many ways to specify these values, the easiest is to create a *~.snakebiterc* configuration file. Example 1-6 contains a sample config with the NameNode hostname of `localhost` and RPC port of `9000`.

Example 1-6. ~/.snakebiterc

```
{
    "config_version": 2,
    "skiptrash": true,
    "namenodes": [
        {"host": "localhost", "port": 9000, "version": 9},
    ]
}
```

The values for `host` and `port` can be found in the *hadoop/conf/core-site.xml* configuration file under the property `fs.defaultFS`.

For more information on configuring the CLI, see the Snakebite CLI documentation online.

Usage

To use the Snakebite CLI client from the command line, simply use the command `snakebite`. Use the `ls` option to display the contents of a directory:

```
$ snakebite ls /
Found 2 items
drwx------   - hadoop    supergroup    0 2015-09-20 14:36 /tmp
drwxr-xr-x   - hadoop    supergroup    0 2015-09-20 11:40 /user
```

Like the `hdfs dfs` command, the CLI client supports many familiar file manipulation commands (e.g., `ls`, `mkdir`, `df`, `du`, etc.).

The major difference between `snakebite` and `hdfs dfs` is that `snakebite` is a pure Python client and does not need to load any Java libraries to communicate with HDFS. This results in quicker interactions with HDFS from the command line.

CLI Command Reference

The following is a full listing of file manipulation commands possible with the `snakebite` CLI client. This listing can be displayed from the command line by specifying `snakebite` without any arguments. To view help with a specific command, use `snakebite [cmd] --help`, where `cmd` is a valid `snakebite` command.

```
snakebite [general options] cmd [arguments]
general options:
  -D --debug               Show debug information
  -V --version             Hadoop protocol version (default:9)
  -h --help                show help
  -j --json                JSON output
  -n --namenode            namenode host
  -p --port                namenode RPC port (default: 8020)
  -v --ver                 Display snakebite version

commands:
  cat [paths]                   copy source paths to stdout
  chgrp <grp> [paths]           change group
  chmod <mode> [paths]          change file mode (octal)
  chown <owner:grp> [paths]     change owner
  copyToLocal [paths] dst       copy paths to local
```

```
                                  file system destination
  count [paths]                 display stats for paths
  df                            display fs stats
  du [paths]                    display disk usage statistics
  get file dst                  copy files to local
                                  file system destination
  getmerge dir dst              concatenates files in source dir
                                  into destination local file
  ls [paths]                    list a path
  mkdir [paths]                 create directories
  mkdirp [paths]                create directories and their
                                  parents
  mv [paths] dst                move paths to destination
  rm [paths]                    remove paths
  rmdir [dirs]                  delete a directory
  serverdefaults                show server information
  setrep <rep> [paths]          set replication factor
  stat [paths]                  stat information
  tail path                     display last kilobyte of the
                                  file to stdout
  test path                     test a path
  text path [paths]             output file in text format
  touchz [paths]                creates a file of zero length
  usage <cmd>                   show cmd usage

to see command-specific options use: snakebite [cmd] --help
```

Chapter Summary

This chapter introduced and described the core concepts of HDFS. It explained how to interact with the filesystem using the built-in `hdfs dfs` command. It also introduced the Python library, Snakebite. Snakebite's client library was explained in detail with multiple examples. The `snakebite` CLI was also introduced as a Python alternative to the `hdfs dfs` command.

CHAPTER 2

MapReduce with Python

MapReduce is a programming model that enables large volumes of data to be processed and generated by dividing work into independent tasks and executing the tasks in parallel across a cluster of machines. The MapReduce programming style was inspired by the functional programming constructs map and reduce, which are commonly used to process lists of data. At a high level, every MapReduce program transforms a list of input data elements into a list of output data elements twice, once in the map phase and once in the reduce phase.

This chapter begins by introducing the MapReduce programming model and describing how data flows through the different phases of the model. Examples then show how MapReduce jobs can be written in Python.

Data Flow

The MapReduce framework is composed of three major phases: map, shuffle and sort, and reduce. This section describes each phase in detail.

Map

The first phase of a MapReduce application is the map phase. Within the map phase, a function (called the mapper) processes a series of key-value pairs. The mapper sequentially processes each

key-value pair individually, producing zero or more output key-value pairs (Figure 2-1).

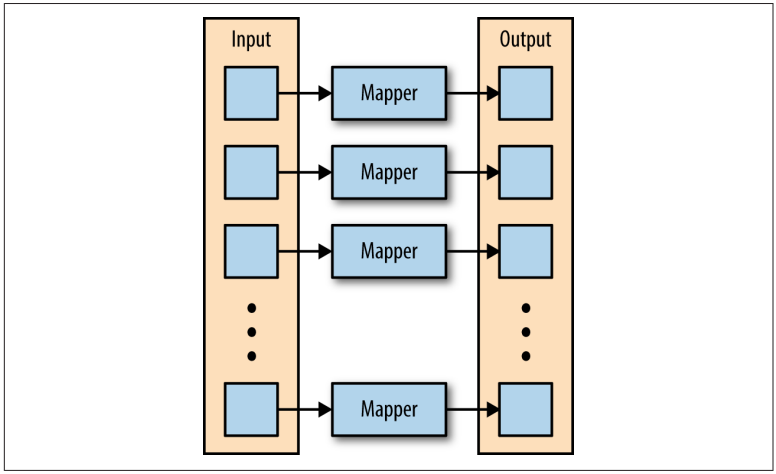

Figure 2-1. The mapper is applied to each input key-value pair, producing an output key-value pair

As an example, consider a mapper whose purpose is to transform sentences into words. The input to this mapper would be strings that contain sentences, and the mapper's function would be to split the sentences into words and output the words (Figure 2-2).

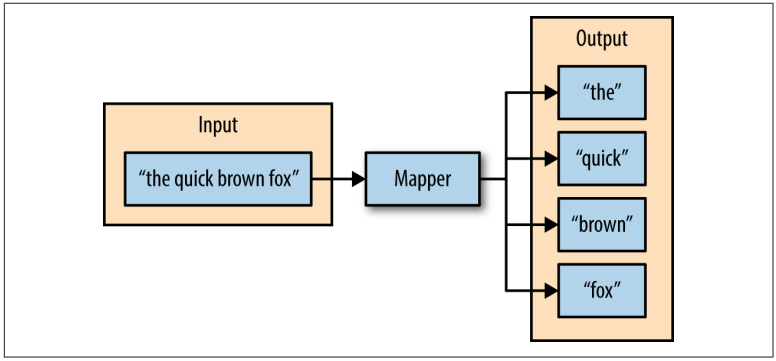

Figure 2-2. The input of the mapper is a string, and the function of the mapper is to split the input on spaces; the resulting output is the individual words from the mapper's input

Shuffle and Sort

The second phase of MapReduce is the shuffle and sort. As the mappers begin completing, the intermediate outputs from the map phase are moved to the reducers. This process of moving output from the mappers to the reducers is known as shuffling.

Shuffling is handled by a partition function, known as the partitioner. The partitioner is used to control the flow of key-value pairs from mappers to reducers. The partitioner is given the mapper's output key and the number of reducers, and returns the index of the intended reducer. The partitioner ensures that all of the values for the same key are sent to the same reducer. The default partitioner is hash-based. It computes a hash value of the mapper's output key and assigns a partition based on this result.

The final stage before the reducers start processing data is the sorting process. The intermediate keys and values for each partition are sorted by the Hadoop framework before being presented to the reducer.

Reduce

The third phase of MapReduce is the reduce phase. Within the reducer phase, an iterator of values is provided to a function known as the reducer. The iterator of values is a nonunique set of values for each unique key from the output of the map phase. The reducer aggregates the values for each unique key and produces zero or more output key-value pairs (Figure 2-3).

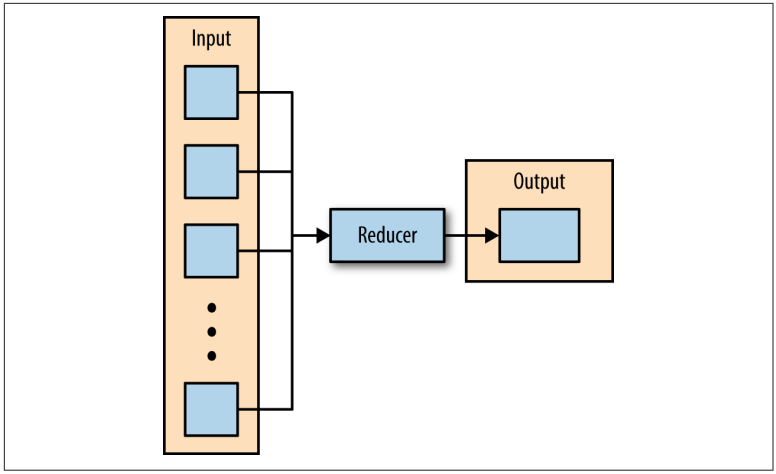

Figure 2-3. The reducer iterates over the input values, producing an output key-value pair

As an example, consider a reducer whose purpose is to sum all of the values for a key. The input to this reducer is an iterator of all of the values for a key, and the reducer sums all of the values. The reducer then outputs a key-value pair that contains the input key and the sum of the input key values (Figure 2-4).

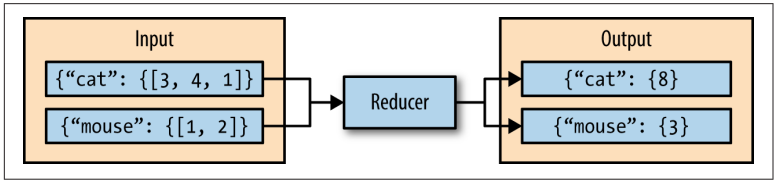

Figure 2-4. This reducer sums the values for the keys "cat" and "mouse"

The next section describes a simple MapReduce application and its implementation in Python.

Hadoop Streaming

Hadoop streaming is a utility that comes packaged with the Hadoop distribution and allows MapReduce jobs to be created with any executable as the mapper and/or the reducer. The Hadoop streaming utility enables Python, shell scripts, or any other language to be used as a mapper, reducer, or both.

How It Works

The mapper and reducer are both executables that read input, line by line, from the standard input (stdin), and write output to the standard output (stdout). The Hadoop streaming utility creates a MapReduce job, submits the job to the cluster, and monitors its progress until it is complete.

When the mapper is initialized, each map task launches the specified executable as a separate process. The mapper reads the input file and presents each line to the executable via stdin. After the executable processes each line of input, the mapper collects the output from stdout and converts each line to a key-value pair. The key consists of the part of the line before the first tab character, and the value consists of the part of the line after the first tab character. If a line contains no tab character, the entire line is considered the key and the value is null.

When the reducer is initialized, each reduce task launches the specified executable as a separate process. The reducer converts the input key-value pair to lines that are presented to the executable via stdin. The reducer collects the executables result from stdout and converts each line to a key-value pair. Similar to the mapper, the executable specifies key-value pairs by separating the key and value by a tab character.

A Python Example

To demonstrate how the Hadoop streaming utility can run Python as a MapReduce application on a Hadoop cluster, the WordCount application can be implemented as two Python programs: *mapper.py* and *reducer.py*.

mapper.py is the Python program that implements the logic in the map phase of WordCount. It reads data from stdin, splits the lines into words, and outputs each word with its intermediate count to stdout. The code in Example 2-1 implements the logic in *mapper.py*.

Example 2-1. python/MapReduce/HadoopStreaming/mapper.py

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

import sys

# Read each line from stdin
```

```
for line in sys.stdin:

  # Get the words in each line
  words = line.split()

  # Generate the count for each word
  for word in words:

     # Write the key-value pair to stdout to be processed by
     # the reducer.
     # The key is anything before the first tab character and the
     #value is anything after the first tab character.
     print '{0}\t{1}'.format(word, 1)
```

reducer.py is the Python program that implements the logic in the reduce phase of WordCount. It reads the results of *mapper.py* from stdin, sums the occurrences of each word, and writes the result to stdout. The code in Example 2-2 implements the logic in *reducer.py*.

Example 2-2. python/MapReduce/HadoopStreaming/reducer.py

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

import sys

curr_word = None
curr_count = 0

# Process each key-value pair from the mapper
for line in sys.stdin:

  # Get the key and value from the current line
  word, count = line.split('\t')

  # Convert the count to an int
  count = int(count)

  # If the current word is the same as the previous word,
  # increment its count, otherwise print the words count
  # to stdout
  if word == curr_word:
     curr_count += count
  else:

     # Write word and its number of occurrences as a key-value
     # pair to stdout
     if curr_word:
        print '{0}\t{1}'.format(curr_word, curr_count)

     curr_word = word
```

```
    curr_count = count

# Output the count for the last word
if curr_word == word:
  print '{0}\t{1}'.format(curr_word, curr_count)
```

Before attempting to execute the code, ensure that the *mapper.py* and *reducer.py* files have execution permission. The following command will enable this for both files:

```
$ chmod a+x mapper.py reducer.py
```

Also ensure that the first line of each file contains the proper path to Python. This line enables *mapper.py* and *reducer.py* to execute as standalone executables. The value `#!/usr/bin/env python` should work for most systems, but if it does not, replace `/usr/bin/env python` with the path to the Python executable on your system.

To test the Python programs locally before running them as a MapReduce job, they can be run from within the shell using the `echo` and `sort` commands. It is highly recommended to test all programs locally before running them across a Hadoop cluster.

```
$ echo 'jack be nimble jack be quick' | ./mapper.py
| sort -t 1 | ./reducer.py
be      2
jack    2
nimble  1
quick   1
```

Once the mapper and reducer programs are executing successfully against tests, they can be run as a MapReduce application using the Hadoop streaming utility. The command to run the Python programs *mapper.py* and *reducer.py* on a Hadoop cluster is as follows:

```
$ $HADOOP_HOME/bin/hadoop jar
  $HADOOP_HOME/mapred/contrib/streaming/hadoop-streaming*.jar \
-files mapper.py,reducer.py \
-mapper mapper.py \
-reducer reducer.py \
-input /user/hduser/input.txt -output /user/hduser/output
```

The options used with the Hadoop streaming utility are listed in Table 2-1.

Table 2-1. Options for Hadoop streaming

Option	Description
`-files`	A command-separated list of files to be copied to the MapReduce cluster
`-mapper`	The command to be run as the mapper
`-reducer`	The command to be run as the reducer
`-input`	The DFS input path for the Map step
`-output`	The DFS output directory for the Reduce step

mrjob

mrjob is a Python MapReduce library, created by Yelp, that wraps Hadoop streaming, allowing MapReduce applications to be written in a more Pythonic manner. mrjob enables multistep MapReduce jobs to be written in pure Python. MapReduce jobs written with mrjob can be tested locally, run on a Hadoop cluster, or run in the cloud using Amazon Elastic MapReduce (EMR).

Writing MapReduce applications with mrjob has many benefits:

- mrjob is currently a very actively developed framework with multiple commits every week.
- mrjob has extensive documentation, more than any other framework or library that supports Python on Hadoop.
- mrjob applications can be executed and tested without having Hadoop installed, enabling development and testing before deploying to a Hadoop cluster.
- mrjob allows MapReduce applications to be written in a single class, instead of writing separate programs for the mapper and reducer.

While mrjob is a great solution, it does have its drawbacks. mrjob is simplified, so it doesn't give the same level of access to Hadoop that other APIs offer. mrjob does not use typedbytes, so other libraries may be faster.

Installation

The installation of mrjob is simple; it can be installed with `pip` by using the following command:

```
$ pip install mrjob
```

Or it can be installed from source (a git clone):

```
$ python setup.py install
```

WordCount in mrjob

Example 2-3 uses mrjob to implement the WordCount algorithm.

Example 2-3. python/MapReduce/mrjob/word_count.py

```
from mrjob.job import MRJob

class MRWordCount(MRJob):

   def mapper(self, _, line):
      for word in line.split():
         yield(word, 1)

   def reducer(self, word, counts):
      yield(word, sum(counts))

if __name__ == '__main__':
   MRWordCount.run()
```

To run the mrjob locally, the only thing needed is a body of text. To run the job locally and count the frequency of words within a file named *input.txt*, use the following command:

```
$ python word_count.py input.txt
```

The output depends on the contents of the input file, but should look similar to Example 2-4.

Example 2-4. Output from word_count.py

```
"be"    2
"jack"    2
"nimble"    1
"quick"    1
```

What Is Happening

The MapReduce job is defined as the class, MRWordCount. Within the mrjob library, the class that inherits from MRJob contains the methods that define the steps of the MapReduce job. The steps within an mrjob application are mapper, combiner, and reducer. The class inheriting MRJob only needs to define one of these steps.

The `mapper()` method defines the mapper for the MapReduce job. It takes key and value as arguments and yields tuples of (output_key, output_value). In the WordCount example (Example 2-4), the mapper ignored the input key and split the input value to produce words and counts.

The `combiner()` method defines the combiner for the MapReduce job. The combiner is a process that runs after the mapper and before the reducer. It receives, as input, all of the data emitted by the mapper, and the output of the combiner is sent to the reducer. The combiner's input is a key, which was yielded by the mapper, and a value, which is a generator that yields all values yielded by one mapper that corresponds to the key. The combiner yields tuples of (output_key, output_value) as output.

The `reducer()` method defines the reducer for the MapReduce job. It takes a key and an iterator of values as arguments and yields tuples of (output_key, output_value). In Example 2-4, the reducer sums the value for each key, which represents the frequency of words in the input.

The final component of a MapReduce job written with the mrjob library is the two lines at the end of the file:

```
if __name__ == '__main__':
    MRWordCount.run()
```

These lines enable the execution of mrjob; without them, the application will not work.

Executing mrjob

Executing a MapReduce application with mrjob is similar to executing any other Python program. The command line must contain the name of the mrjob application and the input file:

```
$ python mr_job.py input.txt
```

By default, mrjob writes output to stdout.

Multiple files can be passed to mrjob as inputs by specifying the filenames on the command line:

```
$ python mr_job.py input1.txt input2.txt input3.txt
```

mrjob can also handle input via stdin:

```
$ python mr_job.py < input.txt
```

By default, mrjob runs locally, allowing code to be developed and debugged before being submitted to a Hadoop cluster.

To change how the job is run, specify the `-r/--runner` option. Table 2-2 contains a description of the valid choices for the runner options.

Table 2-2. mrjob runner choices

`-r inline`	(Default) Run in a single Python process
`-r local`	Run locally in a few subprocesses simulating some Hadoop features
`-r hadoop`	Run on a Hadoop cluster
`-r emr`	Run on Amazon Elastic Map Reduce (EMR)

Using the runner option allows the mrjob program to be run on a Hadoop cluster, with input being specified from HDFS:

```
$ python mr_job.py -r hadoop hdfs://input/input.txt
```

mrjob also allows applications to be run on EMR directly from the command line:

```
$ python mr_job.py -r emr s3://input-bucket/input.txt
```

Top Salaries

Example 2-5 uses mrjob to compute employee top annual salaries and gross pay. The dataset used is the salary information from the city of Baltimore for 2014.

Example 2-5. python/MapReduce/mrjob/top_salary.py

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import csv

cols = 'Name,JobTitle,AgencyID,Agency,HireDate,AnnualSalary,Gross
Pay'.split(',')

class salarymax(MRJob):

    def mapper(self, _, line):
        # Convert each line into a dictionary
                row = dict(zip(cols, [ a.strip() for a in
csv.reader([line]).next()]))

        # Yield the salary
```

```
            yield 'salary', (float(row['AnnualSalary'][1:]), line)

            # Yield the gross pay
            try:
                yield 'gross', (float(row['GrossPay'][1:]), line)
            except ValueError:
                self.increment_counter('warn', 'missing gross', 1)

    def reducer(self, key, values):
        topten = []

        # For 'salary' and 'gross' compute the top 10
        for p in values:
            topten.append(p)
            topten.sort()
            topten = topten[-10:]

        for p in topten:
            yield key, p

    combiner = reducer

if __name__ == '__main__':
    salarymax.run()
```

Use the following command to execute the MapReduce job on Hadoop:

```
$ python top_salary.py -r hadoop hdfs:///user/hduser/input/
salaries.csv
```

Chapter Summary

This chapter introduced the MapReduce programming model and described how data flows through the different phases of the model. Hadoop Streaming and mrjob were then used to highlight how MapReduce jobs can be written in Python.