MODULE 02

Essential Hadoop Tools

Using Apache Pig

Apache Pig is a high-level language that enables programmers to write complex MapReduce transformations using a simple scripting language. Pig Latin (the actual language) defines a set of transformations on a data set such as aggregate, join, and sort. Pig is often used to extract, transform, and load (ETL) data pipelines, quick research on raw data, and iterative data processing.

Apache Pig has several usage modes. The first is a local mode in which all processing is done on the local machine. The non-local (cluster) modes are MapReduce and Tez. These modes execute the job on the cluster using either the MapReduce engine or the optimized Tez engine. (Tez, which is Hindi for "speed," optimizes multistep Hadoop jobs such as those found in many Pig queries.) There are also interactive and batch modes available; they enable Pig applications to be developed locally in interactive modes, using small amounts of data, and then run at scale on the cluster in a production mode. The modes are summarized in Table 7.1.

	Local Mode	Tez Local Mode	MapReduce Mode	Tez Mode
Interactive Mode	Yes	Experimental	Yes	Yes
Batch Mode	Yes	Experimental	Yes	Yes

Table 7.1 **Apache Pig Usage Modes**

Pig Example Walk-Through

For this example, the following software environment is assumed. Other environments should work in a similar fashion.

- OS: Linux
- Platform: RHEL 6.6
- Hortonworks HDP 2.2 with Hadoop version: 2.6
- Pig version: 0.14.0

To begin the example, copy the `passwd` file to a working directory for local Pig operation:

$ cp /etc/passwd .

Next, copy the data file into HDFS for Hadoop MapReduce operation:

$ hdfs dfs -put passwd passwd

You can confirm the file is in HDFS by entering the following command:

```
hdfs dfs -ls passwd
-rw-r—r-- 2 hdfs hdfs 2526 2015-03-17 11:08 passwd
```

In the following example of local Pig operation, all processing is done on the local machine (Hadoop is not used). First, the interactive command line is started:

$ pig -x local

If Pig starts correctly, you will see a `grunt>` prompt. You may also see a bunch of INFO messages, which you can ignore. Next, enter the following commands to load the `passwd` file and then grab the user name and dump it to the terminal. Note that Pig commands must end with a semicolon (`;`).

```
grunt>A=load passwd' using PigStorage(':');
grunt>B=for each A generate $0 as id;
grunt> dump B;
```

The processing will start and a list of user names will be printed to the screen. To exit the interactive session, enter the command `quit`.

$ grunt> quit

To use Hadoop MapReduce, start Pig as follows (or just enter `pig`):

$ pig -x mapreduce

The same sequence of commands can be entered at the `grunt>` prompt. You may wish to change the `$0` argument to pull out other items in the `passwd` file. In the case of this simple script, you will notice that the MapReduce version takes much longer. Also, because we are running this application under Hadoop, make sure the file is placed in HDFS.

If you are using the Hortonworks HDP distribution with `tez` installed, the `tez` engine can be used as follows:

$ pig -x tez

Pig can also be run from a script. An example script (`id.pig`) is available from the example code download. This script, which is repeated here, is designed to do the same things as the interactive version:

```
/* id.pig */
A = load 'passwd' using PigStorage(':'); -- load the passwd file
B = foreach A generate $0 as id; -- extract the user IDs
dump B;
store B into 'id.out'; -- write the results to a directory name id.out
```

Comments are delineated by `/* */` and `--` at the end of a line. The script will create a directory called `id.out` for the results. First, ensure that the `id.out` directory is not in your local directory, and then start Pig with the script on the command line:

```
$/bin/rm -r id.out/
$ pig -x local id.pig
```

If the script worked correctly, you should see at least one data file with the results and a zero-length file with the name `_SUCCESS`. To run the MapReduce version, use the same procedure; the only difference is that now all reading and writing takes place in HDFS.

```
$ hdfs dfs -rm -r id.out
$ pig id.pig
```

If Apache `tez` is installed, you can run the example script using the `-x tez` option.

Using Apache Hive

Apache Hive is a data warehouse infrastructure built on top of Hadoop for providing data summarization, ad hoc queries, and the analysis of large data sets using a SQL-like language called HiveQL. Hive is considered the de facto standard for interactive SQL queries over petabytes of data using Hadoop and offers the following features:

- Tools to enable easy data extraction, transformation, and loading (ETL)
- A mechanism to impose structure on a variety of data formats
- Access to files stored either directly in HDFS or in other data storage systems such as HBase
- Query execution via MapReduce and Tez (optimized MapReduce)

Hive Example Walk-Through

For this example, the following software environment is assumed. Other environments should work in a similar fashion.

- OS: Linux
- Platform: RHEL 6.6
- Hortonworks HDP 2.2 with Hadoop version: 2.6
- Hive version: 0.14.0

To start Hive, simply enter the `hive` command. If Hive starts correctly, you should get a `hive>` prompt.

```
$ hive
(some messages may show up here)
hive>
```

As a simple test, create and drop a table. Note that Hive commands must end with a semicolon (`;`).

```
hive>CREATE TABLE pokes (foo INT, bar STRING);
OK
Time taken: 1.705 seconds
hive> SHOW TABLES;
```

```
OK
pokes
Time taken: 0.174 seconds, Fetched: 1 row(s)
hive> DROP TABLE pokes;
OK
Time taken: 4.038 seconds
```

A more detailed example can be developed using a web server log file to summarize message types. First, create a table using the following command:

```
hive> CREATE TABLE logs(t1 string, t2 string, t3 string, t4 string, t5 string, t6 string, t7 string) ROW FORMAT DELIMITED FIELDS TERMINATED BY ' ';
OK
Time taken: 0.129 seconds
```

Next, load the data—in this case, from the `sample.log` file.

```
hive> LOAD DATA LOCAL INPATH 'sample.log' OVERWRITE INTO TABLE logs;
Loading data to table default.logs
Table default.logs stats: [numFiles=1, numRows=0, totalSize=99271, rawDataSize=0]
OK
Time taken: 0.953 seconds
```

Finally, apply the select step to the file. Note that this invokes a Hadoop MapReduce operation. The results appear at the end of the output (e.g., totals for the message types `DEBUG`, `ERROR`, and so on).

```
hive> SELECT t4 AS sev, COUNT(*) AS cnt FROM logs WHERE t4 LIKE '[%' GROUP BY t4;
```

To exit Hive, simply type `exit;`:

```
hive> exit;
```

Using Apache Sqoop to Acquire Relational Data

Sqoop is a tool designed to transfer data between Hadoop and relational databases. You can use Sqoop to import data from a relational database management system (RDBMS) into the Hadoop Distributed File System (HDFS), transform the data in Hadoop, and then export the data back into an RDBMS.

Apache Sqoop Import and Export Methods

Figure 7.1 describes the Sqoop data import (to HDFS) process. The data import is done in two steps. In the first step, shown in the figure, Sqoop examines the database to gather the necessary metadata for the data to be imported. The second step is a map-only (no reduce step) Hadoop job that Sqoop submits to the cluster. This job does the actual data transfer using the metadata capture

in the previous step. Note that each node doing the import must have access to the database.

Figure 7.1 Two-step Apache Sqoop data import method (Adapted from Apache Sqoop Documentation)

The imported data are saved in an HDFS directory. Sqoop will use the database name for the directory, or the user can specify any alternative directory where the files should be populated. By default, these files contain comma-delimited fields, with new lines separating different records. You can easily override the format in which data are copied over by explicitly specifying the field separator and record terminator characters. Once placed in HDFS, the data are ready for processing.

Data export from the cluster works in a similar fashion. The export is done in two steps, as shown in Figure 7.2. As in the import process, the first step is to examine the database for metadata. The export step again uses a map-only Hadoop job to write the data to the database. Sqoop divides the input data set into splits, then uses individual map tasks to push the splits to the database. Again, this process assumes the map tasks have access to the database.

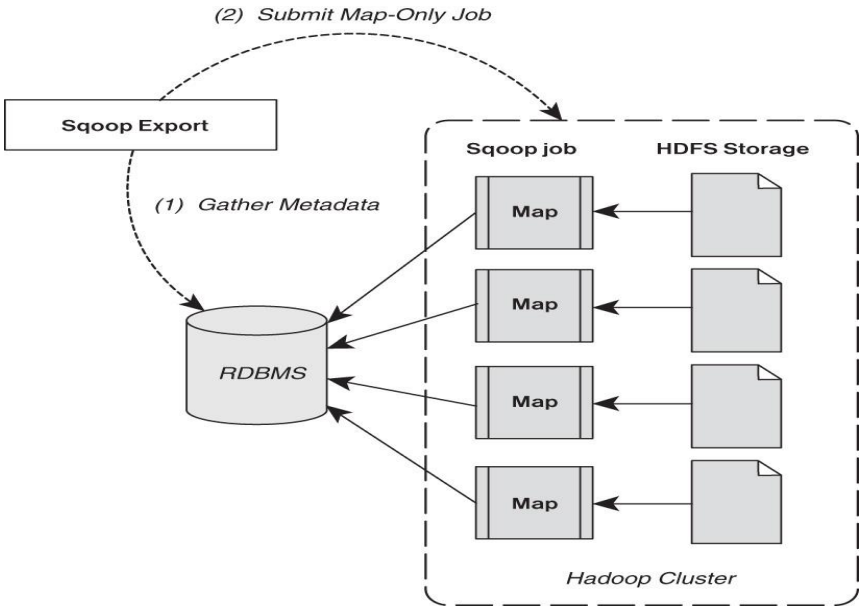

Figure 7.2 Two-step Sqoop data export method (Adapted from Apache Sqoop Documentation)

Apache Sqoop Version Changes

Sqoop Version 1 uses specialized connectors to access external systems. These connectors are often optimized for various RDBMSs or for systems that do not support JDBC. Connectors are plug-in components based on Sqoop's extension framework and can be added to any existing Sqoop installation. Once a connector is installed, Sqoop can use it to efficiently transfer data between Hadoop and the external store supported by the connector. By default, Sqoop version 1 includes connectors for popular databases such as MySQL, PostgreSQL, Oracle, SQL Server, and DB2. It also supports direct transfer to and from the RDBMS to HBase or Hive.

In contrast, to streamline the Sqoop input methods, Sqoop version 2 no longer supports specialized connectors or direct import into HBase or Hive. All imports and exports are done through the JDBC interface. Table 7.2 summarizes the changes from version 1 to version 2. Due to these changes, any new development should be done with Sqoop version 2.

Feature	Sqoop Version 1	Sqoop Version 2
Connectors for all major RDBMSs	Supported.	Not supported. Use the generic JDBC connector.
Kerberos security integration	Supported.	Not supported.
Data transfer from RDBMS to Hive or HBase	Supported.	Not supported. First import data from RDBMS into HDFS, then load data into Hive or HBase manually.
Data transfer from Hive or HBase to RDBMS	Not supported. First export data from Hive or HBase into HDFS, and then use Sqoop for export.	Not supported. First export data from Hive or HBase into HDFS, then use Sqoop for export.

Table 7.2 **Apache Sqoop Version Comparison**

Sqoop Example Walk-Through

The following simple example illustrates use of Sqoop. It can be used as a foundation from which to explore the other capabilities offered by Apache Sqoop. The following steps will be performed:

1. Download Sqoop.

2. Download and load sample MySQL data.

3. Add Sqoop user permissions for the local machine and cluster.

4. Import data from MySQL to HDFS.

5. Export data from HDFS to MySQL.

For this example, the following software environment is assumed. Other environments should work in a similar fashion.

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

- Sqoop version: 1.4.5
- A working installation of MySQL on the host

Step 1: Download Sqoop and Load Sample MySQL Database

If you have not done so already, make sure Sqoop is installed on your cluster. Sqoop is needed on only a single node in your cluster. This Sqoop node will then serve as an entry point for all connecting Sqoop clients. Because the Sqoop node is a Hadoop MapReduce client, it requires both a Hadoop installation and access to HDFS.

```
$wget http://downloads.mysql.com/docs/world_innodb.sql.gz
$ gunzip world_innodb.sql.gz
```

Next, log into MySQL (assumes you have privileges to create a database) and import the desired database by following these steps:

```
$ mysql -u root -p
mysql> CREATE DATABASE world;
mysql> USE world;
mysql> SOURCE world_innodb.sql;
mysql> SHOW TABLES;
```

The following MySQL command will let you see the table details (output omitted for clarity):

```
mysql> SHOW CREATE TABLE Country;
mysql> SHOW CREATE TABLE City;
mysql> SHOW CREATE TABLE CountryLanguage;
```

Step 2: Add Sqoop User Permissions for the Local Machine and Cluster

In MySQL, add the following privileges for user `sqoop` to MySQL. Note that you must use both the local host name and the cluster subnet for Sqoop to work properly. Also, for the purposes of this example, the sqoop password is `sqoop`.

```
mysql> GRANT ALL PRIVILEGES ON world.* To 'sqoop'@'limulus' IDENTIFIED BY 'sqoop';
mysql> GRANT ALL PRIVILEGES ON world.* To 'sqoop'@'10.0.0.%' IDENTIFIED BY 'sqoop';
mysql> quit
```

Next, log in as `sqoop` to test the permissions:

```
$ mysql -u sqoop -p
mysql> USE world;
mysql> SHOW TABLES;
```

mysql> quit

Step 3: Import Data Using Sqoop

As a test, we can use Sqoop to list databases in MySQL. The results appear after the warnings at the end of the output. Note the use of local host name (`limulus`) in the JDBC statement.

$ sqoop list-databases --connect jdbc:mysql://limulus/world --username sqoop --password sqoop

Step 4: Export Data from HDFS to MySQL

Sqoop can also be used to export data from HDFS. The first step is to create tables for exported data. There are actually two tables needed for each exported table. The first table holds the exported data (`CityExport`), and the second is used for staging the exported data (`CityExportStaging`). NULL DEFAULT '0', PRIMARY KEY ('ID'));

Using Apache Flume to Acquire Data Streams

Apache Flume is an independent agent designed to collect, transport, and store data into HDFS. Often data transport involves a number of Flume agents that may traverse a series of machines and locations. Flume is often used for log files, social media-generated data, email messages, and just about any continuous data source.

As shown in Figure 7.3, a Flume agent is composed of three components.

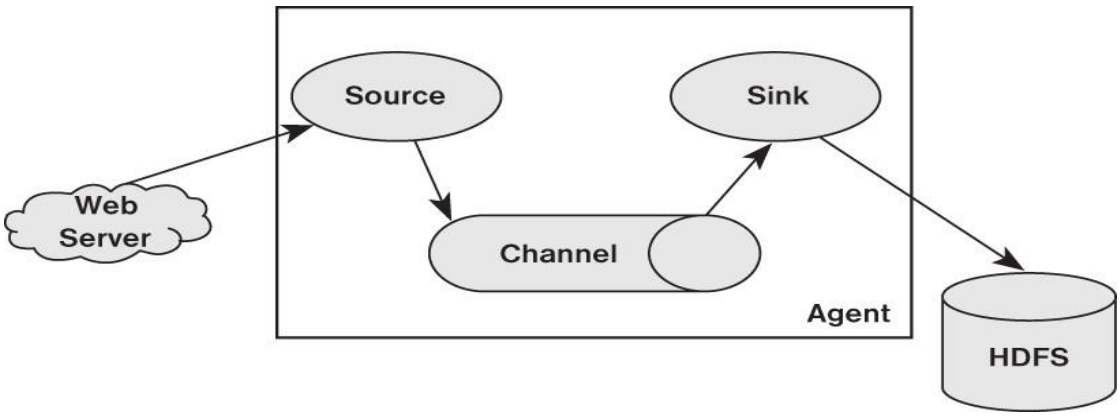

Figure 7.3 Flume agent with source, channel, and sink (Adapted from Apache Flume Documentation)

■ *Source.* The source component receives data and sends it to a channel. It can send the data to more than one channel. The input data can be from a real-time source (e.g., weblog) or another Flume agent.

■ *Channel.* A channel is a data queue that forwards the source data to the sink destination. It can be thought of as a buffer that manages input (source) and output (sink) flow rates.

■ *Sink.* The sink delivers data to destination such as HDFS, a local file, or another Flume agent.

A Flume agent must have all three of these components defined. A Flume agent can have several sources, channels, and sinks. Sources can write to multiple channels, but a sink can take data from only a single channel. Data written to a channel remain in the channel until a sink removes the data. By default, the data in a channel are kept in memory but may be optionally stored on disk to prevent data loss in the event of a network failure.

As shown in Figure 7.4, Sqoop agents may be placed in a pipeline, possibly to traverse several machines or domains. This configuration is normally used when data are collected on one machine (e.g., a web server) and sent to another machine that has access to HDFS.

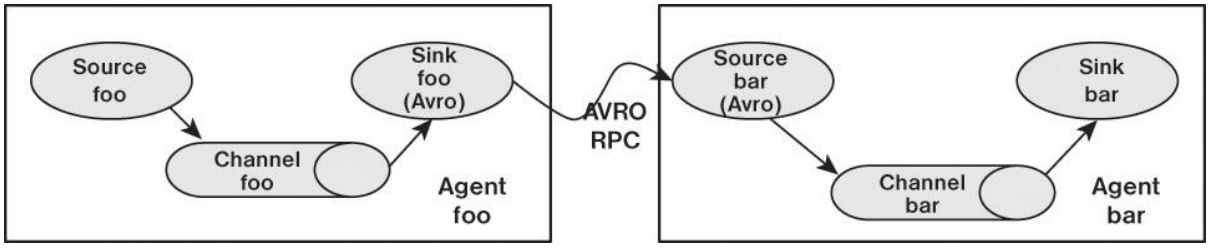

Figure 7.4 Pipeline created by connecting Flume agents (Adapted from Apache Flume Sqoop Documentation)

In a Flume pipeline, the sink from one agent is connected to the source of another. The data transfer format normally used by Flume, which is called Apache Avro, provides several useful features. First, Avro is a data serialization/deserialization system that uses a compact binary format. The schema is sent as part of the data exchange and is defined using JSON (JavaScript Object Notation). Avro also uses remote procedure calls (RPCs) to send data. That is, an Avro sink will contact an Avro source to send data.

Another useful Flume configuration is shown in Figure 7.5. In this configuration, Flume is used to consolidate several data sources before committing them to HDFS.

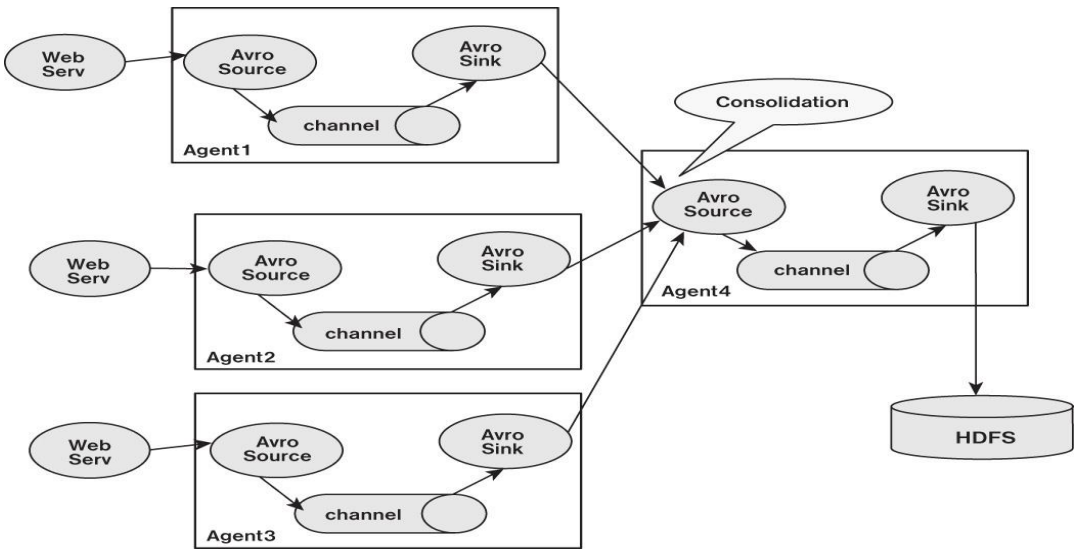

Figure 7.5 A Flume consolidation network (Adapted from Apache Flume Documentation)

There are many possible ways to construct Flume transport networks. In addition, other Flume features not described in depth here include plug-ins and interceptors that can enhance Flume pipelines.

Flume Example Walk-Through

Follow these steps to walk through a Flume example.

Step 1: Download and Install Apache Flume

For this example, the following software environment is assumed. Other environments should work in a similar fashion.

- OS: Linux
- Platform: RHEL 6.6
- Hortonworks HDP 2.2 with Hadoop version: 2.6
- Flume version: 1.5.2

If Flume is not installed and you are using the Hortonworks HDP repository, you can add Flume with the following command:

```
# yum install flume flume-agent
```

In addition, for the simple example, `telnet` will be needed:

```
# yum install telnet
```

The following examples will also require some configuration files.

Step 2: Simple Test

A simple test of Flume can be done on a single machine. To start the Flume agent, enter the `flume-ng` command shown here. This command uses the `simple-example.conf` file to configure the agent.

```
$ flume-ng agent --conf conf --conf-file simple-example.conf --name simple_agent -Dflume.root.logger=INFO,console
```

In another terminal window, use `telnet` to contact the agent:

Step 3: Weblog Example

In this example, a record from the weblogs from the local machine (Ambari output) will be placed into HDFS using Flume. This example is easily modified to use other weblogs from different machines. Two files are needed to configure Flume.

Manage Hadoop Workflows with Apache Oozie

Oozie is a workflow director system designed to run and manage multiple related Apache Hadoop jobs. For instance, complete data input and analysis may require several discrete Hadoop jobs to be run as a workflow in which the output of one job serves as the input for a successive job. Oozie is designed to construct and manage these workflows. Oozie is not a substitute for the YARN scheduler. That is, YARN manages resources for individual Hadoop jobs, and Oozie provides a way to connect and control Hadoop jobs on the cluster.

Oozie workflow jobs are represented as directed acyclic graphs (DAGs) of actions. (DAGs are basically graphs that cannot have directed loops.) Three types of Oozie jobs are permitted:

■ Workflow—a specified sequence of Hadoop jobs with outcome-based decision points and control dependency. Progress from one action to another cannot happen until the first action is complete.

■ Coordinator—a scheduled workflow job that can run at various time intervals or when data become available.

■ Bundle—a higher-level Oozie abstraction that will batch a set of coordinator jobs.

Oozie is integrated with the rest of the Hadoop stack, supporting several types of Hadoop jobs out of the box (e.g., Java MapReduce, Streaming MapReduce, Pig, Hive, and Sqoop) as well as system-specific jobs (e.g., Java programs and shell scripts). Oozie also provides a CLI and a web UI for monitoring jobs.

Figure 7.6 depicts a simple Oozie workflow. In this case, Oozie runs a basic MapReduce operation. If the application was successful, the job ends; if an error occurred, the job is killed.

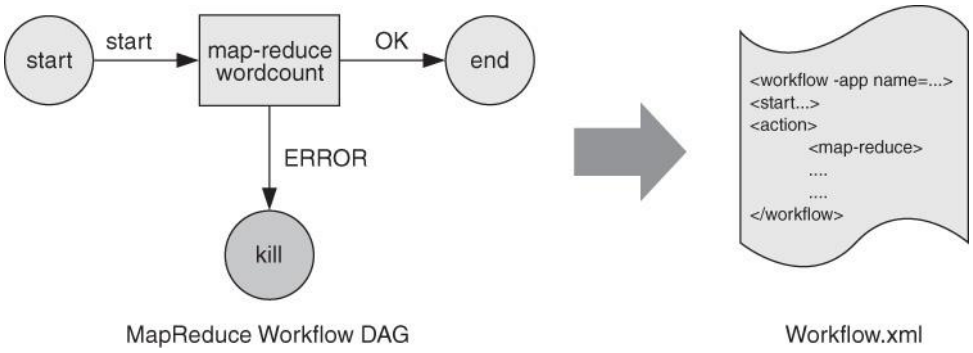

Figure 7.6 A simple Oozie DAG workflow (Adapted from Apache Oozie Documentation)

Oozie workflow definitions are written in hPDL (an XML Process Definition Language). Such workflows contain several types of nodes:

■ **Control flow nodes** define the beginning and the end of a workflow. They include start, end, and optional fail nodes.

■ **Action nodes** are where the actual processing tasks are defined. When an action node finishes, the remote systems notify Oozie and the next node in the workflow is executed. Action nodes can also include HDFS commands.

■ **Fork/join nodes** enable parallel execution of tasks in the workflow. The fork node enables two or more tasks to run at the same time. A join node represents a rendezvous point that must wait until all forked tasks complete.

■ **Control flow nodes** enable decisions to be made about the previous task. Control decisions are based on the results of the previous action (e.g., file size or file existence). Decision nodes are essentially switch-case statements that use JSP EL (Java Server Pages—Expression Language) that evaluate to either true or false.

Figure 7.7 depicts a more complex workflow that uses all of these node types. More information on Oozie can be found at http://oozie.apache.org/docs/4.1.0/index.html.

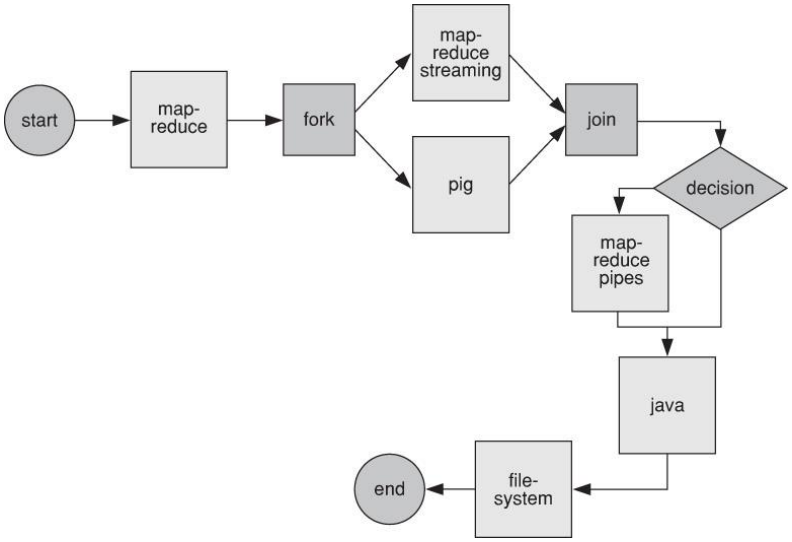

Figure 7.7 A more complex Oozie DAG workflow (Adapted from Apache Oozie Documentation)

Oozie Example Walk-Through

For this example, the following software environment is assumed. Other environments should work in a similar fashion.

■ OS: Linux

■ Platform: CentOS 6.6

■ Hortonworks HDP 2.2 with Hadoop version: 2.6

■ Oozie version: 4.1.0

Oozie is also installed as part of the Hortonworks HDP Sandbox.

Step 1: Download Oozie Examples

For HDP 2.1, the following command can be used to extract the files into the working directory used for the demo:

$ tar xvzf /usr/share/doc/oozie-4.0.0.2.1.2.1/oozie-examples.tar.gz

For HDP 2.2, the following command will extract the files:

$ tar xvzf /usr/hdp/2.2.4.2-2/oozie/doc/oozie-examples.tar.gz

Once extracted, rename the examples directory to `oozie-examples` so that you will not confuse it with the other examples directories.

$ mv examples oozie-examples

The examples must also be placed in HDFS. Enter the following command to move the example files into HDFS:

$ hdfs dfs -put oozie-examples/ oozie-examples

The Oozie shared library must be installed in HDFS. If you are using the Ambari installation of HDP 2.x, this library is already found in HDFS: `/user/oozie/share/lib`.

Step 2: Run the Simple MapReduce Example

Move to the simple MapReduce example directory:

$ cd oozie-examples/apps/map-reduce/

This directory contains two files and a `lib` directory. The files are:

■ The `job.properties` file defines parameters (e.g., path names, ports) for a job. This file may change per job.

■ The `workflow.xml` file provides the actual workflow for the job. In this case, it is a simple MapReduce (pass/fail). This file usually stays the same between jobs.

Step 3: Run the Oozie Demo Application

A more sophisticated example can be found in the demo directory (`oozie-examples/apps/demo`). This workflow includes MapReduce, Pig, and file system tasks as well as fork, join, decision, action, start, stop, kill, and end nodes.

Move to the demo directory and edit the `job.properties` file as described previously. Entering the following command runs the workflow (assuming the `OOZIE_URL` environment variable has been set):

$ oozie job -run -config job.properties

You can track the job using either the Oozie command-line interface or the Oozie web console. To start the web console from within Ambari, click on the Oozie service, and then click on the Quick Links pull-down menu and select Oozie Web UI. Alternatively, you can start the Oozie web UI by connecting to the Oozie server directly. For example, the following command will bring up the Oozie UI (use your Oozie server host name in place of "limulus"):

$ firefox http://limulus:11000/oozie/

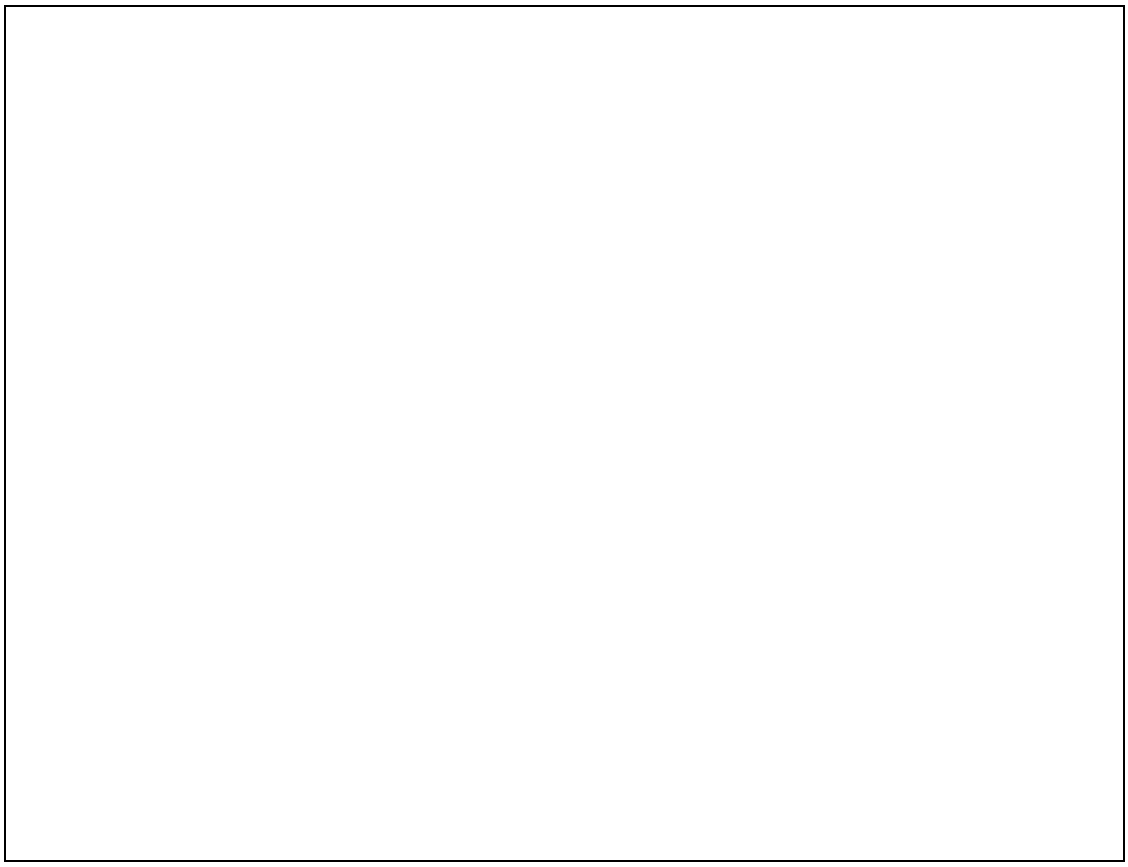

Figure 7.10 Oozie-generated workflow DAG for the demo example, as it appears on the screen

A Short Summary of Oozie Job Commands

The following summary lists some of the more commonly encountered Oozie commands. See the latest documentation at http://oozie.apache.org for more information. (Note that the examples here assume `OOZIE_URL` is defined.)

■ Run a workflow job (returns `_OOZIE_JOB_ID_`):

$ oozie job -run -config JOB_PROPERITES

■ Submit a workflow job (returns `_OOZIE_JOB_ID_` but does not start):

$ oozie job -submit -config JOB_PROPERTIES

■ Start a submitted job:

$ oozie job -start _OOZIE_JOB_ID_

■ Check a job's status:

$ oozie job -info _OOZIE_JOB_ID_

■ Suspend a workflow:

$ oozie job -suspend _OOZIE_JOB_ID_

■ Resume a workflow:

$ oozie job -resume _OOZIE_JOB_ID_

■ Rerun a workflow:

$ oozie job -rerun _OOZIE_JOB_ID_ -config JOB_PROPERTIES

■ Kill a job:

$ oozie job -kill _OOZIE_JOB_ID_

■ View server logs:

$ oozie job -logs _OOZIE_JOB_ID_

Full logs are available at `/var/log/oozie` on the Oozie server.

Using Apache HBase

Apache HBase is an open source, distributed, versioned, nonrelational database modeled after Google's Bigtable Like Bigtable, HBase leverages the distributed data storage provided by the underlying distributed file systems spread across commodity servers. Apache HBase provides Bigtable-like capabilities on top of Hadoop and HDFS. Some of the more important features include the following capabilities:

■ Linear and modular scalability

■ Strictly consistent reads and writes

■ Automatic and configurable sharding of tables

■ Automatic failover support between RegionServers

■ Convenient base classes for backing Hadoop MapReduce jobs with Apache HBase tables

■ Easy-to-use Java API for client access

HBase Data Model Overview

A table in HBase is similar to other databases, having rows and columns. Columns in HBase are grouped into column families, all with the same prefix. For example, consider a table of daily stock prices. There may be a column family called "price" that has four members—price:open, price:close, price:low, and price:high. A column does not need to be a family. For instance, the stock table may have a column named "volume" indicating how many shares were traded. All column family members are stored together in the physical file system.

Specific HBase cell values are identified by a row key, column (column family and column), and version (timestamp). It is possible to have many versions of data within an HBase cell. A version is specified as a timestamp and is created each time data are written to a cell. Almost anything can serve as a row key, from strings to binary representations of longs to serialized data structures. Rows are lexicographically sorted with the lowest order appearing first in a table. The empty byte array denotes both the start and the end of a table's namespace. All table accesses are via the table row key, which is considered its primary key.

HBase Example Walk-Through

For this example, the following software environment is assumed. Other environments should work in a similar fashion.

- OS: Linux
- Platform: CentOS 6.6
- Hortonworks HDP 2.2 with Hadoop version: 2.6
- HBase version: 0.98.4

If you are using the pseudo-distributed installation from Chapter 2 or want to install HBase by hand, see the installation instructions on the HBase website: http://hbase.apache.org. HBase is also installed as part of the Hortonworks HDP Sandbox.

The following example illustrates a small subset of HBase commands. Consult the HBase website for more background. HBase provides a shell for interactive use. To enter the shell, type the following as a user:

```
$ hbase shell
hbase(main):001:0>
```

To exit the shell, type `exit`.

Various commands can be conveniently entered from the shell prompt. For instance, the `status` command provides the system status:

```
hbase(main):001:0> status
4 servers, 0 dead, 1.0000 average load
```

Additional arguments can be added to the status command, including `'simple'`, `'summary'`, or `'detailed'`. The single quotes are needed for proper operation. For example, the following command will provide simple status information for the four HBase servers (actual server statistics have been removed for clarity):

```
hbase(main):002:0> status 'simple'
4 live servers
n1:60020 1429912048329
...
n2:60020 1429912040653
...
limulus:60020 1429912041396
....
n0:60020 1429912042885
...
0 dead servers
Aggregate load: 0, regions: 4
```

Other basic commands, such as `version` or `whoami`, can be entered directly at the `hbase(main)` prompt. In the example that follows, we will use a small set of daily stock price data for Apple computer. The data have the following form:

Date	Open	High	Low	Close	Volume
6-May-15	126.56	126.75	123.36	125.01	71820387

The data can be downloaded from Google using the following command. Note that other stock prices are available by changing the `NASDAQ:AAPL` argument to any other valid exchange and stock name (e.g., `NYSE: IBM`).

```
$ wget -O Apple-stock.csv
http://www.google.com/finance/historical?q=NASDAQ:AAPL\&authuser=0\&output=csv
```

The Apple stock price database is in comma-separated format (csv) and will be used to illustrate some basic operations in the HBase shell.

Create the Database

The next step is to create the database in HBase using the following command:

```
hbase(main):006:0> create 'apple', 'price' , 'volume'
0 row(s) in 0.8150 seconds
```

In this case, the table name is `apple`, and two columns are defined. The date will be used as the row key. The `price` column is a family of four values (`open`, `close`, `low`, `high`). The `put` command is used to add data to the database from within the shell. For instance, the preceding data can be entered by using the following commands:

```
put'apple','6-May-15','price:open','126.56'
put'apple','6-May-15','price:high','126.75'
put'apple','6-May-15','price:low','123.36'
put'apple','6-May-15','price:close','125.01'
put 'apple','6-May-15','volume','71820387'
```

Note that these commands can be copied and pasted into HBase shell and are available from the book download files (see Appendix A). The shell also keeps a history for the session, and previous commands can be retrieved and edited for resubmission.

Inspect the Database

The entire database can be listed using the `scan` command. Be careful when using this command with large databases. This example is for one row.

```
scan 'apple'
hbase(main):006:0> scan 'apple'
ROWCOLUMN+CELL
 6-May-15 column=price:close, timestamp=1430955128359, value=125.01
 6-May-15 column=price:high, timestamp=1430955126024, value=126.75
 6-May-15 column=price:low, timestamp=1430955126053, value=123.36
 6-May-15 column=price:open, timestamp=1430955125977, value=126.56
 6-May-15 column=volume:, timestamp=1430955141440, value=71820387
```

Get a Row

You can use the row key to access an individual row. In the stock price database, the date is the row key.

```
hbase(main):008:0> get 'apple', '6-May-15'
COLUMN CELL
price:close timestamp=1430955128359, value=125.01
price:high timestamp=1430955126024, value=126.75
price:low timestamp=1430955126053, value=123.36
price:open timestamp=1430955125977, value=126.56
volume: timestamp=1430955141440, value=71820387
5 row(s) in 0.0130 seconds
```

Get Table Cells

A single cell can be accessed using the `get` command and the `COLUMN` option:

```
hbase(main):013:0> get 'apple', '5-May-15', {COLUMN => 'price:low'}
COLUMN      CELL
 price:low  timestamp=1431020767444,                    value=125.78
1 row(s) in 0.0080 seconds
```

In a similar fashion, multiple columns can be accessed as follows:

```
hbase(main):012:0> get 'apple', '5-May-15', {COLUMN => ['price:low', 'price:high']}
COLUMN      CELL
 price:high timestamp=1431020767444,                    value=128.45
 price:low  timestamp=1431020767444,                    value=125.78
2 row(s) in 0.0070 seconds
```

Delete a Cell

A specific cell can be deleted using the following command:

```
hbase(main):009:0> delete 'apple', '6-May-15' , 'price:low'
```

If the row is inspected using `get`, the `price:low` cell is not listed.

```
hbase(main):010:0> get 'apple', '6-May-15'
COLUMN      CELL
 price:close timestamp=1430955128359,                   value=125.01
 price:high timestamp=1430955126024,                    value=126.75
 price:open timestamp=1430955125977,                    value=126.46
 volume:    timestamp=1430955141440,                   value=71820387
4 row(s) in 0.0130 seconds
```

Delete a Row

You can delete an entire row by giving the `deleteall` command as follows:

```
hbase(main):009:0> deleteall 'apple', '6-May-15'
```

Remove a Table

To remove (drop) a table, you must first disable it. The following two commands remove the `apple` table from Hbase:

hbase(main):009:0> disable 'apple'
hbase(main):010:0> drop 'apple'

Scripting Input

Commands to the HBase shell can be placed in bash scripts for automated processing. For instance, the following can be placed in a bash script:

```
echo "put 'apple','6-May-15','price:open','126.56'" | hbase shell
```

The book software page includes a script (`input_to_hbase.sh`) that imports the `Apple-stock.csv` file into HBase using this method. It also removes the column titles in the first line. The script will load the entire file into HBase when you issue the following command:

```
$ input_to_hbase.sh Apple-stock.csv
```

While the script can be easily modified to accommodate other types of data, it is not recommended for production use because the upload is very inefficient and slow. Instead, this script is best used to experiment with small data files and different types of data.

Hadoop YARN Applications

YARN Distributed-Shell

The Hadoop YARN project includes the Distributed-Shell application, which is an example of a Hadoop non-MapReduce application built on top of YARN. Distributed-Shell is a simple mechanism for running shell commands and scripts in containers on multiple nodes in a Hadoop cluster. This application is not meant to be a production administration tool, but rather a demonstration of the non-MapReduce capability that can be implemented on top of YARN. There are multiple mature implementations of a distributed shell that administrators typically use to manage a cluster of machines.

Using the YARN Distributed-Shell

For the purpose of the examples presented in the remainder of this chapter, we assume and assign the following installation path, based on Hortonworks HDP 2.2, the Distributed-Shell application:

```
$ export YARN_DS=/usr/hdp/current/hadoop-yarn-client/hadoop-yarn-applications-distributedshell.jar
```

For the pseudo-distributed install using Apache Hadoop version 2.6.0, the following path will run the Distributed-Shell application

```
$ export YARN_DS=$HADOOP_HOME/share/hadoop/yarn/hadoop-yarn-applications-distributedshell-2.6.0.jar
```

If another distribution is used, search for the file `hadoop-yarn-applications-distributedshell*.jar` and set `$YARN_DS` based on its location. Distributed-Shell exposes various options that can be found by running the following command:

$ yarn org.apache.hadoop.yarn.applications.distributedshell.Client -jar $YARN_DS -help

A Simple Example

The simplest use-case for the Distributed-Shell application is to run an arbitrary shell command in a container. We will demonstrate the use of the `uptime` command as an example. This command is run on the cluster using Distributed-Shell as follows:

$ yarn org.apache.hadoop.yarn.applications.distributedshell.Client -jar $YARN_DS -shell_command uptime

By default, Distributed-Shell spawns only one instance of a given shell command. When this command is run, you can see progress messages on the screen but nothing about the actual shell command. If the shell command succeeds, the following should appear at the end of the output:

15/05/27 14:48:53 INFO distributedshell.Client: Application completed successfully

If the shell command did not work for whatever reason, the following message will be displayed:

15/05/27 14:58:42 ERROR distributedshell.Client: Application failed to complete successfully

Assuming log aggregation is enabled, the results for each instance of the command can be found by using the `yarn logs` command. For the previous `uptime` example, the following command can be used to inspect the logs:

$ yarn logs -applicationId application_1432831236474_0001

Structure of YARN Applications

A full explanation of writing YARN programs is beyond the scope of this book. The structure and operation of a YARN application are covered briefly in this section.

The central YARN ResourceManager runs as a scheduling daemon on a dedicated machine and acts as the central authority for allocating resources to the various competing applications in the cluster. The ResourceManager has a central and global view of all cluster resources and, therefore, can ensure fairness, capacity, and locality are shared across all users. Depending on the application demand, scheduling priorities, and resource availability, the ResourceManager dynamically allocates resource containers to applications to run on particular nodes. A container is a logical bundle of resources (e.g., memory, cores) bound to a particular cluster node. To enforce and track such assignments, the ResourceManager interacts with a special system daemon running on each node called the NodeManager. Communications between the ResourceManager and NodeManagers are heartbeat based for scalability. NodeManagers are responsible for local monitoring of resource availability, fault reporting, and container life-cycle management (e.g., starting and killing jobs). The ResourceManager depends on the NodeManagers for its “global view” of the cluster.

User applications are submitted to the ResourceManager via a public protocol and go through an admission control phase during which security credentials are validated and various operational and administrative checks are performed. Those applications that are accepted pass to the scheduler and are allowed to run. Once the scheduler has enough resources to satisfy the request, the application is moved from an accepted state to a running state. Aside from internal bookkeeping, this process involves allocating a container for the single ApplicationMaster and spawning it on a node in the cluster. Often called container 0, the ApplicationMaster does not have any additional resources at this point, but rather must request additional resources from the ResourceManager.

The ApplicationMaster is the "master" user job that manages all application life-cycle aspects, including dynamically increasing and decreasing resource consumption (i.e., containers), managing the flow of execution (e.g., in case of MapReduce jobs, running reducers against the output of maps), handling faults and computation skew, and performing other local optimizations. The ApplicationMaster is designed to run arbitrary user code that can be written in any programming language, as all communication with the ResourceManager and NodeManager is encoded using extensible network protocols (i.e., Google Protocol Buffers, http://code.google.com/p/protobuf/).

YARN makes few assumptions about the ApplicationMaster, although in practice it expects most jobs will use a higher-level programming framework. By delegating all these functions to ApplicationMasters, YARN's architecture gains a great deal of scalability, programming model flexibility, and improved user agility. For example, upgrading and testing a new MapReduce framework can be done independently of other running MapReduce frameworks.

Typically, an ApplicationMaster will need to harness the processing power of multiple servers to complete a job. To achieve this, the ApplicationMaster issues resource requests to the ResourceManager. The form of these requests includes specification of locality preferences (e.g., to accommodate HDFS use) and properties of the containers. The ResourceManager will attempt to satisfy the resource requests coming from each application according to availability and scheduling policies. When a resource is scheduled on behalf of an ApplicationMaster, the ResourceManager generates a lease for the resource, which is acquired by a subsequent ApplicationMaster heartbeat. The ApplicationMaster then works with the NodeManagers to start the resource. A token-based security mechanism guarantees its authenticity when the ApplicationMaster presents the container lease to the NodeManager. In a typical situation, running containers will communicate with the ApplicationMaster through an application-specific protocol to report status and health information and to receive framework-specific commands. In this way, YARN provides a basic infrastructure for monitoring and life-cycle management of containers, while each framework manages application-specific semantics independently. This design stands in sharp contrast to the original Hadoop version 1 design, in which scheduling was designed and integrated around managing only MapReduce tasks.

Figure 8.1 illustrates the relationship between the application and YARN components. The YARN components appear as the large outer boxes (ResourceManager and NodeManagers), and the two applications appear as smaller boxes (containers), one dark and one light. Each application uses a different ApplicationMaster; the darker client is running a Message Passing Interface (MPI) application and the lighter client is running a traditional MapReduce application.

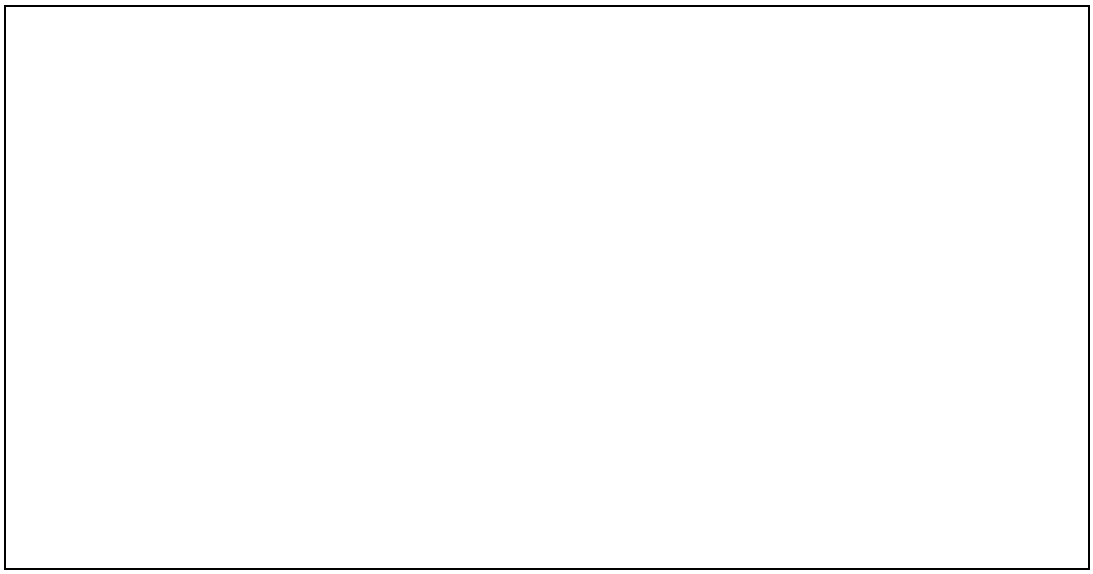

Figure 8.1 YARN architecture with two clients (MapReduce and MPI).

YARN Application Frameworks

One of the most exciting aspects of Hadoop version 2 is the capability to run all types of applications on a Hadoop cluster. In Hadoop version 1, the only processing model available to users is MapReduce. In Hadoop version 2, MapReduce is separated from the resource management layer of Hadoop and placed into its own application framework.

YARN presents a resource management platform, which provides services such as scheduling, fault monitoring, data locality, and more to MapReduce and other frameworks. Figure 8.2 illustrates some of the various frameworks that will run under YARN. Note that the Hadoop version 1 applications (e.g., Pig and Hive) run under the MapReduce framework.

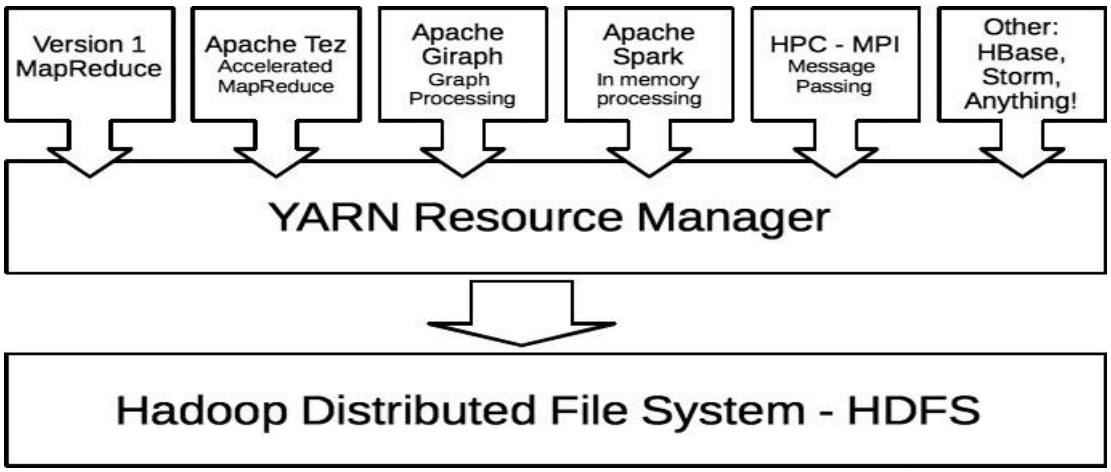

Figure 8.2 Example of the Hadoop version 2 ecosystem. Hadoop version 1 supports batch MapReduce applications only.

Distributed-Shell

As described earlier in this chapter, Distributed-Shell is an example application included with the Hadoop core components that demonstrates how to write applications on top of YARN. It provides

a simple method for running shell commands and scripts in containers in parallel on a Hadoop YARN cluster.

Hadoop MapReduce

MapReduce was the first YARN framework and drove many of YARN's requirements. It is integrated tightly with the rest of the Hadoop ecosystem projects, such as Apache Pig, Apache Hive, and Apache Oozie.

Apache Tez

One great example of a new YARN framework is Apache Tez. Many Hadoop jobs involve the execution of a complex directed acyclic graph (DAG) of tasks using separate MapReduce stages. Apache Tez generalizes this process and enables these tasks to be spread across stages so that they can be run as a single, all-encompassing job.

Tez can be used as a MapReduce replacement for projects such as Apache Hive and Apache Pig. No changes are needed to the Hive or Pig applications.

Apache Giraph

Apache Giraph is an iterative graph processing system built for high scalability. Facebook, Twitter, and LinkedIn use it to create social graphs of users. Giraph was originally written to run on standard Hadoop V1 using the MapReduce framework, but that approach proved inefficient and totally unnatural for various reasons. The native Giraph implementation under YARN provides the user with an iterative processing model that is not directly available with MapReduce. Support for YARN has been present in Giraph since its own version 1.0 release. In addition, using the flexibility of YARN, the Giraph developers plan on implementing their own web interface to monitor job progress.

Hoya: HBase on YARN

The Hoya project creates dynamic and elastic Apache HBase clusters on top of YARN. A client application creates the persistent configuration files, sets up the HBase cluster XML files, and then asks YARN to create an ApplicationMaster. YARN copies all files listed in the client's application-launch request from HDFS into the local file system of the chosen server, and then executes the command to start the Hoya ApplicationMaster. Hoya also asks YARN for the number of containers matching the number of HBase region servers it needs.

Dryad on YARN

Similar to Apache Tez, Microsoft's Dryad provides a DAG as the abstraction of execution flow. This framework is ported to run natively on YARN and is fully compatible with its non-YARN version. The code is written completely in native C++ and C# for worker nodes and uses a thin layer of Java within the application.

Apache Spark

Spark was initially developed for applications in which keeping data in memory improves performance, such as iterative algorithms, which are common in machine learning, and interactive data mining. Spark differs from classic MapReduce in two important ways. First, Spark holds intermediate results in memory, rather than writing them to disk. Second, Spark supports more than just MapReduce functions; that is, it greatly expands the set of possible analyses that can be executed over HDFS data stores. It also provides APIs in Scala, Java, and Python.

Since 2013, Spark has been running on production YARN clusters at Yahoo!. The advantage of porting and running Spark on top of YARN is the common resource management and a single underlying file system.

Apache Storm

Traditional MapReduce jobs are expected to eventually finish, but Apache Storm continuously processes messages until it is stopped. This framework is designed to process unbounded streams of data in real time. It can be used in any programming language. The basic Storm use-cases include real-time analytics, online machine learning, continuous computation, distributed RPC (remote procedure calls), ETL (extract, transform, and load), and more. Storm provides fast performance, is scalable, is fault tolerant, and provides processing guarantees. It works directly under YARN and takes advantage of the common data and resource management substrate.

Apache REEF: Retainable Evaluator Execution Framework

YARN's flexibility sometimes requires significant effort on the part of application implementers. The steps involved in writing a custom application on YARN include building your own ApplicationMaster, performing client and container management, and handling aspects of fault tolerance, execution flow, coordination, and other concerns. The REEF project by Microsoft recognizes this challenge and factors out several components that are common to many applications, such as storage management, data caching, fault detection, and checkpoints. Framework designers can build their applications on top of REEF more easily than they can build those same applications directly on YARN, and can reuse these common services/libraries. REEF's design makes it suitable for both MapReduce and DAG-like executions as well as iterative and interactive computations.

Hamster: Hadoop and MPI on the Same Cluster

The Message Passing Interface (MPI) is widely used in high-performance computing (HPC). MPI is primarily a set of optimized message-passing library calls for C, C++, and Fortran that operate over popular server interconnects such as Ethernet and InfiniBand. Because users have full control over their YARN containers, there is no reason why MPI applications cannot run within a Hadoop cluster. The Hamster effort is a work-in-progress that provides a good discussion of the issues involved in mapping MPI to a YARN cluster (see https://issues.apache.org/jira/browse/MAPREDUCE-2911). Currently, an alpha version of MPICH2 is available for YARN that can be used to run MPI applications.

Apache Flink: Scalable Batch and Stream Data Processing

Apache Flink is a platform for efficient, distributed, general-purpose data processing. It features powerful programming abstractions in Java and Scala, a high-performance run time, and automatic program optimization. It also offers native support for iterations, incremental iterations, and programs consisting of large DAGs of operations.

Flink is primarily a stream-processing framework that can look like a batch-processing environment. The immediate benefit from this approach is the ability to use the same algorithms for both streaming and batch modes (exactly as is done in Apache Spark). However, Flink can provide low-latency similar to that found in Apache Storm, but which is not available in Apache Spark.

In addition, Flink has its own memory management system, separate from Java's garbage collector. By managing memory explicitly, Flink almost eliminates the memory spikes often seen on Spark clusters.

Apache Slider: Dynamic Application Management

Apache Slider (incubating) is a YARN application to deploy existing distributed applications on YARN, monitor them, and make them larger or smaller as desired in real time.

Applications can be stopped and then started; the distribution of the deployed application across the YARN cluster is persistent and allows for best-effort placement close to the previous locations. Applications that remember the previous placement of data (such as HBase) can exhibit fast startup times by capitalizing on this feature.

YARN monitors the health of "YARN containers" that are hosting parts of the deployed applications. If a container fails, the Slider manager is notified. Slider then requests a new replacement container from the YARN ResourceManager. Some of Slider's other features include user creation of on-demand applications, the ability to stop and restart applications as needed (preemption), and the ability to expand or reduce the number of application containers as needed. The Slider tool is a Java command-line application.

Managing Hadoop with Apache Ambari

Quick Tour of Apache Ambari

After completing the initial installation and logging into Ambari, a dashboard similar to that shown in Figure 9.1 is presented. The same four-node cluster as created in will be used to explore Ambari in this chapter. If you need to reopen the Ambari dashboard interface, simply enter the following command (which assumes you are using the Firefox browser, although other browsers may also be used):

$ firefox localhost:8080

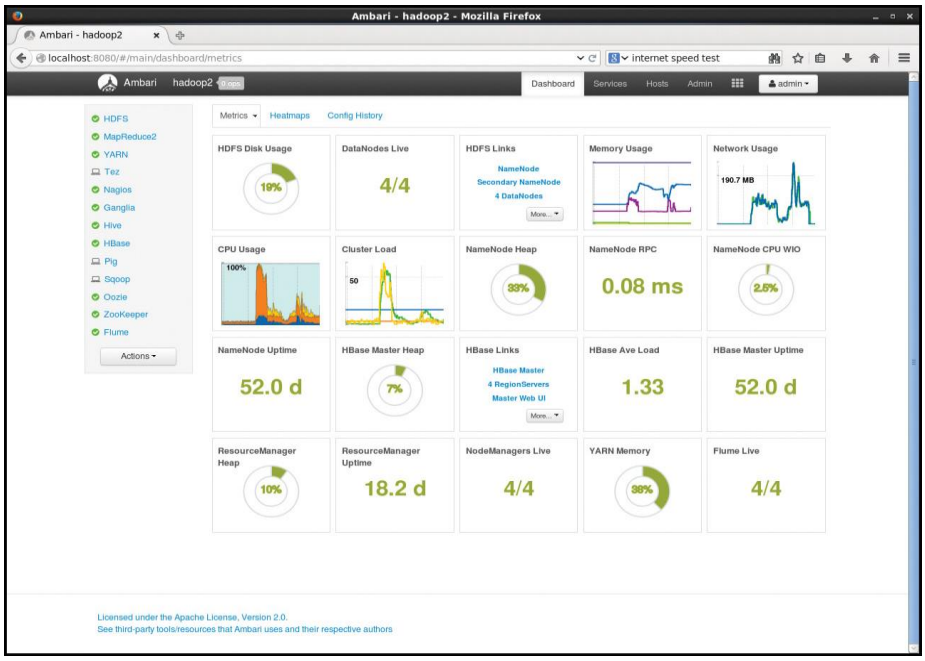

Figure 9.1 Apache Ambari dashboard view of a Hadoop cluster

The default login and password are `admin` and `admin`, respectively. Before continuing any further, you should change the default password. To change the password, select Manage Ambari from the Admin pull-down menu in the upper-right corner. In the management window, click Users under User + Group Management, and then click the `admin` user name. Select Change Password and enter a new password. When you are finished, click the Go To Dashboard link on the left side of the window to return to the dashboard view.

To leave the Ambari interface, select the Admin pull-down menu at the left side of the main menu bar and click Sign out.

The dashboard view provides a number of high-level metrics for many of the installed services. A glance at the dashboard should allow you to get a sense of how the cluster is performing.

The top navigation menu bar, shown in Figure 9.1, provides access to the Dashboard, Services, Hosts, Admin, and Views features (the 3 × 3 cube is the Views menu). The status (up/down) of various Hadoop services is displayed on the left using green/orange dots. Note that two of the services managed by Ambari are Nagios and Ganglia; the standard cluster management services installed by Ambari, they are used to provide cluster monitoring (Nagios) and metrics (Ganglia).

Dashboard View

The Dashboard view provides small status widgets for many of the services running on the cluster. The actual services are listed on the left-side vertical menu.

■ **Moving:** Click and hold a widget while it is moved about the grid.

■ **Edit:** Place the mouse on the widget and click the gray edit symbol in the upper-right corner of the widget. You can change several different aspects (including thresholds) of the widget.

■ **Remove:** Place the mouse on the widget and click the X in the upper-left corner.

■ **Add:** Click the small triangle next to the Metrics tab and select Add. The available widgets will be displayed. Select the widgets you want to add and click Apply.

Some widgets provide additional information when you move the mouse over them. For instance, the DataNodes widget displays the number of live, dead, and decommissioning hosts. Clicking directly on a graph widget provides an enlarged view. For instance, Figure 9.2 provides a detailed view of the CPU Usage widget from Figure 9.1.

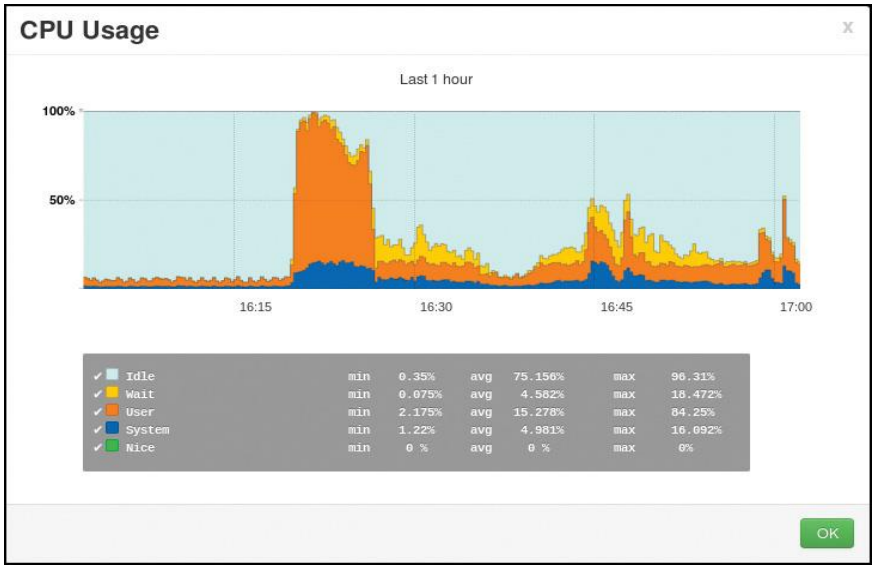

Figure 9.2 Enlarged view of Ambari CPU Usage widget

The Dashboard view also includes a heatmap view of the cluster. Cluster heatmaps physically map selected metrics across the cluster. When you click the Heatmaps tab, a heatmap for the cluster will be displayed. To select the metric used for the heatmap, choose the desired option from the Select Metric pull-down menu. Note that the scale and color ranges are different for each metric. The heatmap for percentage host memory used is displayed in Figure 9.3.

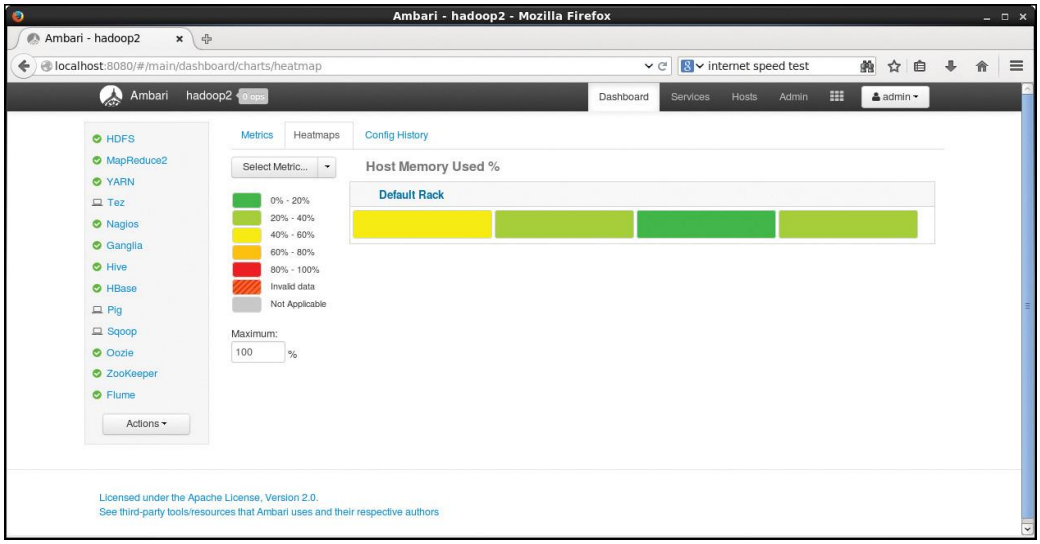

Figure 9.3 Ambari heatmap for Host memory usage

Configuration history is the final tab in the dashboard window. This view provides a list of configuration changes made to the cluster. As shown in Figure 9.4, Ambari enables configurations to be sorted by Service, Configuration Group, Data, and Author. To find the specific configuration settings, click the service name. More information on configuration settings is provided later in the chapter.

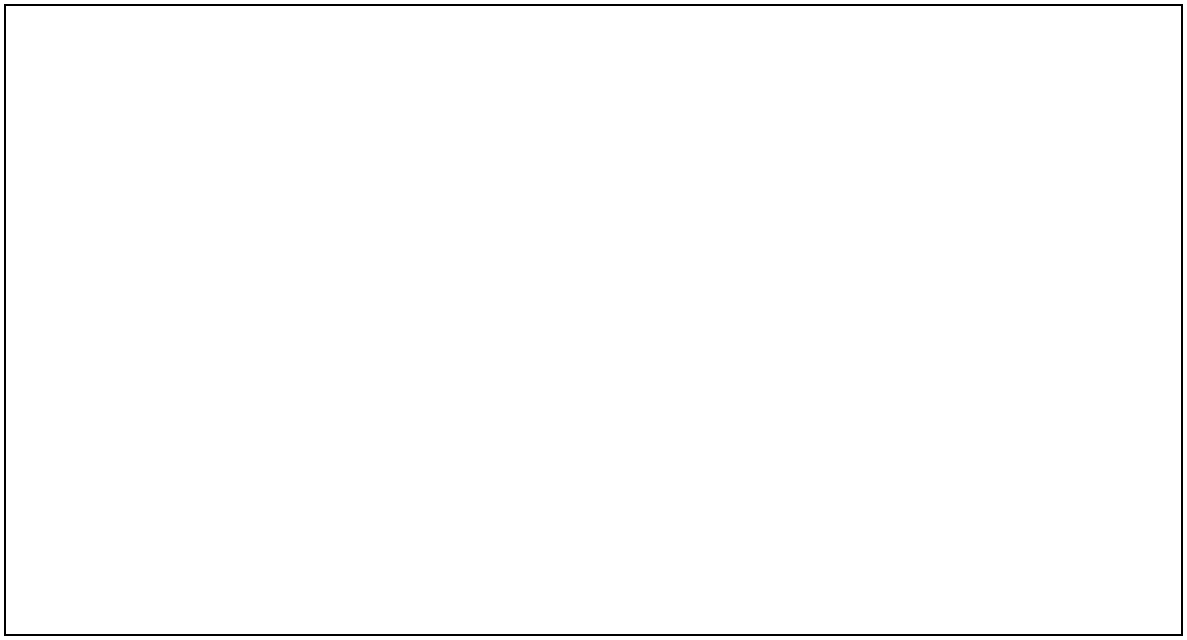

Figure 9.4 Ambari master configuration changes list

Services View

The Services menu provides a detailed look at each service running on the cluster. It also provides a graphical method for configuring each service (i.e., instead of hand-editing the `/etc/hadoop/conf` XML files). The summary tab provides a current Summary view of important service metrics and an Alerts and Health Checks sub-window.

Similar to the Dashboard view, the currently installed services are listed on the left-side menu. To select a service, click the service name in the menu. When applicable, each service will have its own Summary, Alerts and Health Monitoring, and Service Metrics windows. For example, Figure 9.5 shows the Service view for HDFS. Important information such as the status of NameNode, SecondaryNameNode, DataNodes, uptime, and available disk space is displayed in the Summary window. The Alerts and Health Checks window provides the latest status of the service and its component systems. Finally, several important real-time service metrics are displayed as widgets at the bottom of the screen. As on the dashboard, these widgets can be expanded to display a more detailed view.

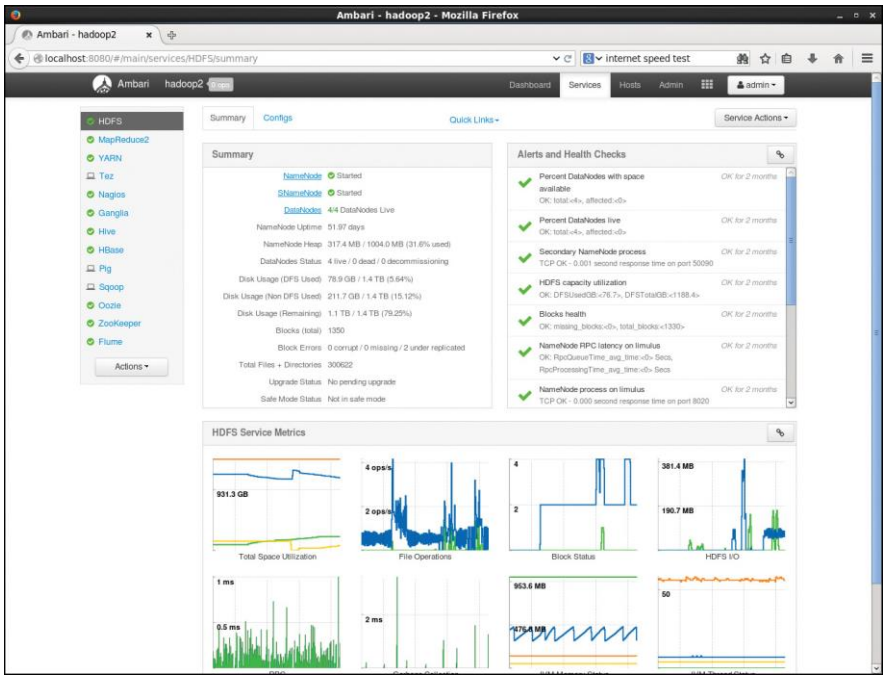

Figure 9.5 HDFS service summary window

Clicking the Configs tab will open an options form, shown in Figure 9.6, for the service. The options (properties) are the same ones that are set in the Hadoop XML files. When using Ambari, the user has complete control over the XML files and should manage them only through the Ambari interface—that is, the user should *not* edit the files by hand.

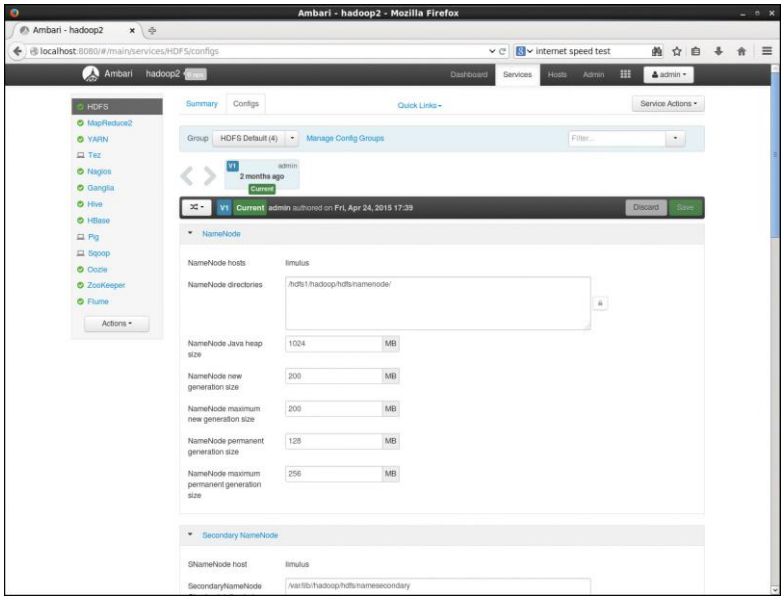

Figure 9.6 Ambari service options for HDFS

The current settings available for each service are shown in the form. The administrator can set each of these properties by changing the values in the form. Placing the mouse in the input box of the property displays a short description of each property. Where possible, properties are grouped by functionality. The form also has provisions for adding properties that are not listed. An example of

changing service properties and restarting the service components is provided in the "Managing Hadoop Services" section.

If a service provides its own graphical interface (e.g., HDFS, YARN, Oozie), then that interface can be opened in a separate browser tab by using the Quick Links pull-down menu located in top middle of the window.

Finally, the Service Action pull-down menu in the upper-left corner provides a method for starting and stopping each service and/or its component daemons across the cluster. Some services may have a set of unique actions (such as rebalancing HDFS) that apply to only certain situations. Finally, every service has a Service Check option to make sure the service is working property. The service check is initially run as part of the installation process and can be valuable when diagnosing problems.

Hosts View

Selecting the Hosts menu item provides the information shown in Figure 9.7. The host name, IP address, number of cores, memory, disk usage, current load average, and Hadoop components are listed in this window in tabular form.

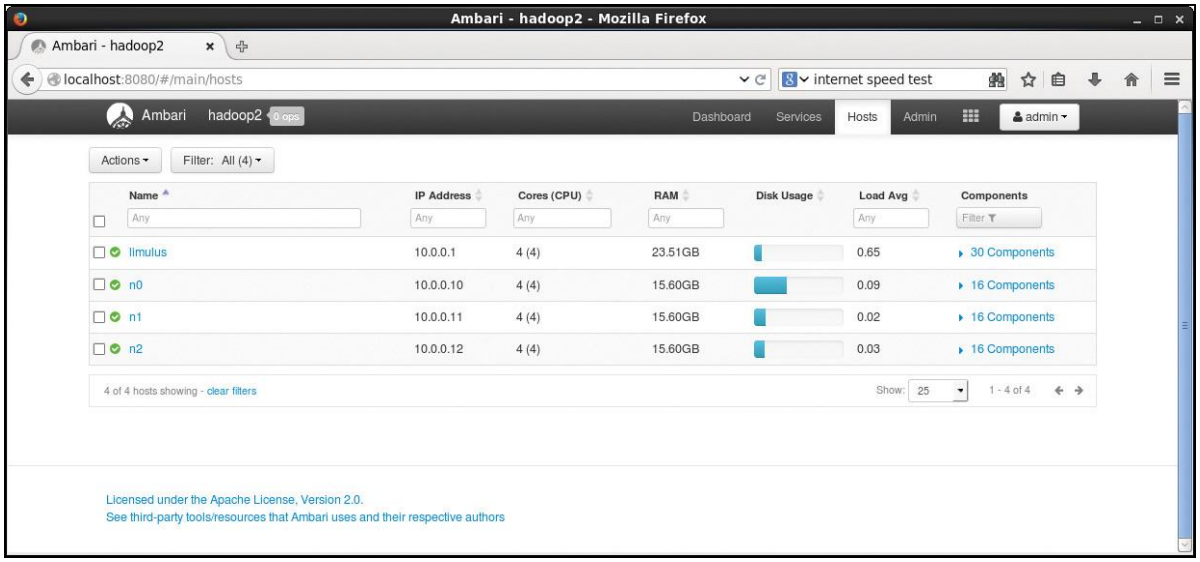

Figure 9.7 Ambari main Hosts screen

To display the Hadoop components installed on each host, click the links in the rightmost columns. You can also add new hosts by using the Actions pull-down menu. The remaining options in the Actions pull-down menu provide control over the various service components running on the hosts.

Further details for a particular host can be found by clicking the host name in the left column. As shown in Figure 9.8, the individual host view provides three sub-windows: Components, Host Metrics, and Summary information. The Components window lists the services that are currently running on the host. Each service can be stopped, restarted, decommissioned, or placed in maintenance mode. The Metrics window displays widgets that provide important metrics (e.g., CPU, memory, disk, and network usage). Clicking the widget displays a larger version of the

graphic. The Summary window provides basic information about the host, including the last time a heartbeat was received.

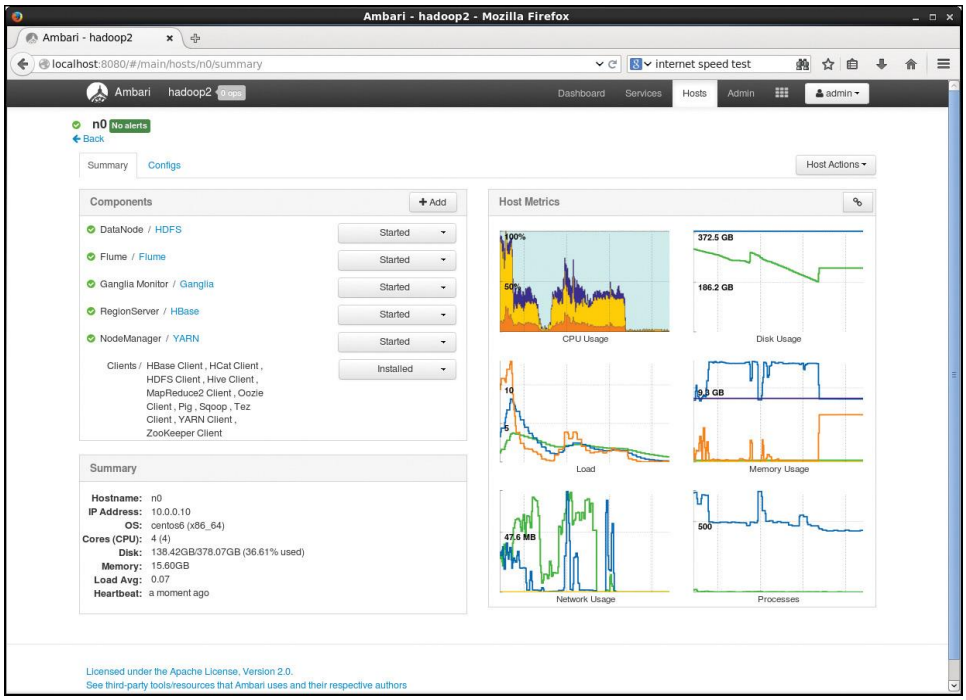

Figure 9.8 Ambari cluster host detail view

Admin View

The Administration (Admin) view provides three options. The first, as shown in Figure 9.9, displays a list of installed software. This Repositories listing generally reflects the version of Hortonworks Data Platform (HDP) used during the installation process. The Service Accounts option lists the service accounts added when the system was installed. These accounts are used to run various services and tests for Ambari. The third option, Security, sets the security on the cluster. A fully secured Hadoop cluster is important in many instances and should be explored if a secure environment is needed. This aspect of Ambari is beyond the scope of this book.

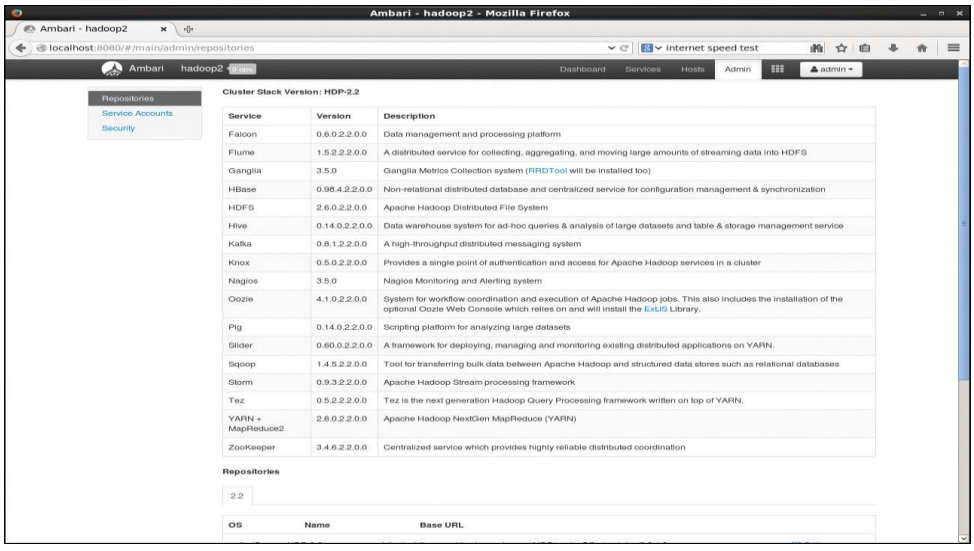

Figure 9.9 Ambari installed packages with versions, numbers, and descriptions

Views View

Ambari Views is a framework offering a systematic way to plug in user interface capabilities that provide for custom visualization, management, and monitoring features in Ambari. Views allows you to extend and customize Ambari to meet your specific needs.

Admin Pull-Down Menu

The Administrative (Admin) pull-down menu provides the following options:

- **About**—Provides the current version of Ambari.
- **Manage Ambari**—Open the management screen where Users, Groups, Permissions, and Ambari Views can be created and configured.
- **Settings**—Provides the option to turn off the progress window. (See Figure 9.15.)
- **Sign Out**—Exits the interface.

Managing Hadoop Services

During the course of normal Hadoop cluster operation, services may fail for any number of reasons. Ambari monitors all of the Hadoop services and reports any service interruption to the dashboard. In addition, when the system was installed, an administrative email for the Nagios monitoring system was required. All service interruption notifications are sent to this email address.

Figure 9.10 shows the Ambari dashboard reporting a down DataNode. The service error indicator numbers next to the HDFS service and Hosts menu item indicate this condition. The DataNode widget also has turned red and indicates that 3/4 DataNodes are operating.

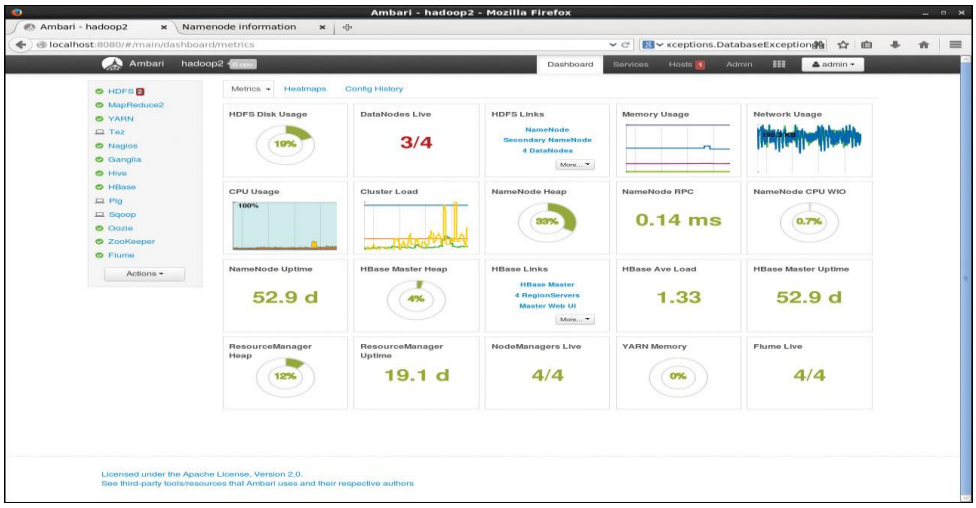

Figure 9.10 Ambari main dashboard indicating a DataNode issue

Clicking the HDFS service link in the left vertical menu will bring up the service summary screen shown in Figure 9.11. The Alerts and Health Checks window confirms that a DataNode is down.

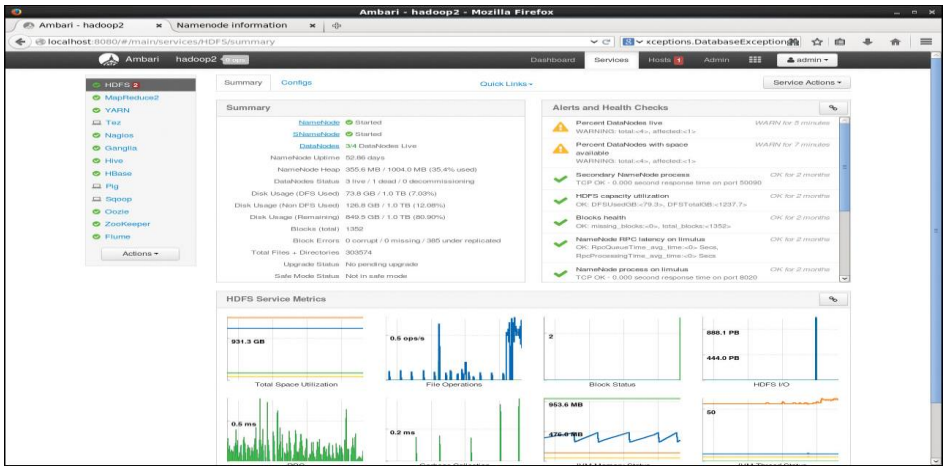

Figure 9.11 Ambari HDFS service summary window indicating a down DataNode

The specific host (or hosts) with an issue can be found by examining the Hosts window. As shown in Figure 9.12, the status of host n1 has changed from a green dot with a check mark inside to a yellow dot with a dash inside. An orange dot with a question mark inside indicates the host is not responding and is probably down. Other service interruption indicators may also be set as a result of the unresponsive node.

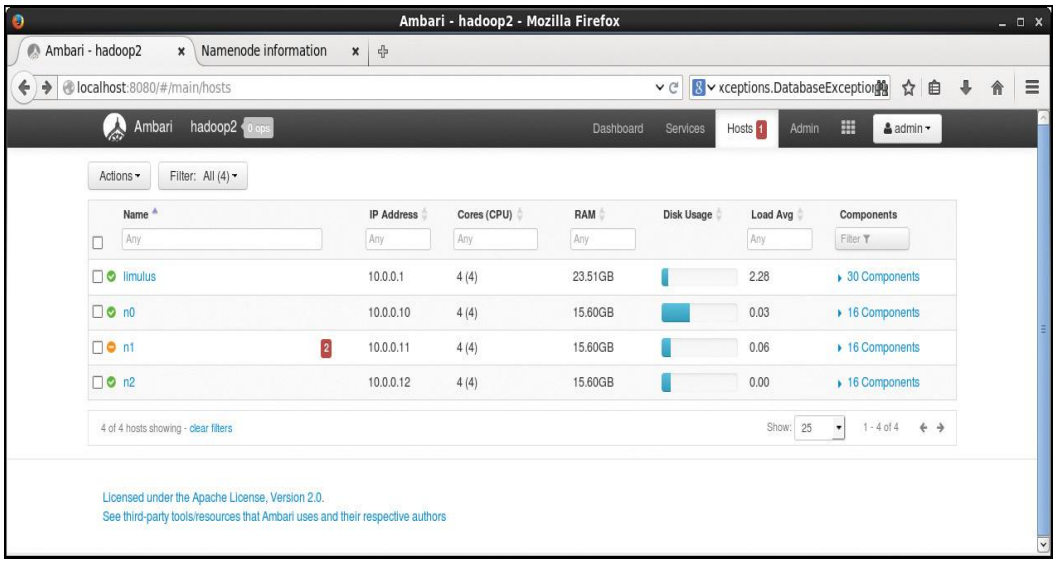

Figure 9.12 Ambari Hosts screen indicating an issue with host n1

Clicking on the n1 host link opens the view in Figure 9.13. Inspecting the Components sub-window reveals that the DataNode daemon has stopped on the host. At this point, checking the DataNode logs on host n1 will help identify the actual cause of the failure. Assuming the failure is resolved, the DataNode daemon can be started using the Start option in the pull-down menu next to the service name.

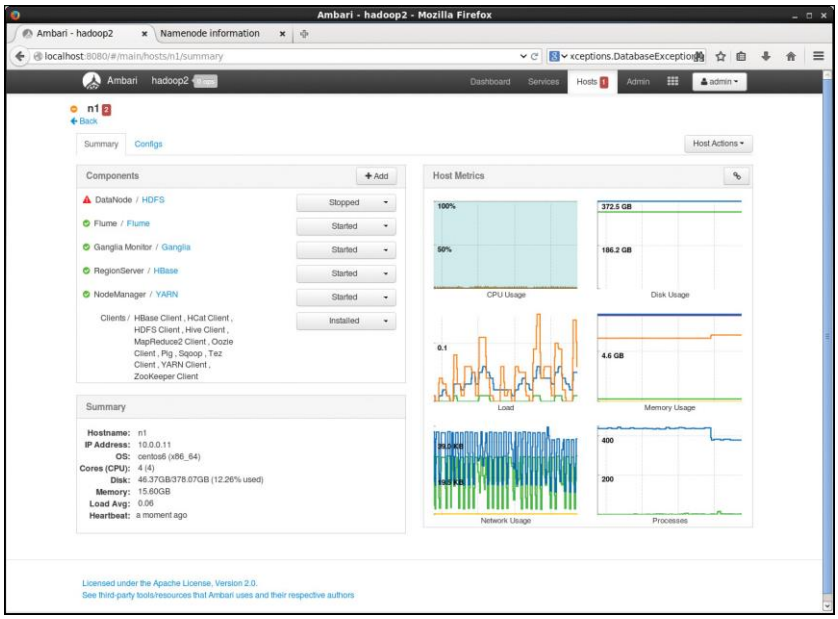

Figure 9.13 Ambari window for host n1 indicating the DataNode/HDFS service has stopped

When the DataNode daemon is restarted, a confirmation similar to Figure 9.14 is required from the user.

Figure 9.14 Ambari restart confirmation

When a service daemon is started or stopped, a progress window similar to Figure 9.15 is opened. The progress bar indicates the status of each action. Note that previous actions are part of this window. If something goes wrong during the action, the progress bar will turn red. If the system generates a warning about the action, the process bar will turn orange.

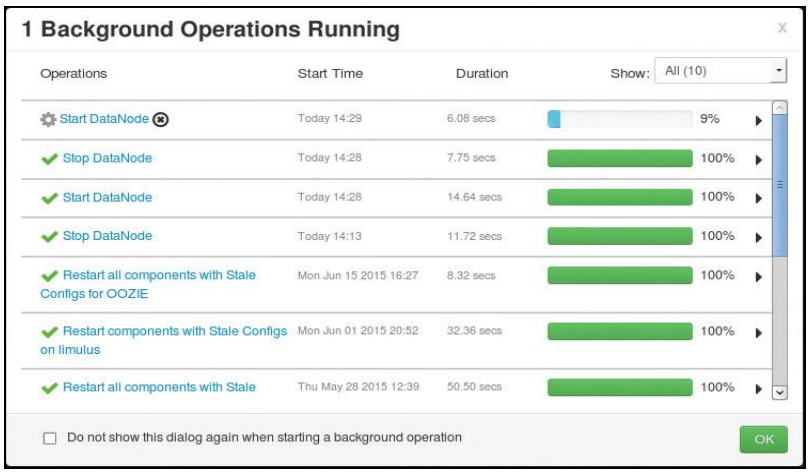

Figure 9.15 Ambari progress window for DataNode restart

When these background operations are running, the small ops (operations) bubble on the top menu bar will indicate how many operations are running. (If different service daemons are started or stopped, each process will be run to completion before the next one starts.)

Once the DataNode has been restarted successfully, the dashboard will reflect the new status (e.g., 4/4 DataNodes are Live). As shown in Figure 9.16, all four DataNodes are now working and the service error indicators are beginning to slowly disappear. The service error indicators may lag behind the real-time widget updates for several minutes.

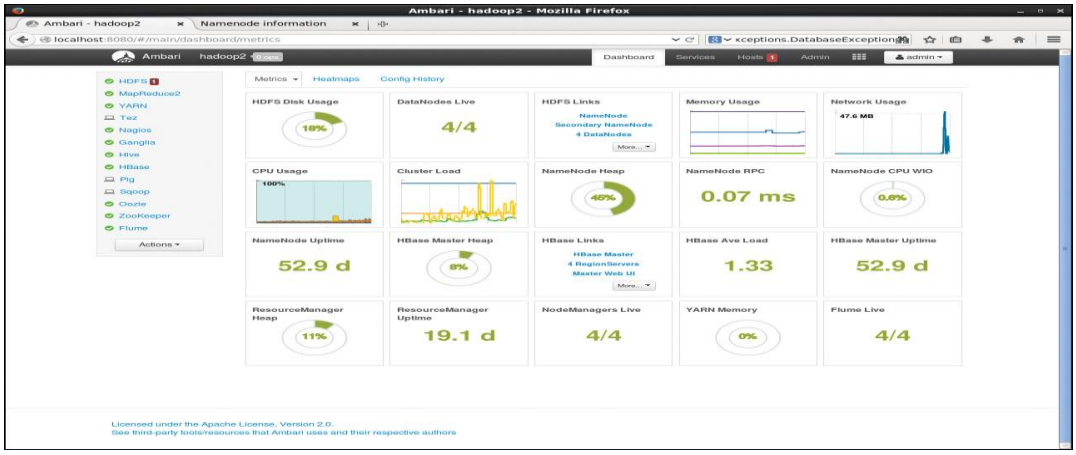

Figure 9.16 Ambari dashboard indicating all DataNodes are running (The service error indicators will slowly drop off the screen.)

Changing Hadoop Properties

One of the challenges of managing a Hadoop cluster is managing changes to cluster-wide configuration properties. In addition to modifying a large number of properties, making changes to a property often requires restarting daemons (and dependent daemons) across the entire cluster. This process is tedious and time consuming. Fortunately, Ambari provides an easy way to manage this process.

As described previously, each service provides a Configs tab that opens a form displaying all the possible service properties. Any service property can be changed (or added) using this interface. As an example, the configuration properties for the YARN scheduler are shown in Figure 9.17.

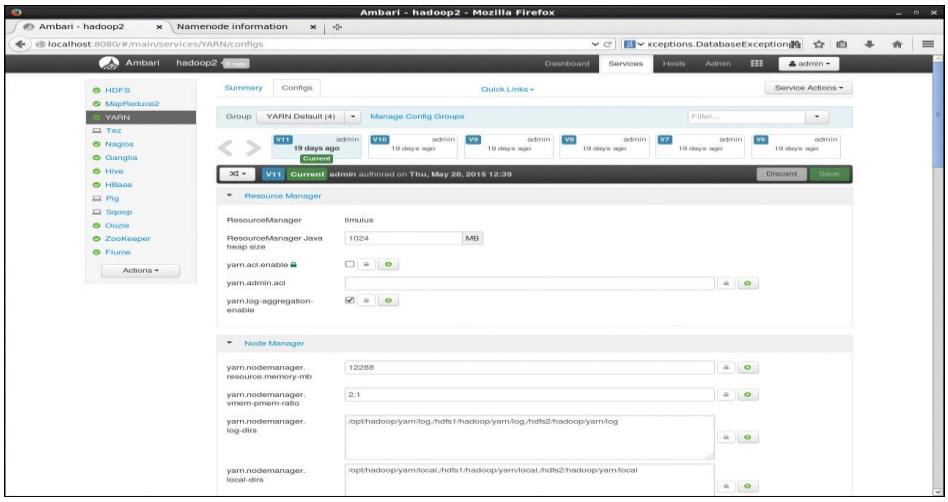

Figure 9.17 Ambari YARN properties view

The number of options offered depends on the service; the full range of YARN properties can be viewed by scrolling down the form. Both Chapter 4, “Running Example Programs and Benchmarks,” and Chapter 6, “MapReduce Programming,” discussed the YARN property `yarn.log-aggregation-enable`. To easily view the application logs, this property must be set to true. This property is normally on by default. As an example for our purposes here, we will use the Ambari interface to disable this feature. As shown in Figure 9.18, when a property is changed, the green Save button becomes activated.

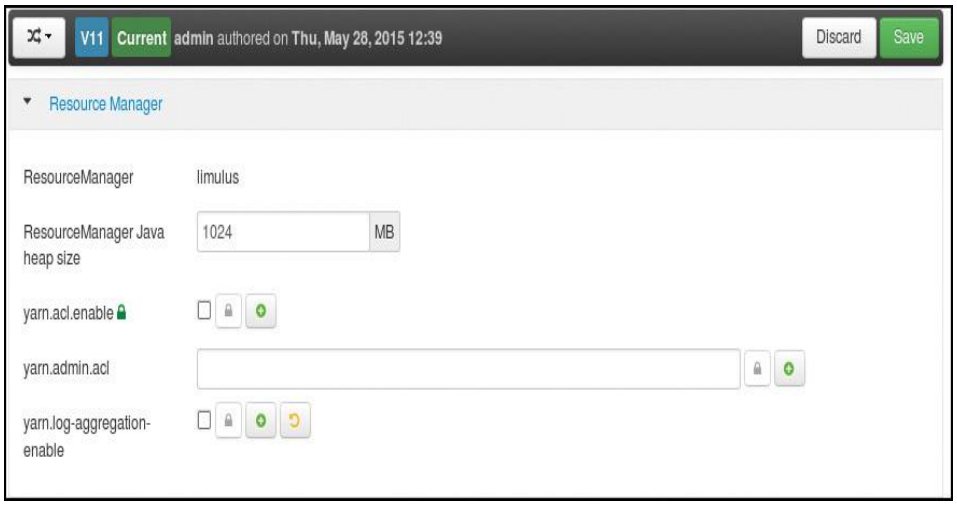

Figure 9.18 YARN properties with log aggregation turned off

Changes do not become permanent until the user clicks the Save button. A save/notes window will then be displayed. It is highly recommended that historical notes concerning the change be added to this window.

Once the user adds any notes and clicks the Save button, another window, shown in Figure 9.20, is presented. This window confirms that the properties have been saved.

Figure 9.19 Ambari configuration save/notes window

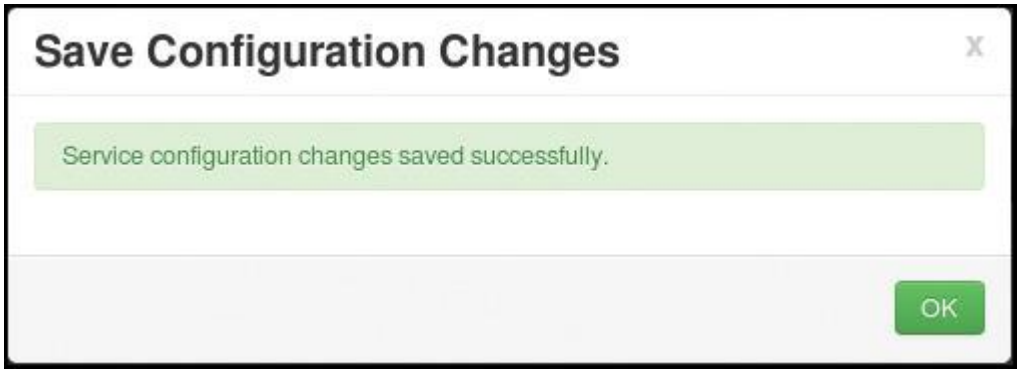

Figure 9.20 Ambari configuration change notification

Once the new property is changed, an orange Restart button will appear at the top left of the window. The new property will not take effect until the required services are restarted. As shown in Figure 9.21, the Restart button provides two options: Restart All and Restart NodeManagers. To be safe, the Restart All should be used. Note that Restart All does not mean all the Hadoop services will be restarted; rather, only those that use the new property will be restarted.

Figure 9.21 Ambari Restart function appears after changes in service properties

After the user clicks Restart All, a confirmation window, shown in Figure 9.22 will be displayed. Click Confirm Restart All to begin the cluster-wide restart.

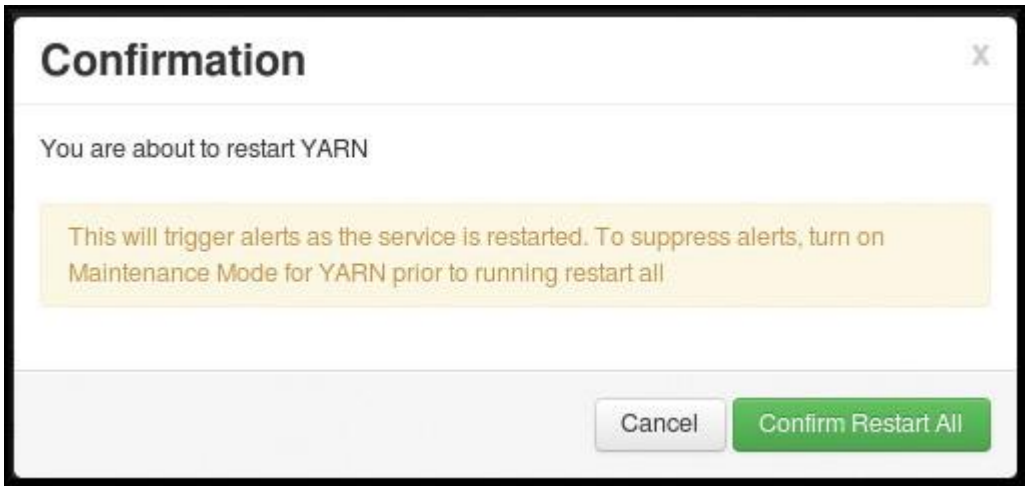

Figure 9.22 Ambari confirmation box for service restart

Similar to the DataNode restart example, a progress window will be displayed. Again, the progress bar is for the entire YARN restart. Details from the logs can be found by clicking the arrow to the right of the bar (see Figure 9.23).

Figure 9.23 Ambari progress window for cluster-wide YARN restart

Once the restart is complete, run a simple example (see Chapter 4) and attempt to view the logs using the YARN ResourceManager Applications UI. (You can access the UI from the Quick Links pull-down menu in the middle of the YARN series window.) A message similar to that in Figure 9.24 will be displayed (compare this message to the log data in Figure 6.1).

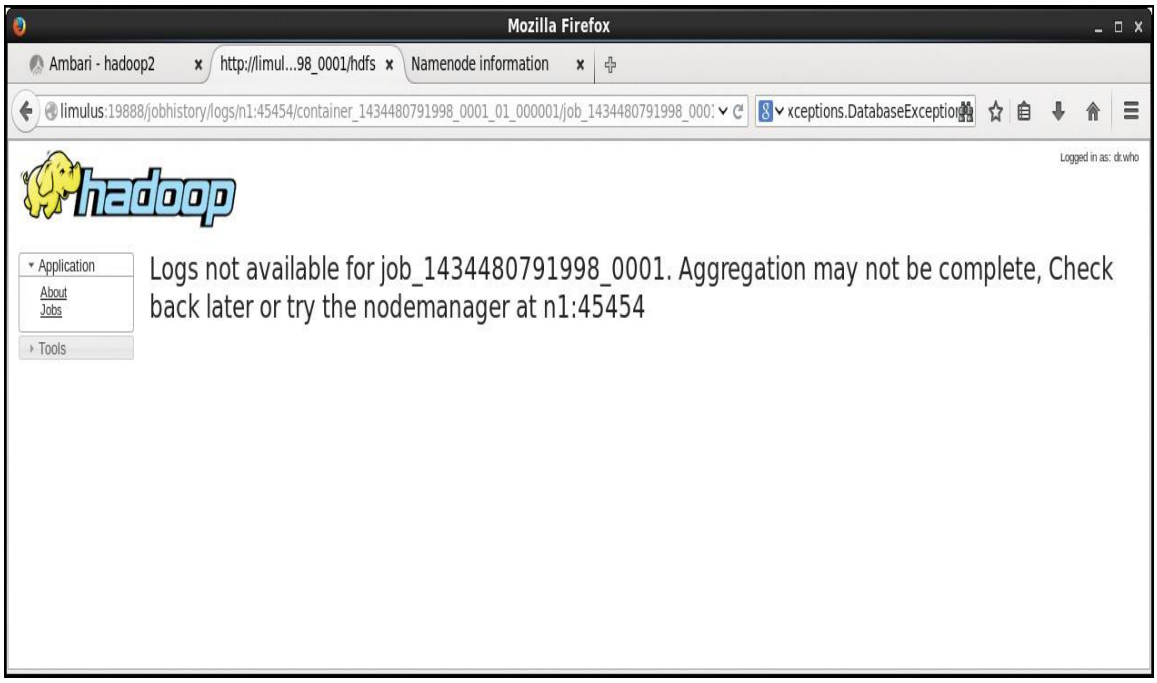

Figure 9.24 YARN ResourceManager interface with log aggregation turned off (compare to Figure 6.1)

Ambari tracks all changes made to system properties. As can be seen in Figure 9.17 and in more detail in Figure 9.25, each time a configuration is changed, a new version is created. Reverting back to a previous version results in a new version. You can reduce the potential for version confusion by providing meaningful comments for each change (e.g., Figure 9.19 and Figure 9.27). In the preceding example, we created version 12 (V12). The current version is indicated by a green Current label in the horizontal version boxes or in the dark horizontal bar. Scrolling though the version boxes or pulling down the menu on the left-hand side of the dark horizontal bar will display the previous configuration versions.

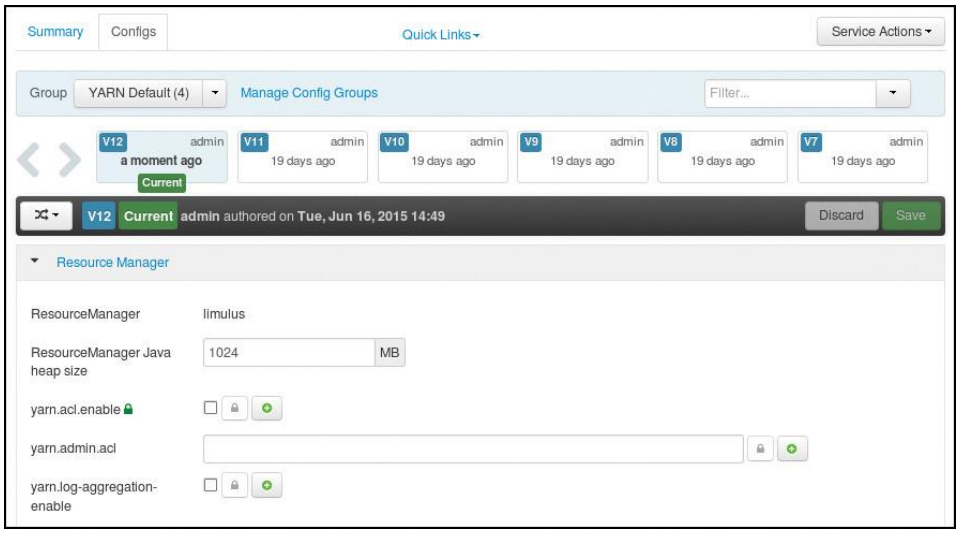

Figure 9.25 Ambari configuration change management for YARN service (Version V12 is current)

To revert to a previous version, simply select the version from the version boxes or the pull-down menu. In Figure 9.26, the user has selected the previous version by clicking the Make Current button in the information box. This configuration will return to the previous state where log aggregation is enabled.

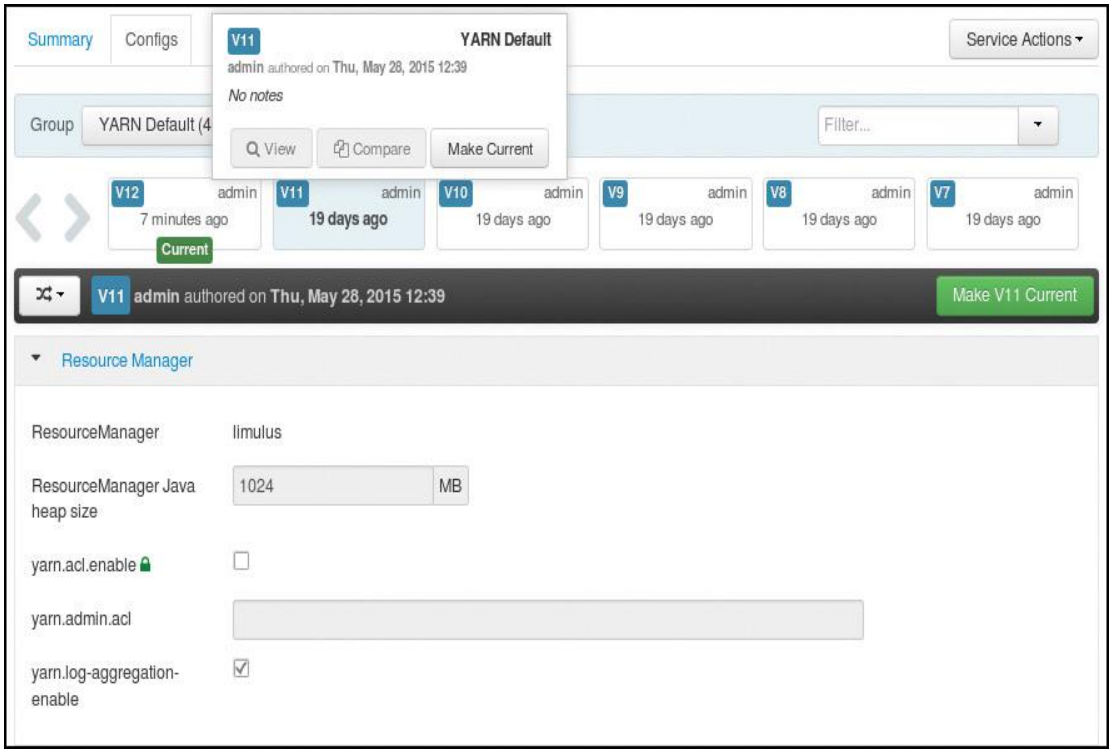

Figure 9.26 Reverting to previous YARN configuration (V11) with Ambari

As shown in Figure 9.27, a confirmation/notes window will open before the new configuration is saved. Again, it is suggested that you provide notes about the change in the Notes text box. When the save step is complete, the Make Current button will restore the previous configuration. The orange Restart button will appear and indicate that a service restart is needed before the changes take effect.

Figure 9.27 Ambari confirmation window for a new configuration

There are several important points to remember about the Ambari versioning tool:

■ Every time you change the configuration, a new version is created. Reverting to a previous version creates a new version.

■ You can view or compare a version to other versions without having to change or restart services. (See the buttons in the V11 box in Figure 9.26.)

■ Each service has its own version record.

■ Every time you change the properties, you must restart the service by using the Restart button. When in doubt, restart all services.

Basic Hadoop Administration Procedures

Basic Hadoop YARN Administration

YARN has several built-in administrative features and commands. The main administration command is `yarn rmadmin` (resource manager administration). Enter `yarn rmadmin -help` to learn more about the various options.

Decommissioning YARN Nodes

If a NodeManager host/node needs to be removed from the cluster, it should be decommissioned first. Assuming the node is responding, you can easily decommission it from the Ambari web UI. Simply go to the Hosts view, click on the host, and select Decommission from the pull-down menu next to the NodeManager component. Note that the host may also be acting as a HDFS DataNode. Use the Ambari Hosts view to decommission the HDFS host in a similar fashion.

YARN WebProxy

The Web Application Proxy is a separate proxy server in YARN that addresses security issues with the cluster web interface on ApplicationMasters. By default, the proxy runs as part of the Resource Manager itself, but it can be configured to run in a stand-alone mode by adding the configuration property `yarn.web-proxy.address` to `yarn-site.xml`. (Using Ambari, go to the YARN Configs view, scroll to the bottom, and select Custom yarn-site.xml/Add property.) In stand-alone mode, `yarn.web-proxy.principal` and `yarn.web-proxy.keytab` control the Kerberos principal name and the corresponding keytab, respectively, for use in secure mode. These elements can be added to the `yarn-site.xml` if required.

Using the JobHistoryServer

The removal of the JobTracker and migration of MapReduce from a system to an application-level framework necessitated creation of a place to store MapReduce job history. The JobHistoryServer provides all YARN MapReduce applications with a central location in which to aggregate completed jobs for historical reference and debugging. The settings for the JobHistoryServer can be found in the `mapred-site.xml` file.

Managing YARN Jobs

YARN jobs can be managed using the `yarn application` command. The following options, including `-kill`, `-list`, and `-status`, are available to the administrator with this command. MapReduce jobs can also be controlled with the `mapred job` command.

Setting Container Memory

YARN manages application resource containers over the entire cluster. Controlling the amount of container memory takes place through three important values in the `yarn-site.xml` file:

- `yarn.nodemanager.resource.memory-mb` is the amount of memory the NodeManager can use for containers.
- `scheduler.minimum-allocation-mb` is the smallest container allowed by the ResourceManager. A requested container smaller than this value will result in an allocated container of this size (default 1024MB).
- `yarn.scheduler.maximum-allocation-mb` is the largest container allowed by the ResourceManager (default 8192MB).

Setting Container Cores

You can set the number of cores for containers using the following properties in the `yarn-stie.xml`:

- `yarn.scheduler.minimum-allocation-vcores`: The minimum allocation for every container request at the ResourceManager, in terms of virtual CPU cores. Requests smaller than this allocation will not take effect, and the specified value will be allocated the minimum number of cores. The default is 1 core.
- `yarn.scheduler.maximum-allocation-vcores`: The maximum allocation for every container request at the ResourceManager, in terms of virtual CPU cores. Requests larger than this allocation will not take effect, and the number of cores will be capped at this value. The default is 32.
- `yarn.nodemanager.resource.cpu-vcores`: The number of CPU cores that can be allocated for containers. The default is 8.

Setting MapReduce Properties

As noted throughout this book, MapReduce now runs as a YARN application. Consequently, it may be necessary to adjust some of the `mapred-site.xml` properties as they relate to the map and reduce containers. The following properties are used to set some Java arguments and memory size for both the map and reduce containers:

- `mapred.child.java.opts` provides a larger or smaller heap size for child JVMs of maps (e.g., `--Xmx2048m`).

■ `mapreduce.map.memory.mb` provides a larger or smaller resource limit for maps (default = 1536MB).

■ `mapreduce.reduce.memory.mb` provides a larger heap size for child JVMs of maps (default = 3072MB).

■ `mapreduce.reduce.java.opts` provides a larger or smaller heap size for child reducers.

Basic HDFS Administration

The following section covers some basic administration aspects of HDFS.

The NameNode User Interface

Monitoring HDFS can be done in several ways. One of the more convenient ways to get a quick view of HDFS status is through the NameNode user interface. This web-based tool provides essential information about HDFS and offers the capability to browse the HDFS namespace and logs.

The web-based UI can be started from within Ambari or from a web browser connected to the NameNode. In Ambari, simply select the HDFS service window and click on the Quick Links pull-down menu in the top middle of the page. Select NameNode UI. A new browser tab will open with the UI shown in Figure 10.1. You can also start the UI directly by entering the following command (the command given here assumes the Firefox browser is used, but other browsers should work as well):

$ firefox http://localhost:50070

10. Basic Hadoop Administration Procedures

In This Chapter:

■ Several basic Hadoop YARN administration topics are presented, including decommissioning YARN nodes, managing YARN applications, and important YARN properties.

■ Basic HDFS administration procedures are described, including using the NameNode UI, adding users, performing file system checks, balancing DataNodes, taking HDFS snapshots, and using the HDFS NFSv3 gateway.

■ The Capacity scheduler is discussed.

■ Hadoop version 2 MapReduce compatibility and node capacity are discussed.

Hadoop has two main areas of administration: the YARN resource manager and the HDFS file system. Other application frameworks (e.g., the MapReduce framework) and tools have their own management files. Hadoop configuration is accomplished through the use of XML configuration files. The basic files and their function are as follows:

■ `core-default.xml`: System-wide properties

■ `hdfs-default.xml`: Hadoop Distributed File System properties

- `mapred-default.xml`: Properties for the YARN MapReduce framework
- `yarn-default.xml`: YARN properties

Basic Hadoop YARN Administration

YARN has several built-in administrative features and commands.

Decommissioning YARN Nodes

If a NodeManager host/node needs to be removed from the cluster, it should be decommissioned first. Assuming the node is responding, you can easily decommission it from the Ambari web UI. Simply go to the Hosts view, click on the host, and select Decommission from the pull-down menu next to the NodeManager component. Note that the host may also be acting as a HDFS DataNode. Use the Ambari Hosts view to decommission the HDFS host in a similar fashion.

YARN WebProxy

The Web Application Proxy is a separate proxy server in YARN that addresses security issues with the cluster web interface on ApplicationMasters. By default, the proxy runs as part of the Resource Manager itself, but it can be configured to run in a stand-alone mode by adding the configuration property `yarn.web-proxy.address` to `yarn-site.xml`. (Using Ambari, go to the YARN Configs view, scroll to the bottom, and select Custom yarn-site.xml/Add property.) In stand-alone mode, `yarn.web-proxy.principal` and `yarn.web-proxy.keytab` control the Kerberos principal name and the corresponding keytab, respectively, for use in secure mode. These elements can be added to the `yarn-site.xml` if required.

Using the JobHistoryServer

The removal of the JobTracker and migration of MapReduce from a system to an application-level framework necessitated creation of a place to store MapReduce job history. The JobHistoryServer provides all YARN MapReduce applications with a central location in which to aggregate completed jobs for historical reference and debugging. The settings for the JobHistoryServer can be found in the `mapred-site.xml` file.

Managing YARN Jobs

YARN jobs can be managed using the `yarn application` command. The following options, including `-kill`, `-list`, and `-status`, are available to the administrator with this command. MapReduce jobs can also be controlled with the `mapred job` command.

Setting Container Memory

YARN manages application resource containers over the entire cluster. Controlling the amount of container memory takes place through three important values in the `yarn-site.xml` file:

- `yarn.nodemanager.resource.memory-mb` is the amount of memory the NodeManager can use for containers.

■ `scheduler.minimum-allocation-mb` is the smallest container allowed by the ResourceManager. A requested container smaller than this value will result in an allocated container of this size (default 1024MB).

■ `yarn.scheduler.maximum-allocation-mb` is the largest container allowed by the ResourceManager (default 8192MB).

Setting Container Cores

You can set the number of cores for containers using the following properties in the `yarn-stie.xml`:

■ `yarn.scheduler.minimum-allocation-vcores`: The minimum allocation for every container request at the ResourceManager, in terms of virtual CPU cores. Requests smaller than this allocation will not take effect, and the specified value will be allocated the minimum number of cores. The default is 1 core.

■ `yarn.scheduler.maximum-allocation-vcores`: The maximum allocation for every container request at the ResourceManager, in terms of virtual CPU cores. Requests larger than this allocation will not take effect, and the number of cores will be capped at this value. The default is 32.

■ `yarn.nodemanager.resource.cpu-vcores`: The number of CPU cores that can be allocated for containers. The default is 8.

Setting MapReduce Properties

As noted throughout this book, MapReduce now runs as a YARN application. Consequently, it may be necessary to adjust some of the `mapred-site.xml` properties as they relate to the map and reduce containers. The following properties are used to set some Java arguments and memory size for both the map and reduce containers:

■ `mapred.child.java.opts` provides a larger or smaller heap size for child JVMs of maps (e.g., `--Xmx2048m`).

■ `mapreduce.map.memory.mb` provides a larger or smaller resource limit for maps (default = 1536MB).

■ `mapreduce.reduce.memory.mb` provides a larger heap size for child JVMs of maps (default = 3072MB).

■ `mapreduce.reduce.java.opts` provides a larger or smaller heap size for child reducers.

Basic HDFS Administration

The following section covers some basic administration aspects of HDFS.

The NameNode User Interface

Monitoring HDFS can be done in several ways. One of the more convenient ways to get a quick view of HDFS status is through the NameNode user interface. This web-based tool provides essential information about HDFS and offers the capability to browse the HDFS namespace and logs.

The web-based UI can be started from within Ambari or from a web browser connected to the NameNode. In Ambari, simply select the HDFS service window and click on the Quick Links pull-down menu in the top middle of the page. Select NameNode UI. A new browser tab will open with the UI shown in Figure 10.1. You can also start the UI directly by entering the following command (the command given here assumes the Firefox browser is used, but other browsers should work as well):

$ firefox http://localhost:50070

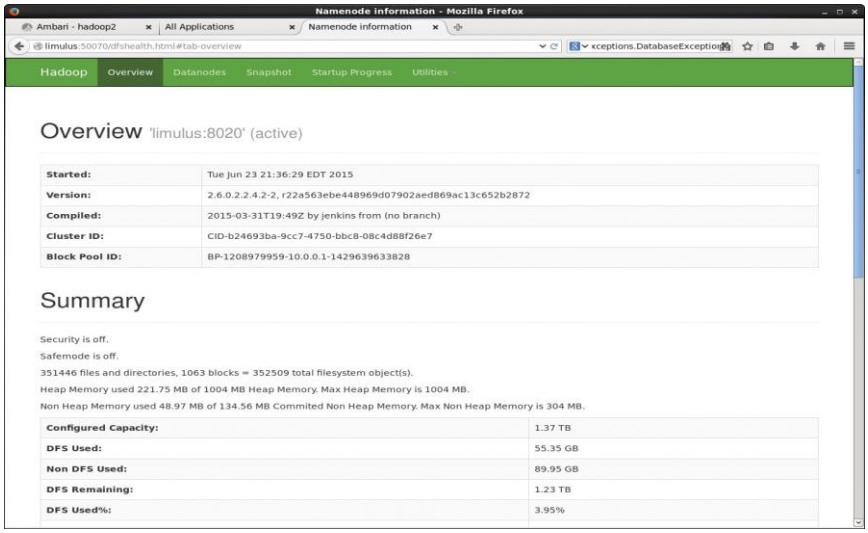

Figure 10.1 Overview page for NameNode user interface

There are five tabs on the UI: Overview, Datanodes, Snapshot, Startup Progress, and Utilities. The Overview page provides much of the essential information that the command-line tools also offer, but in a much easier-to-read format. The Datanodes tab displays node information like that shown in Figure 10.2.

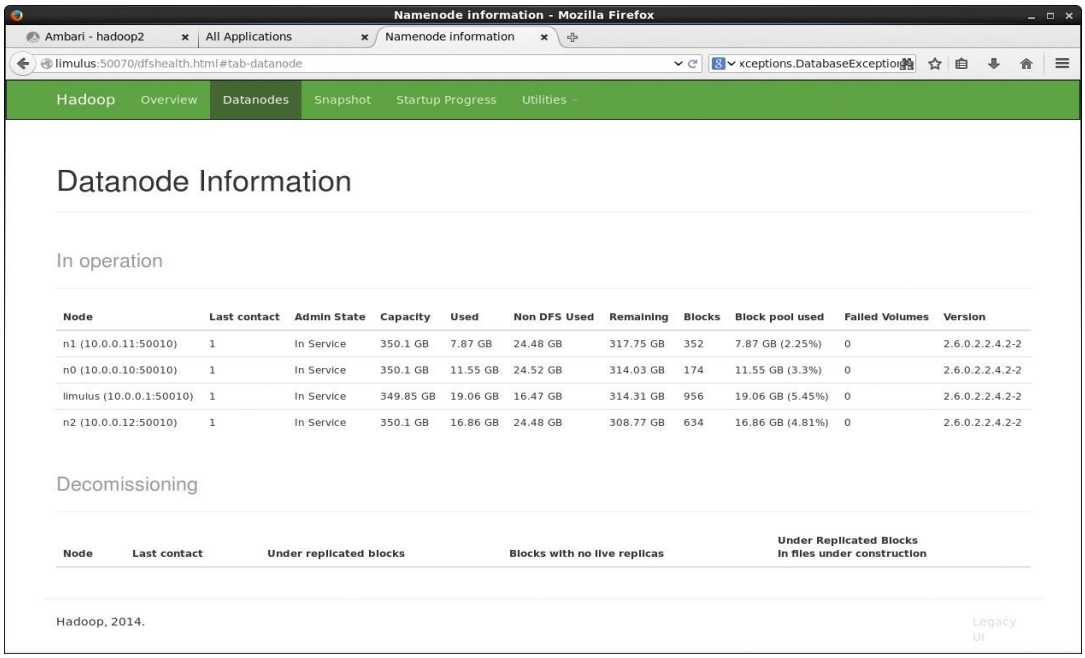

Figure 10.2 NameNode web interface showing status of DataNodes

The Snapshot window (shown later in this chapter in Figure 10.5) lists the "snapshottable" directories and the snapshots. Further information on snapshots can be found in the "HDFS Snapshots" section.

Figure 10.3 provides a NameNode startup progress view. In Figure 10.3, all the phases have been completed, and as indicated in the overview window in Figure 10.1, the system is out of safe mode.

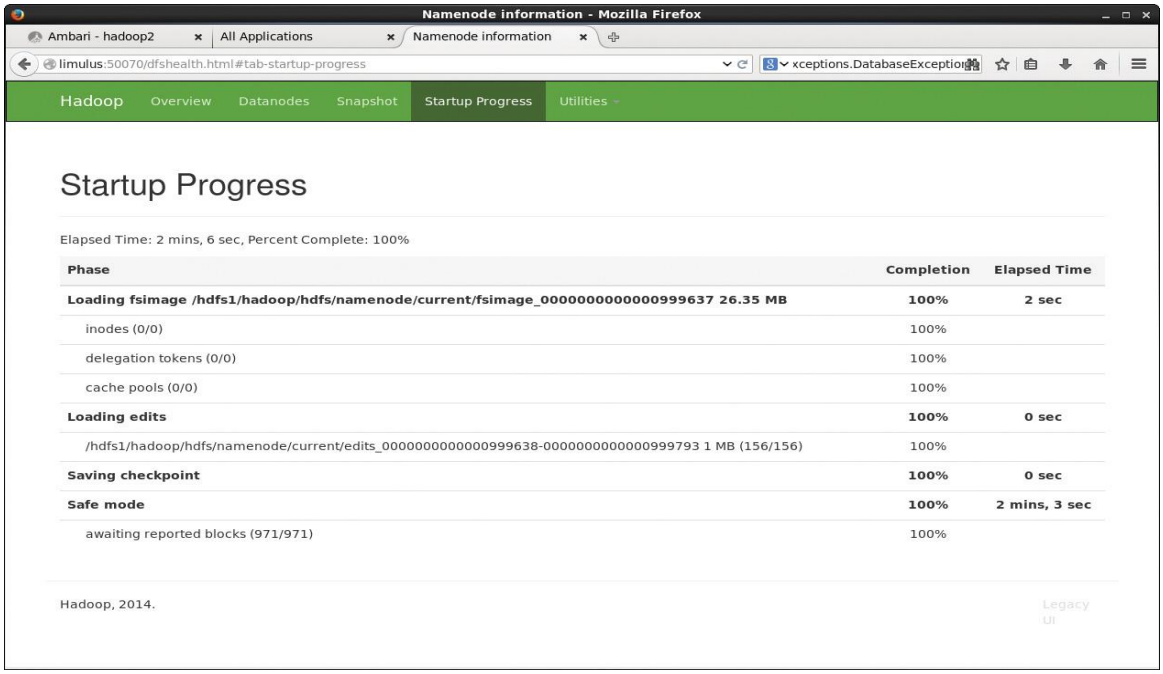

Figure 10.3 NameNode web interface showing startup progress

The Utilities menu offers two options. The first, as shown in Figure 10.4, is a file system browser. From this window, you can easily explore the HDFS namespace. The second option, which is not shown, links to the various NameNode logs.

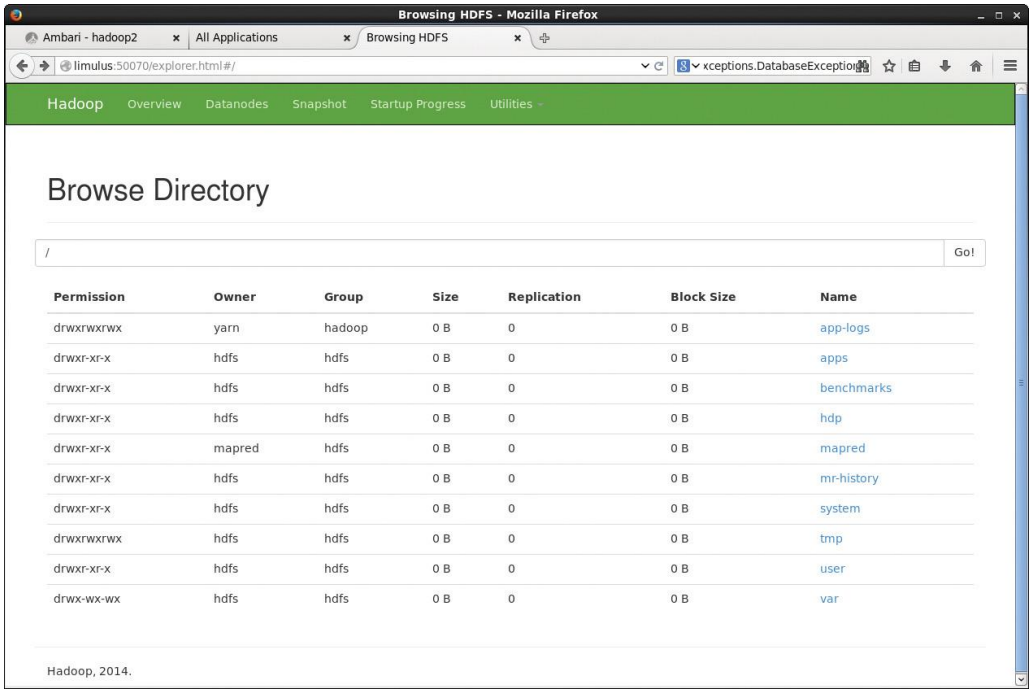

Figure 10.4 NameNode web interface directory browser

Adding Users to HDFS

To quickly create user accounts manually on a Linux-based system, perform the following steps:

1. Add the user to the group for your operating system on the HDFS client system. In most cases, the `groupname` should be that of the HDFS superuser, which is often `hadoop` or `hdfs`.

useradd -G <groupname> <username>

2. Create the `username` directory in HDFS.

hdfs dfs -mkdir /user/<username>

3. Give that account ownership over its directory in HDFS.

hdfs dfs -chown <username>:<groupname> /user/<username>

Perform an FSCK on HDFS

To check the health of HDFS, you can issue the `hdfs fsck <path>` (file system check) command. The entire HDFS namespace can be checked, or a subdirectory can be entered as an argument to the command. The following example checks the entire HDFS namespace.

$ hdfs fsck /

Other options provide more detail, include snapshots and open files, and management of corrupted files.

- `-move` moves corrupted files to `/lost+found`.
- `-delete` deletes corrupted files.
- `-files` prints out files being checked.
- `-openforwrite` prints out files opened for writes during check.
- `-includeSnapshots` includes snapshot data. The path indicates the existence of a snapshottable directory or the presence of snapshottable directories under it.
- `-list-corruptfileblocks` prints out a list of missing blocks and the files to which they belong.
- `-blocks` prints out a block report.
- `-locations` prints out locations for every block.
- `-racks` prints out network topology for data-node locations.

Balancing HDFS

Based on usage patterns and DataNode availability, the number of data blocks across the DataNodes may become unbalanced. To avoid over-utilized DataNodes, the HDFS balancer tool rebalances data blocks across the available DataNodes. Data blocks are moved from over-utilized to under-utilized nodes to within a certain percent threshold. Rebalancing can be done when new DataNodes are added or when a DataNode is removed from service. This step does not create more space in HDFS, but rather improves efficiency.

The HDFS superuser must run the balancer. The simplest way to run the balancer is to enter the following command:

$ hdfs balancer

By default, the balancer will continue to rebalance the nodes until the number of data blocks on all DataNodes are within 10% of each other. The balancer can be stopped, without harming HDFS, at any time by entering a Ctrl-C. Lower or higher thresholds can be set using the `-threshold` argument. For example, giving the following command sets a 5% threshold:

$ hdfs balancer -threshold 5

The lower the threshold, the longer the balancer will run. To ensure the balancer does not swamp the cluster networks, you can set a bandwidth limit before running the balancer, as follows:

$ dfsadmin -setBalancerBandwidth newbandwidth

The `newbandwidth` option is the maximum amount of network bandwidth, in bytes per second, that each DataNode can use during the balancing operation.

Balancing data blocks can also break HBase locality. When HBase regions are moved, some data locality is lost, and the RegionServers will then request the data over the network from remote

DataNode(s). This condition will persist until a major HBase compaction event takes place (which may either occur at regular intervals or be initiated by the administrator).

HDFS Safe Mode

The administrator can place HDFS in Safe Mode by giving the following command:

$ hdfs dfsadmin -safemode enter

Entering the following command turns off Safe Mode:

$ hdfs dfsadmin -safemode leave

HDFS may drop into Safe Mode if a major issue arises within the file system (e.g., a full DataNode). The file system will not leave Safe Mode until the situation is resolved. To check whether HDFS is in Safe Mode, enter the following command:

$ hdfs dfsadmin -safemode get

Decommissioning HDFS Nodes

If you need to remove a DataNode host/node from the cluster, you should decommission it first. Assuming the node is responding, it can be easily decommissioned from the Ambari web UI. Simply go to the Hosts view, click on the host, and selected Decommission from the pull-down menu next to the DataNode component. Note that the host may also be acting as a Yarn NodeManager. Use the Ambari Hosts view to decommission the YARN host in a similar fashion.

SecondaryNameNode

To avoid long NameNode restarts and other issues, the performance of the SecondaryNameNode should be verified. Recall that the SecondaryNameNode takes the previous file system image file (`fsimage*`) and adds the NameNode file system edits to create a new file system image file for the NameNode to use when it restarts. The `hdfs-site.xml` defines a property called `fs.checkpoint.period` (called HDFS Maximum Checkpoint Delay in Ambari). This property provides the time in seconds between the SecondaryNameNode checkpoints.

When a checkpoint occurs, a new `fsimage*` file is created in the directory corresponding to the value of `dfs.namenode.checkpoint.dir` in the `hdfs-site.xml` file. This file is also placed in the NameNode directory corresponding to the `dfs.namenode.name.dir` path designated in the `hdfs-site.xml` file. To test the checkpoint process, a short time period (e.g., 300 seconds) can be used for `fs.checkpoint.period` and HDFS restarted. After five minutes, two identical `fsimage*` files should be present in each of the two previously mentioned directories. If these files are not recent or are missing, consult the NameNode and SecondaryNameNode logs.

Once the SecondaryNameNode process is confirmed to be working correctly, reset the `fs.checkpoint.period` to the previous value and restart HDFS. (Ambari versioning is helpful with this type or procedure.) If the SecondaryNameNode is not running, a checkpoint can be forced by running the following command:

$ hdfs secondarynamenode -checkpoint force

HDFS Snapshots

HDFS snapshots are read-only, point-in-time copies of HDFS. Snapshots can be taken on a subtree of the file system or the entire file system. Some common use-cases for snapshots are data backup, protection against user errors, and disaster recovery.

Snapshots can be taken on any directory once the directory has been set as **snapshottable.** A snapshottable directory is able to accommodate 65,536 simultaneous snapshots. There is no limit on the number of snapshottable directories. Administrators may set any directory to be snapshottable, but nested snapshottable directories are not allowed. For example, a directory cannot be set to snapshottable if one of its ancestors/descendants is a snapshottable directory.

The following example walks through the procedure for creating a snapshot. The first step is to declare a directory as "snapshottable" using the following command:

$ hdfs dfsadmin -allowSnapshot /user/hdfs/war-and-peace-input
Allowing snapshot on /user/hdfs/war-and-peace-input succeeded

Once the directory has been made snapshottable, the snapshot can be taken with the following command. The command requires the directory path and a name for the snapshot—in this case, `wapi-snap-1`.

Configuring an NFSv3 Gateway to HDFS

HDFS supports an NFS version 3 (NFSv3) gateway. This feature enables files to be easily moved between HDFS and client systems. The NFS gateway supports NFSv3 and allows HDFS to be mounted as part of the client's local file system. Currently the NFSv3 gateway supports the following capabilities:

■ Users can browse the HDFS file system through their local file system using an NFSv3 client-compatible operating system.

■ Users can download files from the HDFS file system to their local file system.

■ Users can upload files from their local file system directly to the HDFS file system.

■ Users can stream data directly to HDFS through the mount point. File append is supported, but random write is *not* supported.

In the following example, a simple four-node cluster is used to demonstrate the steps for enabling the NFSv3 gateway. Other potential options, including those related to security, are not addressed in this example. A DataNode is used as the gateway node in this example, and HDFS is mounted on the main (login) cluster node.

Step 1: Set Configuration Files

Several Hadoop configuration files need to be changed. In this example, the Ambari GUI will be used to alter the HDFS configuration files. The following environment is assumed:

■ OS: Linux

■ Platform: RHEL 6.6

■ Hortonworks HDP 2.2 with Hadoop version: 2.6

Several properties need to be added to the `/etc/hadoop/config/core-site.xml` file. Using Ambari, go to the HDFS service window and select the Configs tab. Toward the bottom of the screen, select the Add Property link in the Custom core-site.xml section. Add the following two properties (the item used for the key field in Ambari is the `name` field included in this code):

```
<property>
  <name>hadoop.proxyuser.root.groups</name>
  <value>*</value>
</property>

<property>
 <name>hadoop.proxyuser.root.hosts</name>
 <value>*</value>
</property>
```

The name of the user who will start the Hadoop NFSv3 gateway is placed in the `name` field. In the previous example, `root` is used for this purpose. This setting can be any user who starts the gateway. If, for instance, user `nfsadmin` starts the gateway, then the two names would be `hadoop.proxyuser.nfsadmin.groups` and `hadoop.proxyuser.nfsadmin.hosts`. The `*` value, entered in the preceding lines, opens the gateway to all groups and allows it to run on any host. Access is restricted by entering groups (comma separated) in the group's property. Entering a host name for the host's property can restrict the host running the gateway.

Next, move to the Advanced hdfs-site.xml section and set the following property:

```
<property>
  <name>dfs.namenode.accesstime.precision</name>
  <value>3600000</value>
 </property>
```

This property ensures client mounts with access time updates work properly. (See the `mount` default `atime` option.)

Finally, move to the Custom hdfs-site section, click the Add Property link, and add the following property:

```
property>
 <name>dfs.nfs3.dump.dir</name>
 <value>/tmp/.hdfs-nfs</value>
</property>
```

The NFSv3 `dump` directory is needed because the NFS client often reorders writes. Sequential writes can arrive at the NFS gateway in random order. This directory is used to temporarily save out-of-order writes before writing to HDFS. Make sure the `dump` directory has enough space. For

example, if the application uploads 10 files, each of size 100MB, it is recommended that this directory have 1GB of space to cover a worst-case write reorder for every file.

Once all the changes have been made, click the green Save button and note the changes you made to the Notes box in the Save confirmation dialog. Then restart all of HDFS by clicking the orange Restart button.

Step 2: Start the Gateway

Log into a DataNode and make sure all NFS services are stopped. In this example, DataNode `n0` is used as the gateway.

```
# service rpcbind stop
# service nfs stop
```

Next, start the HDFS gateway by using the `hadoop-daemon` script to start `portmap` and `nfs3` as follows:

```
# /usr/hdp/2.2.4.2-2/hadoop/sbin/hadoop-daemon.sh start portmap
# /usr/hdp/2.2.4.2-2/hadoop/sbin/hadoop-daemon.sh start nfs3
```

The `portmap` daemon will write its log to

/var/log/hadoop/root/hadoop-root-portmap-n0.log

The `nfs3` daemon will write its log to

/var/log/hadoop/root/hadoop-root-nfs3-n0.log

To confirm the gateway is working, issue the following command. The output should look like the following:

```
# rpcinfo -p n0
```

Finally, make sure the mount is available by issuing the following command:

```
# showmount -e n0
Export list for n0:
/ *
```

If the `rpcinfo` or `showmount` command does not work correctly, check the previously mentioned log files for problems.

Step 3: Mount HDFS

The final step is to mount HDFS on a client node. In this example, the main login node is used. To mount the HDFS files, exit from the gateway node (in this case node `n0`) and create the following directory:

```
# mkdir /mnt/hdfs
```

The mount command is as follows. Note that the name of the gateway node will be different on other clusters, and an IP address can be used instead of the node name.

```
# mount -t nfs -o vers=3,proto=tcp,nolock n0:/  /mnt/hdfs/
```

Once the file system is mounted, the files will be visible to the client users. The following command will list the mounted file system:

```
# ls /mnt/hdfs

app-logs apps benchmarks hdp mapred mr-history system tmp user var
```

The gateway in the current Hadoop release uses `AUTH_UNIX`-style authentication and requires that the login user name on the client match the user name that NFS passes to HDFS. For example, if the NFS client is user `admin`, the NFS gateway will access HDFS as user `admin` and existing HDFS permissions will prevail.

The system administrator must ensure that the user on the NFS client machine has the same user name and user ID as that on the NFS gateway machine. This is usually not a problem if you use the same user management system, such as LDAP/NIS, to create and deploy users to cluster nodes.

Capacity Scheduler Background

The Capacity scheduler is the default scheduler for YARN that enables multiple groups to securely share a large Hadoop cluster. Developed by the original Hadoop team at Yahoo!, the Capacity scheduler has successfully run many of the largest Hadoop clusters.

To use the Capacity scheduler, one or more queues are configured with a predetermined fraction of the total slot (or processor) capacity. This assignment guarantees a minimum amount of resources for each queue. Administrators can configure soft limits and optional hard limits on the capacity allocated to each queue. Each queue has strict ACLs (Access Control Lists) that control which users can submit applications to individual queues. Also, safeguards are in place to ensure that users cannot view or modify applications from other users.

The Capacity scheduler permits sharing a cluster while giving each user or group certain minimum capacity guarantees. These minimum amounts are not given away in the absence of demand (i.e., a group is always guaranteed a minimum number of resources is available). Excess slots are given to the most starved queues, based on the number of running tasks divided by the queue capacity. Thus, the fullest queues as defined by their initial minimum capacity guarantee get the most needed resources. Idle capacity can be assigned and provides elasticity for the users in a cost-effective manner.

Administrators can change queue definitions and properties, such as capacity and ACLs, at run time without disrupting users. They can also add more queues at run time, but cannot delete queues at run time. In addition, administrators can stop queues at run time to ensure that while existing applications run to completion, no new applications can be submitted.

The Capacity scheduler currently supports memory-intensive applications, where an application can optionally specify higher memory resource requirements than the default. Using information from the NodeManagers, the Capacity scheduler can then place containers on the best-suited nodes.

The Capacity scheduler works best when the workloads are well known, which helps in assigning the minimum capacity. For this scheduler to work most effectively, each queue should be assigned a minimal capacity that is less than the maximal expected workload. Within each queue, multiple jobs are scheduled using hierarchical (first in, first out) FIFO queues similar to the approach used with the stand-alone FIFO scheduler. If there are no queues configured, all jobs are placed in the default queue.

The ResourceManager UI provides a graphical representation of the scheduler queues and their utilization. Figure 10.6 shows two jobs running on a four-node cluster. To select the scheduler view, click the Scheduler option at the bottom of the left-side vertical menu.

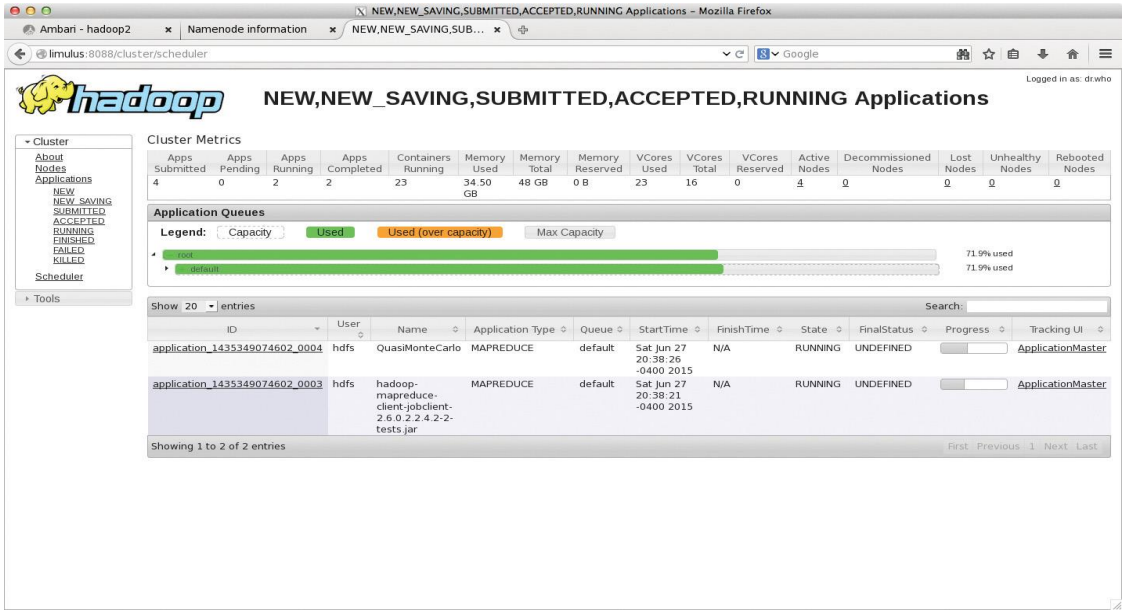

Figure 10.6 Apache YARN ResourceManager web interface showing Capacity scheduler information

In addition to the Capacity scheduler, Hadoop YARN offers a Fair scheduler. More information can be found on the Hadoop website.

Hadoop Version 2 MapReduce Compatibility

Hadoop version 1 is essentially a monolithic MapReduce engine. Moving this technology to YARN as a separate application framework was a complex task because MapReduce requires many important processing features, including data locality, fault tolerance, and application priorities.

To provide data locality, the MapReduce ApplicationMaster is required to locate blocks for processing and then request containers on these blocks. To implement fault tolerance, the capability to handle failed map or reduce tasks and request them again on other nodes was needed. Fault tolerance moved hand-in-hand with the complex intra-application priorities.

The logic to handle complex intra-application priorities for map and reduce tasks had to be built into the ApplicationMaster. There was no need to start idle reducers before mappers finished processing enough data. Reducers were now under control of the ApplicationMaster and were not

fixed, as they had been in Hadoop version 1. This design actually made Hadoop version 2 much more efficient and increased cluster throughput.

The following sections provide basic background on how the MapReduce framework operates under YARN. The new Hadoop version 2 MapReduce (often referred to as MRv2) was designed to provide as much backward compatibility with Hadoop version 1 MapReduce (MRv1) as possible. As with the other topics in this chapter, the following discussion provides an overview of some of the important considerations when administering the Hadoop version 2 MapReduce framework.

Enabling ApplicationMaster Restarts

Should an error occur in a MapReduce job, the ApplicationMaster can be automatically restarted by YARN. To enable ApplicationMaster restarts, set the following properties:

- Inside `yarn-site.xml`, you can tune the property `yarn.resourcemanager.am.max-retries`. The default is 2.

- Inside `mapred-site.xml`, you can more directly tune how many times a MapReduce ApplicationMaster should restart with the property `mapreduce.am.max-attempts`. The default is 2.

Calculating the Capacity of a Node

YARN has removed the hard-partitioned mapper and reducer slots of Hadoop version 1. To determine the MapReduce capacity of a cluster node, new capacity calculations are required. Estimates as to the number of mapper and reducer tasks that can efficiently run on a node help determine the amount of computing resources made available to Hadoop users. There are eight important parameters for calculating a node's MapReduce capacity; they are found in the `mapred-site.xml` and `yarn-site.xml` files.

- `mapred-site.xml`

  - `mapreduce.map.memory.mb`
  `mapreduce.reduce.memory.mb`

  The hard limit enforced by Hadoop on the mapper or reducer task.

  - `mapreduce.map.java.opts`
  `mapreduce.reduce.java.opts`

  The heap size of the `jvm -Xmx` for the mapper or reducer task. Remember to leave room for the JVM Perm Gen and Native Libs used. This value should always be smaller than `mapreduce.[map|reduce].memory.mb`.

- `yarn-site.xml`

  - `yarn.scheduler.minimum-allocation-mb`

  The smallest container YARN will allow.

  - `yarn.scheduler.maximum-allocation-mb`

The largest container YARN will allow.

- `yarn.nodemanager.resource.memory-mb`

The amount of physical memory (RAM) on the compute node for containers. It is important that this value is not equal to the total RAM on the node, as other Hadoop services also require RAM.

- `yarn.nodemanager.vmem-pmem-ratio`

The amount of virtual memory each container is allowed. It is calculated with the following formula:

containerMemoryRequest*vmem-pmem-ratio.

As an example, consider a configuration with the settings in Table 10.1. Using these settings, we have given each map and reduce task a generous 512MB of overhead for the container, as seen with the difference between the `mapreduce.[map|reduce].memory.mb` and the `mapreduce.[map|reduce].java.opts`.

Property	Value
mapreduce.map.memory.mb	1536
mapreduce.reduce.memory.mb	2560
mapreduce.map.java.opts	-Xmx1024m
mapreduce.reduce.java.opts	-Xmx2048m
yarn.scheduler.minimum-allocation-mb	512
yarn.scheduler.maximum-allocation-mb	4096
yarn.nodemanager.resource.memory-mb	36864
yarn.nodemanager.vmem-pmem-ratio	2.1

Table 10.1 **Example YARN MapReduce Settings**

Next, we have configured YARN to allow a container no smaller than 512MB and no larger than 4GB. Assuming the compute nodes have 36GB of RAM available for containers, and with a virtual memory ratio of 2.1 (the default value), each map can have as much as 3225.6MB of RAM and a reducer can have 5376MB of virtual RAM. Thus our compute node configured for 36GB of container space can support up to 24 maps or 14 reducers, or any combination of mappers and reducers allowed by the available resources on the node.

Running Hadoop Version 1 Applications

To ease the transition from Hadoop version 1 to version 2 with YARN, a major goal of YARN and the MapReduce framework implementation on top of YARN is to ensure that existing MapReduce applications that were programmed and compiled against previous MapReduce APIs (MRv1 applications) can continue to run with little work on top of YARN (MRv2 applications).

Binary Compatibility of org.apache.hadoop.mapred APIs

For the vast majority of users who use the `org.apache.hadoop.mapred` APIs, MapReduce on YARN ensures full binary compatibility. These existing applications can run on YARN directly without recompilation. You can use jar files of your existing application that code against MapReduce APIs and use `bin/hadoop` to submit them directly to YARN.

Source Compatibility of org.apache.hadoop.mapreduce APIs

Unfortunately, it has proved difficult to ensure full binary compatibility of applications that were originally compiled against MRv1 `org.apache.hadoop.mapreduce` APIs. These APIs have gone through lots of changes. For example, many of the classes stopped being abstract classes and changed to interfaces. The YARN community eventually reached a compromise on this issue, supporting source compatibility only for `org.apache.hadoop.mapreduce` APIs. Existing applications using MapReduce APIs are source compatible and can run on YARN either with no changes, with simple recompilation against MRv2 jar files that are shipped with Hadoop version 2, or with minor updates.

Compatibility of Command-Line Scripts

Most of the command-line scripts from Hadoop 1.x should work without any tweaking. The only exception is `mradmin`, whose functionality was removed from MRv2 because the JobTracker and TaskTracker no longer exist. The `mradmin` functionality has been replaced with `rmadmin`. The suggested method to invoke `rmadmin` is through the command line, even though you can directly invoke the APIs. In YARN, when `mradmin` commands are executed, warning messages will appear, reminding users to use YARN commands (i.e., `rmadmin` commands). Conversely, if the user's applications programmatically invoke `mradmin`, those applications will break when running on top of YARN. There is no support for either binary or source compatibility under YARN.

Running Apache Pig Scripts on YARN

Pig is one of the two major data process applications in the Hadoop ecosystem, with the other being Hive. Thanks to the significant efforts made by the Pig community, Pig scripts of existing users do not need any modifications. Pig on YARN in Hadoop 0.23 has been supported since version 0.10.0, and Pig working with Hadoop 2.x has been supported since version 0.10.1.

Existing Pig scripts that work with Pig 0.10.1 and beyond will work just fine on top of YARN. In contrast, versions earlier than Pig 0.10.x may not run directly on YARN due to some of the incompatible MapReduce APIs and configuration.

Running Apache Hive Queries on YARN

Hive queries of existing users do not need any changes to work on top of YARN, starting with Hive 0.10.0, thanks to the work done by Hive community. Support for Hive to work on YARN in the Hadoop 0.23 and 2.x releases has been in place since version 0.10.0. Queries that work in Hive 0.10.0 and beyond will work without changes on top of YARN. However, as with Pig, earlier versions of Hive may not run directly on YARN, as those Hive releases do not support Hadoop 0.23 and 2.x.

Running Apache Oozie Workflows on YARN

Like the Pig and Hive communities, the Apache Oozie community worked to ensure existing Oozie workflows would run in a completely backward-compatible manner on Hadoop version 2. Support for Hadoop 0.23 and 2.x is available starting with Oozie release 3.2.0. Existing Oozie workflows can start taking advantage of YARN in versions 0.23 and 2.x with Oozie 3.2.0 and above.