Hadoop MapReduce Cookbook

Recipes for analyzing large and complex datasets with Hadoop MapReduce

Srinath Perera
Thilina Gunarathne

Hadoop MapReduce Cookbook

Recipes for analyzing large and complex datasets with Hadoop MapReduce

Srinath Perera

Thilina Gunarathne

Hadoop MapReduce Cookbook

Credits

About the Authors

Srinath Perera is a Senior Software Architect at WSO2 Inc., where he overlooks the overall WSO2 platform architecture with the CTO. He also serves as a Research Scientist at Lanka Software Foundation and teaches as a visiting faculty at Department of Computer Science and Engineering, University of Moratuwa. He is a co-founder of Apache Axis2 open source project, and he has been involved with the Apache Web Service project since 2002, and is a member of Apache Software foundation and Apache Web Service project PMC. Srinath is also a committer of Apache open source projects Axis, Axis2, and Geronimo.

He received his Ph.D. and M.Sc. in Computer Sciences from Indiana University, Bloomington, USA and received his Bachelor of Science in Computer Science and Engineering from University of Moratuwa, Sri Lanka.

Srinath has authored many technical and peer reviewed research articles, and more detail can be found from his website. He is also a frequent speaker at technical venues.

He has worked with large-scale distributed systems for a long time. He closely works with Big Data technologies, such as Hadoop and Cassandra daily. He also teaches a parallel programming graduate class at University of Moratuwa, which is primarily based on Hadoop.

Thilina Gunarathne is a Ph.D. candidate at the School of Informatics and Computing of Indiana University. He has extensive experience in using Apache Hadoop and related technologies for large-scale data intensive computations. His current work focuses on developing technologies to perform scalable and efficient large-scale data intensive computations on cloud environments.

Thilina has published many articles and peer reviewed research papers in the areas of distributed and parallel computing, including several papers on extending MapReduce model to perform efficient data mining and data analytics computations on clouds. Thilina is a regular presenter in both academic as well as industry settings.

Thilina has contributed to several open source projects at Apache Software Foundation as a committer and a PMC member since 2005. Before starting the graduate studies, Thilina worked as a Senior Software Engineer at WSO2 Inc., focusing on open source middleware development. Thilina received his B.Sc. in Computer Science and Engineering from University of Moratuwa, Sri Lanka, in 2006 and received his M.Sc. in Computer Science from Indiana University, Bloomington, in 2009. Thilina expects to receive his doctorate in the field of distributed and parallel computing in 2013.

About the Reviewers

Masatake Iwasaki is Software Engineer at NTT DATA Corporation. He provides technical consultation for Open Source software such as Hadoop, HBase, and PostgreSQL.

Shinichi Yamashita is a Chief Engineer at OSS professional service unit in NTT DATA Corporation in Japan. He has more than seven years' experience in software and middleware (Apache, Tomcat, PostgreSQL, and Hadoop eco system) engineering. NTT DATA is your Innovation Partner anywhere around the world. It provides professional services from consulting, and system development to business IT outsourcing. In Japan, he has authored some books on Hadoop.

I thank my co-workers.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ▸ Fully searchable across every book published by Packt
- ▸ Copy and paste, print and bookmark content
- ▸ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Hadoop MapReduce Cookbook helps readers learn to process large and complex datasets. The book starts in a simple manner, but still provides in-depth knowledge of Hadoop. It is a simple one-stop guide on how to get things done. It has 90 recipes, presented in a simple and straightforward manner, with step-by-step instructions and real world examples.

This product includes software developed at The Apache Software Foundation (`http://www.apache.org/`).

What this book covers

Chapter 1, *Getting Hadoop Up and Running in a Cluster*, explains how to install and run Hadoop both as a single node as well as a cluster.

Chapter 2, *Advanced HDFS*, introduces a set of advanced HDFS operations that would be useful when performing large-scale data processing with Hadoop MapReduce as well as with non-MapReduce use cases.

Chapter 3, *Advanced Hadoop MapReduce Administration*, explains how to change configurations and security of a Hadoop installation and how to debug.

Chapter 4, *Developing Complex Hadoop MapReduce Applications*, introduces you to several advanced Hadoop MapReduce features that will help you to develop highly customized, efficient MapReduce applications.

Chapter 5, *Hadoop Ecosystem*, introduces the other projects related to Hadoop such HBase, Hive, and Pig.

Chapter 6, *Analytics*, explains how to calculate basic analytics using Hadoop.

Chapter 7, *Searching and Indexing*, introduces you to several tools and techniques that you can use with Apache Hadoop to perform large-scale searching and indexing.

Chapter 8, *Classifications, Recommendations, and Finding Relationships*, explains how to implement complex algorithms such as classifications, recommendations, and finding relationships using Hadoop.

Chapter 9, *Mass Text Data Processing*, explains how to use Hadoop and Mahout to process large text datasets, and how to perform data preprocessing and loading operations using Hadoop.

Chapter 10, *Cloud Deployments: Using Hadoop on Clouds*, explains how to use Amazon Elastic MapReduce (EMR) and Apache Whirr to deploy and execute Hadoop MapReduce, Pig, Hive, and HBase computations on cloud infrastructures.

What you need for this book

All you need is access to a computer running Linux Operating system, and Internet. Also, Java knowledge is required.

Who this book is for

For big data enthusiasts and would be Hadoop programmers. The books for Java programmers who either have not worked with Hadoop at all, or who knows Hadoop and MapReduce but want to try out things and get into details. It is also a one-stop reference for most of your Hadoop tasks.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "From this point onward, we shall call the unpacked Hadoop directory HADOOP_HOME."

A block of code is set as follows:

```
public void map(Object key, Text value, Context context)
  throws IOException, InterruptedException
  {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens())
    {
      word.set(itr.nextToken());
      context.write(word, new IntWritable(1));
    }
  }
```

Any command-line input or output is written as follows:

```
>tar -zxvf hadoop-1.x.x.tar.gz
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Create a S3 bucket to upload the input data by clicking on **Create Bucket**".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Hadoop Up and Running in a Cluster

In this chapter, we will cover:

- ▶ Setting up Hadoop on your machine
- ▶ Writing the WordCount MapReduce sample, bundling it, and running it using standalone Hadoop
- ▶ Adding the combiner step to the WordCount MapReduce program
- ▶ Setting up HDFS
- ▶ Using the HDFS monitoring UI
- ▶ HDFS basic command-line file operations
- ▶ Setting Hadoop in a distributed cluster environment
- ▶ Running the WordCount program in a distributed cluster environment
- ▶ Using the MapReduce monitoring UI

Introduction

For many years, users who want to store and analyze data would store the data in a database and process it via SQL queries. The Web has changed most of the assumptions of this era. On the Web, the data is unstructured and large, and the databases can neither capture the data into a schema nor scale it to store and process it.

Google was one of the first organizations to face the problem, where they wanted to download the whole of the Internet and index it to support search queries. They built a framework for large-scale data processing borrowing from the "map" and "reduce" functions of the functional programming paradigm. They called the paradigm **MapReduce**.

Hadoop is the most widely known and widely used implementation of the MapReduce paradigm. This chapter introduces Hadoop, describes how to install Hadoop, and shows you how to run your first MapReduce job with Hadoop.

Hadoop installation consists of four types of nodes—a **NameNode**, **DataNodes**, a **JobTracker**, and **TaskTracker** HDFS nodes (NameNode and DataNodes) provide a distributed filesystem where the JobTracker manages the jobs and TaskTrackers run tasks that perform parts of the job. Users submit MapReduce jobs to the JobTracker, which runs each of the Map and Reduce parts of the initial job in TaskTrackers, collects results, and finally emits the results.

Hadoop provides three installation choices:

  ▸ **Local mode**: This is an unzip and run mode to get you started right away where all parts of Hadoop run within the same JVM
  ▸ **Pseudo distributed mode**: This mode will be run on different parts of Hadoop as different Java processors, but within a single machine
  ▸ **Distributed mode**: This is the real setup that spans multiple machines

We will discuss the local mode in the first three recipes, and Pseudo distributed and distributed modes in the last three recipes.

Setting up Hadoop on your machine

This recipe describes how to run Hadoop in the local mode.

Getting ready

Download and install Java 1.6 or higher version from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

How to do it...

Now let us do the Hadoop installation:

  1. Download the most recent Hadoop 1.0 branch distribution from `http://hadoop.apache.org/`.

  2. Unzip the Hadoop distribution using the following command. You will have to change the `x.x` in the filename with the actual release you have downloaded. If you are using Windows, you should use your favorite archive program such as WinZip or WinRAR for extracting the distribution. From this point onward, we shall call the unpacked Hadoop directory `HADOOP_HOME`.

     ```
     >tar -zxvf hadoop-1.x.x.tar.gz
     ```

3. You can use Hadoop local mode after unzipping the distribution. Your installation is done. Now, you can run Hadoop jobs through `bin/hadoop` command, and we will elaborate that further in the next recipe.

How it works...

Hadoop local mode does not start any servers but does all the work within the same JVM. When you submit a job to Hadoop in the local mode, that job starts a JVM to run the job, and that JVM carries out the job. The output and the behavior of the job is the same as a distributed Hadoop job, except for the fact that the job can only use the current node for running tasks. In the next recipe, we will discover how to run a MapReduce program using the unzipped Hadoop distribution.

> **Downloading the example code**
>
> You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Writing a WordCount MapReduce sample, bundling it, and running it using standalone Hadoop

This recipe explains how to write a simple MapReduce program and how to execute it.
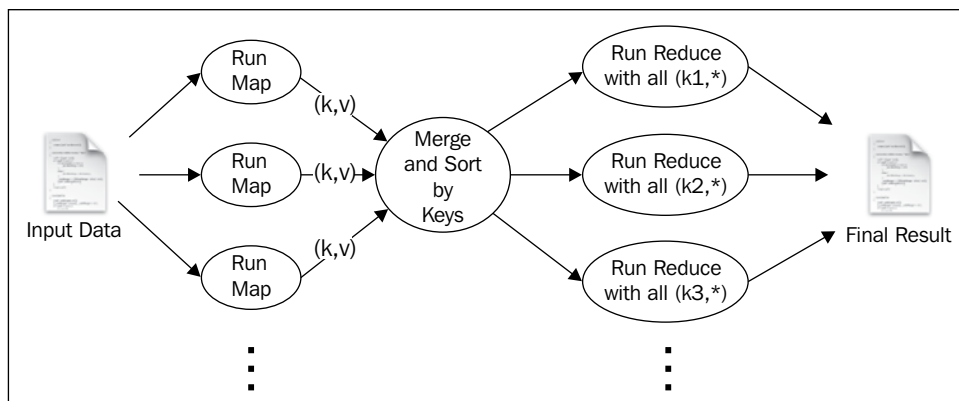
To run a MapReduce job, users should furnish a `map` function, a `reduce` function, input data, and an output data location. When executed, Hadoop carries out the following steps:

1. Hadoop breaks the input data into multiple data items by new lines and runs the `map` function once for each data item, giving the item as the input for the function. When executed, the `map` function outputs one or more key-value pairs.

2. Hadoop collects all the key-value pairs generated from the `map` function, sorts them by the key, and groups together the values with the same key.

3. For each distinct key, Hadoop runs the reduce function once while passing the key and list of values for that key as input.

4. The `reduce` function may output one or more key-value pairs, and Hadoop writes them to a file as the final result.

Getting ready

From the source code available with this book, select the source code for the first chapter, `chapter1_src.zip`. Then, set it up with your favorite **Java Integrated Development Environment** (**IDE**); for example, Eclipse. You need to add the `hadoop-core` JAR file in `HADOOP_HOME` and all other JAR files in the `HADOOP_HOME/lib` directory to the classpath of the IDE.

Download and install Apache Ant from `http://ant.apache.org/`.

How to do it...

Now let us write our first Hadoop MapReduce program.

1. The WordCount sample uses MapReduce to count the number of word occurrences within a set of input documents. Locate the sample code from `src/chapter1/Wordcount.java`. The code has three parts—mapper, reducer, and the main program.

2. The mapper extends from the `org.apache.hadoop.mapreduce.Mapper` interface. When Hadoop runs, it receives each new line in the input files as an input to the mapper. The `map` function breaks each line into substrings using whitespace characters such as the separator, and for each token (word) emits `(word,1)` as the output.

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

```
        {
            word.set(itr.nextToken());
            context.write(word, new IntWritable(1));
        }
    }
```

3. The `reduce` function receives all the values that have the same key as the input, and it outputs the key and the number of occurrences of the key as the output.

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

4. The `main` program puts the configuration together and submits the job to Hadoop.

```
Configuration conf = new Configuration();
String[] otherArgs = new GenericOptionsParser(conf, args).
getRemainingArgs();
if (otherArgs.length != 2) {
System.err.println("Usage: wordcount <in><out>");
System.exit(2);
}
Job job = new Job(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
//Uncomment this to
//job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

5. You can compile the sample by running the following command, which uses Apache Ant, from the root directory of the sample code:

```
>ant build
```

   If you have not done this already, you should install Apache Ant by following the instructions given at `http://ant.apache.org/manual/install.html`. Alternatively, you can use the compiled JAR file included with the source code.

6. Change the directory to `HADOOP_HOME`, and copy the `hadoop-cookbook-chapter1.jar` file to the `HADOOP_HOME` directory. To be used as the input, create a directory called `input` under `HADOOP_HOME` and copy the `README.txt` file to the directory. Alternatively, you can copy any text file to the `input` directory.

7. Run the sample using the following command. Here, `chapter1.WordCount` is the name of the `main` class we need to run. When you have run the command, you will see the following terminal output:

```
>bin/hadoop jar hadoop-cookbook-chapter1.jar chapter1.WordCount
input output

12/04/11 08:12:44 INFO input.FileInputFormat: Total input paths to
process : 16

12/04/11 08:12:45 INFO mapred.JobClient: Running job: job_
local_0001

12/04/11 08:12:45 INFO mapred.Task: Task:attempt_
local_0001_m_000000_0 is done. And is in the process of commiting

...........

.....

12/04/11 08:13:37 INFO mapred.JobClient: Job complete: job_
local_0001

.....
```

8. The output directory will have a file named like `part-r-XXXXX`, which will have the count of each word in the document. Congratulations! You have successfully run your first MapReduce program.

How it works...

In the preceding sample, MapReduce worked in the local mode without starting any servers and using the local filesystem as the storage system for inputs, outputs, and working data. The following diagram shows what happened in the WordCount program under the covers:
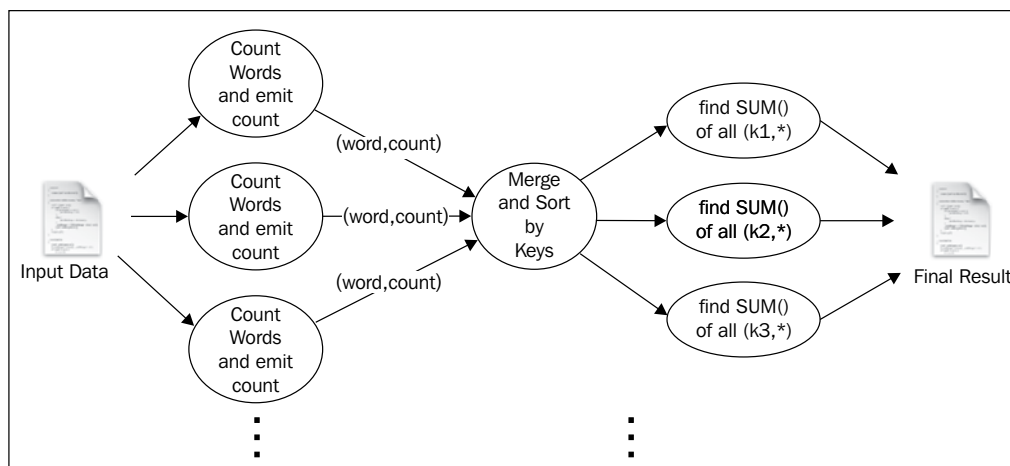
The workflow is as follows:

1. Hadoop reads the input, breaks it by new line characters as the separator and then runs the map function passing each line as an argument.

2. The map function tokenizes the line, and for each token (word), emits a key value pair `(word, 1)`.

3. Hadoop collects all the `(word,1)` pairs, sorts them by the word, groups all the values emitted against each unique key, and invokes the reduce once for each unique key passing the key and values for that key as an argument.

4. The reduce function counts the number of occurrences of each word using the values and emits it as a key-value pair.

5. Hadoop writes the final output to the output directory.

There's more...

As an optional step, copy the `input` directory to the top level of the IDE-based project (Eclipse project) that you created for samples. Now you can run the `WordCount` class directly from your IDE passing `input output` as arguments. This will run the sample the same as before. Running MapReduce jobs from IDE in this manner is very useful for debugging your MapReduce jobs.

Although you ran the sample with Hadoop installed in your local machine, you can run it using distributed Hadoop cluster setup with a HDFS-distributed filesystem. The recipes of this chapter, *Setting up HDFS* and *Setting Hadoop in a distributed cluster environment* will discuss how to run this sample in a distributed setup.

Adding the combiner step to the WordCount MapReduce program

After running the `map` function, if there are many key-value pairs with the same key, Hadoop has to move all those values to the `reduce` function. This can incur a significant overhead. To optimize such scenarios, Hadoop supports a special function called **combiner**. If provided, Hadoop will call the combiner from the same node as the map node before invoking the reducer and after running the mapper. This can significantly reduce the amount of data transferred to the reduce step.

This recipe explains how to use the combiner with the WordCount sample introduced in the previous recipe.

How to do it...

Now let us run the MapReduce job adding the combiner:

1. Combiner must have the same interface as the `reduce` function. For the WordCount sample, we will reuse the `reduce` function as the combiner.

2. To ask the MapReduce job to use the combiner, let us uncomment the line `//job.setCombinerClass(IntSumReducer.class);` in the sample and recompile the code.

3. Copy the `hadoop-cookbook-chapter1.jar` file to the `HADOOP_HOME` directory and run the WordCount as done in the earlier recipe. Make sure to delete the old output directory before running the job.

4. Final results will be available from the `output` directory.

How it works...

To activate a combiner, users should provide a mapper, a reducer, and a combiner as input to the MapReduce job. In that setting, Hadoop executes the combiner in the same node as the mapper function just after running the mapper. With this method, the combiner can pre-process the data generated by the mapper before sending it to the reducer, thus reducing the amount of data that is getting transferred.

For example, with the WordCount, combiner receives `(word,1)` pairs from the map step as input and outputs a single `(word, N)` pair. For example, if an input document has 10,000 occurrences of word "the", the mapper will generate 10,000 `(the,1)` pairs, while the combiner will generate one `(the,10,000)` thus reducing the amount of data transferred to the reduce task.

However, the combiner only works with commutative and associative functions. For example, the same idea does not work when calculating mean. As mean is not communicative and associative, a combiner in that case will yield a wrong result.

There's more...

Although in the sample we reused the `reduce` function implementation as the combiner function, you may write your own combiner function just like we did for the `map` and `reduce` functions in the previous recipe. However, the signature of the combiner function must be identical to that of the `reduce` function.

In a local setup, using a combiner will not yield significant gains. However, in the distributed setups as described in *Setting Hadoop in a distributed cluster environment* recipe, combiner can give significant gains.

Setting up HDFS

HDFS is the distributed filesystem that is available with Hadoop. MapReduce tasks use HDFS to read and write data. HDFS deployment includes a single NameNode and multiple DataNodes.
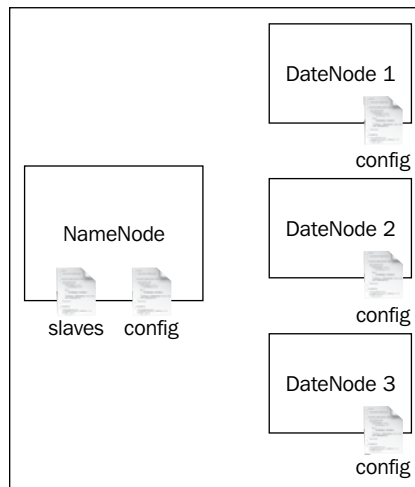
For the HDFS setup, we need to configure NameNodes and DataNodes, and then specify the DataNodes in the `slaves` file. When we start the NameNode, startup script will start the DataNodes.

Getting ready

You may follow this recipe either using a single machine or multiple machines. If you are using multiple machines, you should choose one machine as the master node where you will run the HDFS NameNode. If you are using a single machine, use it as both the NameNode as well as the DataNode.

1. Install Java in all machines that will be used to set up the HDFS cluster.

2. If you are using Windows machines, install Cygwin and SSH server in each machine. The link `http://pigtail.net/LRP/printsrv/cygwin-sshd.html` provides step-by-step instructions.

How to do it...

Now let us set up HDFS in the distributed mode.

1. Enable SSH from master nodes to slave nodes. Check that you can login to the localhost and all other nodes using SSH without a passphrase by running one of the following commands:

   □ `>ssh localhost`

   □ `>ssh IPaddress`

2. If the above command returns an error or asks for a password, create SSH keys by executing the following command:

   `>ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa`

   Move the `~/.ssh/id_dsa.pub` file to the all the nodes in the cluster. Then add the SSH keys to the `~/.ssh/authorized_keys` file in each node by running the following command (if the `authorized_keys` file does not exist, run the following command. Else, skip to the `cat` command):

   `>touch ~/.ssh/authorized_keys && chmod 600 ~/.ssh/authorized_keys`

   Now with permissions set, add your key to the `~/.ssh/authorized_keys` file.

   `>cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys`

   Then you can log in with the following command:

   `>ssh localhost`

   This command creates an SSH key pair in the `.ssh/`directory of the home directory, and registers the generated public key with SSH as a trusted key.

3. In each machine, create a directory for storing HDFS data. Let's call that directory `HADOOP_DATA_DIR`. Now let us create two sub directories, `HADOOP_DATA_DIR/data` and `HADOOP_DATA_DIR/name`. Change the directory permissions to `755` by running the following command for each directory:

```
>chmod 755 <name of dir>
```

4. In the NameNode, change directory to the unzipped `HADOOP_HOME` directory. Then place the IP address of all slave nodes in the `HADOOP_HOME/conf/slaves` file, each on a separate line. When we start the NameNode, it will use the `slaves` file to start the DataNodes.

5. In all machines, edit the `HADOOP_HOME/conf/hadoop-env.sh` file by uncommenting the `JAVA_HOME` line and pointing it to your local Java installation. For example, if Java is in `/opt/jdk1.6`, change the `JAVA_HOME` line to `export JAVA_HOME=/opt/jdk1.6`.

6. Inside each node's `HADOOP_HOME/conf` directory, add the following code to the `core-site.xml` and `hdfs-site.xml` files. Before adding the configurations, replace the `MASTER_NODE` strings with the IP address of the master node and `HADOOP_DATA_DIR` with the directory you created in the first step.

HADOOP_HOME/conf/core-site.xml

```
<configuration>
<property>
<name>fs.default.name</name>
<!-- URL of MasterNode/NameNode -->
<value>hdfs://MASTER_NODE:9000/</value>
</property>
</configuration>
```

HADOOP_HOME/conf/hdfs-site.xml

```
<configuration>
<property>
<name>dfs.name.dir</name>
<!-- Path to store namespace and transaction logs -->
<value>HADOOP_DATA_DIR/name</value>
</property>
<property>
<name>dfs.data.dir</name>
<!-- Path to store data blocks in datanode -->
<value>HADOOP_DATA_DIR/data</value>
</property>
</configuration>
```

7. From the NameNode, run the following command to format a new filesystem:

    ```
    >bin/hadoop namenode –format
    ```

    ```
    12/04/09 08:44:50 INFO namenode.NameNode: STARTUP_MSG:
    /************************************************************
    …
    12/04/09 08:44:51 INFO common.Storage: Storage directory /Users/
    srinath/playground/hadoop-book/hadoop-temp/dfs/name has been
    successfully formatted.
    12/04/09 08:44:51 INFO namenode.NameNode: SHUTDOWN_MSG:
    /************************************************************
    SHUTDOWN_MSG: Shutting down NameNode at Srinath-s-MacBook-Pro.
    local/172.16.91.1
    ************************************************************/
    ```

8. Start the HDFS setup with the following command:

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

    This command will first start a NameNode. It will then look at the `HADOOP_HOME/ conf/slaves` file and start the DataNodes. It will print a message like the following to the console.

    ```
    starting namenode, logging to /root/hadoop-setup-srinath/
    hadoop-1.0.0/libexec/../logs/hadoop-root-namenode-node7.beta.out
    ```

    ```
    209.126.198.72: starting datanode, logging to /root/hadoop-setup-
    srinath/hadoop-1.0.0/libexec/../logs/hadoop-root-datanode-node7.
    beta.out
    ```

    ```
    209.126.198.71: starting datanode, logging to /root/hadoop-setup-
    srinath/hadoop-1.0.0/libexec/../logs/hadoop-root-datanode-node6.
    beta.out
    ```

    ```
    209.126.198.72: starting secondarynamenode, logging to /root/
    hadoop-setup-srinath/hadoop-1.0.0/libexec/../logs/hadoop-root-
    secondarynamenode-node7.beta.out
    ```

    Hadoop uses a centralized architecture for metadata. In this design, the NameNode holds the information of all the files and where the data blocks for each file are located. The NameNode is a single point of failure, and on failure it will stop all the operations of the HDFS cluster. To avoid this, Hadoop supports a secondary NameNode that will hold a copy of all data in NameNode. If the NameNode fails, the secondary NameNode takes its place.

9. Access the link `http://MASTER_NODE:50070/` and verify that you can see the HDFS startup page. Here, replace `MASTER_NODE` with the IP address of the master node running the HDFS NameNode.

10. Finally, shut down the HDFS cluster using the following command:

```
>bin/stop-dfs.sh
```

How it works...

When started, the NameNode will read the `HADOOP_HOME/conf/slaves` files, find the DataNodes that need to be started, start them, and set up the HDFS cluster. In the *HDFS basic command line file operations* recipe, we will explore how to use HDFS to store and manage files.

HDFS setup is only a part of the Hadoop installation. The *Setting Hadoop in a distributed cluster environment* recipe describes how to set up the rest of the Hadoop.

Using HDFS monitoring UI

HDFS comes with a monitoring web console to verify the installation and monitor the HDFS cluster. It also lets users explore the content of the HDFS filesystem. In this recipe, we will look at how we can access the HDFS monitoring UI and verify the installation.

Getting ready

Start the HDFS cluster as described in the previous recipe.

How to do it...

Let us access the HDFS web console.

1. Access the link `http://MASTER_NODE:50070/` using your browser, and verify that you can see the HDFS startup page. Here, replace `MASTER_NODE` with the IP address of the master node running the HDFS NameNode.

2. The following screenshot shows the current status of the HDFS installation including the number of nodes, total storage, storage taken by each node. It also allows users to browse the HDFS filesystem.

HDFS basic command-line file operations

HDFS is a distributed filesystem, and just like a Unix filesystem, it allows users to manipulate the filesystem using shell commands. This recipe explains how to use the HDFS basic command line to execute those commands.

It is worth noting that HDFS commands have a one-to-one correspondence with Unix commands. For example, consider the following command:

```
>hadoop dfs –cat /data/foo.txt
```

The command reads the /data/foo.txt file and prints it to the screen, just like the cat command in Unix system.

Getting ready

Start the HDFS server by following the *Setting up HDFS* recipe.

How to do it...

1. Change the directory to `HADOOP_HOME`.

2. Run the following command to create a new directory called `/test`:

   >bin/hadoop dfs -mkdir /test

3. HDFS filesystem has / as the root directory just like the Unix filesystem. Run the following command to list the content of the HDFS root directory:

   >bin/hadoop dfs -ls /

4. Run the following command to copy the local readme file to /test

   >bin/hadoop dfs -put README.txt /test

5. Run the following command to list the `/test` directory:

   >bin/hadoop dfs -ls /test

   Found 1 items

   **-rw-r--r--   1 srinath supergroup   1366 2012-04-10 07:06 /
   test/README.txt**

6. Run the following command to copy the `/test/README.txt` to local directory:

   >bin/hadoop dfs -get /test/README.txt README-NEW.txt

How it works...

When a command is issued, the client will talk to the HDFS NameNode on the user's behalf and carry out the operation. Generally, we refer to a file or a folder using the path starting with /; for example, /data, and the client will pick up the NameNode from configurations in the HADOOP_HOME/conf directory.

However, if needed, we can use a fully qualified path to force the client to talk to a specific NameNode. For example, hdfs://bar.foo.com:9000/data will ask the client to talk to NameNode running on bar.foo.com at the port 9000.

There's more...

HDFS supports most of the Unix commands such as cp, mv, and chown, and they follow the same pattern as the commands discussed above. The document http://hadoop.apache. org/docs/r1.0.3/file_system_shell.html provides a list of all commands. We will use these commands throughout, in the recipes of the book.

Setting Hadoop in a distributed cluster environment

Hadoop deployment includes a HDFS deployment, a single job tracker, and multiple TaskTrackers. In the preceding recipe, *Setting up HDFS*, we discussed the HDFS deployment. For the Hadoop setup, we need to configure JobTrackers and TaskTrackers and then specify the TaskTrackers in the `HADOOP_HOME/conf/slaves` file. When we start the JobTracker, it will start the TaskTracker nodes. The following diagram illustrates a Hadoop deployment:
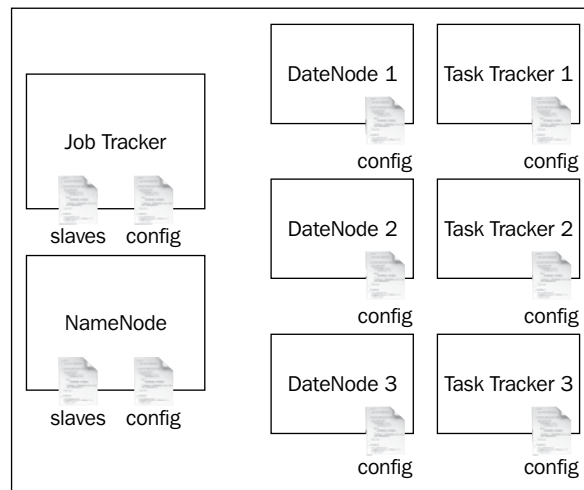
Getting ready

You may follow this recipe either using a single machine or multiple machines. If you are using multiple machines, you should choose one machine as the master node where you will run the HDFS NameNode and the JobTracker. If you are using a single machine, use it as both the master node as well as a slave node.

1. Install Java in all machines that will be used to set up Hadoop.

2. If you are using Windows machines, first install Cygwin and SSH server in each machine. The link `http://pigtail.net/LRP/printsrv/cygwin-sshd.html` provides step-by-step instructions.

How to do it...

Let us set up Hadoop by setting up the JobTracker and TaskTrackers.

1. In each machine, create a directory for Hadoop data. Let's call this directory `HADOOP_DATA_DIR`. Then create three directories, `HADOOP_DATA_DIR/data`, `HADOOP_DATA_DIR/local`, and `HADOOP_DATA_DIR/name`.

2. Set up SSH keys to all machines so that we can log in to all from the master node. The *Setting up HDFS* recipe describes the SSH setup in detail.

3. Unzip the Hadoop distribution at the same location in all machines using the `>tar -zxvf hadoop-1.x.x.tar.gz` command. You can use any of the Hadoop 1.0 branch distributions.

4. In all machines, edit the `HADOOP_HOME/conf/hadoop-env.sh` file by uncommenting the `JAVA_HOME` line and point it to your local Java installation. For example, if Java is in `/opt/jdk1.6`, change the `JAVA_HOME` line to `export JAVA_HOME=/opt/jdk1.6`.

5. Place the IP address of the node used as the master (for running JobTracker and NameNode) in `HADOOP_HOME/conf/masters` in a single line. If you are doing a single-node deployment, leave the current value, `localhost`, as it is.

    ```
    209.126.198.72
    ```

6. Place the IP addresses of all slave nodes in the `HADOOP_HOME/conf/slaves` file, each in a separate line.

    ```
    209.126.198.72
    209.126.198.71
    ```

7. Inside each node's `HADOOP_HOME/conf` directory, add the following to the `core-site.xml`, `hdfs-site.xml` and `mapred-site.xml`. Before adding the configurations, replace the `MASTER_NODE` with the IP of the master node and `HADOOP_DATA_DIR` with the directory you created in the first step.

    Add URL of the NameNode to `HADOOP_HOME/conf/core-site.xml`.

    ```
    <configuration>
    <property>
    <name>fs.default.name</name>
    <value>hdfs://MASTER_NODE:9000/</value>
    </property>
    </configuration>
    ```

    Add locations to store metadata (names) and data within `HADOOP_HOME/conf/hdfs-site.xml` to submit jobs:

    ```
    <configuration>
    <property>
    ```

```
<name>dfs.name.dir</name>
<value>HADOOP_DATA_DIR/name</value>
</property>
<property>
<name>dfs.data.dir</name>
<value>HADOOP_DATA_DIR/data</value>
</property>
</configuration>
```

Map reduce local directory is the location used by Hadoop to store temporary files used. Add JobTracker location to `HADOOP_HOME/conf/mapred-site.xml`. Hadoop will use this for the jobs. The final property sets the maximum map tasks per node, set it the same as the amount of cores (CPU).

```
<configuration>
<property>
<name>mapred.job.tracker</name>
<value>MASTER_NODE:9001</value>
</property>
<property>
<name>mapred.local.dir</name>
<value>HADOOP_DATA_DIR/local</value>
</property>
<property>
<name>mapred.tasktracker.map.tasks.maximum</name>
<value>8</value>
</property>
</configuration>
```

8. To format a new HDFS filesystem, run the following command from the Hadoop NameNode (master node). If you have done this as part of the HDFS installation in earlier recipe, you can skip this step.

```
>bin/hadoop namenode –format

...

/Users/srinath/playground/hadoop-book/hadoop-temp/dfs/name has
been successfully formatted.

12/04/09 08:44:51 INFO namenode.NameNode: SHUTDOWN_MSG:

/************************************************************

SHUTDOWN_MSG: Shutting down NameNode at Srinath-s-MacBook-Pro.
local/172.16.91.1

************************************************************/
```

9. In the master node, change the directory to `HADOOP_HOME` and run the following commands:

    `>bin/start-dfs.sh`

    `starting namenode, logging to /root/hadoop-setup-srinath/`
    `hadoop-1.0.0/libexec/../logs/hadoop-root-namenode-node7.beta.out`

    `209.126.198.72: starting datanode, logging to /root/hadoop-setup-`
    `srinath/hadoop-1.0.0/libexec/../logs/hadoop-root-datanode-node7.`
    `beta.out`

    `209.126.198.71: starting datanode, logging to /root/hadoop-setup-`
    `srinath/hadoop-1.0.0/libexec/../logs/hadoop-root-datanode-node6.`
    `beta.out`

    `209.126.198.72: starting secondarynamenode, logging to /root/`
    `hadoop-setup-srinath/hadoop-1.0.0/libexec/../logs/hadoop-root-`
    `secondarynamenode-node7.beta.out`

    `>bin/start-mapred.sh`

    `starting jobtracker, logging to /root/hadoop-setup-srinath/`
    `hadoop-1.0.0/libexec/../logs/hadoop-root-jobtracker-node7.beta.out`

    `209.126.198.72: starting tasktracker, logging to /root/`
    `hadoop-setup-srinath/hadoop-1.0.0/libexec/../logs/hadoop-root-`
    `tasktracker-node7.beta.out`

    `209.126.198.71: starting tasktracker, logging to /root/`
    `hadoop-setup-srinath/hadoop-1.0.0/libexec/../logs/hadoop-root-`
    `tasktracker-node6.beta.out`

10. Verify the installation by listing the processes through the `ps | grep java` command (if you are using Linux) or via Task Manager (if you are in Windows), in the master node and slave nodes. Master node will list four processes—NameNode, DataNode, JobTracker, and TaskTracker and slaves will have a DataNode and TaskTracker.

11. Browse the web-based monitoring pages for namenode and JobTracker:

    ❑ **NameNode**: `http://MASTER_NODE:50070/`.

    ❑ **JobTracker**: `http://MASTER_NODE:50030/`.

12. You can find the logfiles under `${HADOOP_HOME}/logs`.

13. Make sure HDFS setup is OK by listing the files using HDFS command line.

    `bin/hadoop dfs -ls /`

    `Found 2 items`
    `drwxr-xr-x   - srinath supergroup    0 2012-04-09 08:47 /Users`
    `drwxr-xr-x   - srinath supergroup    0 2012-04-09 08:47 /tmp`

How it works...

As described in the introduction to the chapter, Hadoop installation consists of HDFS nodes, a JobTracker and worker nodes. When we start the NameNode, it finds the slaves through the `HADOOP_HOME/slaves` file and uses SSH to start the DataNodes in the remote server at the startup. Also when we start the JobTracker, it finds the slaves through the `HADOOP_HOME/slaves` file and starts the TaskTrackers.

There's more...

In the next recipe, we will discuss how to run the aforementioned WordCount program using the distributed setup. The following recipes will discuss how to use MapReduce monitoring UI to monitor the distributed Hadoop setup.

Running the WordCount program in a distributed cluster environment

This recipe describes how to run a job in a distributed cluster.

Getting ready

Start the Hadoop cluster.

How to do it...

Now let us run the WordCount sample in the distributed Hadoop setup.

1. To use as inputs to the WordCount MapReduce sample that we wrote in the earlier recipe, copy the `README.txt` file in your Hadoop distribution to the HDFS filesystem at the location `/data/input1`.

   ```
   >bin/hadoop dfs -mkdir /data/
   >bin/hadoop dfs -mkdir /data/input1
   >bin/hadoop dfs -put README.txt /data/input1/README.txt
   >bin/hadoop dfs -ls /data/input1


   Found 1 items
   -rw-r--r--   1 srinath supergroup      1366 2012-04-09 08:59 /data/input1/README.txt
   ```

2. Now, let's run the WordCount example from the `HADOOP_HOME` directory.

```
>bin/hadoop jar hadoop-examples-1.0.0.jar wordcount /data/input1 /
data/output1
```

```
12/04/09 09:04:25 INFO input.FileInputFormat: Total input paths to
process : 1
```

```
12/04/09 09:04:26 INFO mapred.JobClient: Running job:
job_201204090847_0001
```

```
12/04/09 09:04:27 INFO mapred.JobClient:  map 0% reduce 0%
```

```
12/04/09 09:04:42 INFO mapred.JobClient:  map 100% reduce 0%
```

```
12/04/09 09:04:54 INFO mapred.JobClient:  map 100% reduce 100%
```

```
12/04/09 09:04:59 INFO mapred.JobClient: Job complete:
job_201204090847_0001
```

```
.....
```

3. Run the following commands to list the output directory and then look at the results.

```
>bin/hadoop dfs -ls /data/output1
```

```
Found 3 items
```

```
-rw-r--r--   1 srinath supergroup          0 2012-04-09 09:04 /
data/output1/_SUCCESS
```

```
drwxr-xr-x   - srinath supergroup          0 2012-04-09 09:04 /
data/output1/_logs
```

```
-rw-r--r--   1 srinath supergroup       1306 2012-04-09 09:04 /
data/output1/part-r-00000
```

```
>bin/hadoop dfs -cat /data/output1/*
```

```
(BIS),  1
(ECCN)   1
(TSU)  1
(see  1
5D002.C.1,  1
740.13)  1
```

How it works...

Job submission to the distributed Hadoop works in a similar way to the job submissions to local Hadoop installation, as described in the *Writing a WordCount MapReduce sample, bundling it and running it using standalone Hadoop* recipe. However, there are two main differences.

First, Hadoop stores both the inputs for the jobs and output generated by the job in HDFS filesystem. Therefore, we use step 1 to store the inputs in the HDFS filesystem and we use step 3 read outputs from the HDFS filesystem.

Secondly, when job is submitted, local Hadoop installation runs the job as a local JVM execution. However, the distributed cluster submits it to the JobTracker, and it executes the job using nodes in the distributed Hadoop cluster.

There's more...

You can see the results of the WordCount application also through the HDFS monitoring UI, as described in the *Using HDFS monitoring UI* recipe, and also you can see the statistics about the WordCount job as explained in the next recipe, *Using MapReduce Monitoring UI*.

Using MapReduce monitoring UI

This recipe describes how to use the Hadoop monitoring web console to verify Hadoop installation, and to monitor the allocations and uses of each part of the Hadoop cluster.

How to do it...

Now let us visit the Hadoop monitoring web console.

1. Access `http://MASTER_NODE:50030/` using the browser where `MASTER_NODE` is the IP address of the master node.

2. The web page shows the current status of the MapReduce installation, including running and completed jobs.

How it works...

Hadoop monitoring UI lets users access the JobTracker of the Hadoop installation and find different nodes in the installation, their configurations, and usage. For example, users can use the UI to see current running jobs and logs associated with the jobs.

2
Advanced HDFS

In this chapter, we will cover:

- ▸ Benchmarking HDFS
- ▸ Adding a new DataNode
- ▸ Decommissioning DataNodes
- ▸ Using multiple disks/volumes and limiting HDFS disk usage
- ▸ Setting HDFS block size
- ▸ Setting the file replication factor
- ▸ Using HDFS Java API
- ▸ Using HDFS C API (libhdfs)
- ▸ Mounting HDFS (Fuse-DFS)
- ▸ Merging files in HDFS

Introduction

Hadoop Distributed File System (**HDFS**) is a block-structured, distributed filesystem that is designed to run on a low-cost commodity hardware. HDFS supports storing massive amounts of data and provides high-throughput access to the data. HDFS stores file data across multiple nodes with redundancy to ensure fault-tolerance and high aggregate bandwidth.

HDFS is the default distributed filesystem used by the Hadoop MapReduce computations. Hadoop supports data locality aware processing of the data stored in HDFS. However, HDFS can be used as a general purpose distributed filesystem as well. HDFS architecture consists mainly of a centralized NameNode that handles the filesystem metadata and DataNodes that store the real data blocks. HDFS data blocks are typically coarser grained and perform better with large data products.

Setting up HDFS and other related recipes in *Chapter 1, Getting Hadoop Up and Running in a Cluster*, show how to deploy HDFS and give an overview of the basic operation of HDFS. In this chapter, you will be introduced to a selected set of advanced HDFS operations that would be useful when performing large-scale data processing with Hadoop MapReduce, as well as when using HDFS as a standalone distributed filesystem for non-MapReduce use cases.

Benchmarking HDFS

Running benchmarks is a good way to verify whether your HDFS cluster is set up properly and performs as expected. **DFSIO** is a benchmark test that comes with Hadoop, which can be used to analyze the I/O performance of a HDFS cluster. This recipe shows how to use DFSIO to benchmark the read and write performance of a HDFS cluster.

Getting ready

You must set up and deploy HDFS and Hadoop MapReduce prior to running these benchmarks. Export the HADOOP_HOME environment variable to point to your Hadoop installation root directory:

```
>export HADOOP_HOME=/../hadoop-1.0.4
```

The benchmark programs are in the $HADOOP_HOME/hadoop-*test.jar file.

How to do it...

The following steps show you how to run the write performance benchmark:

1. To run the write performance benchmark, execute the following command in the $HADOOP_HOME directory. The –nrFiles parameter specifies the number of files and the -fileSize parameter specifies the file size in MB.

   ```
   >bin/hadoop jar $HADOOP_HOME/hadoop-test-*.jar TestDFSIO -write
   -nrFiles 5 –fileSize 100
   ```

2. The benchmark writes to the console, as well as appends to a file named TestDFSIO_results.log. You can provide your own result file name using the –resFile parameter.

The following steps show you how to run the read performance benchmark:

1. The read performance benchmark uses the files written by the write benchmark in step 1. Hence, the write benchmark should be executed before running the read benchmark and the files written by the write benchmark should exist in the HDFS for the read benchmark to work.

2. Execute the following command to run the read benchmark. Benchmark writes the results to the console and appends the results to a logfile similarly to the write benchmark.

```
>bin/hadoop jar $HADOOP_HOME/hadoop-test-*.jar TestDFSIO -read
-nrFiles5 –fileSize 100
```

To clean the files generated by these benchmarks, use the following command:

```
>bin/hadoop jar $HADOOP_HOME hadoop-test-*.jar TestDFSIO -clean
```

How it works...

DFSIO executes a MapReduce job where the map tasks write and read the files in parallel, while the reduce tasks are used to collect and summarize the performance numbers.

There's more...

Running these tests together with monitoring systems can help you identify the bottlenecks much more easily.

See also

▶   The *Running benchmarks to verify the Hadoop installation* recipe in *Chapter 3, Advanced Hadoop MapReduce Administration*.

Adding a new DataNode

This recipe shows how to add new nodes to an existing HDFS cluster without restarting the whole cluster, and how to force HDFS to rebalance after the addition of new nodes.

Getting ready

To get started, follow these steps:

1. Install Hadoop on the new node and replicate the configuration files of your existing Hadoop cluster. You can use `rsync` to copy the Hadoop configuration from another node. For example:

```
>rsync -a <master_node_ip>:hadoop-1.0.x/conf $HADOOP_HOME/conf
```

2. Ensure that the master node of your Hadoop/HDFS cluster can perform password-less SSH to the new node. Password-less SSH setup is optional, if you are not planning on using the `bin/*.sh` scripts from the master node to start/stop the cluster.

How to do it...

The following steps will show you how to add a new DataNode to an existing HDFS cluster:

1. Add the IP or the DNS of the new node to the `$HADOOP_HOME/conf/slaves` file in the master node.

2. Start the DataNode in the newly added **slave node** by using the following command.

   ```
   >bin/hadoop-deamon.sh start datanode
   ```

   > You can also use the `$HADOOP_HOME/bin/start-dfs.sh` script from the master node to start the DataNode daemons in the newly added nodes. This is helpful if you are adding more than one new DataNodes to the cluster.
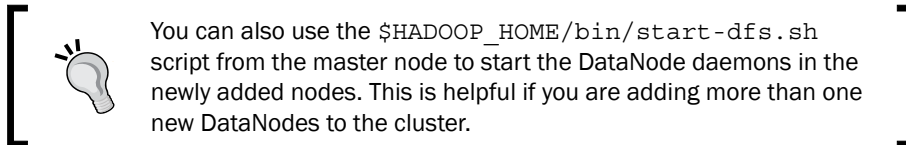
3. Check the `$HADOOP_HOME/logs/hadoop-*-datanode-*.log` in the new slave node for any errors.

The preceding steps apply both to adding a new node as well as re-joining a node that has been crashed and restarted.

There's more...

Similarly, you can add a new node to the Hadoop MapReduce cluster as well.

1. Start the **TaskTracker** in the new node using the following command:

   ```
   >bin/hadoop-deamon.sh start tasktracker
   ```

2. Check the `$HADOOP_HOME/logs/hadoop-*-tasktracker-*.log` in the new slave node for any errors.

Rebalancing HDFS

When you add new nodes, HDFS will not rebalance automatically. However, HDFS provides a **rebalancer** tool that can be invoked manually. This tool will balance the data blocks across cluster up to an optional threshold percentage. Rebalancing would be very helpful if you are having space issues in the other existing nodes.

1. Execute the following command. The optional `-threshold` parameter specifies the percentage of disk capacity leeway to consider when identifying a node as under- or over-utilized. An under-utilized data node is a node whose utilization is less than *average utilization – threshold*. An over-utilized data node is a node whose utilization is greater than *average utilization + threshold*. Smaller threshold values will achieve more evenly balanced nodes, but would take more time for the rebalancing. Default threshold value is 10 percent.

   ```
   >bin/start-balancer.sh -threshold 15
   ```

2. Rebalancing can be stopped by executing the `bin/stop-balancer.sh` command.

3. A summary of the rebalancing will be available at the `$HADOOP_HOME/logs/ hadoop-*-balancer*.out` file.

See also

▸ The *Decommissioning data nodes* recipe in this chapter.

Decommissioning DataNodes

There can be multiple situations where you want to decommission one or more data nodes from an HDFS cluster. This recipe shows how to gracefully decommission the DataNodes without incurring data loss and without having to restart the cluster.

How to do it...

The following steps show you how to decommission data nodes gracefully:

1. If your cluster doesn't have it, add an **exclude file** to the cluster. Create an empty file in the NameNode and point to it from the `conf/hdfs-site.xml` file by adding the following property.

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

2. Add the hostnames of the nodes that are to be decommissioned to the `exclude` file.

3. Run the following command to reload the NameNode configuration. This will start the decommissioning process. The decommissioning process can take a significant time, as it requires replication of data blocks without overwhelming the other tasks of the cluster.

   `>bin/hadoop dfsadmin -refreshNodes`

4. The decommissioning progress is shown in the HDFS UI under the **Decommissioning Nodes** page. The decommissioning progress can be monitored using the following command as well. Do not shut down the nodes until the decommissioning is complete.

```
>bin/hadoop dfsadmin -report
.....
.....
Name: myhost:50010
Decommission Status : Decommission in progress
Configured Capacity: ....
.....
```

5. You can remove the nodes from the `exclude` file and execute the `bin/hadoop dfsadmin -refreshNodes` command when you want to add the nodes back in to the cluster.

6. The decommissioning process can be stopped by removing the node's name from the `exclude` file and then executing the `bin/hadoop dfsadmin -refreshNodes` command.

How it works...

When a node is in the decommissioning process, HDFS replicates the blocks in that node to the other nodes in the cluster. Decommissioning can be a slow process as HDFS purposely does it slowly to avoid overwhelming the cluster. Shutting down nodes without decommissioning may result in data loss.

After the decommissioning is completed, the nodes mentioned in the `exclude` file are not allowed to communicate with the NameNode.

See also

▶ The *Rebalancing HDFS* section of the *Adding a new node* recipe in this chapter.

Using multiple disks/volumes and limiting HDFS disk usage

Hadoop supports specifying multiple directories for DataNode data directory. This feature allows us to utilize multiple disks/volumes to store the data blocks in DataNodes. Hadoop will try to store equal amounts of data in each directory. Hadoop also supports limiting the amount of disk space used by HDFS.

How to do it...

The following steps will show you how to add multiple disk volumes:

1. Create HDFS data storage directories in each volume.

2. In the `$HADOOP_HOME/conf/hdfs-site.xml`, provide a comma-separated list of directories corresponding to the data storage locations in each volume under the `dfs.data.dir` directory.

```
<property>
  <name>dfs.data.dir</name>
  <value>/u1/hadoop/data,/u2/hadoop/data</value>
</property>
```

3. To limit the HDFS disk usage, add the following property to `$HADOOP_HOME/conf/hdfs-site.xml` to reserve space for non-DFS usage. The value specifies the number of bytes that HDFS cannot use per volume.

```
<property>
  <name>dfs.datanode.du.reserved</name>
  <value>6000000000</value>
  <description>Reserved space in bytes per volume. Always leave
this much space free for non dfs use.
  </description>
</property>
```

Setting HDFS block size

HDFS stores files across the cluster by breaking them down in to coarser grained, fixed-size blocks. The default HDFS block size is 64 MB. The block size of a data product can affect the performance of the filesystem operations where larger block sizes would be more effective, if you are storing and processing very large files. The block size of a data product can affect the performance of MapReduce computations, as the default behavior of Hadoop is to create one map task for each data block of the input files.

How to do it...

1. To use the NameNode configuration file to set the HDFS block size, add or modify the following in the `$HADOOP_HOME/conf/hdfs-site.xml`. Block size is provided using the number of bytes. This change would not change the block size of the files that are already in the HDFS. Only the files copied after the change will have the new block size.

```
<property>
  <name>dfs.block.size</name>
  <value>134217728</value>
</property>
```

2. To specify the HDFS block size for specific file paths, you can specify the block size when uploading the file from the command line as follows:

```
>bin/hadoop fs -Ddfs.blocksize=134217728 -put data.in /user/foo
```

There's more...

You can also specify the block size when creating files using the HDFS Java API as well.

```
public FSDataOutputStream create(Path f,boolean overwrite, int
bufferSize, short replication,long blockSize)
```

You can use the `fsck` command to find the block size and block locations of a particular file path in the HDFS. You can find this information by browsing the filesystem from the HDFS monitoring console as well.

```
>bin/hadoop fsck /user/foo/data.in -blocks -files -locations
......
/user/foo/data.in 215227246 bytes, 2 block(s): ....
0. blk_6981535920477261584_1059len=134217728 repl=1 [hostname:50010]
1. blk_-8238102374790373371_1059 len=81009518 repl=1 [hostname:50010]


......
```

See also

▸ The *Setting file replication factor* recipe in this chapter.

Setting the file replication factor

HDFS stores files across the cluster by breaking them down in to coarser grained fixed-size blocks. These coarser grained data blocks are replicated in different DataNodes mainly for the fault-tolerance purposes. Data block replication also has the ability to increase the data locality of the MapReduce computations and to increase the total data access bandwidth as well. Reducing the replication factor helps save the storage space in HDFS.

HDFS replication factor is a file-level property that can be set per file basis. This recipe shows how to change the default replication factor of a HDFS deployment affecting the new files that would be created afterwards, how to specify a custom replication factor at the time of file creation in HDFS, and how to change the replication factor of the existing files in HDFS.

How to do it...

1. To set the file replication factor using the NameNode configuration, add or modify the `dfs.replication` property in `$HADOOP_HOME/conf/hdfs-site.xml`. This change would not change the replication factor of the files that are already in the HDFS. Only the files copied after the change will have the new replication factor.

    ```
    <property>

      <name>dfs.replication</name>

      <value>2</value>

    </property>
    ```

2. To set the file replication factor when uploading the files, you can specify the replication factor from the command line, as follows:

    ```
    >bin/hadoop fs -D dfs.replication=1 -copyFromLocal non-critical-
    file.txt /user/foo
    ```

3. The `setrep` command can be used to change the replication factor of files or file paths that are already in the HDFS.

    ```
    > bin/hadoop fs -setrep 2 non-critical-file.txt

    Replication 3 set: hdfs://myhost:9000/user/foo/non-critical-file.
    txt
    ```

How it works...

The `setrep` command syntax is as follows:

```
hadoop fs -setrep [-R] <path>
```

The `<path>` parameter of the `setrep` command specifies the HDFS path where the replication factor has to be changed. The `-R` option recursively sets the replication factor for files and directories within a directory.

There's more...

The replication factor of a file is displayed when listing the files using the `ls` command.

```
>bin/hadoop fs -ls

Found 1 item

-rw-r--r--2foo supergroup ... /user/foo/non-critical-file.txt
```

The replication factor of files is displayed when browsing files in the HDFS monitoring UI.

► The *Setting HDFS block size* recipe in this chapter.

Using HDFS Java API

HDFS Java API can be used to interact with HDFS from any Java program. This API gives us the ability to utilize the data stored in HDFS from other Java programs as well as to process that data with other non-Hadoop computational frameworks. Occasionally you may also come across a use case where you want to access HDFS directly from inside a MapReduce application. However, if you are writing or modifying files in HDFS directly from a Map or Reduce task, be aware that you are violating the side effect free nature of MapReduce that might lead to data consistency issues based on your use case.

Getting ready

Set the `HADOOP_HOME` environment variable to point to your Hadoop installation root directory.

How to do it...

The following steps show you how to use the HDFS Java API to perform filesystem operations on a HDFS installation using a Java program:

1. The following sample program creates a new file in HDFS, writes some text to the newly created file, and reads the file back from the HDFS:

```java
importjava.io.IOException;

importorg.apache.hadoop.conf.Configuration;
importorg.apache.hadoop.fs.FSDataInputStream;
importorg.apache.hadoop.fs.FSDataOutputStream;
importorg.apache.hadoop.fs.FileSystem;
importorg.apache.hadoop.fs.Path;

public class HDFSJavaAPIDemo {

  public static void main(String[] args) throws IOException {
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(conf);
    System.out.println(fs.getUri());

    Path file = new Path("demo.txt");

    if (fs.exists(file)) {
```

```
          System.out.println("File exists.");
      } else {
        // Writing to file
        FSDataOutputStream outStream = fs.create(file);
        outStream.writeUTF("Welcome to HDFS Java API!!!");
        outStream.close();
      }

      // Reading from file
      FSDataInputStream inStream = fs.open(file);
      String data = inStream.readUTF();
      System.out.println(data);
      inStream.close();

      fs.close();
    }
```

2. Compile and package the above program in to a JAR package. Unzip the source package for this chapter, go to the `HDFS_Java_API` folder and run the Ant build. The `HDFSJavaAPI.jar` file will be created in the `build` folder.

   >cd HDFS_java_API

   >ant

   You can use the following Ant build file to compile the above sample program:

```xml
<project name="HDFSJavaAPI" default="compile" basedir=".">
  <property name="build" location="build"/>
  <property environment="env"/>

  <path id="hadoop-classpath">
    <fileset dir="${env.HADOOP_HOME}/lib">
      <include name="**/*.jar"/>
    </fileset>
    <fileset dir="${env.HADOOP_HOME}">
      <include name="**/*.jar"/>
    </fileset>
  </path>

  <target name="compile">
    <mkdir dir="${build}"/>
    <javac srcdir="src" destdir="${build}">
      <classpath refid="hadoop-classpath"/>
    </javac>
    <jar jarfile="HDFSJavaAPI.jar" basedir="${build}"/>
```

```
      </target>

      <target name="clean">
        <delete dir="${build}"/>
      </target>
</project>
```

3. You can execute the above sample with Hadoop using the following command. Running samples using the `hadoop` script ensures that it uses the currently configured HDFS and the necessary dependencies from the Hadoop classpath.

    >bin/hadoop jar HDFSJavaAPI.jar HDFSJavaAPIDemo

    hdfs://yourhost:9000

    Welcome to HDFS Java API!!!

4. Use the `ls` command to list the newly created file:

    >/bin/hadoop fs -ls

    Found 1 items

    -rw-r--r--   3 foosupergroup   20 2012-04-27 16:57 /user/ foo/demo.txt

How it works...

In order to interact with the HDFS programmatically, we first obtain a handle to the currently configured filesystem. Instantiating a Configuration object and obtaining a `FileSystem` handle within a Hadoop environment will point it to the HDFS NameNode of that environment. Several alternative methods to configure a `FileSystem` object are discussed in the *Configuring the FileSystem object* section.

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
```

The `FileSystem.create(filePath)` method creates a new file in the given path and provides us with a `FSDataOutputStream` object to the newly created file. `FSDataOutputStream` wraps the `java.io.DataOutputStream` and allows the program to write primitive Java data types to the file. The `FileSystem.Create()` method overrides if the file exists. In this example, the file will be created relative to the users' home directory in the HDFS, which would result in a path similar to `/user/<user_name>/demo.txt`.

```
Path file = new Path("demo.txt");
FSDataOutputStream outStream = fs.create(file);
outStream.writeUTF("Welcome to HDFS Java API!!!");
outStream.close();
```

`FileSystem.open(filepath)` opens a `FSDataInputStream` to the given file. `FSDataInputStream` wraps the `java.io.DataInputStream` and allows the program to read primitive Java data types from the file.

```
FSDataInputStream inStream = fs.open(file);
String data = inStream.readUTF();
System.out.println(data);
inStream.close();
```

There's more...

HDFS Java API supports many more filesystem operations than we have used in the above sample. The full API documentation can be found at `http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/fs/FileSystem.html`.

Configuring the FileSystem object

We can use the HDFS Java API from outside the Hadoop environment as well. When doing so, we have to explicitly configure the HDFS NameNode and the port. The following are a couple of ways to perform that configuration:

► You can load the configuration files to the `Configuration` object before retrieving the `FileSystem` object as follows. Make sure to add all the Hadoop and dependency libraries to the classpath.

```
Configuration conf = new Configuration();
conf.addResource(new Path("../hadoop/conf/core-site.xml"));
conf.addResource(new Path("../hadoop/conf/hdfs-site.xml"));

FileSystem fileSystem = FileSystem.get(conf);
```

► You can also specify the NameNode and the port as follows. Replace the `NAMENODE_HOSTNAME` and `PORT` with the hostname and the port of the NameNode of your HDFS installation.

```
Configuration conf = new Configuration();
conf.set("fs.default.name", "hdfs://NAMENODE_HOSTNAME:PORT");
FileSystem fileSystem = FileSystem.get(conf);
```

HDFS filesystem API is an abstraction that supports several filesystems. In case the above program could not find a valid HDFS configuration, it will point to the local filesystem instead of the HDFS. You can identify the current filesystem of the `FileSystem` object using the `getUri()` function as follows. It would result in `hdfs://your_namenode:port` in the case it's using a properly configured HDFS and `file:///` in the case it is using the local filesystem.

```
fileSystem.getUri();
```

Retrieving the list of data blocks of a file

The `getFileBlockLocations()` function of the `FileSystem` object allows you to retrieve the list of data blocks of a file stored in HDFS, together with hostnames where the blocks are stored and the block offsets. This information would be very useful if you are planning for performing any data local operations on the file's data using a framework other than Hadoop MapReduce.

```
FileStatus fileStatus = fs.getFileStatus(file);
BlockLocation[] blocks = fs.getFileBlockLocations(
        fileStatus, 0, fileStatus.getLen());
```

See also

▸   The *Using HDFS C API* recipe in this chapter.

Using HDFS C API (libhdfs)

libhdfs—a native shared library—provides a C API that enables non-Java programs to interact with HDFS. libhdfs uses JNI to interact with HDFS through Java.

Getting ready

Current Hadoop distributions contain the pre-compiled libhdfs libraries for 32-bit and 64-bit Linux operating systems. You may have to download the Hadoop standard distribution and compile the libhdfs library from the source code, if your operating system is not compatible with the pre-compiled libraries. Refer to the *Mounting HDFS (Fuse-DFS)* recipe for information on compiling the libhdfs library.

How to do it...

The following steps show you how to perform operations on a HDFS installation using a HDFS C API:

1. The following sample program creates a new file in HDFS, writes some text to the newly created file and reads the file back from the HDFS. Replace NAMENODE_ HOSTNAME and PORT with the relevant values corresponding to the NameNode of your HDFS cluster. The `hdfs_cpp_demo.c` source file is provided in the HDFS_C_ API directory of the source code bundle for this folder.

    ```
    #include "hdfs.h"

    int main(intargc, char **argv) {

    hdfsFS fs =hdfsConnect("NAMENODE_HOSTNAME,PORT);
    ```

```
if (!fs) {
        fprintf(stderr, "Cannot connect to HDFS.\n");
        exit(-1);
    }

char* fileName = "demo_c.txt";
char* message = "Welcome to HDFS C API!!!";
int size = strlen(message);

int exists = hdfsExists(fs, fileName);

if (exists > -1) {
    fprintf(stdout, "File %s exists!\n", fileName);
}else{
  // Create and open file for writing
  hdfsFile outFile = hdfsOpenFile(fs, fileName, O_WRONLY|O_CREAT,
0, 0, 0);
if (!outFile) {
  fprintf(stderr, "Failed to open %s for writing!\n", fileName);
            exit(-2);
    }

    // write to file
hdfsWrite(fs, outFile, (void*)message, size);
    hdfsCloseFile(fs, outFile);
    }

    // Open file for reading
hdfsFile inFile = hdfsOpenFile(fs, fileName, O_RDONLY, 0, 0, 0);
    if (!inFile) {
fprintf(stderr, "Failed to open %s for reading!\n", fileName);
        exit(-2);
    }

    char* data = malloc(sizeof(char) * size);
    // Read from file.
tSize readSize = hdfsRead(fs, inFile, (void*)data, size);
fprintf(stdout, "%s\n", data);
    free(data);

hdfsCloseFile(fs, inFile);
hdfsDisconnect(fs);
    return 0;
}
```

2. Compile the above program by using `gcc` as follows. When compiling you have to link with the `libhdfs` and the JVM libraries. You also have to include the JNI header files of your Java installation. An example compiling command would look like the following. Replace the ARCH and the architecture dependent paths with the paths relevant for your system.

```
>gcc hdfs_cpp_demo.c \

-I $HADOOP_HOME/src/c++/libhdfs \

-I $JAVA_HOME/include \

-I $JAVA_HOME/include/linux/ \

-L $HADOOP_HOME/c++/ARCH/lib/ \

-L $JAVA_HOME/jre/lib/ARCH/server\

-l hdfs -ljvm -o hdfs_cpp_demo
```

3. Export an environment variable named `CLASSPATH` with the Hadoop dependencies. A safe approach is to include all the jar files in `$HADOOP_HOME` and in the `$HADOOP_HOME/lib`.

```
export CLASSPATH=$HADOOP_HOME/hadoop-core-xx.jar:....
```

Ant build script to generate the classpath

Add the following Ant target to the `build` file given in step 2 of the HDFS Java API recipe. The modified `build.xml` script is provided in the `HDFS_C_API` folder of the source package for this chapter.

```
<target name="print-cp">
    <property name="classpath"
      refid="hadoop-classpath"/>
    <echo message="classpath= ${classpath}"/>
  </target>
```

Execute the Ant build using `ant print-cp` to generate a string with all the jars in `$HADOOP_HOME` and `$HADOOP_HOME/lib`. Copy and export this string as the `CLASSPATH` environmental variable.

4. Execute the program.

```
>LD_LIBRARY_PATH=$HADOOP_HOME/c++/ARCH/lib:$JAVA_HOME/jre/lib/
ARCH/server ./hdfs_cpp_demo

Welcome to HDFS C API!!!
```

How it works...

First we connect to a HDFS cluster using the `hdfsConnect` command by providing the hostname (or the IP address) and port of the NameNode of the HDFS cluster. The `hdfsConnectAsUser` command can be used to connect to a HDFS cluster as a specific user.

```
hdfsFS fs =hdfsConnect("NAMENODE_HOSTNAME",PORT);
```

We create new file and obtain a handle to the newly created file using the `hdfsOpenFile` command. The `O_WRONLY|O_CREAT` flags create a new file or override the existing file and open it in write only mode. Other supported flags are `O_RDONLY` and `O_APPEND`. The fourth, fifth, and sixth parameters of the `hdfsOpenFile` command are the buffer size for read/write operations, block replication factor and block size for the newly created file. Specify `0` if you want to use the default values for these three parameters.

```
hdfsFile outFile = hdfsOpenFile(fs, fileName,flags, 0, 0, 0);
```

The `hdfsWrite` command writes the provided data in to the file specified by the `outFile` handle. Data size needs to be specified using the number of bytes.

```
hdfsWrite(fs, outFile, (void*)message, size);
```

The `hdfsRead` command reads data from the file specified by the `inFile`. The size of the buffer in bytes needs to be provided as the fourth parameter. The `hdfsRead` command returns the actual number of bytes read from the file that might be less than the buffer size. If you want to ensure certain amounts of bytes that are read from the file, it is advisable to use the `hdfsRead` command from inside a loop until the specified number of bytes are read.

```
char* data = malloc(sizeof(char) * size);
tSize readSize = hdfsRead(fs, inFile, (void*)data, size);
```

There's more...

HDFS C API (libhdfs) supports many more filesystem operations than the functions we have used in the preceding sample. Refer to the `$HADOOP_HOME/src/c++/libhdfs/hdfs.h` header file for more information.

Configuring using HDFS configuration files

You can also use the HDFS configuration files to point libhdfs to your HDFS NameNode, instead of specifying the NameNode hostname and the port number in the `hdfsConnect` command.

1. Change the NameNode hostname and the port of the `hdfsConnect` command to `'default'` and `0`. (Setting the host as NULL would make libhdfs to use the local filesystem).

   ```
   hdfsFS fs = hdfsConnect("default",0);
   ```

2. Add the `conf` directory of your HDFS installation to the `CLASSPATH` environmental variable.

    ```
    export CLASSPATH=$HADOOP_HOME/hadoop-core-xx.jar:....:$HADOOP_
    HOME/conf
    ```

See also

▸   The *HDFS Java API* and *Mounting HDFS* recipes in this chapter.

Mounting HDFS (Fuse-DFS)

The Fuse-DFS project allows us to mount HDFS on Linux (supports many other flavors of Unix as well) as a standard filesystem. This allows any program or user to access and interact with HDFS similar to a traditional filesystem.

Getting ready

You must have the following software installed in your system.

▸   Apache Ant (`http://ant.apache.org/`).

▸   Fuse and fuse development packages. Fuse development files can be found in fuse-devel RPM for Redhat/Fedora and in libfuse-dev package for Debian/Ubuntu.

`JAVA_HOME` must be set to point to a JDK, not to a JRE.

You must have the root privileges for the node in which you are planning to mount the HDFS filesystem.

The following recipe assumes you already have pre-built libhdfs libraries. Hadoop contains pre-built libhdfs libraries for the Linux x86_64/i386 platforms. If you are using some other platform, first follow the *Building libhdfs* sub section in the more info section to build the libhdfs libraries.

How to do it...

The following steps show you how to mount an HDFS filesystem as a standard file system on Linux:

1. Go to `$HADOOP_HOME` and create a new directory named `build`.

    ```
    >cd $HADOOP_HOME
    >mkdir build
    ```

2. Create a symbolic link to the `libhdfs` libraries inside the `build` directory.

   ```
   >ln -s c++/Linux-amd64-64/lib/ build/libhdfs
   ```

3. Copy the `c++` directory to the `build` folder.

   ```
   >cp -R c++/ build/
   ```

4. Build fuse-dfs by executing the following command in `$HADOOP_HOME`. This command will generate the `fuse_dfs` and `fuse_dfs_wrapper.sh` files in the `build/contrib/fuse-dfs/` directory.

   ```
   > ant compile-contrib -Dlibhdfs=1 -Dfusedfs=1
   ```

> If the build fails with messages similar to `undefined reference to 'fuse_get_context'`, then append the following to the end of the `src/contrib/fuse-dfs/src/Makefile.am` file:
>
> ```
>     fuse_dfs_LDADD=-lfuse -lhdfs -ljvm -lm
> ```

5. Verify the paths in `fuse_dfs_wrapper.sh` and correct them. You may have to change the libhdfs path in the following line as follows:

   ```
   export LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/$OS_ARCH/server:$HADOOP_
   HOME/build/libhdfs/:/usr/local/lib
   ```

6. If it exists, uncomment the `user_allow_other` in `/etc/fuse.conf`.

7. Create a directory as the mount point:

   ```
   >mkdir /u/hdfs
   ```

8. Execute the following command from the `build/contrib/fuse-dfs/` directory. You have to execute this command with root privileges. Make sure that the `HADOOP_HOME` and `JAVA_HOME` environmental variables are set properly in the root environment as well. The optional `-d` parameter enables the debug mode. It would be helpful to run the following command in the debug mode to identify any error when you run it for the first time. The `rw` parameter mounts the filesystem read-write (`ro` for read-only). `-oserver` must point to the NameNode hostname. `-oport` should provide the NameNode port number.

   ```
   >chmod a+x fuse_dfs_wrapper.sh
   ```

   ```
   >./fuse_dfs_wrapper.sh rw -oserver=localhost -oport=9000  /u/hdfs/
   -d
   ```

How it works...

Fuse-DFS is based on the filesystem in user space. The FUSE project (`http://fuse.sourceforge.net/`) makes it possible to implement filesystems in the user space. Fuse-DFS interacts with HDFS filesystem using the libhdfs C API. libhdfs uses JNI to spawn a JVM that communicates with the configured HDFS NameNode.

There's more...

Many instances of HDFS can be mounted on to different directories using the Fuse-DFS as mentioned in the preceding sections.

Building libhdfs

In order to build libhdfs, you must have the following software installed in your system:

- The `ant-nodeps` and `ant-trax` packages
- The `automake` package
- The `Libtool` package
- The `zlib-devel` package
- JDK 1.5—needed in the compile time for Apache Forrest
- Apache Forrest (`http://forrest.apache.org/`)—use the 0.8 release

Compile libhdfs by executing the following command in `$HADOOP_HOME`:

```
>ant compile-c++-libhdfs -Dislibhdfs=1
```

Package the distribution together with `libhdfs` by executing the following command. Provide the path to JDK 1.5 using the `-Djava5.home` property. Provide the path to the Apache Forrest installation using the `-Dforrest.home` property.

```
>ant package -Djava5.home=/u/jdk1.5 -Dforrest.home=/u/apache-forrest-0.8
```

Check whether the `build/libhdfs` directory contains the `libhdfs.*` files. If it doesn't, copy those files to `build/libhdfs` from the `build/c++/<your_architecture>/lib` directory.

```
>cp -R build/c++/<Your_OS_Architecture/lib>/ build/libhdfs
```

See also

- The *HDFS C API* recipe in this chapter.

Merging files in HDFS

This recipe shows how to merge files in HDFS to create a single file. This is useful when retrieving the output of a MapReduce computation with multiple reducers where each reducer produces a part of the output.

How to do it...

1. The HDFS `getMerge` command can copy the files in a given path in HDFS to a single concatenated file in the local filesystem.

   ```
   >bin/hadoop fs -getmerge /user/foo/demofiles merged.txt
   ```

How it works...

The `getmerge` command has the following syntax:

```
hadoopfs -getmerge <src> <localdst> [addnl]
```

The `getmerge` command has three parameters. The first parameter, `<src files>` is the HDFS path to the directory that contains the files to be concatenated. `<dist file>` is the local filename of the merged file. `addnl` is an optional parameter that adds a new line in the result file, after the data from each merged file.

3

Advanced Hadoop MapReduce Administration

In this chapter, we will cover:

- ▸ Tuning Hadoop configurations for cluster deployments
- ▸ Running benchmarks to verify the Hadoop installation
- ▸ Reusing Java VMs to improve the performance
- ▸ Fault tolerance and speculative execution
- ▸ Debug scripts – analyzing task failures
- ▸ Setting failure percentages and skipping bad records
- ▸ Shared-user Hadoop clusters – using fair and other schedulers
- ▸ Hadoop security – integrating with Kerberos
- ▸ Using the Hadoop Tool interface

Introduction

This chapter describes how to perform advanced administration steps for your Hadoop Cluster. This chapter assumes that you have followed *Chapter 1*, *Getting Hadoop Up and Running in a Cluster*, and have installed Hadoop in a clustered or pseudo-distributed setup.

Tuning Hadoop configurations for cluster deployments

Getting ready

Shut down the Hadoop cluster if it is already running, by executing the `bin/stop-dfs.sh` and `bin/stop-mapred.sh` commands from `HADOOP_HOME`.

How to do it...

We can control Hadoop configurations through the following three configuration files:

- ▸ `conf/core-site.xml`: This contains the configurations common to whole Hadoop distribution

- ▸ `conf/hdfs-site.xml`: This contains configurations for HDFS

- ▸ `conf/mapred-site.xml`: This contains configurations for MapReduce

Each configuration file has name-value pairs expressed in an XML format, and they define the workings of different aspects of Hadoop. The following code snippet shows an example of a property in the configuration file. Here, the `<configuration>` tag is the top-level XML container, and the `<property>` tags that define individual properties go as child elements of the `<configuration>` tag.

```
<configuration>
<property>
<name>mapred.reduce.parallel.copies</name>
<value>20</value>
</property>
...
</configuration>
```

The following instructions show how to change the directory to which we write Hadoop logs and configure the maximum number of map and reduce tasks:

1. Create a directory to store the logfiles. For example, `/root/hadoop_logs`.

2. Uncomment the line that includes `HADOOP_LOG_DIR` in `HADOOP_HOME/conf/hadoop-env.sh` and point it to the new directory.

3. Add the following lines to the `HADOOP_HOME/conf/mapred-site.xml` file:

```
<property>
<name>mapred.tasktracker.map.tasks.maximum</name>
<value>2 </value>
</property>
<property>
```

```
<name>mapred.tasktracker.reduce.tasks.maximum</name>
<value>2 </value>
</property>
```

4. Restart the Hadoop cluster by running the `bin/stop-mapred.sh` and `bin/start-mapred.sh` commands from the `HADOOP_HOME` directory.

5. You can verify the number of processes created using OS process monitoring tools. If you are in Linux, run the `watch ps -ef|grep hadoop` command. If you are in Windows or MacOS use the Task Manager.

How it works...

`HADOOP_LOG_DIR` redefines the location to which Hadoop writes its logs. The `mapred.tasktracker.map.tasks.maximum` and `mapred.tasktracker.reduce.tasks.maximum` properties define the maximum number of map and reduce tasks that can run within a single TaskTracker at a given moment.

These and other server-side parameters are defined in the `HADOOP_HOME/conf/*-site.xml` files. Hadoop reloads these configurations after a restart.

There's more...

There are many similar configuration properties defined in Hadoop. You can see some of them in the following tables.

The configuration properties for `conf/core-site.xml` are listed in the following table:

Name	Default value	Description
`fs.inmemory.size.mb`	100	This is the amount of memory allocated to the in-memory filesystem that is used to merge map outputs at reducers in MBs.
`io.sort.factor`	100	This is the maximum number of streams merged while sorting files.
`io.file.buffer.size`	131072	This is the size of the read/write buffer used by sequence files.

The configuration properties for `conf/mapred-site.xml` are listed in the following table:

Name	Default value	Description
`mapred.reduce.parallel.copies`	5	This is the maximum number of parallel copies the reduce step will execute to fetch output from many parallel jobs.
`mapred.map.child.java.opts`	`-Xmx200M`	This is for passing Java options into the map JVM.
`mapred.reduce.child.java.opts`	`-Xmx200M`	This is for passing Java options into the reduce JVM.
`io.sort.mb`	200	The memory limit while sorting data in MBs.

The configuration properties for `conf/hdfs-site.xml` are listed in the following table:

Name	Default value	Description
`dfs.block.size`	67108864	This is the HDFS block size.
`dfs.namenode.handler.count`	40	This is the number of server threads to handle RPC calls in the NameNode.

Running benchmarks to verify the Hadoop installation

The Hadoop distribution comes with several **benchmarks**. We can use them to verify our Hadoop installation and measure Hadoop's performance. This recipe introduces these benchmarks and explains how to run them.

Getting ready

Start the Hadoop cluster. You can run these benchmarks either on a cluster setup or on a pseudo-distributed setup.

How to do it...

Let us run the sort benchmark. The sort benchmark consists of two jobs. First, we generate some random data using the `randomwriter` Hadoop job and then sort them using the sort sample.

1. Change the directory to `HADOOP_HOME`.

2. Run the `randomwriter` Hadoop job using the following command:

   ```
   >bin/hadoop jar hadoop-examples-1.0.0.jarrandomwriter
   -Dtest.randomwrite.bytes_per_map=100
   -Dtest.randomwriter.maps_per_host=10 /data/unsorted-data
   ```

   Here the two parameters, `test.randomwrite.bytes_per_map` and `test.randomwriter.maps_per_host` specify the size of data generated by a map and the number of maps respectively.

3. Run the sort program:

   ```
   >bin/hadoop jar hadoop-examples-1.0.0.jar sort /data/unsorted-data
   /data/sorted-data
   ```

4. Verify the final results by running the following command:

   ```
   >bin/hadoop jar hadoop-test-1.0.0.jar testmapredsort -sortInput /
   data/unsorted-data -sortOutput  /data/sorted-data
   ```

Finally, when everything is successful, the following message will be displayed:

```
The job took 66 seconds.
SUCCESS! Validated the MapReduce framework's 'sort' successfully.
```

How it works...

First, the `randomwriter` application runs a Hadoop job to generate random data that can be used by the second sort program. Then, we verify the results through `testmapredsort` job. If your computer has more capacity, you may run the initial `randomwriter` step with increased output sizes.

There's more...

Hadoop includes several other benchmarks.

- ▸ **TestDFSIO**: This tests the input output (I/O) performance of HDFS
- ▸ **nnbench**: This checks the NameNode hardware
- ▸ **mrbench**: This runs many small jobs
- ▸ **TeraSort**: This sorts a one terabyte of data

More information about these benchmarks can be found at `http://www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench/`.

Reusing Java VMs to improve the performance

In its default configuration, Hadoop starts a new JVM for each map or reduce task. However, running multiple tasks from the same JVM can sometimes significantly speed up the execution. This recipe explains how to control this behavior.

How to do it...

1. Run the WordCount sample by passing the following option as an argument:

   ```
   >bin/hadoop jar hadoop-examples-1.0.0.jar wordcount –Dmapred.job.
   reuse.jvm.num.tasks=-1 /data/input1 /data/output1
   ```

2. Monitor the number of processes created by Hadoop (through `ps –ef|grephadoop` command in Unix or task manager in Windows). Hadoop starts only a single JVM per task slot and then reuses it for an unlimited number of tasks in the job.

   However, passing arguments through the `–D` option only works if the job implements the `org.apache.hadoop.util.Tools` interface. Otherwise, you should set the option through the `JobConf.setNumTasksToExecutePerJvm(-1)` method.

How it works...

By setting the job configuration property through `mapred.job.reuse.jvm.num.tasks`, we can control the number of tasks for the JVM run by Hadoop. When the value is set to `-1`, Hadoop runs the tasks in the same JVM.

Fault tolerance and speculative execution

The primary advantage of using Hadoop is its support for fault tolerance. When you run a job, especially a large job, parts of the execution can fail due to external causes such as network failures, disk failures, and node failures.

When a job has been started, Hadoop JobTracker monitors the TaskTrackers to which it has submitted the tasks of the job. If any TaskTrackers are not responsive, Hadoop will resubmit the tasks handled by unresponsive TaskTracker to a new TaskTracker.

Generally, a Hadoop system may be compose of heterogeneous nodes, and as a result there can be very slow nodes as well as fast nodes. Potentially, a few slow nodes can slow down an execution significantly.

To avoid this, Hadoop supports speculative executions. This means if most of the map tasks have completed and Hadoop is waiting for a few more map tasks, Hadoop JobTracker will start these pending jobs also in a new node. The tracker will use the results from the first task that finishes and stop any other identical tasks.

However, the above model is feasible only if the map tasks are side-effects free. If such parallel executions are undesirable, Hadoop lets users turn off speculative executions.

How to do it...

Run the WordCount sample by passing the following option as an argument to turn off the speculative executions:

```
bin/hadoop jar hadoop-examples-1.0.0.jar wordcount–Dmapred.map.tasks.
speculative.execution=false –D mapred.reduce.tasks.speculative.
execution=true /data/input1 /data/output1
```

However, this only works if the job implements the `org.apache.hadoop.util.Tools` interface. Otherwise, you should set the parameter through `JobConf.set(name, value)`.

How it works...

When the option is specified and set to `false`, Hadoop will turn off the speculative executions. Otherwise, it will perform speculative executions by default.

Debug scripts – analyzing task failures

A Hadoop job may consist of many map tasks and reduce tasks. Therefore, debugging a Hadoop job is often a complicated process. It is a good practice to first test a Hadoop job using unit tests by running it with a subset of the data.

However, sometimes it is necessary to debug a Hadoop job in a distributed mode. To support such cases, Hadoop provides a mechanism called **debug scripts**. This recipe explains how to use debug scripts.

Getting ready

Start the Hadoop cluster. Refer to the *Setting Hadoop in a distributed cluster environment* recipe from *Chapter 1, Getting Hadoop Up and Running in a Cluster*.

How to do it...

A debug script is a shell script, and Hadoop executes the script whenever a task encounters an error. The script will have access to the `$script`, `$stdout`, `$stderr`, `$syslog`, and `$jobconf` properties, as environment variables populated by Hadoop. You can find a sample script from `resources/chapter3/debugscript`. We can use the debug scripts to copy all the logfiles to a single location, e-mail them to a single e-mail account, or perform some analysis.

```
LOG_FILE=HADOOP_HOME/error.log

echo "Run the script" >> $LOG_FILE

echo $script >> $LOG_FILE
echo $stdout>> $LOG_FILE
echo $stderr>> $LOG_FILE
echo $syslog >> $LOG_FILE
echo $jobconf>> $LOG_FILE
```

1. Write your own debug script using the above example. In the above example, edit `HADOOP_HOME` to point to your `HADOOP_HOME` directory.

   `src/chapter3/WordcountWithDebugScript.java` extends the WordCount sample to use debug scripts. The following listing shows the code.

   The following code uploads the job scripts to HDFS and configures the job to use these scripts. Also, it sets up the distributed cache.

```
private static final String scriptFileLocation =
"resources/chapter3/debugscript";
public static void setupFailedTaskScript(JobConfconf)
throws Exception {

// create a directory on HDFS where we'll upload the fail
   //scripts
FileSystemfs = FileSystem.get(conf);
Path debugDir = new Path("/debug");

// who knows what's already in this directory; let's just
   //clear it.
if (fs.exists(debugDir)) {
fs.delete(debugDir, true);
}

// ...and then make sure it exists again
fs.mkdirs(debugDir);
```

```
// upload the local scripts into HDFS
fs.copyFromLocalFile(new Path(scriptFileLocation),
new Path("/debug/fail-script"));

conf.setMapDebugScript("./fail-script");
conf.setReduceDebugScript("./fail-script");

DistributedCache.createSymlink(conf);

URI fsUri = fs.getUri();

String mapUriStr = fsUri.toString()
+ "/debug/fail-script#fail-script";

URI mapUri = new URI(mapUriStr);
DistributedCache.addCacheFile(mapUri, conf);
}
```

The following code runs the Hadoop job as we described in *Chapter 1*, *Getting Hadoop Up and Running in a Cluster*. The only difference is that here, we have called the preceding method to configure failed task scripts.

```
public static void main(String[] args) throws Exception
{
  JobConfconf = new JobConf();
  setupFailedTaskScript(conf);
  Job job = new Job(conf, "word count");
  job.setJarByClass(FaultyWordCount.class);
  job.setMapperClass(FaultyWordCount.TokenizerMapper.class);
  job.setReducerClass(FaultyWordCount.IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  job.waitForCompletion(true);
}
```

2. Compile the code base by running Ant from home directory of the source code. Copy the `build/lib/hadoop-cookbook-chapter3.jar` to `HADOOP_HOME`.

3. Then run the job by running the following command:

```
>bin/hadoopjarhadoop-cookbook-chapter3.jarchapter3.
WordcountWithDebugScript /data/input /data/output1
```

The job will run the `FaultyWordCount` task that will always fail. Then Hadoop will execute the debug script, and you can find the results of the debug script from `HADOOP_HOME`.

How it works...

We configured the debug script through `conf.setMapDebugScript("./fail-script")`. However, the input value is not the file location, but the command that needs to be run on the machine when an error occurs. If you have a specific file that is present in all machines that you want to run when an error occurs, you can just add that path through the `conf. setMapDebugScript("./fail-script")` method.

But, Hadoop runs the mappers in multiple nodes, and often in a machine different than the machine running the job's client. Therefore, for the debug script to work, we need to get the script to all the nodes running the mapper.

We do this using the **distributed cache**. As described in the *Using Distributed cache to distribute resources* recipe in *Chapter 4, Developing Complex Hadoop MapReduce Applications*, users can add files that are in the HDFS filesystem to distribute cache. Then, Hadoop automatically copies those files to each node by running map tasks. However, distributed cache copies the files to `mapred.local.dir` of the MapReduce setup, but it runs the job from a different location. Therefore, we link the cache directory to the working directory by creating a symlink using the `DistributedCache.createSymlink(conf)` command.

Then Hadoop copies the script files to each mapper node and symlinks it to the working directory of the job. When an error occurs, Hadoop will run the `./fail-script` command, which will run the script file that has been copied to the node through distributed cache. The debug script will carry out the tasks you have programmed when an error occurs.

Setting failure percentages and skipping bad records

When processing a large amount of data, there may be cases where a small amount of map tasks will fail, but still the final results make sense without the failed map tasks. This could happen due to a number of reasons such as:

- ▶ Bugs in the map task
- ▶ Small percentage of data records are not well formed
- ▶ Bugs in third-party libraries

In the first case, it is best to debug, find the cause for failures, and fix it. However, in the second and third cases, such errors may be unavoidable. It is possible to tell Hadoop that the job should succeed even if some small percentage of map tasks have failed.

This can be done in two ways:

▸ Setting the failure percentages
▸ Asking Hadoop to skip bad records

This recipe explains how to configure this behavior.

Getting ready

Start the Hadoop setup. Refer to the *Setting Hadoop in a distributed cluster environment* recipe from the *Chapter 1*, *Getting Hadoop Up and Running in a Cluster*.

How to do it...

Run the WordCount sample by passing the following options:

```
>bin/hadoop jar hadoop-examples-1.0.0.jar wordcount
-Dmapred.skip.map.max.skip.records=1
-Dmapred.skip.reduce.max.skip.groups=1 /data/input1 /data/output1
```

However, this only works if the job implements the `org.apache.hadoop.util.Tools` interface. Otherwise, you should set it through `JobConf.set(name, value)`.

How it works...

Hadoop does not support skipping bad records by default. We can turn on bad record skipping by setting the following parameters to positive values:

▸ `mapred.skip.map.max.skip.records`: This sets the number of records to skip near a bad record, including the bad record
▸ `mapred.skip.reduce.max.skip.groups`: This sets the number of acceptable skip groups surrounding a bad group

There's more...

You can also limit the percentage of failures in map or reduce tasks by setting the `JobConf.setMaxMapTaskFailuresPercent(percent)` and `JobConf.setMaxReduceTaskFailuresPercent(percent)` options.

Also, Hadoop repeats the tasks in case of a failure. You can control that through `JobConf.setMaxMapAttempts(5)`.

Shared-user Hadoop clusters – using fair and other schedulers

When a user submits a job to Hadoop, this job needs to be assigned a resource (a computer/host) before execution. This process is called **scheduling**, and a scheduler decides when resources are assigned to a given job.

Hadoop is by default configured with a **First in First out** (**FIFO**) **scheduler**, which executes jobs in the same order as they arrive. However, for a deployment that is running many MapReduce jobs and shared by many users, more complex scheduling policies are needed.

The good news is that Hadoop scheduler is pluggable, and it comes with two other schedulers. Therefore, if required, it is possible to write your own scheduler as well.

 ▸ **Fair scheduler**: This defines pools and over time; each pool gets around the same amount of resources.

 ▸ **Capacity scheduler**: This defines queues, and each queue has a guaranteed capacity. The capacity scheduler shares computer resources allocated to a queue with other queues if those resources are not in use.

This recipe describes how to change the scheduler in Hadoop.

Getting ready

For this recipe, you need a working Hadoop deployment. Set up Hadoop using the *Setting Hadoop in a distributed cluster environment* recipe from *Chapter 1, Getting Hadoop Up and Running in a Cluster*.

How to do it...

1. Shut down the Hadoop cluster.

2. You need `hadoop-fairscheduler-1.0.0.jar` in the `HADOOP_HOME/lib`. However, from Hadoop 1.0.0 and higher releases, this JAR file is in the right place in the Hadoop distribution.

3. Add the following code to the `HADOOP_HOME/conf/mapred-site.xml`:

```
<property>
<name>mapred.jobtracker.taskScheduler</name>
<value>org.apache.hadoop.mapred.FairScheduler</value>
</property>
```

4. Restart Hadoop.

5. Verify that the new scheduler has been applied by going to `http://<job-tracker-host>:50030/scheduler` in your installation. If the scheduler has been properly applied, the page will have the heading "Fair Scheduler Administration".

How it works...

When you follow the preceding steps, Hadoop will load the new scheduler settings when it is started. The fair scheduler shares equal amount of resources between users unless it has been configured otherwise.

The fair scheduler supports users to configure it through two ways. There are several parameters of the `mapred.fairscheduler.*` form, and we can configure these parameters via `HADOOP_HOME/conf/mapred-site.xml`. Also additional parameters can be configured via `HADOOP_HOME/conf/fair-scheduler.xml`. More details about fair scheduler can be found from `HADOOP_HOME/docs/fair_scheduler.html`.

There's more...

Hadoop also includes another scheduler called **capacity scheduler** that provides more fine-grained control than the fair scheduler. More details about the capacity scheduler can be found from `HADOOP_HOME/docs/capacity_scheduler.html`.

Hadoop security – integrating with Kerberos

Hadoop by default runs without security. However, it also supports Kerberos-based setup, which provides full security. This recipe describes how to configure Hadoop with Kerberos for security.
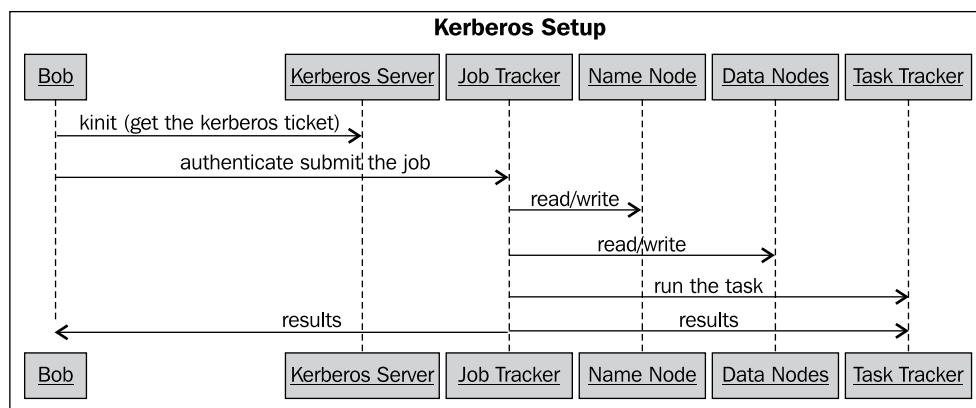
Kerberos setups will include a Hadoop cluster—NameNode, DataNodes, JobTracker, and TaskTrackers—and a Kerberos server. We will define users as **principals** in the Kerberos server. Users can obtain a ticket from the Kerberos server, and use that ticket to log in to any server in Hadoop. We will map each Kerberos principal with a Unix user. Once logged in, the authorization is performed based on the Unix user and group permissions associated with each user.

Getting ready

Set up Hadoop by following *Chapter 1*, *Getting Hadoop Up and Running in a Cluster* either using pseudo-distributed or clustered setup.

We need a machine to use as the Kerberos node for which you have root access. Furthermore, the machine should have the domain name already configured (we will assume DNS name is `hadoop.kbrelam.com`, but you can replace it with another domain). If you want to try this out in a single machine only, you can set up the DNS name through adding your IP address `hadoop.kbrelam.com` to your `/etc/hosts` file.

How to do it...

1. Install Kerberos on your machine. Refer to `http://web.mit.edu/Kerberos/krb5-1.8/krb5-1.8.6/doc/krb5-install.html` for further instructions on setting up Kerberos.

   Provide `hadoop.kbrelam.com` as the realm and the administrative server when installation asks for it. Then run the following command to create a realm:

   `>sudo krb5_newrealm`

2. In Kerberos, we call users "principals". Create a new principal by running following commands:

   `>kadmin.local`

   `>kadmin.local: add principal srinath/admin`

3. Edit `/etc/krb5kdc/kadm5.acl` to include the line `srinath/admin@hadoop.kbrelam.com *` to grant all the permissions.

4. Restart the Kerberos server by running the following command:

   `>sudo /etc/init.d/krb5-admin-server restart.`

5. You can test the new principal by running following commands:

   `>kinitsrinath/admin`

   `>klist`

6. Kerberos will use Unix users in Hadoop machines as Kerberos principals and use local Unix-level user permissions to do authorization. Create the following users and groups with permissions in all the machines on which you plan to run MapReduce.

   We will have three users—`hdfs` to run HDFS server, `mapred` to run MapReduce server, and `bob` to submit jobs.

   ```
   >groupaddhadoop
   >useraddhdfs
   >useraddmapred
   >usermod -g hadoophdfs
   >usermod -g hadoopmapred
   >useradd -G mapred bob
   >usermod -a -G hadoop bob
   ```

7. Now let us create Kerberos principals for these users.

   ```
   >kadmin.local
   >kadmin.local:  addprinc -randkey
   hdfs/hadoop.kbrelam.com
   >kadmin.local:  addprinc –randkey
   mapred/hadoop.kbrelam.com
   >kadmin.local:  addprinc -randkey
   host/hadoop.kbrelam.com
   >kadmin.local:  addprinc -randkey
   bob/hadoop.kbrelam.com
   ```

8. Now, we will create a key tab file that contains credentials for Kerberos principals. We will use these credentials to avoid entering the passwords at Hadoop startup.

   ```
   >kadmin: xst -norandkey -k hdfs.keytab hdfs/hadoop.kbrelam.com
   host/hadoop.kbrelam.com
   >kadmin:  xst -norandkey -k mapred.keytab mapred/hadoop.kbrelam.
   com host/hadoop.kbrelam.com
   >kadmin.local:  xst -norandkey -k bob.keytab bob/hadoop.kbrelam.
   com
   >kadmin.local:  exit
   ```

9. Deploy key tab files by moving them in to the `HADOOP_HOME/conf` directory. Change the directory to `HADOOP_HOME` and run following commands to set the permissions for key tab files:

   ```
   >chownhdfs:hadoopconf/hdfs.keytab
   >chownmapred:hadoopconf/mapred.keytab
   ```

10. Now, set permissions in the filesystem and Hadoop. Change the directory to `HADOOP_HOME` and run the following commands:

```
>chownhdfs:hadoop /opt/hadoop-work/name/
>chownhdfs:hadoop /opt/hadoop-work/data
>chownmapred:hadoop /opt/hadoop-work/local/

>bin/hadoopfs -chownhdfs:hadoop /
>bin/hadoopfs -chmod 755 /
>bin/hadoopfs -mkdir  /mapred
>bin/hadoopfs -mkdir  /mapred/system/
>bin/hadoopfs -chownmapred:hadoop /mapred/system
>bin/hadoopfs -chmod -R 700 /mapred/system
>bin/hadoopfs -chmod 777 /tmp
```

11. Install Unlimited Strength **Java Cryptography Extension** (**JCE**) Policy Files by downloading the policy files from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`and copying the JAR files in the distribution to `JAVA_HOME/jre/lib/security`.

12. Configure Hadoop properties by adding following properties to the associated configuration files. Replace the `HADOOP_HOME` value with the corresponding location. Here, Hadoop will replace the `_HOST` with the localhost name. The following code snippet adds properties to `core-site.xml`:

```
<property>
<name>hadoop.security.authentication</name>
<value>kerberos</value>
</property>
<property>
<name>hadoop.security.authorization</name>
<value>true</value>
</property>
```

13. Copy the configuration parameters defined in `resources/chapter3/kerberos-hdfs-site.xml` of the source code for this chapter to the `HADOOP_HOME/conf/hdfs-site.xml`. Replace the `HADOOP_HOME` value with the corresponding location. Here Hadoop will replace the `_HOST` with the localhost name.

14. Start the NameNode by running the following commands from `HADOOP_HOME`:

```
>sudo -u hdfs bin/hadoopnamenode &
```

15. Test HDFS setup by doing some metadata operations.

    ```
    >kinit hdfs/hadoop.kbrelam.com -k -t conf/hdfs.keytab
    >klist
    >kinit -R
    ```

    In the first command, we specify the name of the principal (for example, `hdfs/hadoop.kbrelam.com`) to apply operations to that principal. The first two commands are theoretically sufficient. However, there is a bug that stops Hadoop from reading the credentials. We can work around this by the last command that rewrites the key in more readable format. Now let's run `hdfs` commands.

    ```
    >bin/hadoopfs -ls /
    ```

16. Start the DataNode (this must be done as the root) by running following command:

    ```
    >su - root
    >cd /opt/hadoop-1.0.3/
    >export HADOOP_SECURE_DN_USER=hdfs
    >export HADOOP_DATANODE_USER=hdfs
    >bin/hadoopdatanode &
    >exit
    ```

17. Configure mapred by adding the following code to `conf/map-red.xml`. Replace `HADOOP_HOME` with the corresponding location.

    ```
    <property>
    <name>mapreduce.jobtracker.kerberos.principal</name>
    <value>mapred/_HOST@hadoop.kbrelam.com</value>
    </property>
    <property>
    <name>mapreduce.jobtracker.kerberos.https.principal</
    name><value>host/_HOST@hadoop.kbrelam.com</value>
    </property>
    <property>
    <name>mapreduce.jobtracker.keytab.file</name>
    <value>HADOOP_HOME/conf/mapred.keytab</value><!-- path to the
    MapReducekeytab -->
    </property><!-- TaskTracker security configs -->
    <property>
    <name>mapreduce.tasktracker.kerberos.principal</name>
    <value>mapred/_HOST@hadoop.kbrelam.com</value>
    </property>
    <property>
    <name>mapreduce.tasktracker.kerberos.https.principal</name>
    <value>host/_HOST@hadoop.kbrelam.com</value>
    ```

```
</property>
<property>
<name>mapreduce.tasktracker.keytab.file</name>
<value>HADOOP_HOME/conf/mapred.keytab</value><!-- path to the
MapReducekeytab -->
</property><!-- TaskController settings -->
<property>
<name>mapred.task.tracker.task-controller</name><value>org.apache.
hadoop.mapred.LinuxTaskController</value>
</property>
<property>
<name>mapreduce.tasktracker.group</name>
<value>mapred</value>
</property>
```

18. Configure the Linux task controller, which must be used for Kerberos setup.

    >mkdir /etc/hadoop

    >cpconf/taskcontroller.cfg /etc/hadoop/taskcontroller.cfg

    >chmod 755 /etc/hadoop/taskcontroller.cfg

19. Add the following code to `/etc/hadoop/taskcontroller.cfg`:

    ```
    mapred.local.dir=/opt/hadoop-work/local/
    hadoop.log.dir=HADOOP_HOME/logs
    mapreduce.tasktracker.group=mapred
    banned.users=mapred,hdfs,bin
    min.user.id=1000
    ```

    Set up the permissions by running the following command from `HADOOP_HOME`, and verify that the final permissions of `bin/task-controller` are `rwsr-x---`. Otherwise, the jobs will fail to execute.

    >chmod 4750 bin/task-controller

    >ls -l bin/task-controller

    >-rwsr-x--- 1 root mapred 63374 May 9 02:05 bin/task-controller

20. Start the JobTracker and TaskTracker:

    >sudo -u mapred bin/hadoopjobtracker

    Wait for the JobTracker to start up and then run the following command:

    >sudo -u mapred bin/hadooptasktracker

21. Run the job by running following commands from `HADOOP_HOME`. If all commands run successfully, you will see the WordCount output as described in *Chapter 1, Getting Hadoop Up and Running in a Cluster*.

```
>su bob
>kinit bob/hadoop.kbrelam.com -k -t conf/bob.keytab
>kinit –R
>bin/hadoopfs -mkdir /data
>bin/hadoopfs -mkdir /data/job1
>bin/hadoopfs -mkdir /data/job1/input
>bin/hadoopfs -put README.txt /data/job1/input


>bin/hadoop jar hadoop-examples-1.0.3.jar wordcount /data/job1 /
data/output
```

How it works...

By running the `kinit` command, the client would obtain a Kerberos ticket and store it in the filesystem. When we run the command, the client uses the Kerberos ticket to get access to the Hadoop nodes and submit jobs. Hadoop resolves the permission based on the user and group permissions of the Linux users that matches the Kerberos principal.

Hadoop Kerberos security settings have many pitfalls. The two tools that might be useful are as follows:

▶ You can enable debugging by adding the environment variable `HADOOP_OPTS="$HADOOP_CLIENT_OPTS -Dsun.security.krb5.debug=true"`

▶ There is a very useful resource that has descriptions for all error codes:

`https://ccp.cloudera.com/display/CDHDOC/Appendix+E+-+Task-controller+Error+Codes`

Also, when you change something, make sure you restart all the processes first by killing all the running processes.

Using the Hadoop Tool interface

Often Hadoop jobs are executed through a command line. Therefore, each Hadoop job has to support reading, parsing, and processing command-line arguments. To avoid each developer having to rewrite this code, Hadoop provides a `org.apache.hadoop.util.Tool` interface.

How to do it...

1. In the source code for this chapter, the `src/chapter3/WordcountWithTools.java` class extends the WordCount example with support for the Tool interface.

```java
public class WordcountWithTools extends
    Configured implements Tool
{
  public int run(String[] args) throws Exception
  {
    if (args.length< 2)
    {
      System.out.println("chapter3.WordCountWithTools
      WordCount<inDir><outDir>");
      ToolRunner.printGenericCommandUsage(System.out);
      System.out.println("");
      return -1;
    }
    Job job = new Job(getConf(), "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.waitForCompletion(true);
    return 0;
  }
  public static void main(String[] args)
    throws Exception
  {
    int res = ToolRunner.run(
        new Configuration(), new WordcountWithTools(), args);
    System.exit(res);
  }
}
```

2. Set up a `input` folder in HDFS with `/data/input/README.txt` if it doesn't already exist. It can be done through following commands:

```
bin/hadoopfs -mkdir /data/output

bin/hadoopfs -mkdir /data/input

bin/hadoopfs -put README.txt /data/input
```

3. Try to run the WordCount without any options, and it will list the available options.

   ```
   bin/hadoop jar hadoop-cookbook-chapter3.jar chapter3.
   WordcountWithToolsWordcount <inDir><outDir>

   Generic options supported are

   -conf<configuration file>      specify an application configuration
   file

   -D <property=value>             use value for given property

   -fs<local|namenode:port>       specify a namenode

   -jt<local|jobtracker:port>     specify a job tracker

   -files<comma separated list of files>    specify comma separated
   files to be copied to the map reduce cluster

   -libjars<comma separated list of jars>    specify comma separated
   jar files to include in the classpath.

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


   The general command line syntax is

   bin/hadoop command [genericOptions] [commandOptions]
   ```

4. Run the WordCount sample with the `mapred.job.reuse.jvm.num.tasks` option to limit the number of JVMs created by the job, as we learned in an earlier recipe.

   ```
   bin/hadoop jar hadoop-cookbook-chapter3.jar

   chapter3.WordcountWithTools

   -D mapred.job.reuse.jvm.num.tasks=1  /data/input /data/output
   ```

How it works...

When a job extends from the `Tool` interface, Hadoop will intercept the command-line arguments, parse the options, and configure the `JobConf` object accordingly. Therefore, the job will support standard generic options.

4
Developing Complex Hadoop MapReduce Applications

In this chapter, we will cover:

- ▶ Choosing appropriate Hadoop data types
- ▶ Implementing a custom Hadoop `Writable` data type
- ▶ Implementing a custom Hadoop `key` type
- ▶ Emitting data of different `value` types from a mapper
- ▶ Choosing a suitable Hadoop `InputFormat` for your input data format
- ▶ Adding support for new input data formats – implementing a custom `InputFormat`
- ▶ Formatting the results of MapReduce computations – using Hadoop `OutputFormat`s
- ▶ Hadoop intermediate (map to reduce) data partitioning
- ▶ Broadcasting and distributing shared resources to tasks in a MapReduce job : Hadoop `DistributedCache`
- ▶ Using Hadoop with legacy applications – Hadoop Streaming
- ▶ Adding dependencies between MapReduce jobs
- ▶ Hadoop counters for reporting custom metrics

Introduction

This chapter introduces you to several advanced Hadoop MapReduce features that will help you to develop highly customized, efficient MapReduce applications.

In this chapter, we will explore the different data types provided by Hadoop and the steps to implement custom data types for Hadoop MapReduce computations. We will also explore the different data input and output formats provided by Hadoop. This chapter will provide you with the basic understanding of how to add support for new data formats in Hadoop. We will also be discussing other advanced Hadoop features such as using `DistributedCache` for distribute data, using Hadoop Streaming for quick prototyping of Hadoop computations, and using Hadoop counters to report custom metrics for your computation as well as adding job dependencies to manage simple DAG-based workflows of Hadoop MapReduce computations.

Choosing appropriate Hadoop data types

Hadoop uses the `Writable` interface based classes as the data types for the MapReduce computations. These data types are used throughout the MapReduce computational flow, starting with reading the input data, transferring intermediate data between Map and Reduce tasks, and finally, when writing the output data. Choosing the appropriate `Writable` data types for your input, intermediate, and output data can have a large effect on the performance and the programmability of your MapReduce programs.

In order to be used as a `value` data type of a MapReduce computation, a data type must implement the `org.apache.hadoop.io.Writable` interface. The `Writable` interface defines how Hadoop should serialize and de-serialize the values when transmitting and storing the data. In order to be used as a `key` data type of a MapReduce computation, a data type must implement the `org.apache.hadoop.io.WritableComparable<T>` interface. In addition to the functionality of the `Writable` interface, the `WritableComparable` interface further defines how to compare the keys of this type with each other for sorting purposes.

Hadoop's Writable versus Java's Serializable

Hadoop's Writable-based serialization framework provides a more efficient and customized serialization and representation of the data for MapReduce programs than using the general-purpose Java's native serialization framework. As opposed to Java's serialization, Hadoop's Writable framework does not write the type name with each object expecting all the clients of the serialized data to be aware of the types used in the serialized data. Omitting the type names makes the serialization process faster and results in compact, random accessible serialized data formats that can be easily interpreted by non-Java clients. Hadoop's Writable-based serialization also has the ability to reduce the object-creation overhead by reusing the `Writable` objects, which is not possible with the Java's native serialization framework.

How to do it...

The following steps show you how to configure the input and output data types of your Hadoop MapReduce application:

1. Specify the data types for the input (key: LongWritable, value: Text) and output (key: Text, value: IntWritable) key-value pairs of your mapper using the generic-type variables.

   ```
   public class SampleMapper extends Mapper<LongWritable, Text, Text,
   IntWritable> {

   public void map(LongWritable key, Text value,
       Context context) … {
   ……    }
   }
   ```

2. Specify the data types for the input (key: Text, value: IntWritable) and output (key: Text, value: IntWritable) key-value pairs of your reducer using the generic-type variables. The reducer's input key-value pair data types should match the mapper's output key-value pairs.

   ```
   public class Reduce extends Reducer<Text, IntWritable, Text,
   IntWritable> {

   public void reduce(Text key,
       Iterable<IntWritable> values, Context context) {
     ……    }
   }
   ```

3. Specify the output data types of the MapReduce computation using the Job object as shown in the following code snippet. These data types will serve as the output types for both, the reducer and the mapper, unless you specifically configure the mapper output types as done in step 4.

   ```
   Job job = new Job(..);
   ….
   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);
   ```

4. Optionally, you can configure the different data types for the mapper's output key-value pairs using the following steps, when your mapper and reducer have different data types for the output key-value pairs.

   ```
   job.setMapOutputKeyClass(Text.class);
   job.setMapOutputValueClass(IntWritable.class);
   ```

There's more…

Hadoop provides several primitive data types such as `IntWritable`, `LongWritable`, `BooleanWritable`, `FloatWritable`, and `ByteWritable`, which are the Writable versions of their respective Java primitive data types. We can use these types as both, the `key` types as well as the `value` types.

The following are several more Hadoop built-in data types that we can use as both, the `key` as well as the `value` types:

- ▸ `Text` : This stores a UTF8 text

- ▸ `BytesWritable`: This stores a sequence of bytes

- ▸ `VIntWritable` and `VLongWritable`: These store variable length integer and long values

- ▸ `NullWritable`: This is a zero-length `Writable` type that can be used when you don't want to use a `key` or `value` type

The following Hadoop build-in collection data types can only be used as `value` types.

- ▸ `ArrayWritable`: This stores an array of values belonging to a `Writable` type. To use `ArrayWritable` type as the `value` type of a reducer's input, you need to create a subclass of `ArrayWritable` to specify the type of the `Writable` values stored in it.

  ```
  public class LongArrayWritable extends ArrayWritable {
    public LongArrayWritable() {
    super(LongWritable.class);
    }
  }
  ```

- ▸ `TwoDArrayWritable`: This stores a matrix of values belonging to the same `Writable` type. To use the `TwoDArrayWritable` type as the `value` type of a reducer's input, you need to specify the type of the stored values by creating a subclass of the `TwoDArrayWritable` type similar to the `ArrayWritable` type.

- ▸ `MapWritable`: This stores a map of key-value pairs. Keys and values should be of the `Writable` data types.

- ▸ `SortedMapWritable`: This stores a sorted map of key-value pairs. Keys should implement the `WritableComparable` interface.

See also

- ▸ *Implementing a custom Hadoop Writable data type*
- ▸ *Implementing a custom Hadoop key type*

Implementing a custom Hadoop Writable data type

There can be use cases where none of the built-in data types matches your requirements or a custom data type optimized for your use case may perform better than a Hadoop built-in data type. In such scenarios, we can easily write a custom `Writable` data type by implementing the `org.apache.hadoop.io.Writable` interface to define the serialization format of your data type. The `Writable` interface-based types can be used as `value` types in Hadoop MapReduce computations.

In this recipe, we implement a sample Hadoop `Writable` data type for HTTP server log entries. For the purpose of this sample, we consider that a log entry consists of the five fields—request host, timestamp, request URL, response size, and the http status code. The following is a sample log entry:

```
192.168.0.2 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/
HTTP/1.0" 200 6245
```

You can download a sample HTTP server log data set from `ftp://ita.ee.lbl.gov/traces/NASA_access_log_Jul95.gz`.

How to do it...

The following are the steps to implement a custom Hadoop Writable data type for the HTTP server log entries:

1. Write a new `LogWritable` class implementing the `org.apache.hadoop.io.Writable` interface.

```
public class LogWritable implements Writable{

  private Text userIP, timestamp, request;
  privateIntWritableresponseSize, status;

  public LogWritable() {
    this.userIP = new Text();
    this.timestamp=  new Text();
    this.request = new Text();
    this.responseSize = new IntWritable();
    this.status = new IntWritable();
  }
  public void readFields(DataInput in) throws IOException {
    userIP.readFields(in);
    timestamp.readFields(in);
    request.readFields(in);
```

```
          responseSize.readFields(in);
          status.readFields(in);
        }

        public void write(DataOutput out) throws IOException {
          userIP.write(out);
          timestamp.write(out);
          request.write(out);
          responseSize.write(out);
          status.write(out);
        }

……… // getters and setters for the fields
}
```

2. Use the new `LogWritable` type as a `value` type in your MapReduce computation. In the following example, we use the `LogWritable` type as the Map output value type.

```
public class LogProcessorMap extends Mapper<LongWritable,
Text, Text, LogWritable> {
….
}

public class LogProcessorReduce extends Reducer<Text,
LogWritable, Text, IntWritable> {

public void reduce(Text key,
Iterable<LogWritable> values, Context context) {
       ……   }
}
```

3. Configure the output types of the job accordingly.

```
Job job = new Job(..);
….
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);job.
setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(LogWritable.class);
```

How it works...

The `Writable` interface consists of the two methods, `readFields()` and `write()`. Inside the `readFields()` method, we de-serialize the input data and populate the fields of the `Writable` object.

```
public void readFields(DataInput in) throws IOException {
  userIP.readFields(in);
  timestamp.readFields(in);
  request.readFields(in);
  responseSize.readFields(in);
  status.readFields(in);
}
```

In the preceding example, we use the `Writable` types as the fields of our custom `Writable` type and use the `readFields()` method of the fields for de-serializing the data from the `DataInput` object. It is also possible to use java primitive data types as the fields of the `Writable` type and to use the corresponding read methods of the `DataInput` object to read the values from the underlying stream, as shown in the following code snippet:

```
int responseSize = in.readInt();
String userIP = in.readUTF();
```

Inside the `write()` method, we write the fields of the `Writable` object to the underlying stream.

```
public void write(DataOutput out) throws IOException {
  userIP.write(out);
  timestamp.write(out);
  request.write(out);
  responseSize.write(out);
  status.write(out);
}
```

In case you are using Java primitive data types as the fields of the `Writable` object, you can use the corresponding write methods of the `DataOutput` object to write the values to the underlying stream as below.

```
out.writeInt(responseSize);
out.writeUTF(userIP);
```

There's more...

Be cautious about the following issues when implementing your custom `Writable` data type:

 ▸ In case you are adding a custom constructor to your custom `Writable` class, make sure to retain the default empty constructor.

▸ `TextOutputFormat` uses the `toString()` method to serialize the `key` and `value` types. In case you are using the `TextOutputFormat` to serialize instances of your custom `Writable` type, make sure to have a meaningful `toString()` implementation for your custom `Writable` data type.

▸ While reading the input data, Hadoop may reuse an instance of the `Writable` class repeatedly. You should not rely on the existing state of the object when populating it inside the `readFields()` method.

See also

▸ *Implementing a custom Hadoop key type*

Implementing a custom Hadoop key type

The instances of Hadoop MapReduce `key` types should have the ability to compare against each other for sorting purposes. In order to be used as a `key` type in a MapReduce a computation, a Hadoop `Writable` data type should implement the `org.apache.hadoop.io.WritableComparable<T>` interface. The `WritableComparable` interface extends the `org.apache.hadoop.io.Writable` interface and adds the `compareTo()` method to perform the comparisons.

In this recipe, we modify the `LogWritable` data type of the *Writing a custom Hadoop Writable data type* recipe to implement the `WritableComparable` interface.

How to do it...

The following are the steps to implement a custom Hadoop `WritableComparable` data type for the HTTP server log entries, which uses the request host name and timestamp for comparison.

1. Modify the `LogWritable` class to implement the `org.apache.hadoop.io.WritableComparable` interface.

```
public class LogWritable implements
    WritableComparable<LogWritable> {

  private Text userIP, timestamp, request;
  private IntWritable responseSize, status;

  public LogWritable() {
    this.userIP = new Text();
    this.timestamp=  new Text();
```

```java
    this.request = new Text();
    this.responseSize = new IntWritable();
    this.status = new IntWritable();
  }

  public void readFields(DataInput in) throws IOException {
    userIP.readFields(in);
    timestamp.readFields(in);
    request.readFields(in);
    responseSize.readFields(in);
    status.readFields(in);
  }

  public void write(DataOutput out) throws IOException {
    userIP.write(out);
    timestamp.write(out);
    request.write(out);
    responseSize.write(out);
    status.write(out);
  }

  public int compareTo(LogWritable o) {
    if (userIP.compareTo(o.userIP)==0){
        return (timestamp.compareTo(o.timestamp));
    }else return (userIP.compareTo(o.userIP);
  }

  public boolean equals(Object o) {
    if (o instanceof LogWritable) {
        LogWritable other = (LogWritable) o;
        return userIP.equals(other.userIP)
                    && timestamp.equals(other.timestamp);
    }
    return false;
  }

  public int hashCode()
  {
    return userIP.hashCode();
  }
   ……… // getters and setters for the fields
}
```

2. You can use the `LogWritable` type as either a `key` type or a `value` type in your MapReduce computation. In the following example, we use the `LogWritable` type as the Map output `key` type.

```
public class LogProcessorMap extends Mapper<LongWritable,
Text, LogWritable,IntWritable> {
…
}

public class LogProcessorReduce extends Reducer<LogWritable,
IntWritable, Text, IntWritable> {

public void reduce(LogWritablekey,
Iterable<IntWritable> values, Context context) {
      ……    }
}
```

3. Configure the output types of the job accordingly.

```
Job job = new Job(..);
…
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
job.setMapOutputKeyClass(LogWritable.class);
job.setMapOutputValueClass(IntWritable.class);
```

How it works...

The `WritableComparable` interface introduces the `comapreTo()` method in addition to the `readFields()` and `write()` methods of the `Writable` interface. The `compareTo()` method should return a negative integer, zero, or a positive integer, if this object is less than, equal to, or greater than the object being compared to respectively. In the `LogWritable` implementation, we consider the objects equal if both the user's IP address and the timestamp are the same. If the objects are not equal, we decide the sort order first based on the user IP address and then based on the timestamp.

```
public int compareTo(LogWritable o) {
  if (userIP.compareTo(o.userIP)==0){
      return (timestamp.compareTo(o.timestamp));
  }else return (userIP.compareTo(o.userIP));
}
```

Hadoop uses `HashPartitioner` as the default **Partitioner** implementation to calculate the distribution of the intermediate data to the reducers. `HashPartitioner` requires the `hashCode()` method of the key objects to satisfy the following two properties:

▸ Provide the same hash value across different JVM instances

▸ Provide a uniform distribution of hash values

Hence, you must implement a stable `hashCode()` method for your custom Hadoop `key` types satisfying the above mentioned two requirements. In the `LogWritable` implementation, we use the hash code of the request hostname/IP address as the hash code of the `LogWritable` instance. This ensures that the intermediate `LogWritable` data will be partitioned based on the request hostname/IP address.

```
public int hashCode()
{
   return userIP.hashCode();
}
```

See also

▸ *Implementing a custom Hadoop Writable data type*

Emitting data of different value types from a mapper

Emitting data products belonging to multiple value types from a mapper is useful when performing reducer-side joins as well as when we need to avoid the complexity of having multiple MapReduce computations to summarize different types of properties in a data set. However, Hadoop reducers do not allow multiple input value types. In these scenarios, we can use the `GenericWritable` class to wrap multiple `value` instances belonging to different data types.

In this recipe, we reuse the HTTP server log entry analyzing sample of the *Implementing a custom Hadoop Writable data type* recipe. However, instead of using a custom data type, in the current recipe we output multiple value types from the mapper. This sample aggregates the total number of bytes served from the web server to a particular host and also outputs a tab-separated list of URLs requested by the particular host. We use `IntWritable` to output the number of bytes from the mapper and `Text` to output the request URL.

How to do it...

The following steps show how to implement a Hadoop `GenericWritable` data type that can wrap instances of either `IntWritable` or `Text` data types.

1. Write a class extending the `org.apache.hadoop.io.GenericWritable` class. Implement the `getTypes()` method to return an array of the `Writable` classes that you will be using. If you are adding a custom constructor, make sure to add a parameter-less default constructor as well.

```
public class MultiValueWritable extends GenericWritable {

  private static Class[] CLASSES =  new Class[]{
    IntWritable.class,
    Text.class
  };

  public MultiValueWritable(){
  }

  public MultiValueWritable(Writable value){
    set(value);
  }

  protected Class[] getTypes() {
    return CLASSES;
  }
}
```

2. Set `MultiValueWritable` as the output value type of the mapper. Wrap the output `Writable` values of the mapper with instances of the `MultiValueWritable` class.

```
public class LogProcessorMap extends
    Mapper<Object, Text, Text, MultiValueWritable> {
  private Text userHostText = new Text();
  private Text requestText = new Text();
  private IntWritableresponseSize = new IntWritable();

  public void map(Object key, Text value,
                              Context context)…{
    ……// parse the value (log entry) using a regex.
    userHostText.set(userHost);
```

```
        requestText.set(request);
        bytesWritable.set(responseSize);

        context.write(userHostText,
          newMultiValueWritable(requestText));
        context.write(userHostText,
          newMultiValueWritable(responseSize));
    }
}
```

3. Set the reducer input value type as `MultiValueWritable`. Implement the
   `reduce()` method to handle multiple value types.

```
public class LogProcessorReduce extends
    Reducer<Text,MultiValueWritable,Text,Text> {
  private Text result = new Text();

    public void reduce(Text key,Iterable<MultiValueWritable>
            values, Context context)…{
    int sum = 0;
    StringBuilder requests = new StringBuilder();
    for (MultiValueWritable multiValueWritable : values) {
    Writable writable = multiValueWritable.get();
    if (writable instanceof IntWritable){
      sum += ((IntWritable)writable).get();
    }else{
      requests.append(((Text)writable).toString());
      requests.append("\t");
    }
  }
result.set(sum + "\t"+requests);
context.write(key, result);
    }
}
```

4. Set `MultiValueWritable` as the Map output value class of this computation.

```
    Configuration conf = new Configuration();
    Job job = new Job(conf, "log-analysis");
    …
    job.setMapOutputValueClass(MultiValueWritable.class);
```

How it works...

The `GenericWritable` implementations should extend `org.apache.hadoop.io.GenericWritable` and should specify a set of the `Writable` value types to wrap, by returning an array of `CLASSES` from the `getTypes()` method. The `GenericWritable` implementations serialize and de-serialize the data using the index to this array of classes.

```
private static Class[] CLASSES =  new Class[]{
  IntWritable.class,
  Text.class
};

protected Class[] getTypes() {
  return CLASSES;
}
```

In the mapper, you wrap each of your values with instances of the `GenericWritable` implementation.

```
private Text requestText = new Text();
context.write(userHostText,
new MultiValueWritable(requestText));
```

The reducer implementation has to take care of the different value types manually.

```
if (writable instanceof IntWritable){
  sum += ((IntWritable)writable).get();
}else{
  requests.append(((Text)writable).toString());
  requests.append("\t");
}
```

There's more...

`org.apache.hadoop.io.ObjectWritable` is another class which can be used to achieve the same objective as `GenericWritable`. The `ObjectWritable` class can handle Java primitive types, strings, and arrays without the need of a Writable wrapper. However, Hadoop serializes the `ObjectWritable` instances by writing the class name of the instance with each serialized entry, making it inefficient compared to a `GenericWritable` class-based implementation.

See also

▶ *Implementing a custom Hadoop Writable data type*

Choosing a suitable Hadoop InputFormat for your input data format

Hadoop supports processing of many different formats and types of data through `InputFormat`. The `InputFormat` of a Hadoop MapReduce computation generates the key-value pair inputs for the mappers by parsing the input data. `InputFormat` also performs the splitting of the input data into logical partitions, essentially determining the number of Map tasks of a MapReduce computation and indirectly deciding the execution location of the Map tasks. Hadoop generates a map task for each logical data partition and invokes the respective mappers with the key-value pairs of the logical splits as the input.

How to do it...

The following steps show you how to use `FileInputFormat` based `KeyValueTextInputFormat` as `InputFormat` for a Hadoop MapReduce computation.

1. In this example, we are going to specify the `KeyValueTextInputFormat` as `InputFormat` for a Hadoop MapReduce computation using the `Job` object as follows:

    ```
    Configuration conf = new Configuration();
    Job job = new Job(conf, "log-analysis");
    ……
    job.SetInputFormat(KeyValueTextInputFormat.class)
    ```

2. Set the input paths to the job.

    ```
    FileInputFormat.setInputPaths(job, new Path(inputPath));
    ```

How it works...

`KeyValueTextInputFormat` is an input format for plain text files, which generates a key-value record for each line of the input text files. Each line of the input data is broken into a key (text) and value (text) pair using a delimiter character. The default delimiter is the tab character. If a line does not contain the delimiter, the whole line will be treated as the key and the value will be empty. We can specify a custom delimiter by setting a property in the job's configuration object as follows, where we use the comma character as the delimiter between the key and value.

```
conf.set("key.value.separator.in.input.line", ",");
```

`KeyValueTextInputFormat` is based on `FileInputFormat`, which is the base class for the file-based InputFormats. Hence, we specify the input path to the MapReduce computation using the `setInputPaths()` method of the `FileInputFormat` class. We have to perform this step when using any InputFormat that is based on the `FileInputFormat` class.

```
FileInputFormat.setInputPaths(job, new Path(inputPath));
```

We can provide multiple HDFS input paths to a MapReduce computation by providing a comma-separated list of paths. You can also use the `addInputPath()` static method of the `FileInputFormat` class to add additional input paths to a computation.

```
public static void setInputPaths(JobConfconf,Path... inputPaths)
public static void addInputPath(JobConfconf, Path path)
```

There's more...

Make sure that your mapper input data types match the data types generated by `InputFormat` used by the MapReduce computation.

The following are some of the `InputFormat` implementations that Hadoop provide to support several common data formats.

- ▶ `TextInputFormat`: This is used for plain text files. `TextInputFormat` generates a key-value record for each line of the input text files. For each line, the key (`LongWritable`) is the byte offset of the line in the file and the value (`Text`) is the line of text. `TextInputFormat` is the default `InputFormat` of Hadoop.

- ▶ `NLineInputFormat`: This is used for plain text files. `NLineInputFormat` splits the input files into logical splits of fixed number of lines. We can use the `NLineInputFormat` when we want our map tasks to receive a fixed number of lines as the input. The key (`LongWritable`) and value (`Text`) records are generated for each line in the split similar to the `TextInputFormat`. By default, `NLineInputFormat` creates a logical split (and a Map task) per line. The number of lines per split (or key-value records per Map task) can be specified as follows. `NLineInputFormat` generates a key-value record for each line of the input text files.

  ```
  NLineInputFormat.setNumLinesPerSplit(job,50);
  ```

- ▶ `SequenceFileInputFormat`: For Hadoop Sequence file input data. Hadoop Sequence files store the data as binary key-value pairs and support data compression. `SequenceFileInputFormat` is useful when using the result of a previous MapReduce computation in Sequence file format as the input of a MapReduce computation.

  - ❑ `SequenceFileAsBinaryInputFormat`: This is a subclass of the `SequenceInputFormat` that presents the key (`BytesWritable`) and the value (`BytesWritable`) pairs in raw binary format

❑ `SequenceFileAsTextInputFormat`: This is a subclass of the `SequenceInputFormat` that presents the key (`Text`) and the value (`Text`) pairs as strings

▶ `DBInputFormat`: This supports reading the input data for MapReduce computation from a SQL table. `DBInputFormat` uses the record number as the key (`LongWritable`) and the query result record as the value (`DBWritable`).

Using multiple input data types and multiple mapper implementations in a single MapReduce application

We can use the `MultipleInputs` feature of Hadoop to run a MapReduce job with multiple input paths, while specifying a different `InputFormat` and (optionally) a mapper for each path. Hadoop will route the outputs of the different mappers to the instances of the single reducer implementation of the MapReduce computation. Multiple inputs with different `InputFormat` implementations is useful when we want to process multiple data sets with the same meaning but are in different input formats (comma-delimited data set and tab-delimited data set).

We can use the following `addInputPath` static method of the `MutlipleInputs` class to add the input paths and the respective input formats to the MapReduce computation.

```
Public static void addInputPath(Job job, Path path,
          Class<?extendsInputFormat>inputFormatClass)
```

The following is an example usage of the preceding method.

```
MultipleInputs.addInputPath(job, path1, CSVInputFormat.class);
MultipleInputs.addInputPath(job, path1, TabInputFormat.class);
```

The multiple inputs feature with both different mappers and `InputFormat` is useful when performing a reduce-side join of two or more data sets.

```
public static void addInputPath(JobConfconf,Path path,
      Class<?extendsInputFormat>inputFormatClass,
      Class<?extends Mapper>mapperClass)
```

The following is an example of using multiple inputs with different input formats and different mapper implementations.

```
MultipleInputs.addInputPath(job, accessLogPath,
      TextInputFormat.class, AccessLogMapper.class);
MultipleInputs.addInputPath(job, userDataPath,
      TextInputFormat.class, UserDataMapper.class);
```

See also

▶ *Adding support for new input data formats– implementing a custom InputFormat*

Adding support for new input data formats – implementing a custom InputFormat

Hadoop enables us to implement and specify custom `InputFormat` implementations for our MapReduce computations. We can implement custom `InputFormat` implementations to gain more control over the input data as well as to support proprietary or application-specific input data file formats as inputs to Hadoop MapReduce computations. A `InputFormat` implementation should extend the `org.apache.hadoop.mapreduce.InputFormat<K,V>` abstract class overriding the `createRecordReader()` and `getSplits()` methods.

In this recipe, we implement a `InputFormat` and a `RecordReader` for the HTTP log files. This `InputFormat` will generate `LongWritable` instances as keys and `LogWritable` instances as the values.

How to do it...

The following are the steps to implement a custom `InputFormat` for the HTTP server log files based on the `FileInputFormat`.

1. `LogFileInputFormat` operates on the data in HDFS files. Hence, we implement the `LogFileInputFormat` extending the `FileInputFormat`.

   ```
   public class LogFileInputFormat extends
           FileInputFormat<LongWritable, LogWritable>{

      public RecordReader<LongWritable, LogWritable>
                     createRecordReader(InputSplit arg0,
                     TaskAttemptContext arg1) throws …… {
        return new LogFileRecordReader();
      }

   }
   ```

2. Implement the `LogFileRecordReader` class.

   ```
   public class LogFileRecordReader extends
       RecordReader<LongWritable, LogWritable>{

      LineRecordReader lineReader;
      LogWritable value;

      public void initialize(InputSplitinputSplit,
                          TaskAttemptContext attempt)…{
        lineReader = new LineRecordReader();
   ```

```
        lineReader.initialize(inputSplit, attempt);
    }

    public boolean nextKeyValue() throws IOException, ..{
        if (!lineReader.nextKeyValue())
            return false;

        String line = lineReader.getCurrentValue().toString();
        ……………//Extract the fields from 'line'using a regex

        value = new LogWritable(userIP, timestamp, request,
            status, bytes);
        return true;
    }

    public LongWritable getCurrentKey() throws..{
        return lineReader.getCurrentKey();
    }

    public LogWritable getCurrentValue() throws ..{
        return value;
    }

    public float getProgress() throws IOException, ..{
        return lineReader.getProgress();
    }

    public void close() throws IOException {
        lineReader.close();
    }
}
```

3. Specify `LogFileInputFormat` as `InputFormat` for the MapReduce computation using the `Job` object as follows. Specify the input paths for the computations using the underlying `FileInputFormat`.

```
Configuration conf = new Configuration();
Job job = new Job(conf, "log-analysis");
……
job.setInputFormatClass(LogFileInputFormat.class);
FileInputFormat.setInputPaths(job, new Path(inputPath));
```

4. Make sure the mappers of the computation use `LongWritable` as the input `key` type and `LogWritable` as the input `value` type.

```
public class LogProcessorMap extends
Mapper<LongWritable, LogWritable, Text, IntWritable>{

    public void map(LongWritablekey,
            LogWritable value, Context context) throws ……{
    ………}
}
```

How it works...

`LogFileInputFormat` extends the `FileInputFormat`, which provides a generic splitting mechanism for HDFS-file based `InputFormat`. We override the `createRecordReader()` method in the `LogFileInputFormat` to provide an instance of our custom `RecordReader` implementation, `LogFileRecordReader`. Optionally, we can also override the `isSplitable()` method of the `FileInputFormat` to control whether the input files are split-up into logical partitions or used as whole files.

```
public RecordReader<LongWritable, LogWritable>
        createRecordReader(InputSplit arg0,
        TaskAttemptContext arg1) throws …… {
    return new LogFileRecordReader();
}
```

The `LogFileRecordReader` class extends the `org.apache.hadoop.mapreduce.RecordReader<K,V>` abstract class and uses `LineRecordReader` internally to perform the basic parsing of the input data. `LineRecordReader` reads lines of text from the input data.

```
        lineReader = new LineRecordReader();
        lineReader.initialize(inputSplit, attempt);
```

We perform the custom parsing of the log entries of the input data in the `nextKeyValue()` method. We use a regular expression to extract the fields out of the HTTP service log entry and populate an instance of the `LogWritable` class using those fields.

```
    public boolean nextKeyValue() throws IOException, ..{
      if (!lineReader.nextKeyValue())
        return false;

      String line = lineReader.getCurrentValue().toString();
      ……………//Extract the fields from 'line' using a regex

      value = new LogWritable(userIP, timestamp, request,
          status, bytes);
      return true;
    }
```

There's more...

We can perform custom splitting of input data by overriding the `getSplits()` method of the `InputFormat` class. The `getSplits()` method should return a list of `InputSplit` objects. A `InputSplit` object represents a logical partition of the input data and will be assigned to a single Map `task`. `InputSplit` classes extend the `InputSplit` abstract class and should override the `getLocations()` and `getLength()` methods. The `getLength()` method should provide the length of the split and the `getLocations()` method should provide a list of nodes where the data represented by this split is physically stored. Hadoop uses a list of data local nodes for Map task scheduling. FileInputFormat we use in the above example uses the `org.apache.hadoop.mapreduce.lib.input.FileSplit` as the `InputSplit` implementation.

You can write `InputFormat` implementations for none HDFS data as well. The `org.apache.hadoop.mapreduce.lib.db.DBInputFormat` is one example of `InputFormat`. `DBInputFormat` supports reading the input data from a SQL table.

See also

▸   *Choosing a suitable Hadoop InputFormat for your input data format*

Formatting the results of MapReduce computations – using Hadoop OutputFormats

Often times the output of your MapReduce computation will be consumed by other applications. Hence, it is important to store the result of a MapReduce computation in a format that can be consumed efficiently by the target application. It is also important to store and organize the data in a location that is efficiently accessible by your target application. We can use Hadoop `OutputFormat` interface to define the data storage format, data storage location and the organization of the output data of a MapReduce computation. A `OutputFormat` prepares the output location and provides a `RecordWriter` implementation to perform the actual serialization and storage of the data.

Hadoop uses the `org.apache.hadoop.mapreduce.lib.output.TextOutputFormat<K,V>` as the default `OutputFormat` for the MapReduce computations. `TextOutputFormat` writes the records of the output data to plain text files in HDFS using a separate line for each record. `TextOutputFormat` uses the tab character to delimit between the key and the value of a record. `TextOutputFormat` extends `FileOutputFormat`, which is the base class for all file-based output formats.

How to do it...

The following steps show you how to use the `FileOutputFormat` based `SequenceFileOutputFormat` as the `OutputFormat` for a Hadoop MapReduce computation.

1. In this example, we are going to specify the `org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat<K,V>` as the `OutputFormat` for a Hadoop MapReduce computation using the `Job` object as follows:

    ```
    Configuration conf = new Configuration();
    Job job = new Job(conf, "log-analysis");
    ……
    job.setOutputFormat(SequenceFileOutputFormat.class)
    ```

2. Set the output paths to the job.

    ```
    FileOutputFormat.setOutputPath(job, new Path(outputPath));
    ```

How it works...

`SequenceFileOutputFormat` serializes the data to Hadoop Sequence files. Hadoop Sequence files store the data as binary key-value pairs and supports data compression. Sequence files are efficient specially for storing non-text data. We can use the Sequence files to store the result of a MapReduce computation, if the output of the MapReduce computation going to be the input of another Hadoop MapReduce computation.

`SequenceFileOutputFormat` is based on the `FileOutputFormat`, which is the base class for the file-based `OutputFormat`. Hence, we specify the output path to the MapReduce computation using the `setOutputPath()` method of the `FileOutputFormat`. We have to perform this step when using any `OutputFormat` that is based on the `FileOutputFormat`.

```
FileOutputFormat.setOutputPath(job, new Path(outputPath));
```

There's more...

You can implement custom `OutputFormat` classes to write the output of your MapReduce computations in a proprietary or custom data format and/or to store the result in storage other than HDFS by extending the `org.apache.hadoop.mapreduce.OutputFormat<K,V>` abstract class. In case your `OutputFormat` implementation stores the data in a filesystem, you can extend from the `FileOutputFormat` class to make your life easier.

Hadoop intermediate (map to reduce) data partitioning

Hadoop partitions the intermediate data generated from the Map tasks across the reduce tasks of the computations. A proper partitioning function ensuring balanced load for each reduce task is crucial to the performance of MapReduce computations. Partitioning can also be used to group together related set of records to specific reduce tasks, where you want the certain outputs to be processed or grouped together.

Hadoop partitions the intermediate data based on the key space of the intermediate data and decides which reduce task will receive which intermediate record. The sorted set of keys and their values of a partition would be the input for a reduce task. In Hadoop, the total number of partitions should be equal to the number of reduce tasks for the MapReduce computation. Hadoop **Partitioners** should extend the `org.apache.hadoop.mapreduce.Partitioner<KEY, VALUE>` abstract class. Hadoop uses `org.apache.hadoop.mapreduce.lib.partition.HashPartitioner` as the default `Partitioner` for the MapReduce computations. **HashPartitioner** partitions the keys based on their `hashcode()`, using the formula *key.hashcode() mod r*, where *r* is the number of reduce tasks. The following diagram illustrates `HashPartitioner` for a computation with two reduce tasks:
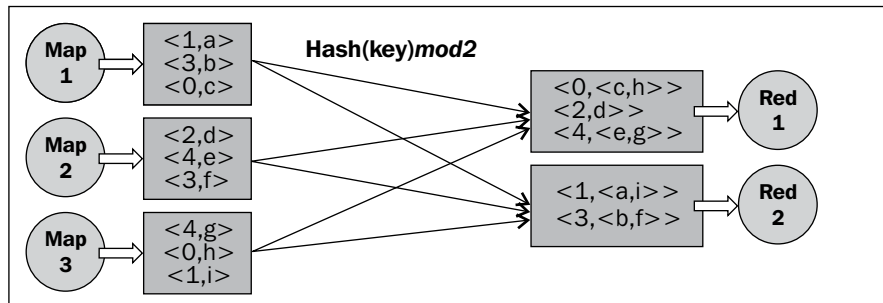
There can be scenarios where our computations logic would require or can be better implemented using an application's specific data-partitioning schema. In this recipe, we implement a custom `Partitioner` for our HTTP log processing application, which partitions the keys (IP addresses) based on their geographic regions.

How to do it...

The following steps show you how to implement a custom `Partitioner` that partitions the intermediate data based on the location of the request IP address or the hostname.

1. Implement the `IPBasedPartitioner` extending the `Partitioner` abstract class.

```
public class IPBasedPartitioner extends Partitioner<Text,
    IntWritable>{

  public int getPartition(Text ipAddress,
            IntWritable value, int numPartitions) {
    String region = getGeoLocation(ipAddress);

    if (region!=null){
              return ((region.hashCode() &
               Integer.MAX_VALUE) % numPartitions);
     }
    return 0;
    }
}
```

2. Set the `Partitioner` class parameter in the `Job` object.

```
Job job = new Job(getConf(), "log-analysis");
……
job.setPartitionerClass(IPBasedPartitioner.class);
```

How it works...

In the above example, we perform the partitioning of the intermediate data, such that the requests from the same geographic region will be sent to the same reducer instance. The `getGeoLocation()` method returns the geographic location of the given IP address. We omit the implementation details of the `getGeoLocation()` method as it's not essential for the understanding of this example. We then obtain the `hashCode()` of the geographic location and perform a modulo operation to choose the reducer bucket for the request.

```
public int getPartition(Text ipAddress,
    IntWritable value, int numPartitions) {
  String region = getGeoLocation(ipAddress);

  if (region!= null && !region.isEmpty())){
      return ((region.hashCode() &
            Integer.MAX_VALUE) % numPartitions);
  }
  return 0;
}
```

There's more...

`TotalOrderPartitioner` and `KeyFieldPartitioner` are two of the several built-in `Partitioner` implementations provided by Hadoop.

TotalOrderPartitioner

TotalOrderPartitioner extends `org.apache.hadoop.mapreduce.lib.partition.` `TotalOrderPartitioner<K,V>`. The set of input records to a reducer are in a sorted order ensuring proper ordering within an input partition. However, the Hadoop default partitioning strategy (`HashPartitioner`) does not enforce an ordering when partitioning the intermediate data and scatters the keys among the partitions. In use cases where we want to ensure a global order, we can use the `TotalOrderPartitioner` to enforce a total order to the reduce input records across the reducer task. `TotalOrderPartitioner` requires a partition file as the input defining the ranges of the `partitions.org.apache.hadoop.` `mapreduce.lib.partition.InputSampler` utility allows us to generate a partition file for the `TotalOrderPartitioner` by sampling the input data. `TotalOrderPartitioner` is used in the Hadoop TeraSort benchmark.

```
Job job = new Job(getConf(), "Sort");
……
job.setPartitionerClass(TotalOrderPartitioner.class);
TotalOrderPartitioner.setPartitionFile(jobConf,partitionFile);
```

KeyFieldBasedPartitioner

`org.apache.hadoop.mapreduce.lib.partition.KeyFieldBasedPartitioner<K,V>` can be used to partition the intermediate data based on parts of the key. A key can be split into a set of fields by using a separator string. We can specify the indexes of the set of fields to be considered when partitioning. We can also specify the index of the characters within fields as well.

Broadcasting and distributing shared resources to tasks in a MapReduce job – Hadoop DistributedCache

We can use the Hadoop DistributedCache to distribute read-only file based resources to the Map and Reduce tasks. These resources can be simple data files, archives or JAR files that are needed for the computations performed by the mappers or the reducers.

How to do it...

The following steps show you how to add a file to the Hadoop `DistributedCache` and how to retrieve it from the Map and Reduce tasks.

1. Copy the resource to the HDFS. You can also use files that are already in the HDFS as well.

   ```
   > bin/hadoop fs -copyFromLocal ip2loc.dat ip2loc.dat
   ```

2. Add the resource to the `DistributedCache` from your driver program.

   ```
   Job job = new Job(getConf(), "log-analysis");
   ……
   DistributedCache.addCacheFile(new
         URI("ip2loc.dat#ip2location"),job.getConfiguration());
   ```

3. Retrieve the resource in the `setup()` method of your mapper or reducer and use the data in the `Map()` or `Reduce()` function.

   ```
   public class LogProcessorMap extends
           Mapper<Object, LogWritable, Text, IntWritable> {
     private IPLookup lookupTable;

     public void setup(Context context) throws IOException{

         File lookupDb = new File("ip2location");
         // Load the IP lookup table to memory
         lookupTable = IPLookup.LoadData(lookupDb);
     }

     public void map(…) {
        String country =
                   lookupTable.getCountry(value.ipAddress);
      ……
     }
   }
   ```

How it works...

Hadoop copies the files added to the `DistributedCache` to all the worker nodes before the execution of any task of the job. `DistributedCache` copies these files only once per the job. Hadoop also supports creating symlinks to the `DistributedCache` files in the working directory of the computation by adding a fragment with the desired symlink name to the URI. In the following example, we are using `ip2location` as the symlink to the `ip2loc.dat` file in the `DistributedCache`.

```
DistributedCache.addCacheFile(new
        URI("/data/ip2loc.dat#ip2location"),
        job.getConfiguration());
```

We parse and load the data from the `DistributedCache` in the `setup()` method of the mapper or the reducer. Files with symlinks are accessible from the working directory using the provided symlink's name.

```
private IPLookup lookup;
public void setup(Context context) throws IOException{

  File lookupDb = new File("ip2location");
  // Load the IP lookup table to memory
  lookup = IPLookup.LoadData(lookupDb);
}

public void map(…) {
  String location =lookup.getGeoLocation(value.ipAddress);

    ……
}
```

We can also access the data in the `DistributedCache` directly using the `getLocalCacheFiles()` method, without using the symlink.

```
Path[] cacheFiles = DistributedCache.getLocalCacheFiles(conf);
```

> `DistributedCache` do not work in the Hadoop local mode.

There's more...

The following sections show you how to distribute the compressed archives using `DistributedCache`, how to add resources to the `DistributedCache` using the |command line and how to use the `DistributedCache` to add resources to the classpath of the mapper and the reducer.

Distributing archives using the DistributedCache

We can use the `DistributedCache` to distribute archives as well. Hadoop extracts the archives in the worker nodes. You also can provide symlinks to the archives using the URI fragments. In the following example, we use the `ip2locationdb` symlink for the `ip2locationdb.tar.gz` archive.

Consider the following MapReduce driver program:

```
Job job = new Job(getConf(), "log-analysis");
DistributedCache.addCacheArchive(
       new URI("/data/ip2locationdb.tar.gz#ip2locationdb"),
       job.getConfiguration());
```

The extracted directory of the archive can be accessible from the working directory of the mapper or the reducer using the above provided symlink.

Consider the following mapper program:

```
public void setup(Context context) throws IOException{
  Configuration conf = context.getConfiguration();

  File lookupDbDir = new File("ip2locationdb");
  String[] children = lookupDbDir.list();


  …
}
```

You can also access the non-extracted `DistributedCache` archived files directly using the following method in the mapper or reducer implementation:

```
Path[] cachePath;

public void setup(Context context) throws IOException{
  Configuration conf = context.getConfiguration();
  cachePath = DistributedCache.getLocalCacheArchives(conf);
    ….
}
```

Adding resources to the DistributedCache from the command line

Hadoop supports adding files or archives to the `DistributedCache` using the command line, provided that your MapReduce driver programs implement the `org.apache.hadoop.util.Tool` interface or utilize the `org.apache.hadoop.util.GenericOptionsParser`. Files can be added to the `DistributedCache` using the `-files` command-line option, while archives can be added using the `-archives` command-line option. Files or archives can be in any filesystem accessible for Hadoop, including your local filesystem. These options support a comma-separated list of paths and the creation of symlinks using the URI fragments.

```
> bin/hadoop jar C4LogProcessor.jar LogProcessor
      -files ip2location.dat#ip2location  indir outdir
> bin/hadoop jar C4LogProcessor.jar LogProcessor
      -archives ip2locationdb.tar.gz#ip2locationdb indir outdir
```

Adding resources to the classpath using DistributedCache

You can use `DistributedCache` to distribute JAR files and other dependent libraries to the mapper or reducer. You can use the following methods in your driver program to add the JAR files to the classpath of the JVM running the mapper or the reducer.

```
public static void addFileToClassPath(
              Path file,Configuration conf,FileSystem fs)


public static void addArchiveToClassPath(
              Path archive,Configuration conf, FileSystem fs)
```

Similar to the `–files` and `–archives` command-line options we describe in *Adding resources to the DistributedCache from the command line* subsection, we can also add the JAR files to the classpath of our MapReduce computations by using the `–libjars` command-line option as well. In order for the `–libjars` command-line option to work, MapReduce driver programs should implement the `Tool` interface or should utilize the `GenericOptionsParser`.

```
> bin/hadoop jar C4LogProcessor.jar LogProcessor
    -libjars ip2LocationResolver.jar  indir outdir
```

See also

- The *Debug scripts – analyzing task failures* recipe in *Chapter 3*, *Advanced Hadoop MapReduce Administration*.

Using Hadoop with legacy applications – Hadoop Streaming

Hadoop Streaming feature allows us to use any executable or a script as the mapper or the reducer of a Hadoop MapReduce job. Hadoop Streaming enables us to perform rapid prototyping of the MapReduce computations using Linux shell utility programs or using scripting languages. Hadoop Streaming also allows the users with some or no Java knowledge to utilize Hadoop to process data stored in HDFS.

In this recipe, we implement a mapper for our HTTP log processing application using Python and use a Hadoop aggregate package based reducer.

How to do it...

The following are the steps to use a Python program as the mapper to process the HTTP server log files.

1. Write the `logProcessor.py` python script.

    ```python
    #!/usr/bin/python
    import sys;
    import re;
    def main(argv):
      regex =re.compile('<regex to parse log entries>');
      line = sys.stdin.readline();
      try:
        while line:
          fields = regex.match(line);
          if(fields!=None):
            print"LongValueSum:"+fields.group(1)+
                                  "\t"+fields.group(7);
          line = sys.stdin.readline();
      except "end of file":
        return None

    if __name__ =="__main__":
      main(sys.argv)
    ```

2. Use the following command from the Hadoop installation directory to execute the Streaming MapReduce computation.

    ```
    > bin/hadoop jar \
        contrib/streaming/hadoop-streaming-1.0.2.jar \
        -input indir \
        -output outdir \
        -mapper logProcessor.py \
        -reducer aggregate \
        -file logProcessor.py
    ```

How it works...

Each Map task launches the Hadoop Streaming executable as a separate process in the worker nodes. The input records (the entries or lines of the log file, not broken in to key value pairs) to the Mapper are provided as lines to the standard input of that process. The executable should read and process the records from the standard input until the end of the file is reached.

```python
line = sys.stdin.readline();
  try:
```

```
    while line:
      ………
    line =sys.stdin.readline();
  except "end of file":
    return None
```

Hadoop Streaming collects the outputs of the executable from the standard output of the process. Hadoop Streaming converts each line of the standard output to a key-value pair, where the text up to the first tab character is considered the key and the rest of the line as the value. The `logProcessor.py` python script outputs the key-value pairs, according to this convention, as follows:

```
If (fields!=None):
    print "LongValueSum:"+fields.group(1)+ "\t"+fields.group(7);
```

In our example, we use the Hadoop **Aggregate** package for the reduction part of our computation. Hadoop aggregate package provides reducer and combiner implementations for simple aggregate operations such as `sum`, `max`, `unique value count`, and `histogram`. When used with the Hadoop Streaming, the mapper outputs must specify the type of aggregation operation of the current computation as a prefix to the output key, which is the `LongValueSum` in our example.

Hadoop Streaming also supports the distribution of files to the worker nodes using the `–file` option. We can use this option to distribute executable files, scripts or any other auxiliary file needed for the Streaming computation. We can specify multiple `–file` options for a computation.

```
> bin/hadoop jar …… \
  -mapper logProcessor.py \
  -reducer aggregate \
  -file logProcessor.py
```

There's more...

We can specify Java classes as the mapper and/or reducer and/or combiner programs of Hadoop Streaming computations. We can also specify `InputFormat` and other options to a Hadoop Streaming computation.

Hadoop Streaming also allows us to use Linux shell utility programs as mapper and reducer as well. The following example shows the usage of `grep` as the mapper of a Hadoop Streaming computation.

```
> bin/hadoop jar
  contrib/streaming/hadoop-streaming-1.0.2.jar \
  –input indir \
  -output outdir \
  -mapper 'grep "wiki"'
```

Hadoop streaming provides the reducer input records of the each key group line by line to the standard input of the process that is executing the executable. However, Hadoop Streaming does not have a mechanism to distinguish when it starts to feed records of a new key to the process. Hence, the scripts or the executables for reducer programs should keep track of the last seen key of the input records to demarcate between key groups.

Extensive documentation on Hadoop Streaming is available at `http://hadoop.apache. org/mapreduce/docs/current/streaming.html`.

See also

  ▸ The *Data extract, cleaning, and format conversion using Hadoop streaming and python* and *Data de-duplication using Hadoop streaming* recipes in *Chapter 7, Mass Data Processing*.

Adding dependencies between MapReduce jobs

Often times we require multiple MapReduce applications to be executed in a workflow-like manner to achieve our objective. Hadoop `ControlledJob` and `JobControl` classes provide a mechanism to execute a simple workflow graph of MapReduce jobs by specifying the dependencies between them.

In this recipe, we execute the `log-grep` MapReduce computation followed by the `log-analysis` MapReduce computation on a HTTP server log data set. The `log-grep` computation filters the input data based on a regular expression. The `log-analysis` computation analyses the filtered data. Hence, the `log-analysis` computation is dependent on the `log-grep` computation. We use the `ControlledJob` to express this dependency and use the `JobControl` to execute the two related MapReduce computations.

How to do it...

The following steps show you how to add a MapReduce computation as a dependency of another MapReduce computation.

  1. Create the `Configuration` and the `Job` objects for the first MapReduce job and populate them with the other needed configurations.

```
Job job1 = new Job(getConf(), "log-grep");
job1.setJarByClass(RegexMapper.class);
job1.setMapperClass(RegexMapper.class);
FileInputFormat.setInputPaths(job1, new Path(inputPath));
FileOutputFormat.setOutputPath(job1, new
                                Path(intermedPath));
……
```

2. Create the `Configuration` and `Job` objects for the second MapReduce job and populate them with the necessary configurations.

```
Job job2 = new Job(getConf(), "log-analysis");
job2.setJarByClass(LogProcessorMap.class);
job2.setMapperClass(LogProcessorMap.class);
job2.setReducerClass(LogProcessorReduce.class);
FileOutputFormat.setOutputPath(job2, new Path(outputPath));
………
```

3. Set the output directory of the first job as the input directory of the second job.

```
FileInputFormat.setInputPaths
       (job2, new Path(intermedPath +"/part*"));
```

4. Create `ControlledJob` objects using the above-created `Job` objects.

```
ControlledJob controlledJob1 =
       new ControlledJob(job1.getConfiguration());
ControlledJob controlledJob2 =
       new ControlledJob(job2.getConfiguration());
```

5. Add the first job as a dependency to the second job.

```
controlledJob2.addDependingJob(controlledJob1);
```

6. Create the `JobControl` object for this group of jobs and add the `ControlledJob` created in step 4 to the newly created `JobControl` object.

```
JobControl jobControl = new
         JobControl("JobControlDemoGroup");
jobControl.addJob(controlledJob1);
jobControl.addJob(controlledJob2);
```

7. Create a new thread to run the group of jobs added to the `JobControl` object. Start the thread and wait for the completion.

```
Thread jobControlThread = new Thread(jobControl);
jobControlThread.start();
while (!jobControl.allFinished()){
  Thread.sleep(500);
}
jobControl.stop();
```

How it works...

The `ControlledJob` class encapsulates MapReduce job and provides the functionality to track the dependencies for the job. A `ControlledJob` class with depending jobs becomes ready for submission only when all of its depending jobs are completed successfully. A `ControlledJob` fails if any of the depending jobs fail.

The `JobControl` class encapsulates a set of `ControlledJob`s and their dependencies. `JobControl` tracks the status of the encapsulated `ControlledJob`s and contains a thread that submits the jobs that are in the *READY* state.

If you want to use the output of a MapReduce job as the input of a dependent job, the input paths to the dependent job has to be set manually. By default, Hadoop generates an output folder per reduce task name with the `part` prefix. We can specify all the `part` prefixed subdirectories as input to the dependent job using wildcards.

```
FileInputFormat.setInputPaths
        (job2, new Path(job1OutPath +"/part*"));
```

There's more...

We can use the `JobControl` class to execute and track a group of non-dependent tasks as well.

Apache **Oozie** is a workflow system for Hadoop MapReduce computations. You can use Oozie to execute **Directed Acyclic Graphs** (**DAG**) of MapReduce computations. You can find more information on Oozie from the project's home page at `http://oozie.apache.org/`.

The `ChainMapper` class, available in the older version of Hadoop MapReduce API, allowed us to execute a pipeline of mapper classes inside a single Map task computation in a pipeline. `ChainReducer` provided the similar support for reduce tasks.

Hadoop counters for reporting custom metrics

Hadoop uses a set of counters to aggregate the metrics for MapReduce computations. Hadoop counters are helpful to understand the behavior of our MapReduce programs and to track the progress of the MapReduce computations. We can define custom counters to track the application specific metrics in MapReduce computations.

How to do it...

The following steps show you how to define a custom counter to count the number of bad or corrupted records in our log processing application.

1. Define the list of custom counters using an enum.

```
public static num LOG_PROCESSOR_COUNTER {
    BAD_RECORDS
    };
```

2. Increment the counter in your mapper or reducer:

```
context.getCounter(LOG_PROCESSOR_COUNTER.BAD_RECORDS).
    increment(1);
```

3. Add the following to your driver program to access the counters:

```
Job job = new Job(getConf(), "log-analysis");
……
Counters counters = job.getCounters();
Counter badRecordsCounter = counters.findCounter(
    LOG_PROCESSOR_COUNTER.BAD_RECORDS);
System.out.println("# of Bad Records:"+
                             badRecordsCounter.getValue());
```

4. Execute your Hadoop MapReduce computation. You can also view the counter values in the admin console or in the command line.

```
> bin/hadoop jar C4LogProcessor.jar \
             demo.LogProcessor in out 1
………
12/07/29 23:59:01 INFO mapred.JobClient: Job complete:
job_201207271742_0020
12/07/29 23:59:01 INFO mapred.JobClient: Counters: 30
12/07/29 23:59:01 INFO mapred.JobClient:    demo.
    LogProcessorMap$LOG_PROCESSOR_COUNTER
12/07/29 23:59:01 INFO mapred.JobClient:    BAD_RECORDS=1406
12/07/29 23:59:01 INFO mapred.JobClient:    Job Counters
………
12/07/29 23:59:01 INFO mapred.JobClient:     Map output
records=112349
# of Bad Records :1406
```

How it works…

You have to define your custom counters using enums. The set of counters in an enum will form a group of counters. The JobTracker aggregates the counter values reported by the mappers and the reducers.

5

Hadoop Ecosystem

In this chapter, we will cover:

- ▶ Installing HBase
- ▶ Data random access using Java client APIs
- ▶ Running MapReduce jobs on HBase (table input/output)
- ▶ Installing Pig
- ▶ Running your first Pig command
- ▶ Set operations (join, union) and sorting with Pig
- ▶ Installing Hive
- ▶ Running a SQL-style queries with Hive
- ▶ Performing a join with Hive
- ▶ Installing Mahout
- ▶ Running K-means with Mahout
- ▶ Visualizing K-means results

Introduction

Hadoop has a family of projects that are either built on top of Hadoop or work very closely with Hadoop. These projects have given rise to an ecosystem that focuses on large-scale data processing, and often users can use several of these projects in combination to handle their use cases. This chapter introduces several key projects in the Hadoop ecosystem and shows how to get started with each project.

We will focus on the following four projects:

- ▶ **HBase**: This is a NoSQL-style highly scalable data storage
- ▶ **Pig**: This is a dataflow-style data processing language for Hadoop jobs
- ▶ **Hive**: This is a SQL-style data processing language for Hadoop
- ▶ **Mahout**: This is a toolkit of machine-learning and data-mining tools

Installing HBase

HBase is a highly scalable NoSQL data store that supports columnar-style data storage. As we will see in the next recipe, it works very closely with Hadoop.
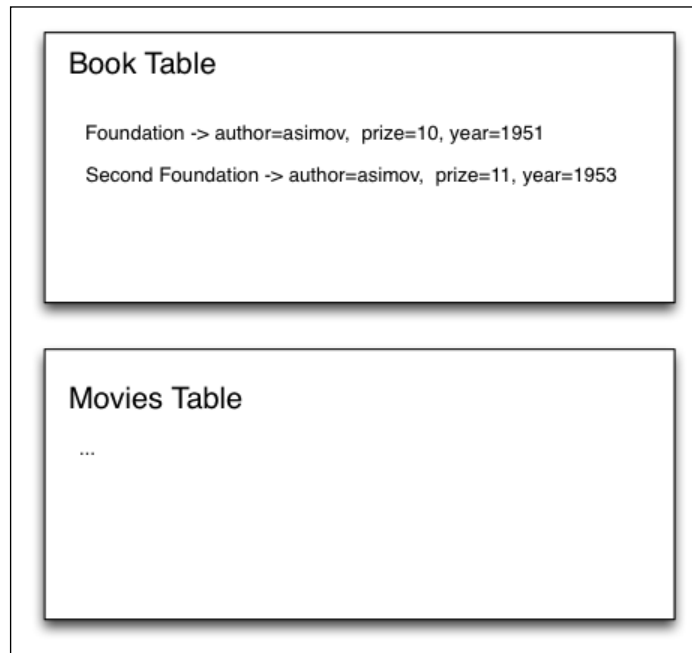
The preceding screenshot depicts the HBase data model. As shown, HBase includes several tables. Each table has zero or more rows where a row consists of a single row ID and multiple name-value pairs. For an example, the first row has the row ID `Foundation`, and several name-value pairs such as `author` with value `asimov`. Although the data model has some similarities with the relational data model, unlike the relational data model, different rows in HBase data model may have different columns. For instance, the second row may contain completely different name-value pairs from the first one. You can find more details about the data model from Google's Bigtable paper `http://research.google.com/archive/bigtable.html`.

Hadoop by default loads data from flat files, and it is a responsibility of the MapReduce job to read and parse the data through data formatters. However, often there are use cases where the data is already in a structured form. Although it is possible to export this data into flat files, parsing and processing the use cases using conventional MapReduce jobs leads to several disadvantages:

- ▶ Processing needs extra steps to convert and export the data
- ▶ Exporting the data needs additional storage
- ▶ Exporting and parsing takes more computing power
- ▶ There arises a need to write specific code to export and parse the data

HBase addresses these concerns by enabling users to read data directly from HBase and write results directly to HBase without having to convert them to flat files.

How to do it...

This section demonstrates how to install HBase.

1. Download HBase 0.94.2 from `http://hbase.apache.org/`.

2. Unzip the distribution by running the following command. We will call the resulting directory `HBASE_HOME`.

   `>tarxfz hbase-0.94.2-SNAPSHOT.tar.gz`

3. Create a data directory to be used by HBase:

   `>cd $HBASE_HOME`

   `>mkdirhbase-data`

4. Add the following to the `HBASE_HOME/conf/hbase-site.xml` file.

   ```
   <configuration>
   <property>
   <name>hbase.rootdir</name>
   <value>file:///Users/srinath/playground/hadoop-book/hbase-0.94.2/
   hbase-data
   </value>
   </property>
   </configuration>
   ```

5. Start the HBase server by running the following command from `HBASE_HOME`:

   `>./bin/start-hbase.sh`

6. Verify the HBase installation by running the shell commands from `HBASE_HOME`:

   ```
   >bin/hbase shell

   HBase Shell; enter 'help<RETURN>' for list of supported commands.

   Type "exit<RETURN>" to leave the HBase Shell

   Version 0.92.1, r1298924, Fri Mar  9 16:58:34 UTC 2012
   ```

7. Create a `test` table and list its content using the following commands:

   ```
   hbase(main):001:0> create 'test', 'cf'

   0 row(s) in 1.8630 seconds


   hbase(main):002:0> list 'test'

   TABLE

   test

   1 row(s) in 0.0180 seconds
   ```

8. Store a value, `row1`, for row ID, column name `test`, and value `val1` to the `test` table using the following commands:

   ```
   hbase(main):004:0> put 'test', 'row1', 'cf:a', 'val1'

   0 row(s) in 0.0680 seconds
   ```

9. Scan the table using the following command. It prints all the data in the table:

   ```
   hbase(main):005:0> scan 'test'

   ROW        COLUMN+CELL

   row1column=cf:a, timestamp=1338485017447, value=val1

   1 row(s) in 0.0320 seconds
   ```

10. Get the value from the table using the following command by giving `row1` as row ID and `test` as the column ID:

    ```
    hbase(main):006:0> get 'test', 'row1'

    COLUMN     CELL

    cf:atimestamp=1338485017447, value=val1

    1 row(s) in 0.0130 seconds


    hbase(main):007:0> exit
    ```

11. The preceding commands verify the HBase installation.

12. When done, finally shut down the HBase by running the following command from the `HBASE_HOME`:

    ```
    > ./bin/stop-hbase.sh

    stoppinghbase..............
    ```

How it works...

The preceding steps configure and run the HBase in the local mode. The server start command starts the HBase server, and HBase shell connects to the server and issues the commands.

There's more...

The preceding commands show how to run HBase in the local mode. The link `http://hbase.apache.org/book/standalone_dist.html#distributed` explains how to run HBase in the distributed mode.

Data random access using Java client APIs

The earlier recipe introduced the command-line interface for HBase. This recipe demonstrates how we can talk to HBase using the Java API.

Getting ready

Install and start HBase as described in the *Installing HBase* recipe.

To compile and run the sample, you would need to have Apache Ant installed in your machine. If Apache Ant has not been installed already, install it by following the instructions given in `http://ant.apache.org/manual/install.html`.

How to do it...

The following steps explain how to connect to HBase via a Java client, store, and retrieve data from the client.

1. Unzip the sample code for this chapter. We will call the new directory `SAMPLE5_DIR`. You can find the Java HBase sample from `SAMPLE5_DIR/src/chapter5/HBaseClient.java`. The client would look like the following. Here, `60000` is the port of HBase and the localhost is the host where HBase master is running. If you connect from a different machine or are running HBase on a different port, you should change these values accordingly.

    ```
    Configuration conf = HBaseConfiguration.create();
    conf.set("hbase.master","localhost:60000");
    HTable table = new HTable(conf, "test");
    ```

2. Store the data in HBase:

    ```
    Put put = new Put("row1".getBytes());
    put.add("cf".getBytes(), "b".getBytes(), "val2".getBytes());
    table.put(put);
    ```

3. Search for data by doing a scan.

    ```
    Scan s = new Scan();
    s.addFamily(Bytes.toBytes("cf"));
    ResultScanner results = table.getScanner(s);
    ```

4. Then let us print the results:

    ```
    try
    {
      for(Result result: results)
      {
        KeyValue[] keyValuePairs = result.raw();
        System.out.println(new String(result.getRow()));
        for(KeyValuekeyValue: keyValuePairs)
        {
          System.out.println(
            new String(keyValue.getFamily()) + " "
            + new String(keyValue.getQualifier()) + "="
            + new String(keyValue.getValue()));
        }
      }
    } finally
    {
    results.close();
    }
    ```

5. Edit the value for the `hbase.home` property in `SAMPLE5_DIR/build.xml`.

6. Compile the Sample by running the following command from `SAMPLE5_DIR`.

    >ant hbase-build

7. Run the sample by running the following command from `SAMPLE5_DIR`.

    >ant hbase-run-javaclient

If all works well, this will print the content of the HBase table to the console.

How it works...

When you run the commands, Ant will run the Java HBase client we had written. It will connect to the HBase server and issue commands to store and search data in HBase storage.

Running MapReduce jobs on HBase (table input/output)

This recipe explains how to run a MapReduce job that reads and writes data directly to and from an HBase storage.

HBase provides abstract mapper and reducer implementations that users can extend to read and write directly from HBase. This recipe explains how to write a sample MapReduce application using these mappers and reducers.

We will use the World Bank's **Human Development Report** (**HDR**) data by country that shows **Gross National Income** (**GNI**) per capita, by countries. The dataset can be found from `http://hdr.undp.org/en/statistics/data/`. Using MapReduce, we will calculate average value for GNI per capita, by countries.

Getting ready

Install and start HBase as described in the *Installing HBase* recipe.

To compile and run the sample, you will need to have Apache Ant installed in your machine. If Apache Ant has not been installed already, install it by following the instructions given at `http://ant.apache.org/manual/install.html`.

How to do it...

This section demonstrates how to run a MapReduce job on data stored in HBase.

1. Unzip the sample code for this chapter. We will call the new directory `SAMPLE5_DIR`.

2. Edit the `hbase.home` value of `SAMPLE5_DIR/build.xml` to point to `HBASE_HOME` of your HBase installation. We will call the Hadoop installation directory as `HADOOP_HOME`.

3. You can find the Java HBase MapReduce sample from `SAMPLE5_DIR/src/chapter5/AverageGINByCountryCalcualtor.java`. The client-side code would look like following:

```
public class AverageGINByCountryCalculator
{
  static class Mapper extends
  TableMapper<ImmutableBytesWritable,
    DoubleWritable>
  {
    privateintnumRecords = 0;
    public void map(ImmutableBytesWritable row,
```

```
      Result values,
      Context context) throws IOException {
    byte[] results = values.getValue(
      "ByCountry".getBytes(),
      "gnip".getBytes());
    ImmutableBytesWritableuserKey = new
    ImmutableBytesWritable("ginp".getBytes());
    try
    {
      context.write(userKey, new
        DoubleWritable(Bytes.toDouble(results)));
    }
    catch (InterruptedException e)
    {
    throw new IOException(e);
    }
    numRecords++;
    if ((numRecords % 50) == 0)
    {
      context.setStatus("mapper processed " +
      numRecords + " records so far");
    }
  }
}
```

HBase provides two classes `TableInputFormat` and `TableOutputFormat` that take off most of the work of reading and writing from an HBase storage. To be used by these classes, the mapper and reducer must extend the `TableMapper` and `TableReducer` classes. When executed, mapper will receive each HBase row as an input.

4. The reducer will use the `Put` construct of the HBase Java API to store the results back to the HBase.

```
public static class Reducer extends
  TableReducer<ImmutableBytesWritable,
  DoubleWritable, ImmutableBytesWritable>
{
  public void reduce(ImmutableBytesWritable key,
    Iterable<DoubleWritable> values, Context context)
    throwsIOException, InterruptedException
  {
    double sum = 0;
    int count = 0;
    for (DoubleWritableval : values)
    {
```

```
        sum += val.get();
        count++;
    }
  Put put = new Put(key.get());
  put.add(Bytes.toBytes("data"),
  Bytes.toBytes("average"),
  Bytes.toBytes(sum / count));
  System.out.println("Processed "+ count +
    " values and avergae =" + sum / count);
  context.write(key, put);
  }
}
```

When running an HBase-based MapReduce job, users should configure from where to read data in HBase and how to write information into HBase via the `TableMapReduceUtilinitTableMapperJob(...)` and `initTableReducerJob(..)` methods.

```
public static void main(String[] args) throws Exception
  {
    Configuration conf = HBaseConfiguration.create();
    Job job = new Job(conf,
      "AverageGINByCountryCalcualtor");
    job.setJarByClass(AverageGINByCountryCalcualtor.class);
    Scan scan = new Scan();
    scan.addFamily("ByCountry".getBytes());
    scan.setFilter(new FirstKeyOnlyFilter());
    TableMapReduceUtil.initTableMapperJob("HDI", scan,
      Mapper.class, ImmutableBytesWritable.class,
      DoubleWritable.class, job);
    TableMapReduceUtil.initTableReducerJob("HDIResult",
    Reducer.class, job);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
```

Here, `initTableMapperJob(..)` instructs Hadoop to read information from the `HDI` table and `initTableReducerJob(..)` instructs Hadoop to write the information to the HBase `HDIResult` table.

5. Run the following command to compile the MapReduce job:

```
>anthbase-build
```

6. Run the following command to upload the data to HBase. (This will use the `HDIDataUploader` to upload the data):

```
>ant hbase-sample1-upload
```

7. Copy the JAR file to `HADOOP_HOME`.

8. Run the MapReduce job by running the following command from `HADOOP_HOME`:

```
>bin/hadoop jar hadoop-cookbook-chapter5.jarchapter5.
AverageGINByCountryCalcualtor
```

9. View the results in HBase by running the following command from the HBase shell. You can start the HBase shell by running `bin/hbaseshell` from `HBASE_HOME`.

```
hbase(main):009:0> scan 'HDIResult'

ROW      COLUMN+CELL

ginpcolumn=data:average, timestamp=1338785279029, value=@\xC8\xF7\
x1Ba2\xA7\x04

1 row(s) in 0.6820 seconds
```

How it works...

When we run the MapReduce job, the `TableMapper` and `TableReducer` classes receive the control. The `TableMapper` class connects to the HBase, reads the data as specified through `initTableMapperJob(…)`, and passes the data directly to the HBase-based mapper that we have written. Here, the `Scan` object we passed into `initTableMapperJob(…)` specifies the search criteria to be used by the mapper when it reads the input data from the HBase.

Similarly, the `TableReducer` lets users emit the data directly to the HBase.

By doing that, `TableMapper` and `TableReducer` build a new programming model based on HBase APIs. With the new programming model, users do not have to worry about parsing and formatting data like with normal MapReduce jobs. The table mapper and reducer map the HBase data to Hadoop name-value pairs and vice versa.

Installing Pig

As we described in the earlier chapters, you can use Hadoop MapReduce interface to program most of the applications. However, if we are writing an application that includes many MapReduce steps, programming them with MapReduce is complicated.

There are several **higher-level programming interfaces** such as Pig and Hive to program parallel applications built on top of MapReduce. We will discuss these two interfaces in the following recipes.

How to do it...

This section demonstrates how to install Pig.

1. Download Pig 0.10.0 from `http://pig.apache.org/releases.html`.

2. Unzip Pig distribution by running the following command. We will call it `PIG_HOME`.

   ```
   > tar xvf pig-0.10.0.tar.gz
   ```

3. To run Pig commands, change the directory to `PIG_HOME` and run the `pig` command. It starts the grunt shell.

   ```
   >cd PIG_HOME
   >bin/pig --help
   >bin/pig-x local
   grunt>
   ```

You can issue the Pig commands from the grunt shell.

How it works...

The preceding instructions set up Pig in the local mode, and you can use the `grunt>` shell to execute the Pig commands.

There's more...

The preceding commands explain how to run Pig in the local mode. The link `http://pig.apache.org/docs/r0.10.0/start.html#Running+the+Pig+Scripts+in+Mapreduce+Mode` explains how to run HBase in the distributed mode.

Running your first Pig command

This recipe runs a basic Pig script. As the sample dataset, we will use **Human Development Report** (**HDR**) data by country. It shows the **Gross National Income** (**GNI**) per capita by country. The dataset can be found from `http://hdr.undp.org/en/statistics/data/`. This recipe will use Pig to process the dataset and create a list of countries that have more than 2000$ of gross national income per capita (GNI) sorted by the GNI value.

How to do it...

This section describes how to use Pig Latin script to find countries with 2000$ GNI sorted by the same criterion from the HDR dataset.

1. From the sample code, copy the dataset from `resources/chapter5/hdi-data.csv` to `PIG_HOME/bin` directory.

2. From the sample code, copy the Pig script `resources/chapter5/countryFilter.pig` to `PIG_HOME/bin`.

3. Open the Pig script through your favorite editor. It will look like the following:

```
A = load 'hdi-data.csv' using PigStorage(',')  AS (id:int,
country:chararray, hdi:float, lifeex:int, mysch:int, eysch:int,
gni:int);
B = FILTER A BY gni> 2000;
C = ORDER B BY gni;
dump C;
```

The first line instructs Pig to load the CSV (comma-separated values) file into the variable A. The `PigStorage(',')` portion tells Pig to load the data using ',' as the separator and assign them to the fields described in the AS clause.

After loading the data, you can process the data using Pig commands. Each Pig command manipulates the data and creates a pipeline of data-processing commands. As each step processes the data and all dependencies are defined as data dependencies, we call Pig a **Dataflow language**.

Finally the `dump` command prints the results to the screen.

4. Run the Pig script by running the following command from `PIG_HOME` directory:

   ```
   >bin/pig-x local bin/countryFilter.pig
   ```

   When executed, the above script will print the following results. As expressed in the script, it will print names of countries that have a GNI value greater than 2000$, sorted by GNI.

   ```
   (126,Kyrgyzstan,0.615,67,9,12,2036)

   (156,Nigeria,0.459,51,5,8,2069)

   (154,Yemen,0.462,65,2,8,2213)

   (138,Lao People's Democratic Republic,0.524,67,4,9,2242)

   (153,Papua New Guinea,0.466,62,4,5,2271)

   (165,Djibouti,0.43,57,3,5,2335)

   (129,Nicaragua,0.589,74,5,10,2430)
   ```

```
(145,Pakistan,0.504,65,4,6,2550)

(114,Occupied Palestinian Territory,0.641,72,8,12,2656)

(128,Viet Nam,0.593,75,5,10,2805)

…
```

How it works...

When we run the Pig script, Pig internally compiles Pig commands to MapReduce jobs in an optimized form and runs it in a MapReduce cluster. Chaining MapReduce jobs using the MapReduce interface is cumbersome, as users will have to write code to pass the output from one job to the other and detect failures. Pig translates such chaining to single-line command and handles the details internally. For complex jobs, the resulting Pig script is easier to write and manage than MapReduce commands that do the same thing.

Set operations (join, union) and sorting with Pig

This recipe explains how to carry out join and sort operations with Pig.

This sample will use two datasets. The first dataset has the **Gross National Income** (**GNI**) per capita by country, and the second dataset has the exports of the country as a percentage of its gross domestic product.

This recipe will use Pig to process the dataset and create a list of countries that have more than 2000$ of gross national income per capita sorted by the GNI value, and then join them with the export dataset.

Getting ready

This recipe needs a working Pig installation. If you have not done it already, follow the earlier recipe and install Pig.

How to do it...

This section will describe how to use Pig to join two datasets.

1. Change the directory to `PIG_HOME`.
2. Copy `resources/chapter5/hdi-data.csv` and `resources/chapter5/ /export-data.csv` to `PIG_HOME/bin`.
3. Copy the `resources/chapter5/countryJoin.pig` script to `PIG_HOME/bin`.

4. Load the script `countryJoin.pig` with your favorite editor. The script
    `countryJoin.pig` joins the HDI data and export data together. Pig calls
    its script "Pig Latin scripts".

    ```
    A = load 'hdi-data.csv' using PigStorage(',')  AS (id:int,
    country:chararray, hdi:float, lifeex:int, mysch:int, eysch:int,
    gni:int);
    B = FILTER A BY gni> 2000;
    C = ORDER B BY gni;
    D = load 'export-data.csv' using PigStorage(',')  AS
    (country:chararray, expct:float);
    E = JOIN C BY country, D by country;
    dump E;
    ```

    The first and forth lines load the data from CSV files. As described in the earlier
    recipe, `PigStorage(',')` asks pig to use `','` as the separator and assigns the
    values to the described fields in the command.

    Then the fifth line joins the two datasets together.

5. Run the Pig Latin script by running the following command from the
    `PIG_HOME` directory.

    ```
    >.bin/pig -x local bin/countryJoin.pig
    (51,Cuba,0.776,79,9,17,5416,Cuba,19.613546)
    (100,Fiji,0.688,69,10,13,4145,Fiji,52.537148)
    (132,Iraq,0.573,69,5,9,3177,Iraq,)
    (89,Oman,0.705,73,5,11,22841,Oman,)
    (80,Peru,0.725,74,8,12,8389,Peru,25.108027)
    (44,Chile,0.805,79,9,14,13329,Chile,38.71985)
    (101,China,0.687,73,7,11,7476,China,29.571701)
    (106,Gabon,0.674,62,7,13,12249,Gabon,61.610462)
    (134,India,0.547,65,4,10,3468,India,21.537624)
    ...
    ```

How it works...

When we run the Pig script, Pig will convert the pig script to MapReduce jobs and execute
them. As described with the Pig Latin script, Pig will load the data from the CSV files, run
transformation commands, and finally join the two data sets.

There's more...

Pig supports many other operations and built-in functions. You can find details about the operations from `http://pig.apache.org/docs/r0.10.0/basic.html` and details about built-in functions from `http://pig.apache.org/docs/r0.10.0/func.html`.

Installing Hive

Just like with Pig, Hive also provides an alternative programming model to write data processing jobs. It allows users to map their data into a relational model and process them through SQL-like commands.

Due to its SQL-style language, Hive is very natural for users who were doing data warehousing using relational databases. Therefore, it is often used as a data warehousing tool.

Getting ready

You need a machine that has Java JDK 1.6 or later version installed.

How to do it...

This section describes how to install Hive.

1. Download Hive 0.9.0 from `http://hive.apache.org/releases.html`.
2. Unzip the distribution by running the following commands.

    > tar xvf hive-0.9.0.tar.gz

3. Download Hadoop 1.0.0 distribution from `http://hadoop.apache.org/common/releases.html`
4. Unzip the Hadoop distribution with the following command.

    > tar xvfhadoop-1.0.0.tar.gz

5. Define the environment variables pointing to Hadoop and Hive distributions.

    >export HIVE_HOME=<hive distribution>

    >export HADOOP_HOME=<hadoopdistribution>

6. Configure Hive by adding the following section to the `conf/hive-site.xml` file.

    ```
    <configuration>
    <property>
    <name>mapred.job.tracker</name>
    <value>local</value>
    </property>
    </configuration>
    ```

7. Delete the `HADOOP_HOME/build` folder to avoid a bug that will cause Hive to fail.

8. Start Hive by running the following commands from `HIVE_HOME`:

   ```
   > cd hive-0.9.0
   ```

   ```
   > bin/hive
   ```

   ```
   WARNING: org.apache.hadoop.metrics.jvm.EventCounter is deprecated.
   Please use org.apache.hadoop.log.metrics.EventCounter in all the
   log4j.properties files.
   ```

   ```
   Logging initialized using configuration in jar:file:/Users/
   srinath/playground/hadoop-book/hive-0.9.0/lib/hive-common-
   0.9.0.jar!/hive-log4j.properties
   ```

   ```
   Hive history file=/tmp/srinath/hive_job_log_
   srinath_201206072032_139699150.txt
   ```

How it works...

The preceding commands will set up Hive, and it will run using the Hadoop distribution as configured in the `HADOOP_HOME`.

Running a SQL-style query with Hive

This recipe explains how you can use Hive to perform data processing operations using its SQL-style language.

In this recipe, we will use a data set that includes **Human Development Report** (**HDR**) by country. HDR describes different countries based on several human development measures. You can find the dataset from `http://hdr.undp.org/en/statistics/data/`.

Getting ready

For this recipe, you need a working Hive installation. If you have not done it already, please follow the previous recipe to install Hive.

How to do it...

This section depicts how to use Hive for filtering and sorting.

1. Copy the `resources/chapter5/hdi-data.csv` file to `HIVE_HOME` directory.

2. Start Hive by changing the directory to `HIVE_HOME` and running the following command:

   ```
   >bin/hive
   ```

3. Let's first define a table to be used to read data, by running the following Hive command.

> The table definition only creates the table layout; it does not put any data into the table.

```
hive> CREATE TABLE HDI(id INT, country STRING, hdi FLOAT, lifeex
INT, mysch INT, eysch INT, gni INT) ROW FORMAT DELIMITED FIELDS
TERMINATED BY ',' STORED AS TEXTFILE;


OK

Time taken: 11.719 seconds
```

4. Let's use the `LOAD` command to load the data to the table. It is worth noting that the `LOAD` command copies the file without any alteration to the storage location of the table as defined by the table definition. Then, it uses the formats defined in the table definition to parse the data and load it to the table. For example, the table definition in step 3 defines a table `HDI` that stores the data as a text file terminated with `','` (CSV format). The input we provide for the `LOAD` command must follow the CSV format as per table definition.

```
hive> LOAD DATA LOCAL INPATH 'hdi-data.csv' INTO TABLE HDI;

Copying data from file:/Users/srinath/playground/hadoop-book/hive-
0.9.0/hdi-data.csv

Copying file: file:/Users/srinath/playground/hadoop-book/hive-
0.9.0/hdi-data.csv

Loading data to table default.hdi

OK

Time taken: 1.447 seconds
```

5. Now we can run the query on the defined table using the Hive SQL-like syntax:

```
hive> SELECT country, gni from HDI WHERE gni> 2000;
```

If the command is successful, Hive will print the following information and finally print the results to the screen.

```
Total MapReduce jobs = 1

Launching Job 1 out of 1

Number of reduce tasks is set to 0 since there's no reduce
operator

Starting Job = job_201206062316_0007, Tracking URL = http://
localhost:50030/jobdetails.jsp?jobid=job_201206062316_0007
```

```
Kill Command = /Users/srinath/playground/hadoop-book/hadoop-1.0.0/
libexec/../bin/hadoop job  -Dmapred.job.tracker=localhost:9001
-kill job_201206062316_0007
Hadoop job information for Stage-1: number of mappers: 1; number
of reducers: 0
2012-06-07 20:45:32,638 Stage-1 map = 0%,  reduce = 0%
2012-06-07 20:45:38,705 Stage-1 map = 100%,  reduce = 0%
2012-06-07 20:45:44,751 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201206062316_0007
MapReduce Jobs Launched:
Job 0: Map: 1   HDFS Read: 9401 HDFS Write: 2435 SUCCESS
Total MapReduce CPU Time Spent: 0 msec
OK
```

The final results will look like following:

```
Norway      47557
Australia     34431
Netherlands     36402
United States     43017
New Zealand     23737
Canada     35166
...
```

How it works...

When we run the Hive, we first define a table and load the data from a file into the table. It is worth noting that the table definition must match the input data file formats, and the LOAD command copies the files into the table's storage location without any change and then tries to parse the file according to the table definitions.

Once the data is loaded, we can use Hive commands to process the data using SQL-like syntax. For example, the following command selects rows from the table that have a GNI value that is more than 2000:

```
SELECT country, gni from HDI WHERE gni> 2000;
```

Performing a join with Hive

This recipe will show how to use Hive to perform joins across two datasets.

The first dataset is the Human Development Report by country. HDR describes different countries based on several human development measures. You can find this dataset at `http://hdr.undp.org/en/statistics/data/`.

This recipe will use Hive to process the dataset and create a list of countries that has more than 2000$ of gross national income per capita, and then join them with export dataset.

Getting ready

This recipe assumes that the earlier recipe has been performed. Install Hive and follow the earlier recipe if you have not done so already.

How to do it...

This section demonstrates how to perform a join using Hive.

1. From the sample directory, copy the `resources/chapter5/export-data.csv` to the `HIVE_HOME` directory.

2. Start Hive by changing the directory to `HIVE_HOME` and running the following command:

   ```
   >bin/hive
   ```

3. We will create a second table to join with the table we loaded in the earlier recipe.

   ```
   hive>  CREATE TABLE EXPO(country STRING, expct FLOAT) ROW FORMAT
   DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;
   OK
   Time taken: 0.758 seconds
   ```

4. We will load the data into the new table by running the following command with Hive. As explained in the earlier recipe, this will move the data to the storage location for the table and parse the data according to the table definition.

   ```
   hive> LOAD DATA LOCAL INPATH 'export-data.csv' INTO TABLE EXPO;

   Copying data from file:/Users/srinath/playground/hadoop-book/hive-
   0.9.0/export-data.csv
   Copying file: file:/Users/srinath/playground/hadoop-book/hive-
   0.9.0/export-data.csv
   Loading data to table default.expo
   OK
   Time taken: 0.391 seconds
   ```

Now we can join the two tables using Hive's SQL-like join command.

```
hive> SELECT h.country, gni, expct FROM HDI h JOIN EXPO e ON
(h.country = e.country) WHERE gni> 2000;
```

If successful it will print the following and print the results to the console:

```
Total MapReduce jobs = 1

Launching Job 1 out of 1

Number of reduce tasks not specified. Estimated from input data
size: 1

In order to change the average load for a reducer (in bytes):

...

2012-06-07 21:19:04,978 Stage-1 map = 0%,  reduce = 0%

2012-06-07 21:19:23,169 Stage-1 map = 50%,  reduce = 0%

..

MapReduce Jobs Launched:

Job 0: Map: 2  Reduce: 1   HDFS Read: 13809 HDFS Write: 2955
SUCCESS

Total MapReduce CPU Time Spent: 0 msec

OK
```

The final result would look like the following:

```
Albania        7803   29.77231

Algeria        7658   30.830406

Andorra        36095  NULL

Angola 4874    56.835884

Antigua and Barbuda 15521  44.08267

Argentina      14527  21.706469

Armenia        5188   20.58361

Australia      34431  19.780243

Austria        35719  53.971355


...

Time taken: 64.856 seconds
```

How it works...

When executed, Hive commands first define and load the second table and data. Then it converts the join command into MapReduce job and carries out the join by running the MapReduce job.

There's more...

Hive supports most SQL commands such as GROUP BY and ORDER BY, with the same semantics as SQL. You can find more details about Hive commands from `https://cwiki.apache.org/confluence/display/Hive/Tutorial`.

Installing Mahout

Hadoop provides a framework for implementing large-scale data processing applications. Often, the users implement their applications on MapReduce from scratch or write their applications using a higher-level programming model such as Pig or Hive.

However, implementing some of the algorithms using MapReduce can be very complex. For example, algorithms such as collaborative filtering, clustering, and recommendations need complex code. This is further agitated by the need to maximize parallel executions.

Mahout is an effort to implement well-known **machine learning** and **data mining** algorithms using MapReduce framework, so that the users can reuse them in their data processing without having to rewrite them from the scratch. This recipe explains how to install Mahout.

How to do it...

This section demonstrates how to install Mahout.

1. Download Mahout from `https://cwiki.apache.org/confluence/display/MAHOUT/Downloads`.

2. Unzip the mahout distribution by running the following command. We will call this folder MAHOUT_HOME.

   ```
   >tar xvf mahout-distribution-0.6.tar.gz
   ```

You can run and verify the Mahout installation by carrying out the following steps:

1. Download the input data from `http://archive.ics.uci.edu/ml/databases/synthetic_control/synthetic_control.data` and copy it to MAHOUT_HOME/testdata.

2. Run the K-mean sample by running the following command:

   ```
   >bin/mahout org.apache.mahout.clustering.syntheticcontrol.kmeans.Job
   ```

   If all goes well, it will process and print out the clusters:

   ```
   12/06/19 21:18:15 INFO kmeans.Job: Running with default arguments
   12/06/19 21:18:15 INFO kmeans.Job: Preparing Input
   ```

```
12/06/19 21:18:15 WARN mapred.JobClient: Use GenericOptionsParser
for parsing the arguments. Applications should implement Tool for
the same.

.....

2/06/19 21:19:38 INFO clustering.ClusterDumper: Wrote 6 clusters

12/06/19 21:19:38 INFO driver.MahoutDriver: Program took 83559 ms
(Minutes: 1.39265)
```

How it works...

Mahout is a collection of MapReduce jobs and you can run them using the `mahout` command. The preceding instructions installed and verified Mahout by running a **K-means** sample that comes with the Mahout distribution.

Running K-means with Mahout

K-means is a clustering algorithm. A clustering algorithm takes data points defined in an **N-dimensional space**, and groups them into multiple **clusters** considering the distance between those data points. A cluster is a set of data points such that the distance between the data points inside the cluster is much less than the distance from data points within the cluster to data points outside the cluster. More details about the K-means clustering can be found from the lecture 4 (`http://www.youtube.com/watch?v=1ZDybXl212Q`) of the *Cluster computing and MapReduce* lecture series by Google.

In this recipe, we will use a data set that includes Human Development Report (HDR) by country. HDR describes different countries based on several human development measures. You can find the data set from `http://hdr.undp.org/en/statistics/data/`.

This recipe will use K-means to cluster countries based on the HDR dimensions.

Getting ready

This recipe needs a Mahout installation. Follow the previous recipe if you have not already done so earlier.

How to do it...

This section demonstrates how to use Mahout K-means algorithm to process with a dataset.

1. Unzip the sample code distribution. We will call this `SAMPLE5_DIR`.
2. Add the `MAHOUT_HOME` to the `mahout.home` property of build.xml file in the sample distribution.

3. The `chapter5.KMeanSample.java` class shows a sample code for running the K-means algorithm using our own dataset.

```
public final class KMean extends AbstractJob {
```

The following code initializes the K-means algorithm with right values

```
public static void main(String[] args) throws Exception
{
  Path output = new Path("output");
  Configuration conf = new Configuration();
  HadoopUtil.delete(conf, output);
  run(conf, new Path("testdata"), output,
    newEuclideanDistanceMeasure(), 6, 0.5, 10);
}
```

The following code shows how to set up K-means from Java code:

```
public static void run(Configuration conf, Path input,
Path output,DistanceMeasure measure, int k, double
convergenceDelta, intmaxIterations)
throws Exception{
    Path directoryContainingConvertedInput = new Path(output,
        DIRECTORY_CONTAINING_CONVERTED_INPUT);
log.info("Preparing Input");
InputDriver.runJob(input,
directoryContainingConvertedInput,
        "org.apache.mahout.math.RandomAccessSparseVector");
log.info("Running random seed to get initial clusters");
    Path clusters = new Path(output,
Cluster.INITIAL_CLUSTERS_DIR);
clusters = RandomSeedGenerator.buildRandom(conf,
directoryContainingConvertedInput, clusters,
k, measure);
log.info("Running KMeans");
KMeansDriver.run(conf, directoryContainingConvertedInput,
clusters, output,
measure, convergenceDelta, maxIterations, true, false);
    // run ClusterDumper
ClusterDumperclusterDumper = new ClusterDumper(
finalClusterPath(conf,
        output, maxIterations),
new Path(output, "clusteredPoints"));
clusterDumper.printClusters(null);
  }
  ...
}
```

4. Compile the sample by running the following command:

```
>ant mahout-build
```

5. From samples, copy the file `resources/chapter5/countries4Kmean.data` to the `MAHOUT_HOME/testdata` directory.

6. Run the sample by running the following command.

```
>ant kmeans-run
```

How it works...

The preceding sample shows how you can configure and use K-means implementation from Java. When we run the code, it initializes the K-means MapReduce job and executes it using the MapReduce framework.

Visualizing K-means results

This recipe explains how you can visualize the results of a K-means run.

Getting ready

This recipe assumes that you have followed the earlier recipe, have run K-means, and have access to the output of the K-means algorithm. If you have not already done so, follow the previous recipe to run K-means.

How to do it...

This section demonstrates how to convert output of the K-means execution to GraphML and visualize it.

1. Running the following command will print the results into **GraphML** format, which is a standard representation of graphs. Here, replace the `<k-means-output-dir>` with the output directory of the k-mean execution.

   ```
   >bin/mahout clusterdump --seqFileDir<k-means-output-dir>/
   clusters-10-final/ --pointsDir<k-means-output-dir>/clusteredPoints
   --outputFormat GRAPH_ML -o clusters.graphml
   ```

2. Download and install Gephi graph visualization toolkit from `http://gephi.org/`.

3. Open the `MAHOUT_HOME/clusters.graphml` file using `File->Open` menu of the Gephi.

4. From the layout window at the lower-left corner of the screen, use YufanHu's multilevel as the layout method, and click on **Run**.

5. Gephi will show a visualization of the graph that looks like the following:
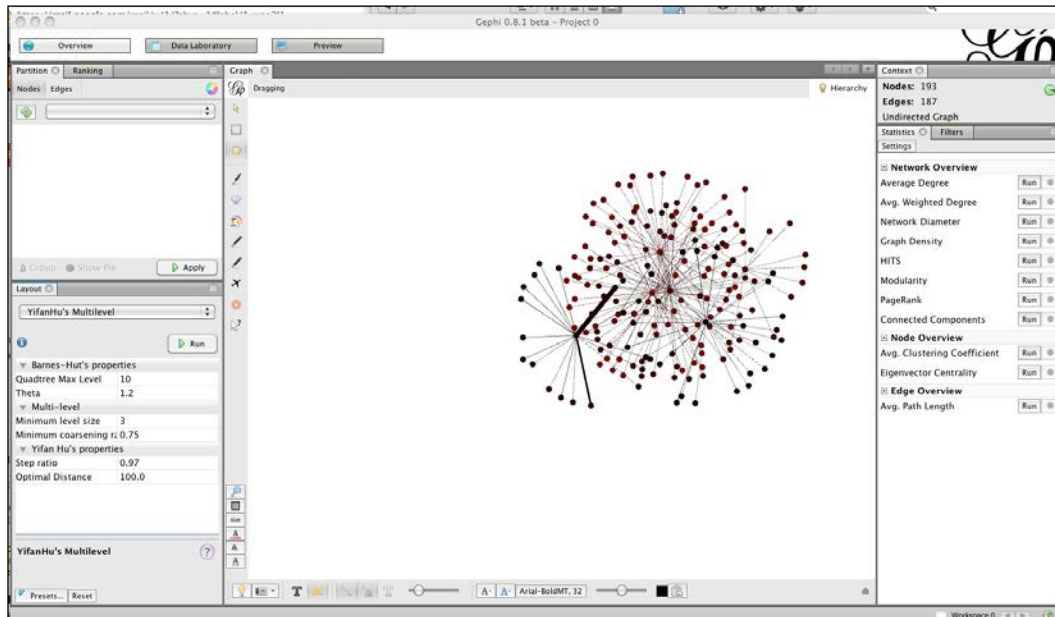
How it works...

K-means output is written as a sequence file. We can use the `clusterdump` command of the Mahout to write them as a GraphML file, which is a standard representation of the graph. Then, we used Gephi graph visualization software to visualize the resulting GraphML file.

6
Analytics

In this chapter, we will cover:

- ▶ Simple analytics using MapReduce
- ▶ Performing Group-By using MapReduce
- ▶ Calculating frequency distributions and sorting using MapReduce
- ▶ Plotting the Hadoop results using GNU Plot
- ▶ Calculating histograms using MapReduce
- ▶ Calculating scatter Plots using MapReduce
- ▶ Parsing a complex dataset with Hadoop
- ▶ Joining two datasets using MapReduce

Introduction

This chapter discusses how we can process a dataset and understand its basic characteristics. We will cover more complex methods like data mining, classification, and so on, in later chapters.

Following are a few instances of basic analytics:

- ▶ Calculating Minimum, Maximum, Mean, Median, Standard deviation, and so on of a dataset. Given a dataset, generally there are multiple dimensions (for example, while processing HTTP access logs, names of the web page, the size of the web page, access time, and so on). We can measure the mentioned analytics using one or more dimensions. For example, we can group the data into multiple groups and calculate the mean value in each case.
- ▶ Histograms used in finding out how many occurrences happen within different value ranges (for example, how many hits happen within each 6-hour period).

▶ Frequency distributions used in finding out how many occurrences of a value happened (for example, how many hits were received by each web page in a site).

▶ Finding a correlation between two dimensions (for example, correlation between access count and the file size of web accesses).

▶ Hypothesis testing, that is, trying to verify or disprove a hypothesis using a given dataset.

This chapter will show how you can calculate basic analytics using a given dataset. For recipes in this chapter, we will use two datasets:

▶ NASA weblog dataset available from `http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html` is a real-life dataset collected using the requests received by NASA web servers.

▶ Apache tomcat developer list e-mail archives available from `http://mail-archives.apache.org/mod_mbox/tomcat-users/`, which is in MBOX format.

Simple analytics using MapReduce

Aggregative values (for example, Mean, Max, Min, standard deviation, and so on) provide the basic analytics about a dataset. You may perform these calculations, either for the whole dataset or a part of the dataset.

In this recipe, we will use Hadoop to calculate the minimum, maximum, and average size of a file downloaded from the NASA servers, by processing the NASA weblog dataset. The following figure shows a summary of the execution:
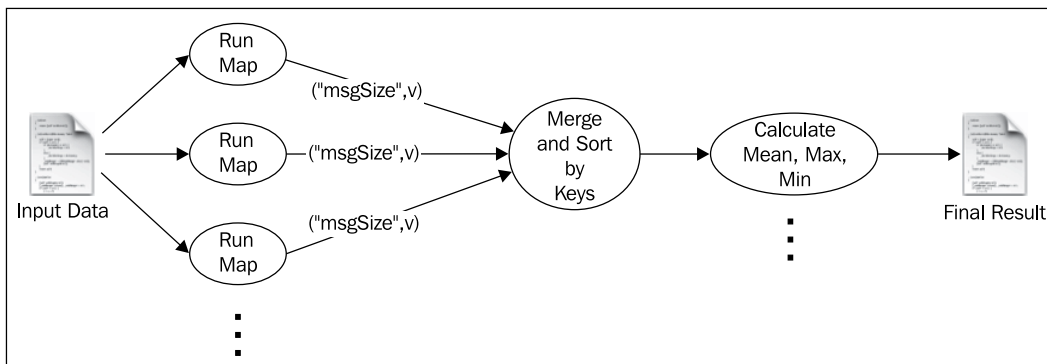
As shown in the figure, mapper task will emit all message sizes under the key **msgSize**, and they are all sent to a one-reducer job. Then the reducer will walk through all of the data and will calculate the aggregate values.

Getting ready

▸ This recipe assumes that you have followed the first chapter and have installed Hadoop. We will use HADOOP_HOME to refer to the Hadoop installation folder.

▸ Start Hadoop by following the instructions in the first chapter.

▸ This recipe assumes that you are aware of how Hadoop processing works. If you have not already done so, you should follow the recipe *Writing a WordCount MapReduce sample, bundling it and running it using standalone Hadoop* from *Chapter 1, Getting Hadoop Up and Running in a Cluster*.

How to do it...

The following steps describe how to use MapReduce to calculate simple analytics about the weblog dataset:

1. Download the weblog dataset from `ftp://ita.ee.lbl.gov/traces/NASA_ access_log_Jul95.gz` and unzip it. We call the extracted folder as DATA_DIR.

2. Upload the data to HDFS by running the following commands from HADOOP_HOME. If /data is already there, clean it up:

    >bin/hadoopdfs -mkdir /data

    > bin/hadoopdfs -mkdir /data/input1

    > bin/hadoopdfs -put <DATA_DIR>/NASA_access_log_Jul95 /data/input1

3. Unzip the source code of this chapter (`chapter6.zip`). We will call that folder CHAPTER_6_SRC.

4. Change the `hadoop.home` property in the CHAPTER_6_SRC/build.xml file to point to your Hadoop installation folder.

5. Compile the source by running the `ant build` command from the CHAPTER_6_SRC folder.

6. Copy the `build/lib/hadoop-cookbook-chapter6.jar` to your HADOOP_HOME.

7. Run the MapReduce job through the following command from HADOOP_HOME:

    >bin/hadoop jar hadoop-cookbook-chapter6.jar chapter6. WebLogMessageSizeAggregator/data/input1 /data/output1

8. Read the results by running the following command:

    $bin/hadoopdfs -cat /data/output1/*

    You will see that it will print the results as following:

    Mean   1150

    Max   6823936

    Min   0

How it works...

You can find the source for the recipe from `src/chapter6/WebLogMessageSizeAggregator.java`.

HTTP logs follow a standard pattern where each log looks like the following. Here the last token includes the size of the web page retrieved:

```
205.212.115.106 - - [01/Jul/1995:00:00:12 -0400] "GET /shuttle/countdown/
countdown.html HTTP/1.0" 200 3985
```

We will use the Java regular expressions' support to parse the log lines, and the `Pattern.compile()` method in the top of the class defines the regular expression. Since most Hadoop jobs involve text processing, regular expressions are a very useful tool while writing Hadoop Jobs:

```
private final static IntWritable one = new IntWritable(1);
public void map(Object key, Text value,
  Context context) throws
  IOException, InterruptedException
{
  Matcher matcher = httplogPattern.matcher(value.
    toString());
    if (matcher.matches())
    {
      int size = Integer.parseInt(matcher.group(5));
      context.write(new Text("msgSize"),one);
    }
}
```

The map task receives each line in the log file as a different key-value pair. It parses the lines using regular expressions and emits the file size against the key `msgSize`.

Then, Hadoop collects all values for the key and invokes the reducer. Reducer walks through all the values and calculates the minimum, maximum, and mean file size of the file downloaded from the web server. It is worth noting that by making the values available as an iterator, Hadoop gives the programmer a chance to process the data without storing them in memory. You should therefore try to process values without storing them in memory whenever possible.

```
public static class AReducer
  extends Reducer<Text, IntWritable, Text, IntWritable>
  {
  public void reduce(Text key, Iterable<IntWritable> values,
  Context context) throws IOException,InterruptedException
  {
    double tot = 0;
    int count = 0;
```

```
      int min = Integer.MAX_VALUE;
      int max = 0;
      Iterator<IntWritable> iterator = values.iterator();
      while (iterator.hasNext())
      {
        int value = iterator.next().get();
        tot = tot + value;
        count++;
        if (value < min)
        {
          min = value;
        }
        if (value > max)
        {
          max = value;
        }
      }
      context.write(new Text("Mean"),
      new IntWritable((int) tot / count));
      context.write(new Text("Max"),
        new IntWritable(max));
      context.write(new Text("Min"),
        new IntWritable(min));
    }
  }
```

The `main()` method of the job looks similar to the WordCount example, except for the highlighted lines that has been changed to accommodate the input and output datatype changes:

```
Job job = new Job(conf, "LogProcessingMessageSizeAggregation");
job.setJarByClass(WebLogMessageSizeAggregator.class);
job.setMapperClass(AMapper.class);
job.setReducerClass(AReducer.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

There's more...

You can learn more about Java regular expressions from the Java tutorial, `http://docs.oracle.com/javase/tutorial/essential/regex/`.

Performing Group-By using MapReduce

This recipe shows how we can use MapReduce to group data into simple groups and calculate the analytics for each group. We will use the same HTTP log dataset. The following figure shows a summary of the execution:
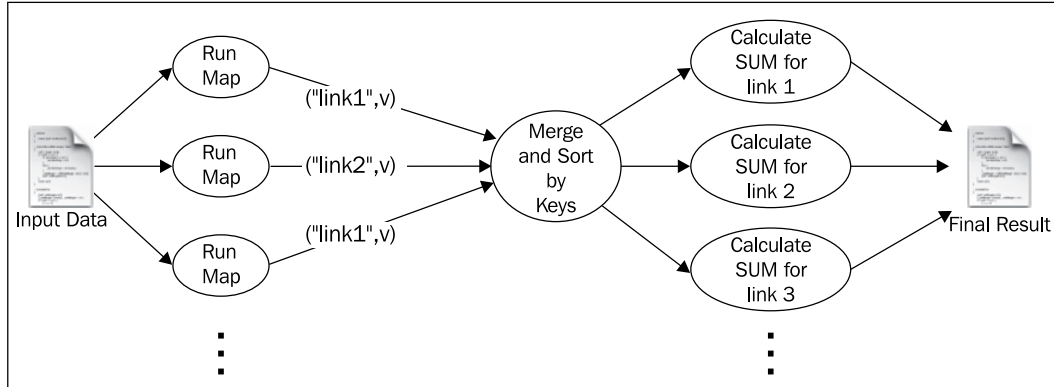
As shown in the figure, the mapper task groups the occurrence of each link under different keys. Then, Hadoop sorts the keys and provides all values for a given key to a reducer, who will count the number of occurrences.

Getting ready

- ▶ This recipe assumes that you have followed the first chapter and have installed Hadoop. We will use the HADOOP_HOME to refer to the Hadoop installation folder.

- ▶ Start Hadoop following the instructions in the first chapter.

- ▶ This recipe assumes that you are aware of how Hadoop processing works. If you have not already done so, you should follow the recipe *Writing a WordCount MapReduce sample, bundling it and running it using standalone Hadoop* from *Chapter 1, Getting Hadoop Up and Running in a Cluster*.

How to do it...

The following steps show how we can group weblog data and calculate analytics.

1. Download the weblog dataset from `ftp://ita.ee.lbl.gov/traces/NASA_ access_log_Jul95.gz` and unzip it. We will call this `DATA_DIR`.

2. Upload the data to HDFS by running the following commands from `HADOOP_HOME`. If `/data` is already there, clean it up:

   ```
   >bin/hadoopdfs -mkdir /data
   >bin/hadoopdfs -mkdir /data/input1
   >bin/hadoopdfs -put <DATA_DIR>/NASA_access_log_Jul95 /data/input1
   ```

3. Unzip the source code of this chapter (`chapter6.zip`). We will call that folder `CHAPTER_6_SRC`.

4. Change the `hadoop.home` property in the `CHAPTER_6_SRC/build.xml` file to point to your Hadoop installation folder.

5. Compile the source by running the `ant build` command from the `CHAPTER_6_SRC` folder.

6. Copy the `build/lib/hadoop-cookbook-chapter6.jar` to `HADOOP_HOME`.

7. Run the MapReduce job using the following command from `HADOOP_HOME`:

   ```
   >bin/hadoop jar hadoop-cookbook-chapter6.jar chapter6.
   WeblogHitsByLinkProcessor/data/input1 /data/output2
   ```

8. Read the results by running the following command:

   ```
   >bin/hadoopdfs -cat /data/output2/*
   ```

   You will see that it will print the results as following:

   ```
   /base-ops/procurement/procurement.html   28
   /biomed/                                 1
   /biomed/bibliography/biblio.html         7
   /biomed/climate/airqual.html             4
   /biomed/climate/climate.html             5
   /biomed/climate/gif/f16pcfinmed.gif      4
   /biomed/climate/gif/f22pcfinmed.gif      3
   /biomed/climate/gif/f23pcfinmed.gif      3
   /biomed/climate/gif/ozonehrlyfin.gif     3
   ```

How it works...

You can find the source for the recipe from `src/chapter6/ WeblogHitsByLinkProcessor.java`.

As described in the earlier recipe, we will use regular expressions to parse HTTP logs. In the following sample the log line `/shuttle/countdown/countdown.html` shows the link (URL) being retrieved.

```
205.212.115.106 - - [01/Jul/1995:00:00:12 -0400] "GET /shuttle/countdown/
countdown.html HTTP/1.0" 200 3985
```

The following code segment shows the mapper:

```
public void map(Object key, Text value,
  Context context) throws IOException,
  InterruptedException
{
  Matcher matcher = httplogPattern.matcher(value.toString());
  if(matcher.matches())
  {
    String linkUrl = matcher.group(4);
    word.set(linkUrl);
    context.write(word, one);
  }
}
```

Map task receives each line in the log file as a different key-value pair. It parses the lines using regular expressions and emits the link as the key, and number `one` as the value.

Then, Hadoop collects all values for different keys (link) and invokes the reducer once for each link. Then each Reducer counts the number of hits for each link.

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

The `main()` method of the job works similar to the earlier recipe.

Calculating frequency distributions and sorting using MapReduce

Frequency distribution is the number of hits received by each URL sorted in the ascending order, by the number hits received by a URL. We have already calculated the number of hits in the earlier recipe. This recipe will sort the list.

Getting ready

▶ This recipe assumes that you have followed the first chapter and have installed Hadoop. We will use the HADOOP_HOME to refer to Hadoop installation folder.

▶ Start Hadoop by following the instructions in the first chapter.

▶ This recipe assumes that you are aware of how Hadoop processing works. If you have not already done so, you should follow the recipe *Writing a WordCount MapReduce sample, bundling it and running it using standalone Hadoop* from *Chapter 1, Getting Hadoop Up and Running in a Cluster*.

▶ This recipe will use the results from the recipe *Performing Group-By using MapReduce* of this chapter. Follow it if you have not done so already.

How to do it...

The following steps show how to calculate frequency distribution using MapReduce:

1. We will use the data from the previous recipe here. So follow the recipe if you have not already done so.

2. Run the MapReduce job using the following command from HADOOP_HOME:

   ```
   > bin/hadoop jar hadoop-cookbook-chapter6.jar chapter6.
   WeblogFrequencyDistributionProcessor/data/output2 /data/output3
   ```

3. Read the results by running the following command:

   ```
   >bin/hadoopdfs -cat /data/output3/*
   ```

   You will see that it will print the results as following:

   ```
   /cgi-bin/imagemap/countdown?91,175       12
   /cgi-bin/imagemap/countdown?105,143      13
   /cgi-bin/imagemap/countdown70?177,284    14
   ```

How it works...

The second recipe of this chapter calculated the number of hits received by each link, and the frequency distribution as a sorted list of those results in that recipe. Therefore, let us sort the results of the second recipe.

MapReduce always sorts the key-value pairs emitted by the mappers by their keys before delivering them to the reducers. We will use this to sort the results.

You can find the source for the recipe from `src/chapter6/WeblogFrequencyDistributionProcessor.java`.

Map task for the job will look like the following:

```
public static class AMapper extends Mapper<Object,
  Text, IntWritable, Text>
{
  public void map(Object key, Text value, Context context) throws
    IOException, InterruptedException
  {
  String[] tokens = value.toString().split("\\s");
  context.write(
    new IntWritable(Integer.parseInt(tokens[1])),
    new Text(tokens[0]));
  }
}
```

Map task receives each line in the log file as a different key-value pair. It parses the lines using regular expressions and emits the number of hits as the key and the URL name as the value. Hadoop sorts the key-value pairs emitted by the mapper before calling the reducers, and therefore the reducer will receive the pairs in sorted order. Hence, it just has to emit them as they arrive.

```
public static class AReducer extends
  Reducer<IntWritable, Text, Text, IntWritable>
{
  public void reduce(IntWritable key, Iterable<Text> values,
    Context context) throws IOException, InterruptedException
  {
    Iterator<Text> iterator = values.iterator();
    if (iterator.hasNext())
    {
      context.write(iterator.next(), key);
    }
  }
}
```

The `main()` method of the job will work similar to the one in the earlier recipe.

Plotting the Hadoop results using GNU Plot

Although Hadoop jobs can generate interesting analytics, making sense of those results and getting a detailed understanding about the data often require us to see the overall trends in the data. We often do that by plotting the data.

The human eye is remarkably good at detecting patterns, and plotting the data often yields us a deeper understanding of the data. Therefore, we often plot the results of Hadoop jobs using some plotting program.

This recipe explains how to use GNU Plot, which is a free and powerful plotting program, to plot Hadoop results.

Getting ready

> ▸ This recipe assumes that you have followed the previous recipe, *Calculating frequency distributions and sorting using MapReduce*. If you have not done so, please follow the recipe.

> ▸ We will use the `HADOOP_HOME` variable to refer to the Hadoop installation folder.

> ▸ Install the GNU Plot plotting program by following the instructions in `http://www.gnuplot.info/`.

How to do it...

The following steps show how to plot Hadoop job results using GNU Plot.

1. Download the results of the last recipe to a local computer by running the following command from `HADOOP_HOME`:

   ```
   > bin/hadoopdfs -get /data/output3/part-r-00000 2.data
   ```

2. Copy all the `*.plot` files from `CHAPTER_6_SRC` to `HADOOP_HOME`.

3. Generate the plot by running the following command from `HADOOP_HOME`:

   ```
   >gnuplot httpfreqdist.plot
   ```

4. It will generate a file called `freqdist.png`, which will look like the following:
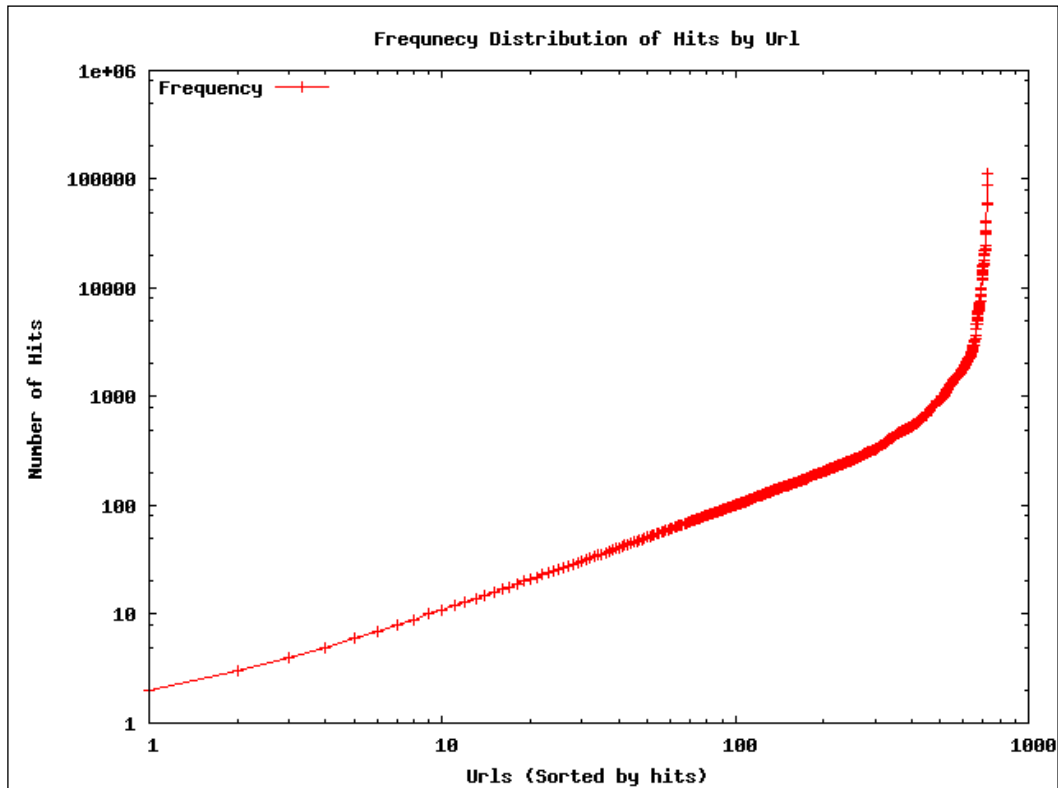
The preceding plot is plotted in log-log scale, and the first part of the distribution follows the **zipf** (power law) distribution, which is a common distribution seen in the web. The last few most popular links have much higher rates than expected from a zipf distribution.

Discussion about more details on this distribution is out of scope of this book. However, this plot demonstrates the kind of insights we can get by plotting the analytical results. In most of the future recipes, we will use the GNU plot to plot and to analyze the results.

How it works...

The following steps describe how plotting with GNU plot works:

1. You can find the source for the GNU plot file from `src/chapter6/resources/httpfreqdist.plot`. The source for the plot will look like the following:

```
set terminal png
set output "freqdist.png"

set title "Frequnecy Distribution of Hits by Url";
set ylabel "Number of Hits";
set xlabel "Urls (Sorted by hits)";
set key left top
set log y
set log x

plot"2.data" using 2 title "Frequency" with linespoints
```

2. Here the first two lines define the output format. This example uses PNG, but GNU plot supports many other terminals like SCREEN, PDF, EPS, and so on.

3. Next four lines define the axis labels and the title.

4. Next two lines define the scale of each axis, and this plot uses log scale for both.

5. Last line defines the plot. Here it is asking GNU plot to read the data from the `2.data` file, and use the data in the second column of the file via `using 2` and to plot it using lines. Columns must be separated by whitespaces.

6. Here if you want to plot one column against other, for example, data from column `1` against column `2`, you should write `using 1:2` instead of `using 2`.

There's more...

You can learn more about GNU plot from `http://www.gnuplot.info/`.

Calculating histograms using MapReduce

Another interesting view into a dataset is a **histogram**. Histogram makes sense only under a continuous dimension (for example, access time and file size). It groups the number of occurrences of some event into several groups in the dimension. For example, in this recipe, if we take the access time from weblogs as the dimension, then we will group the access time by the hour.

The following figure shows a summary of the execution. Here the mapper calculates the hour of the day and emits the "hour of the day" and **1** as the key and value respectively. Then each reducer receives all the occurrences of one hour of a day, and calculates the number of occurrences:
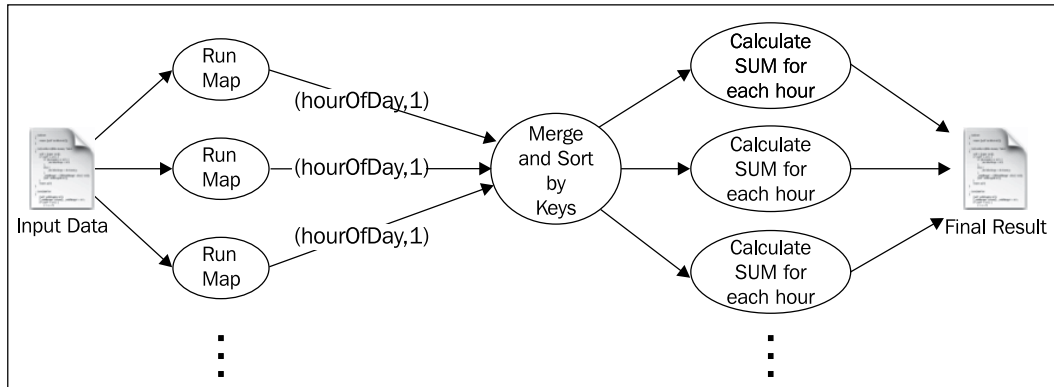
Getting ready

- ▸ This recipe assumes that you have followed the first chapter and have installed Hadoop. We will use the HADOOP_HOME variable to refer to the Hadoop installation folder.

- ▸ Start Hadoop by following the instructions in the first chapter.

- ▸ This recipe assumes that you are aware of how Hadoop processing works. If you have not already done so, you should follow the recipe *Writing a WordCount MapReduce sample, bundling it and running it using standalone Hadoop* from *Chapter 1, Getting Hadoop Up and Running in a Cluster*.

How to do it...

The following steps show how to calculate and plot a Histogram:

1. Download the weblog dataset from `ftp://ita.ee.lbl.gov/traces/NASA_access_log_Jul95.gz` and unzip it. We will call this `DATA_DIR`.

2. Upload the data to HDFS by running the following commands from `HADOOP_HOME`. If `/data` is already there, clean it up:

```
>bin/hadoopdfs -mkdir /data
>bin/hadoopdfs -mkdir /data/input1
>bin/hadoopdfs -put <DATA_DIR>/NASA_access_log_Jul95 /data/input1
```

3. Unzip the source code of this chapter (`chapter6.zip`). We will call that folder `CHAPTER_6_SRC`.

4. Change the `hadoop.home` property in the `CHAPTER_6_SRC/build.xml` file to point to your Hadoop installation folder.

5. Compile the source by running the `ant build` command from the `CHAPTER_6_SRC` folder.

6. Copy the `build/lib/hadoop-cookbook-chapter6.jar` file to `HADOOP_HOME`.

7. Run the MapReduce job through the following command from `HADOOP_HOME`:

   > **bin/hadoop jar hadoop-cookbook-chapter6.jar chapter6.**
   WeblogTimeOfDayHistogramCreator/data/input1 /data/output4

8. Read the results by running the following command:

   >**bin/hadoopdfs -cat /data/output4/***

9. Download the results of the last recipe to a local computer by running the following command from `HADOOP_HOME`:

   > **bin/hadoopdfs -get /data/output4/part-r-00000 3.data**

10. Copy all the `*.plot` files from `CHAPTER_6_SRC` to `HADOOP_HOME`.

11. Generate the plot by running the following command from `HADOOP_HOME`:

   >**gnuplot httphistbyhour.plot**

12. It will generate a file called `hitsbyHour.png`, which will look like following:
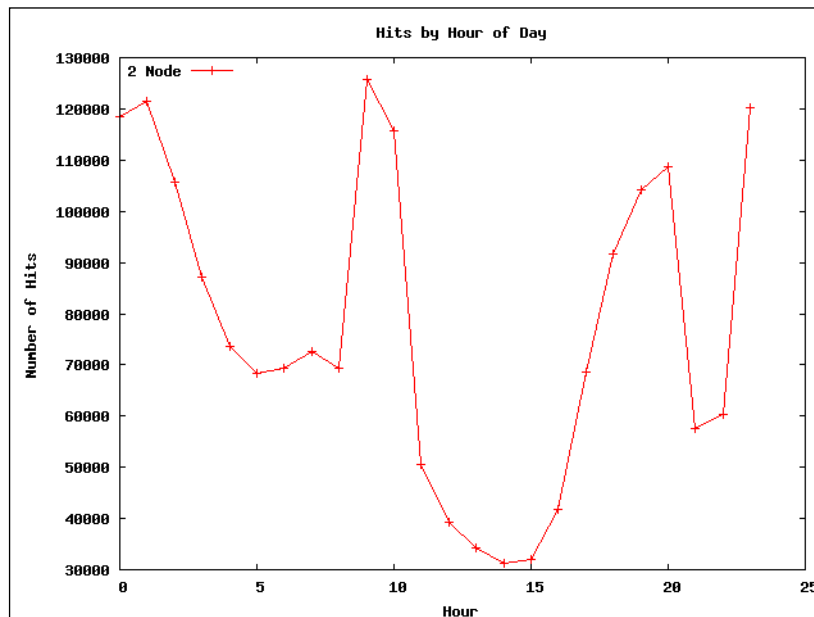
As you can see from the figure, most of the access to NASA is at night, whereas there noontime. Also, two peaks roughly follow the tea times.

How it works...

You can find the source for the recipe from `src/chapter6/ WeblogTimeOfDayHistogramCreator.java`. As explained in the first recipe of this chapter, we will use regular expressions to parse the log file and extract the access time from the log files.

The following code segment shows the mapper function:

```java
public void map(Object key, Text value,
  Context context) throws IOException, InterruptedException
{
  Matcher matcher = httplogPattern.matcher(value.toString());
  if (matcher.matches())
  {
    String timeAsStr = matcher.group(2);
    Date time = dateFormatter.parse(timeAsStr);
    Calendar calendar = GregorianCalendar.getInstance();
    calendar.setTime(time);
    int hours = calendar.get(Calendar.HOUR_OF_DAY);
    context.write(new IntWritable(hours), one);
  }
}
```

Map task receives each line in the log file as a different key-value pair. It parses the lines using regular expressions and extracts the access time for each web page access. Then, the mapper function extracts the hour of the day from the access time and emits the hour of the day and `one` as output of the mapper function.

Then, Hadoop collects all key-value pairs, sorts them, and then invokes the reducer once for each key. Each reducer walks through the values and calculates the count of page accesses for each hour.

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

The `main()` method of the job looks similar to the WordCount example as described in the earlier recipe.

Calculating scatter plots using MapReduce

Another useful tool while analyzing data is a **Scatter plot**. We use Scatter plot to find the relationship between two measurements (dimensions). It plots the two dimensions against each other.

For an example, this recipe analyzes the data to find the relationship between the size of the web pages and the number of hits received by the web page.

The following figure shows a summary of the execution. Here, the mapper calculates and emits the message size (rounded to 1024 bytes) as the key and `one` as the value. Then the reducer calculates the number of occurrences for each message size:
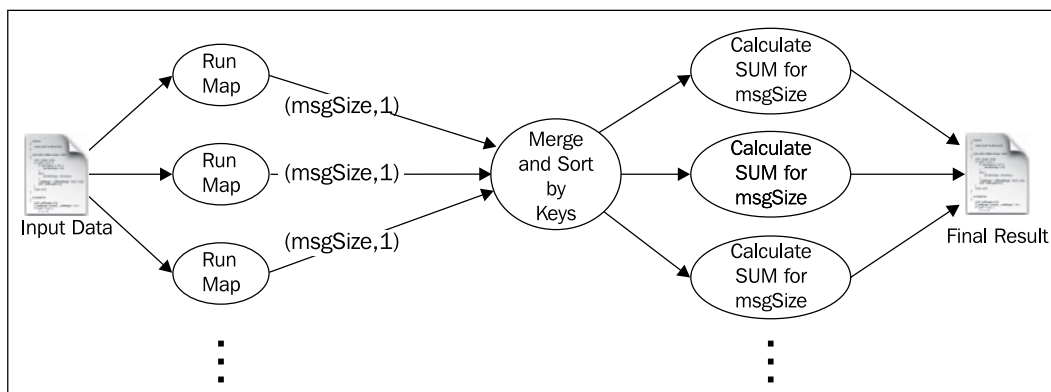
Getting ready

- This recipe assumes that you have followed the first chapter and have installed Hadoop. We will use the `HADOOP_HOME` variable to refer to the Hadoop installation folder.

- Start Hadoop by following the instructions in the first chapter.

- This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the recipe *Writing a WordCount MapReduce sample, bundling it and running it using standalone Hadoop* from *Chapter 1, Getting Hadoop Up and Running in a Cluster*.

How to do it...

The following steps show how to use MapReduce to calculate the correlation between two datasets:

1. Download the weblog dataset from `ftp://ita.ee.lbl.gov/traces/NASA_ access_log_Jul95.gz` and unzip it. We will call this `DATA_DIR`.

2. Upload the data to HDFS by running following commands from `HADOOP_HOME`. If `/ data` is already there, clean it up:

   ```
   >bin/hadoopdfs -mkdir /data
   >bin/hadoopdfs -mkdir /data/input1
   >bin/hadoopdfs -put <DATA_DIR>/NASA_access_log_Jul95 /data/input1
   ```

3. Unzip the source code of this chapter (`chapter6.zip`). We will call that folder `CHAPTER_6_SRC`.

4. Change the `hadoop.home` property in the `CHAPTER_6_SRC/build.xml` file to point to your Hadoop installation folder.

5. Compile the source by running the `ant build` command from the `CHAPTER_6_SRC` folder.

6. Copy the `build/lib/hadoop-cookbook-chapter6.jar` file to your `HADOOP_HOME`.

7. Run the MapReduce job through following command from `HADOOP_HOME`:

   ```
   > bin/hadoop jar hadoop-cookbook-chapter6.jar chapter6.
   WeblogMessagesizevsHitsProcessor/data/input1 /data/output5
   ```

8. Read the results by running the following command:

   ```
   >bin/hadoopdfs -cat /data/output5/*
   ```

9. Download the results of the last recipe to the local computer by running the following command from `HADOOP_HOME`:

   ```
   > bin/hadoopdfs -get /data/output5/part-r-00000 5.data
   ```

10. Copy all the `*.plot` files from `CHAPTER_6_SRC` to `HADOOP_HOME`.

11. Generate the plot by running the following command from `HADOOP_HOME`.

    ```
    >gnuplot httphitsvsmsgsize.plot
    ```

12. It will generate a file called `hitsbymsgSize.png`, which will look like following screenshot:
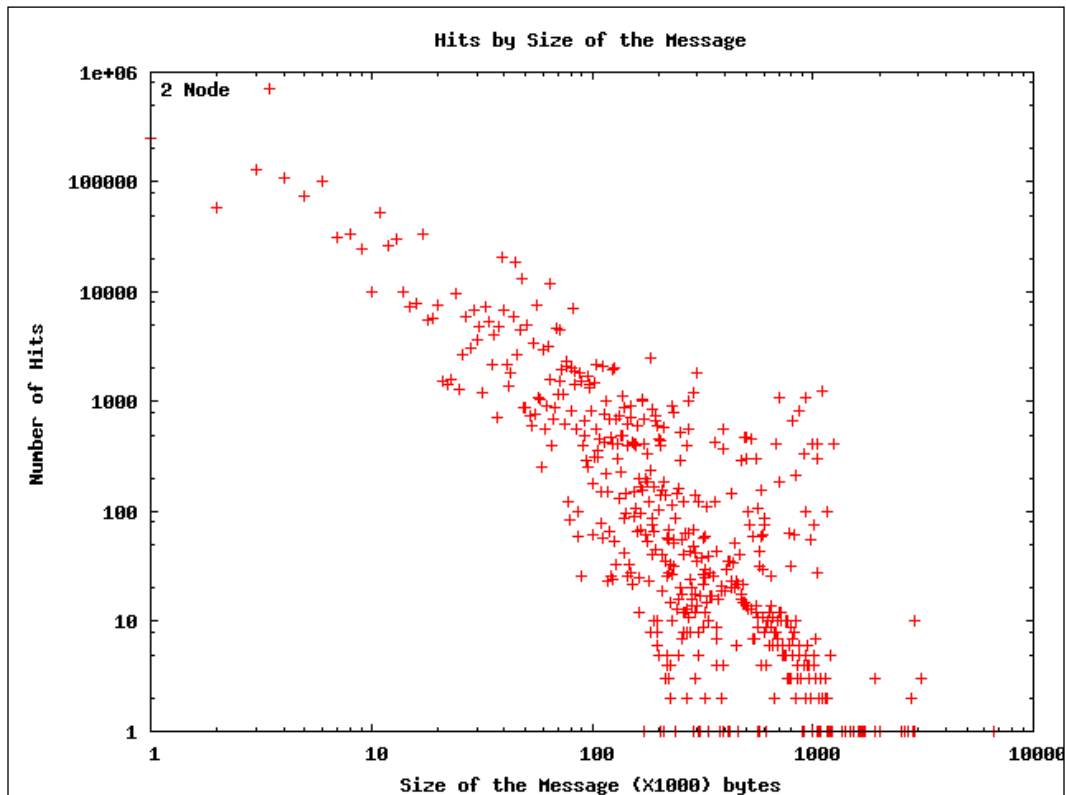
The plot shows a negative correlation between the number of hits and the size of the messages in the log scales, which also suggest a power law distribution.

How it works...

You can find the source for the recipe from `src/chapter6/WeblogMessagesizevsHitsProcessor.java`.

The following code segment shows the code for the mapper. Just like earlier recipes, we will use regular expressions to parse the log entries from log files:

```
public void map(Object key, Text value,
    Context context) throws IOException, InterruptedException
{
    Matcher matcher = httplogPattern.matcher(value.toString());
    if (matcher.matches())
```

```
    {
    int size = Integer.parseInt(matcher.group(5));
    context.write(new IntWritable(size / 1024), one);
    }
}
```

Map task receives each line in the log file as a different key-value pair. It parses the lines using regular expressions and emits the file size as 1024-byte blocks as the key and `one` as the value.

Then, Hadoop collects all key-value pairs, sorts them, and then invokes the reducer once for each key. Each reducer walks through the values and calculates the count of page accesses for each file size.

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

The `main()` method of the job looks similar to the earlier recipes.

Parsing a complex dataset with Hadoop

Datasets we parsed so far were simple, where each data item was contained in a single line. Therefore, we were able to use Hadoop default parsing support to parse those datasets. However, some datasets have much complex formats.

In this recipe, we will analyze Tomcat developer mailing list archives. In the archive, each e-mail is composed of by multiple lines in the log file. Therefore, we will write a Hadoop input formatter to process the e-mail archive.

This recipe parses the complex e-mail list archives, and finds the owner (person who started the thread) and the number of replies received by each e-mail thread.

The following figure shows a summary of the execution. Here the mapper emits the subject of the mail as key and the sender's e-mail address and date as the value. Then Hadoop groups data by the e-mail subject and sends all the data related to that thread to the same reducer.
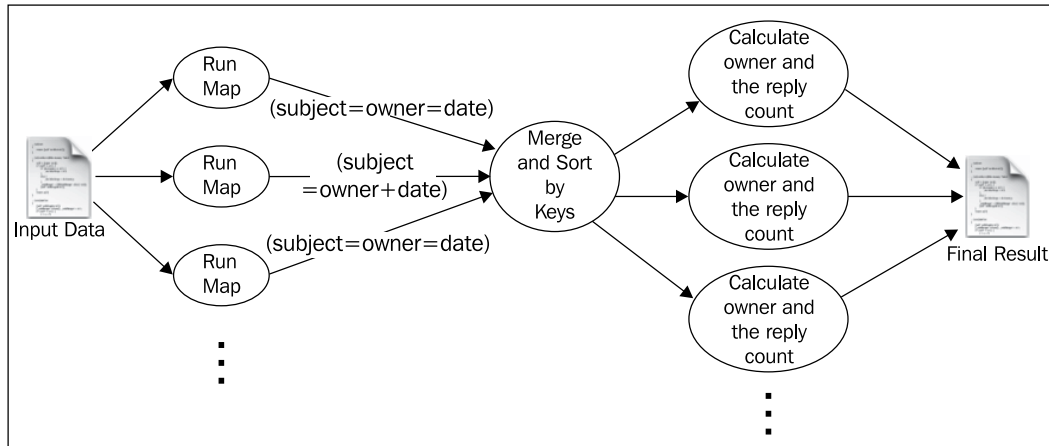
Then, the reducer calculates the owner of the thread it received, and the number of replies received by the thread.

Getting ready

- ▶ This recipe assumes that you have followed the first chapter and have installed Hadoop. We will use the `HADOOP_HOME` variable to refer to the Hadoop installation folder.

- ▶ Start Hadoop by following the instructions in the first chapter.

- ▶ This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the recipe *Writing a WordCount MapReduce sample, bundling it and running it using standalone Hadoop* from *Chapter 1, Getting Hadoop Up and Running in a Cluster.*

How to do it...

The following steps describe how to parse the Tomcat e-mail list dataset that has complex data format using Hadoop by writing an input formatter:

1. Download the Apache Tomcat developer list e-mail archives for the year 2012 available from `http://mail-archives.apache.org/mod_mbox/tomcat-users/`. We call the destination folder as `DATA_DIR`.

2. Upload the data to HDFS by running the following commands from HADOOP_HOME. If /data is already there, clean it up:

   >bin/hadoopdfs -mkdir /data

   >bin/hadoopdfs -mkdir /data/input2

   >bin/hadoopdfs -put <DATA_DIR>/*.mbox /data/input2

3. Unzip the source code for this chapter (chapter6.zip). We will call that folder CHAPTER_6_SRC.

4. Change the hadoop.home property in the CHAPTER_6_SRC/build.xml file to point to your Hadoop installation folder.

5. Compile the source by running the ant build command from the CHAPTER_6_SRC folder.

6. Copy the build/lib/hadoop-cookbook-chapter6.jar file to HADOOP_HOME.

7. Run the MapReduce job through the following command from HADOOP_HOME:

   > bin/hadoop jar hadoop-cookbook-chapter6.jar chapter6.
   MLReceiveReplyProcessor/data/input2 /data/output6

8. Read the results by running the following command:

   >bin/hadoopdfs -cat /data/output6/*

How it works...

As explained before, this dataset has data items that span multiple lines. Therefore, we have to write a custom data formatter to parse the data. You can find the source for the recipe from src/chapter6/WebLogMessageSizeAggregator.java,src/chapter6/MboxFileFormat.java,src/chapter6/MBoxFileReader.java.

When the Hadoop job starts, it invokes the formatter to parse the input files. We add a new formatter via the main() method as highlighted in the following code snippet:

```
Job job = new Job(conf, "LogProcessingHitsByLink");
job.setJarByClass(MLReceiveReplyProcessor.class);
job.setMapperClass(AMapper.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Text.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
job.setReducerClass(AReducer.class);
job.setInputFormatClass(MboxFileFormat.class);
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

As shown by the following code, the new formatter creates a record reader, which is used by Hadoop to read input keys and values:

```
public class MboxFileFormat extends
  FileInputFormat<Text, Text>
{
  private MBoxFileReaderboxFileReader = null;
  public RecordReader<Text, Text>createRecordReader(
    InputSplitinputSplit, TaskAttemptContext attempt)
    throws IOException, InterruptedException
    {
      boxFileReader = new MBoxFileReader();
      boxFileReader.initialize(inputSplit, attempt);
      return boxFileReader;
    }
}
```

The following code snippet shows the record reader:

```
public class MBoxFileReader extends
  RecordReader<Text, Text>
{
  public void initialize(InputSplitinputSplit,
    TaskAttemptContext attempt) throws IOException,
InterruptedException
  {
    Path path = ((FileSplit) inputSplit).getPath();
    FileSystem fs = FileSystem.get(attempt.getConfiguration());
    FSDataInputStream fsStream = fs.open(path);
    reader = new BufferedReader(
    new InputStreamReader(fsStream));
  }
}
```

The `initialize()` method reads the file from HDFS:

```
public Boolean nextKeyValue() throws IOException,
InterruptedException
{
  if (email == null)
  {
    return false;
  }
  count++;
  while ((line = reader.readLine()) != null)
  {
```

```
      Matcher matcher = pattern1.matcher(line);
      if (!matcher.matches())
      {
        email.append(line).append("\n");
      }
      else
      {
        parseEmail(email.toString());
        email = new StringBuffer();
        email.append(line).append("\n");
        return true;
      }
    }
    parseEmail(email.toString());
    email = null; return true;
}
```

Finally, the `nextKeyValue()` method parses the file, and gives users access to the key and values for this dataset. Value has the *from*, *subject*, and *date* of each e-mail separated by a #.

The following code snippet shows the map task source code:

```
public void map(Object key, Text value,
  Context context) throws IOException, InterruptedException
{
  String[] tokens = value.toString().split("#");
  String from = tokens[0];
  String subject = tokens[1];
  String date = tokens[2].replaceAll(",", "");
  subject = subject.replaceAll("Re:", "");
  context.write(new Text(subject), new Text(date + "#" + from));
}
```

The map task receives each line in the log file as a different key-value pair. It parses the lines by breaking it by the #, and emits the `subject` as the key, and `date` and `from` as the value.

Then, Hadoop collects all key-value pairs, sorts them, and then invokes the reducer once for each key. Since we use the e-mail subject as the key, each reducer will receive all the information about each e-mail thread. Then, each reducer walks through all the e-mails and finds out who sent the first e-mail and how many replies have been received by each e-mail thread.

```
public void reduce(Text key, Iterable<Text> values, Context context)
  throws IOException, InterruptedException
{
```

```
  TreeMap<Long, String>replyData = new TreeMap<Long, String>();
  for (Text val : values)
  {
    String[] tokens = val.toString().split("#");
    if(tokens.length != 2)
    {
      throw new IOException("Unexpected token "+ val.toString());
    }
   String from = tokens[1];
   Date date = dateFormatter.parse(tokens[0]);
   replyData.put(date.getTime(), from);
   }
  String owner = replyData.get(replyData.firstKey());
  intreplyCount = replyData.size();
  intselfReplies = 0;
  for(String from: replyData.values())
  {
    if(owner.equals(from))
    {
      selfReplies++;
    }
  }
replyCount = replyCount - selfReplies;
context.write(new Text(owner),new Text(replyCount+"#" + selfReplies));
}
```

Joining two datasets using MapReduce

As we have observed already, Hadoop is very good at reading through a dataset and calculating the analytics. However, often we will have to merge two datasets for analyzing the data. This recipe will explain how to join two datasets using Hadoop.

As an example, this recipe will use the Tomcat developer archives dataset. A common belief among the open source community is that, the more a developer is involved with the community (for example, by replying to threads and helping others and so on), the more quickly he will receive a response to his queries. In this recipe we will test this hypothesis using the Tomcat developer mailing list.

To test this hypothesis, we will run the MapReduce jobs as explained in the following figure:
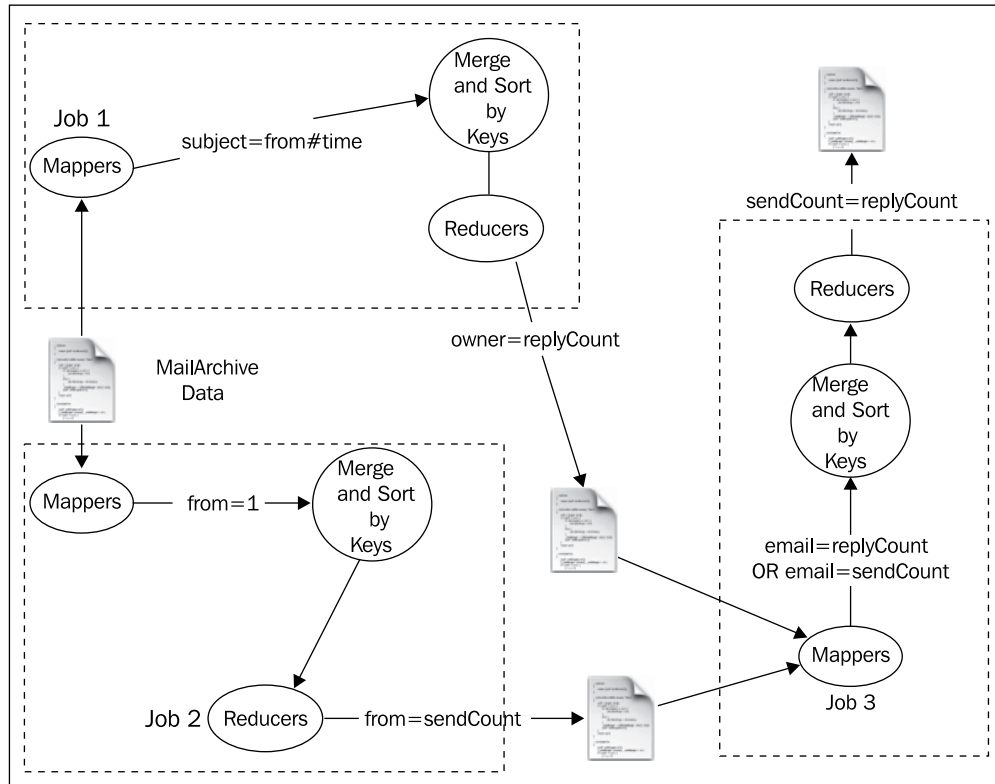
We would start with e-mail archives in the MBOX format, and we will read the mail using the MBOX format class explained in the earlier recipe. Then, the Hadoop job will receive the sender of the e-mail (from), e-mail subject, and the date the e-mail was sent, as inputs.

1. In the first job, mapper will emit the subject as key, and the sender's e-mail address and date as the value. Then, the reducer step will receive all values with the same subject and it will output the subject as the key, and the owner and reply count as the value. We have executed this job in the earlier recipe.

2. In the second job, the mapper step emits the sender's e-mail address as the key and one as the value. Then, the reducer step will receive all the e-mails sent from the same address to the same reducer. Using this data, each reducer will emit the e-mail address as the key and the number of e-mails sent from that e-mail address as the value.

3. Finally, the third job reads both the output from earlier jobs, joins the results, and emits the number of replies sent by each e-mail address and the number of replies received by each e-mail address as the output.

Getting ready

▸ This recipe assumes that you have followed the first chapter and have installed Hadoop. We will use the HADOOP_HOME variable to refer to the Hadoop installation folder.

▸ Start Hadoop by following the instructions in the first chapter.

▸ This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the recipe *Writing a WordCount MapReduce sample, bundling it and running it using standalone Hadoop* from *Chapter 1*, *Getting Hadoop Up and Running in a Cluster*.

How to do it...

The following steps show how to use MapReduce to join two datasets:

1. If you have not already done so, run the previous recipe, which will set up the environment and run the first job as explained in the figure.

2. Run the second MapReduce job through the following command from HADOOP_HOME:

   ```
   > bin/hadoop jar hadoop-cookbook-chapter6.jar chapter6.
   MLSendReplyProcessor/data/input2 /data/output7
   ```

3. Read the results by running the following command:

   ```
   >bin/hadoopdfs -cat /data/output7/*
   ```

4. Create a new folder input3 and copy both results from earlier jobs to that folder in HDFS:

   ```
   > bin/hadoopdfs -mkdir /data/input3
   ```

   ```
   > bin/hadoopdfs  -cp /data/output6/part-r-00000 /data/input3/1.
   data
   ```

   ```
   > bin/hadoopdfs -cp /data/output7/part-r-00000 /data/input3/2.data
   ```

5. Run the third MapReduce job through the following command from HADOOP_HOME:

   ```
   > bin/hadoop jar hadoop-cookbook-chapter6.jar chapter6.
   MLJoinSendReceiveReplies /data/input3 /data/output8
   ```

6. Download the results of the last recipe to the local computer by running the following command from HADOOP_HOME:

   ```
   > bin/hadoopdfs -get /data/output8/part-r-00000 8.data
   ```

7. Copy all the *.plot files from CHAPTER_6_SRC to HADOOP_HOME.

8. Generate the plot by running the following command from HADOOP_HOME:

   ```
   >gnuplot sendvsreceive.plot
   ```

9. It will generate a file called `sendreceive.png`, which will look like following:
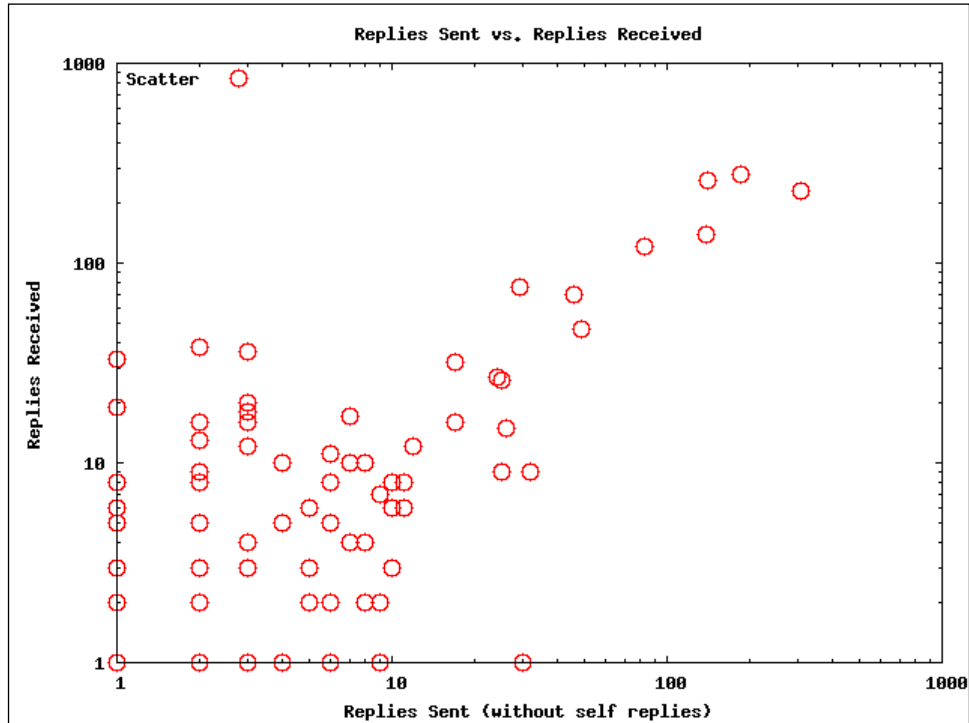
The graph confirms our hypothesis, and like before, the data approximately follows a power law distribution.

How it works...

You can find the source for the recipe from `src/chapter6/MLSendReplyProcessor.java` and `src/chapter6/MLJoinSendReceiveReplies.java`. We have already discussed the working of the first job in the earlier recipe.

The following code snippet shows the `map()` function for the second job. It receives the sender's e-mail, subject, and date separated by # as input, which parses the input and outputs the sender's e-mail as the key and the date the e-mail was sent, as the value:

```java
public void map(Object key, Text value, Context context)
    throws IOException, InterruptedException
{
String[] tokens = value.toString().split("#");
String from = tokens[0]; String date = tokens[2];
context.write(new Text(from), new Text(date));
}
```

The following code snippet shows the `reduce()` function for the second job. Each `reduce()` function receives the time of all the e-mails sent by one sender. The reducer counts the number of replies sent by each sender, and outputs the sender's name as the key and the replies sent, as the value:

```
public void reduce(Text key, Iterable<Text> values,
  Context context ) throws IOException, InterruptedException
{
  int sum = 0;
  for (Text val : values)
  {
    sum = sum +1;
  }
  context.write(key, new IntWritable(sum));
}
```

The following code snippet shows the `map()` function for the third job. It reads the outputs of the first and second jobs and writes them as the key-value pairs:

```
public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
String[] tokens = value.toString().split("\\s");
    String from = tokens[0];
    String replyData = tokens[1];
context.write(new Text(from), new Text(replyData));
  }
```

The following code snippet shows the `reduce()` function for the third job. Since, both the output of the first and the second job has the same key, the sent replies and received replies for a given user are sent to the same reducer. The reducer does some adjustments to remove self-replies, and outputs the sent replies and received replies as the key and value respectively of the reducer, thus joining the two datasets:

```
public void reduce(Text key, Iterable<Text> values, Context context)
  throws IOException, InterruptedException
{
  StringBuffer buf = new StringBuffer("[");
  try
  {
    int sendReplyCount = 0;
    int receiveReplyCount = 0;
    for (Text val : values)
    {
      String strVal = val.toString();
      if(strVal.contains("#"))
      {
```

```
        String[] tokens = strVal.split("#");
        Int repliesOnThisThread = Integer.parseInt(tokens[0]);
        Int selfRepliesOnThisThread = Integer.parseInt(tokens[1]);
        receiveReplyCount = receiveReplyCount + repliesOnThisThread;
        sendReplyCount = sendReplyCount - selfRepliesOnThisThread;
      }
      else
      {
        sendReplyCount = sendReplyCount + Integer.parseInt(strVal);
      }
    }
    context.write(new IntWritable(sendReplyCount),
      new IntWritable(receiveReplyCount)); buf.append("]");
   }
  catch (NumberFormatException e)
  {
    System.out.println("ERROR "+ e.getMessage());
  }
}
```

Here the final job is an example of using the MapReduce to join two datasets. The idea is to send all the values that need to be joined under the same key to the same reducer, and join the data there.

7
Searching and Indexing

In this chapter, we will cover:

- ▸ Generating an inverted index using Hadoop MapReduce
- ▸ Intra-domain web crawling using Apache Nutch
- ▸ Indexing and searching web documents using Apache Solr
- ▸ Configuring Apache HBase as the backend data store for Apache Nutch
- ▸ Deploying Apache HBase on a Hadoop cluster
- ▸ Whole web crawling with Apache Nutch using a Hadoop/HBase cluster
- ▸ ElasticSearch for indexing and searching
- ▸ Generating the in-links graph for crawled web pages

Introduction

MapReduce frameworks are well suited for large-scale search and indexing applications. In fact, Google came up with the original MapReduce framework specifically to facilitate the various operations involved with web searching. The Apache Hadoop project was started as a support project for the **Apache Nutch** search engine, before spawning off as a separate top-level project.

Web searching consists of fetching, indexing, ranking, and retrieval. Given the size of the data, all these operations need to be scalable. In addition, the retrieval also should ensure real-time access. Typically, fetching is performed through web crawling, where the crawlers fetch a set of pages in the fetch queue, extract links from the fetched pages, add the extracted links back to the fetch queue, and repeat this process many times. Indexing parses, organizes, and stores the fetched data in manner that is fast and efficient for querying and retrieval. Search engines perform offline ranking of the documents based on algorithms such as PageRank and real-time ranking of the results based on the query.

In this chapter, we will introduce you to several tools that can be used with Apache Hadoop to perform large-scale searching and indexing.

Generating an inverted index using Hadoop MapReduce

Most of the text searching systems rely on **inverted index** to look up the set of documents that contains a given word or a term. In this recipe, we are going to build a simple inverted index that computes a list of terms in the documents, the set of documents that contains each term, and the term frequency in each of the documents. Retrieval of results from an inverted index can be as simple as returning the set of documents that contains the given terms or can involve much more complex operations such as returning the set of documents ordered based on a particular ranking.

Getting ready

You must have Apache Hadoop (preferably version 1.0.x) configured and installed to follow this recipe. Apache Ant for the compiling and building the source code.

How to do it...

In the following steps, we will use a MapReduce program to build an inverted index for a text dataset.

1. Export the `$HADOOP_HOME` environmental variable pointing to the root of your local Apache Hadoop installation.

2. Create a directory in HDFS and upload a text data set. This data set should consist of one or more text files.

    ```
    > bin/hadoop dfs -mkdir input
    ```

    ```
    > bin/hadoop dfs -put *.txt input
    ```

    > You can download the text versions of Project Gutenberg books by following the instructions given at the following link. Make sure to provide the `filetypes` query parameter of the download request as `txt`. Unzip the downloaded files. You can use the unzipped text files as the text data set for this recipe. `http://www.gutenberg.org/wiki/Gutenberg:Information_About_Robot_Access_to_our_Pages`

3. Unzip the resources bundle for this chapter and change to that directory.

4. Compile the source by running `ant build` command from the unzipped directory.

5. Copy the resulting `build/c7-samples.jar` to your Hadoop home directory.

6. Run the inverted indexing MapReduce job using the following command from the Hadoop home directory. Provide the HDFS directory where you uploaded the input data in step 2 as the first argument and provide a path to store the output as the second argument.

    ```
    > bin/hadoop jar c7-samples.jar chapter7.TextOutInvertedIndexer
    input output
    ```

7. Check the output directory for the results by running the following command. The output will consist of the term followed by a comma-separated list of filename and frequency.

    ```
    > bin/hadoop dfs -cat output/*
    ARE three.txt:1,one.txt:1,four.txt:1,two.txt:1,
    AS three.txt:2,one.txt:2,four.txt:2,two.txt:2,
    AUGUSTA three.txt:1,
    About three.txt:1,two.txt:1,
    Abroad three.txt:2,
    ……
    ```

8. We used the text outputting invert indexing MapReduce program in step 6 for the clarity of understanding the algorithm. The `src/chapter9/InvertIndexer.java` program uses the Hadoop Sequence Files and Map Writable to output an index, which is more friendly for machine processing and more efficient for storage. You can run this version of the program by substituting the command in step 6 with the following command:

    ```
    > bin/hadoop jar c7-samples.jar chapter7.InvertedIndexer input
    output
    ```

How it works...

Map function receives a chunk of an input document as the input and outputs the term and `<docid, 1>` pair for each word. In the Map function, we first replace all the non-alphanumeric characters from the input text value before tokenizing it.

```
public void map(Object key, Text value, ……… {
    String valString = value.toString().replaceAll("[^a-zA-Z0-9]+","
");
    StringTokenizer itr = new StringTokenizer(valString);
     StringTokenizer(value.toString());

    FileSplit fileSplit = (FileSplit) context.getInputSplit();
```

```
        String fileName = fileSplit.getPath().getName();
        while (itr.hasMoreTokens()) {
            term.set(itr.nextToken());
            docFrequency.set(fileName, 1);
            context.write(term, docFrequency);
        }
    }
```

We use the `getInputSplit()` method of the `MapContext` to obtain a reference to
`InputSplit` of assigned to the current Map task. The `InputSplits` for this computation
are instances of `FileSplit` due to the usage of `FileInputFormat` based `InputFormat`.
Then we use the `getPath()` method of `FileSplit` to obtain the path of the file containing
the current split and extract the filename from it. We use this extracted filename as the
document ID when constructing the inverted index.

The reduce function receives IDs and frequencies of all the documents that contain the term
(key) as the input. The reduce function outputs the term and a list of document IDs and the
number of occurrences of the term in each document as the output:

```
public void reduce(Text key, Iterable<TermFrequencyWritable>
values,Context context) …………{

    HashMap<Text, IntWritable> map = new HashMap<Text, IntWritable>();
    for (TermFrequencyWritable val : values) {
        Text docID = new Text(val.getDocumentID());
        int freq = val.getFreq().get();
        if (map.get(docID) != null) {
            map.put(docID, new IntWritable(map.get(docID).get() +
freq));
        } else {
            map.put(docID, new IntWritable(freq));
        }
    }
    MapWritable outputMap = new MapWritable();
    outputMap.putAll(map);
    context.write(key, outputMap);
}
```

In the preceding model, we output a record for each word, generating a large amount of
Map task to Reduce task intermediate data. We use the following combiner to aggregate
the terms emitted by the Map tasks, reducing the size and amount of Map to Reduce
intermediate data transfer.

```
public void reduce(Text key, Iterable<TermFrequencyWritable> values ……
{
    int count = 0;
```

```
      String id = "";
    for (TermFrequencyWritable val : values) {
      count++;
      if (count == 1) {
        id = val.getDocumentID().toString();
      }
    }
    TermFrequencyWritable writable = new TermFrequencyWritable();
    writable.set(id, count);
    context.write(key, writable);
}
```

In the driver program, we set the Mapper, Reducer, and the combiner classes. Also we specify both output value and the map output value properties as we use different value types for the Map tasks and the Reduce tasks.

```
Job job = new Job(conf, "Inverted Indexer");
…
job.setMapperClass(IndexingMapper.class);
job.setReducerClass(IndexingReducer.class);
job.setCombinerClass(IndexingCombiner.class);
…
job.setMapOutputValueClass(TermFrequencyWritable.class);
job.setOutputValueClass(MapWritable.class);
job.setOutputFormatClass(SequenceFileOutputFormat.class);
```

There's more...

The older MapReduce API of Apache Hadoop (`org.apache.hadoop.mapred.*`) supports a file format called **MapFile** that can be used to store an index in to the data stored in SequenceFiles. MapFile is very useful when we need to random access records stored in a large SequenceFile. We can utilize the MapFiles to store a secondary index in to our inverted index. You can use `MapFileOutputFormat` to output MapFiles, which would consist of a SequenceFile containing the actual data and another file containing the index to the SequenceFile.

We can improve this indexing program by performing optimizations into such as filtering-stop words, substituting words with word stems, and storing more information about the context of the word, making the indexing a much more complex problem. Luckily, there exist several open source indexing frameworks that we can use for the indexing purposes. In this chapter we'll be using Apache Lucene-based Apache Solr and ElasticIndex for indexing purposes.

See also

The *Creating TF and TF-IDF vectors for the text data* recipe of *Chapter 9*, *Mass Text Data Processing*.

Intra-domain web crawling using Apache Nutch

Web crawling is the process of visiting and downloading all or a subset of web pages on the Internet. Although the concept of crawling and implementing a simple crawler sounds simple, building a full-fledged crawler takes great deal of work. A full-fledged crawler that needs to be distributed has to obey the best practices such as not overloading servers, follow `robots.txt`, performing periodic crawls, prioritizing the pages to crawl, and identifying many formats of documents. Apache Nutch is an open source search engine that provides a highly scalable crawler. Apache Nutch offers features such as politeness, robustness, and scalability.

In this recipe, we are going to use Apache Nutch in the standalone mode for small-scale, intra-domain web crawling. Almost all the Nutch commands are implemented as Hadoop MapReduce applications, as you would notice when executing the steps 10 to 18 of this recipe. Nutch standalone executes these applications using the Hadoop the local mode.

Getting ready

Set the `JAVA_HOME` environmental variable. Install Apache Ant and add it to the `PATH` environmental variable.

How to do it...

The following steps show you how to use Apache Nutch in standalone mode for small scale web crawling.

1. Apache Nutch standalone mode uses the HyperSQL database as the default data storage. Download HyperSQL from the `http://sourceforge.net/projects/hsqldb/`. Unzip the distribution and go to the data directory.

   ```
   > cd hsqldb-2.2.9/hsqldb
   ```

2. Start a HyperSQL database using the following command. The following database uses `data/nutchdb.*` as the database files and uses `nutchdb` as the database alias name. We'll be using this database alias name in the `gora.sqlstore.jdbc.url` property in the step 7.

   ```
   > java -cp lib/hsqldb.jar org.hsqldb.server.Server --database.0
   file:data/nutchdb --dbname.0 nutchdb

   ......
   ```

```
[Server@79616c7]: Database [index=0, id=0, db=file:data/nutchdb,
alias=nutchdb] opened sucessfully in 523 ms.

......
```

3. Download Apache Nutch 2.X from `http://nutch.apache.org/` and extract it.

4. Go to the extracted directory, which we will refer to as `NUTCH_HOME`, and build Apache Nutch using the following command:

   ```
   > ant runtime
   ```

5. Go to the `runtime/local` directory and run the `bin/nutch` command to verify the Nutch installation. A successful installation would print out the list of Nutch commands, shown as follows:

   ```
   > cd runtime/local
   > bin/nutch
   Usage: nutch COMMAND
   where COMMAND is one of:…..
   ```

6. Add the following to `NUTCH_HOME/runtime/local/conf/nutch-site.xml`. You can give any name to the value of `http.agent.name`.

   ```
   <configuration>
   <property>
    <name>http.agent.name</name>
    <value>NutchCrawler</value>
   </property>
   <property>
     <name>http.robots.agents</name>
     <value>NutchCrawler,*</value>
   </property>
   </configuration>
   ```

7. You can restrict the domain names you wish to crawl by editing the `NUTCH_HOME/runtime/local/conf/regex-urlfiler.txt` file. For an example, in order to restrict the domain to `http://apache.org`,

   Replace the following in the `NUTCH_HOME/runtime/local/conf/regex-urlfilter.txt` file:

   ```
   # accept anything else
   +.
   ```

   Use the following regular expression:

   ```
   +^http://([a-z0-9]*\.)*apache.org/
   ```

8. Ensure that you have the following in the `NUTCH_HOME/runtime/local/conf/gora.properties` file. Provide the database alias named used in step 2.

```
###############################
# Default SqlStore properties #
###############################
gora.sqlstore.jdbc.driver=org.hsqldb.jdbc.JDBCDriver
gora.sqlstore.jdbc.url=jdbc:hsqldb:hsql://localhost/nutchdb
gora.sqlstore.jdbc.user=sa
```

9. Create a directory named `urls` and create a file named `seed.txt` inside that directory. Add your seed URLs to this file. Seed URLs are used to start the crawling and would be pages that are crawled first. We use `http://apache.org` as the seed URL in the following example:

```
> mkdir urls
```

```
> echo http://apache.org/ > urls/seed.txt
```

10. Inject the seed URLs in to the Nutch database using the following command:

```
> bin/nutch inject urls/
InjectorJob: starting
InjectorJob: urlDir: urls
InjectorJob: finished
```

11. Use the following command to verify the injection of the seeds to the Nutch database. `TOTAL urls` printed by this command should match the number of URLs you had in your `seed.txt` file. You can use this command in the later cycles as well to get an idea about the number of web page entries in your database.

```
> bin/nutch readdb  -stats
WebTable statistics start
Statistics for WebTable:
min score:  1.0
....
TOTAL urls:  1
```

12. Use the following command to generate a fetch list from the injected seed URLs. This will prepare list of web pages to be fetched in the first cycle of the crawling. Generation will assign a batch ID to the current generated fetch list, which can be used in the subsequent commands.

```
> bin/nutch generate
GeneratorJob: Selecting best-scoring urls due for fetch.
GeneratorJob: starting
```

```
GeneratorJob: filtering: true
GeneratorJob: done
GeneratorJob: generated batch id: 1350617353-1356796157
```

13. Use the following command to fetch the list of pages prepared in step 12. This step performs the actual fetching of the web pages. The `-all` parameter is used to inform Nutch to fetch all the generated batches.

```
> bin/nutch fetch -all
FetcherJob: starting
FetcherJob: fetching all
FetcherJob: threads: 10
......
fetching http://apache.org/
......
-activeThreads=0
FetcherJob: done
```

14. Use the following command to parse and to extract the useful data from fetched web pages, such as the text content of the pages, metadata of the pages, and the set of pages linked from the fetched pages. We call the set of pages linked from a fetched page as the out-links of that particular fetched page. The out-links data will be used to discover new pages to fetch as well as to rank pages using link analysis algorithms such as PageRank.

```
> bin/nutch parse -all
ParserJob: starting
......
ParserJob: success
```

15. Execute the following command to update the Nutch database with the data extracted in the preceding step. This step includes updating the contents of the fetched pages as well as adding new entries of the pages discovered through the links contained in the fetched pages.

```
> bin/nutch updatedb
DbUpdaterJob: starting
……
DbUpdaterJob: done
```

16. Execute the following command to generate a new fetch list using the information from the previously fetched data. The *topN* parameter limits the number of URLs generated for the next fetch cycle.

```
> bin/nutch generate -topN 100
GeneratorJob: Selecting best-scoring urls due for fetch.
GeneratorJob: starting
......
GeneratorJob: done
GeneratorJob: generated batch id: 1350618261-1660124671
```

17. Fetch the new list, parse it, and update the database.

```
> bin/nutch fetch –all
......
> bin/nutch parse -all
......
> bin/nutch updatedb
......
```

18. Repeat the steps 16 and 17 till you get the desired number of pages or the depth.

See also

The *Whole web crawling with Apache Nutch using a Hadoop/HBase cluster* and *Indexing and searching web documents using Apache Solr* recipes of this chapter.

Refer to `http://www.hsqldb.org/doc/2.0/guide/index.html` for more information on using HyperSQL.

Indexing and searching web documents using Apache Solr

Apache Solr is an open source search platform that is part of the **Apache Lucene** project. It supports powerful full-text search, hit highlighting, faceted search, dynamic clustering, database integration, rich document handling (for example, Word and PDF), and geospatial search. In this recipe, we are going to index the web pages crawled by Apache Nutch for use by Apache Solr and use Apache Solr to search through those web pages.

Getting Ready

Crawl a set of web pages using Apache Nutch by following *the Intra-domain crawling using Apache Nutch* recipe.

How to do it

The following steps show you how to index and search your crawled web pages dataset:

1. Download and extract Apache Solr from `http://lucene.apache.org/solr/`. We use Apache Solr 4.0 for the examples in this chapter. From here on, we call the extracted directory `$SOLR_HOME`.

2. Replace the `$SOLR_HOME/examples/solr/collection1/conf/schema.xml` file using the `$NUTCH_HOME/runtime/local/conf/schema.solr4.xml` file.

   ```
   > cp $NUTCH_HOME/conf/schema-solr4.xml \
        $SOLR_HOME/example/solr/collection1/conf/schema.xml
   ```

3. Open the `example/solr/collection1/conf/solrconfig.xml` file and comment the following tag.

   ```
   <updateLog>

   <str name="dir">${solr.data.dir:}</str>

   </updateLog>
   ```

4. Start Solr by executing the following command from the `$SOLR_HOME/example` directory.

   ```
   > java -jar start.jar
   ```

5. Go to the URL `http://localhost:8983/solr` to verify the Apache Solr installation.

6. Index the data fetched using Apache Nutch in to Apache Solr by issuing the following command from the `$NUTCH_HOME/runtime/local` directory. This command pushes the data crawled by Nutch in to Solr through the Solr web service interface.

   ```
   > bin/nutch solrindex http://127.0.0.1:8983/solr/ -reindex
   ```

7. Go to Apache Solr search UI at `http://localhost:8983/solr/#/collection1/query`. Enter a search term in the **q** textbox and click on **Execute Query**.
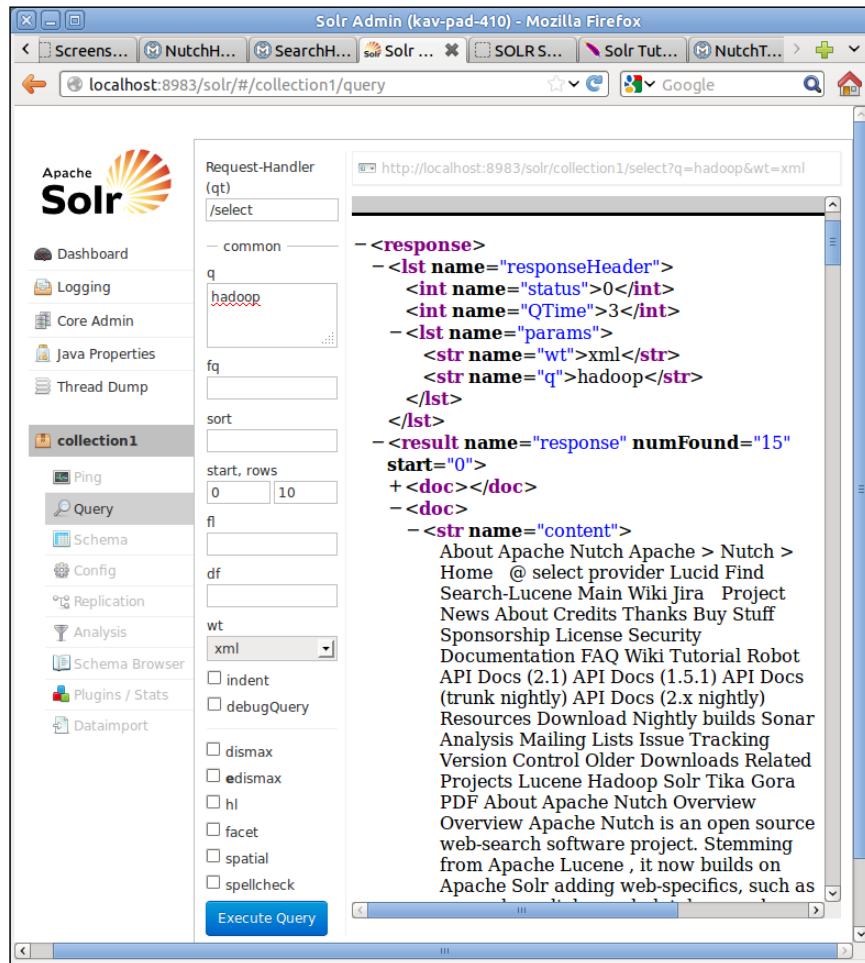
8. You can also issue your search queries directly using the HTTP GET requests. Paste the following to your browser address bar:

   `http://localhost:8983/solr/collection1/select?q=hadoop&start=5&rows=5&wt=xml`

How it works

Apache Solr is built using the Apache Lucene text search library. Apache Solr adds many features on top of Apache Lucene and provides a text search web application that works out of the box. The preceding steps deploy Apache Solr and import the data crawled by Nutch in to the deployed Solr instance.

The metadata about the documents we plan to index and search using Solr needs to be specified through the Solr `schema.xml` file. The Solr schema file should define the data fields in our documents and how these data fields should be processed by Solr. We use the schema file provided with Nutch (`$NUTCH_HOME/conf/schema-solr4.xml`), which defines the schema for the web pages crawled by Nutch, as the Solr schema file for this recipe. More information about the Solr schema file can be found from `http://wiki.apache.org/solr/SchemaXml`.

See also

▶ The *ElasticSearch for Indexing and searching* recipe of this chapter.

▶ Follow the tutorial at `http://lucene.apache.org/solr/tutorial.html` for more information on using Apache Solr.

Configuring Apache HBase as the backend data store for Apache Nutch

Apache Nutch integrates Apache Gora to add support for different backend data stores. In this recipe, we are going to configure Apache HBase as the backend data storage for Apache Nutch. Similarly, it is possible to plug in data stores such as RDBMS databases, Cassandra and others through Gora.

Getting ready

Set the `JAVA_HOME` environmental variable.

Install Apache Ant and add it to the `PATH` environmental variable.

How to do it

The following steps show you how to configure Apache HBase local mode as the backend data store for Apache Nutch to store the crawled data.

1. Download and install Apache HBase. Apache Nutch 4.1 and Apache Gora 0.2 recommend HBase 0.90.4 or the later versions of the 0.90.x branch.

2. Create two directories to store the HDFS data and Zookeeper data. Add the following to the `$HBASE_HOME/conf/hbase-site.xml` file, replacing the values with the paths to the two directories:

```
<configuration>
<property>
    <name>hbase.rootdir</name>
    <value>file:///u/software/hbase-0.90.6/hbase-data</value>
  </property>
<property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>file:///u/software/hbase-0.90.6/zookeeper-data</value>
  </property>
</configuration>
```

Refer to the *Installing HBase* recipe in *Chapter 5*, *Hadoop Ecosystem*, for more information on how to install HBase in the local mode. Test your HBase installation using the HBase shell before proceeding (step 6 of the *Installing HBase* recipe.)

3. In case you have not downloaded Apache Nutch for the earlier recipes in this chapter, download Nutch from the `http://nutch.apache.org` and extract it.

4. Add the following to the `$NUTCH_HOME/conf/nutch-site.xml` file.

```
<property>
 <name>storage.data.store.class</name>
 <value>org.apache.gora.hbase.store.HBaseStore</value>
 <description>Default class for storing data</description>
</property>
```

5. Uncomment the following in the `$NUTCH_HOME/ivy/ivy.xml` file.

```
<dependency org="org.apache.gora" name="gora-hbase" rev="0.2"
conf="*->default" />
```

6. Add the following to the `$NUTCH_HOME/conf/gora.properties` file to set the HBase storage as the default Gora data store.

```
gora.datastore.default=org.apache.gora.hbase.store.HBaseStore
```

7. Execute the following commands in the `$NUTCH_HOME` to build Apache Nutch with HBase as the back end data storage.

```
> ant clean
> ant runtime
```

8. Follow steps 4 to 17 of the *Intra-domain web crawling using Apache Nutch* recipe.

9. Start the Hbase shell and issue the following commands to view the fetched data.

```
> bin/hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.90.6, r1295128, Wed Feb 29 14:29:21 UTC 2012
hbase(main):001:0> list
TABLE
webpage
1 row(s) in 0.4970 seconds


hbase(main):002:0> count 'webpage'
Current count: 1000, row: org.apache.bval:http/release-management.
html
Current count: 2000, row: org.apache.james:http/jspf/index.html
Current count: 3000, row: org.apache.sqoop:http/team-list.html
Current count: 4000, row: org.onesocialweb:http/
4065 row(s) in 1.2870 seconds


hbase(main):005:0> scan 'webpage',{STARTROW => 'org.apache.
nutch:http/', LIMIT=>10}
ROW                              COLUMN+CELL
 org.apache.nutch:http/             column=f:bas,
timestamp=1350800142780, value=http://nutch.apache.org/
 org.apache.nutch:http/             column=f:cnt,
timestamp=1350800142780, value=<....
......
10 row(s) in 0.5160 seconds
```

10. Follow the steps in the *Indexing and searching web documents using Apache Solr to index* recipe and search the fetched data using Apache Solr.

How it works...

The preceding steps configure and run Apache Nutch using Apache HBase as the storage backend. When configured, Nutch stores the fetched web page data and other metadata in HBase tables. In this recipe we use a standalone HBase deployment. However, as shown in the *Whole web crawling with Apache Nutch using a Hadoop/HBase cluster* recipe of this chapter, Nutch can be used with a distributed HBase deployment as well. Usage of HBase as the backend data store provides more scalability and performance for Nutch crawling.

See also

▸   The *Installing HBase* recipe of *Chapter 5*, *Hadoop Ecosystem*, and the *Deploying HBase on a Hadoop cluster* recipe of this chapter.

Deploying Apache HBase on a Hadoop cluster

In this recipe, we are going to deploy Apache HBase 0.90.x on top of an Apache Hadoop 1.0.x cluster. This is required for using Apache Nutch with a Hadoop MapReduce cluster.

Getting ready

We assume you already have your Hadoop cluster (version 1.0.x) deployed. If not, refer to the *Setting Hadoop in a distributed cluster environment* recipe of *Chapter 1*, *Getting Hadoop up and running in a Cluster*, to configure and deploy a Hadoop cluster.

How to do it

The following steps show you how to deploy a distributed Apache HBase cluster on top of an Apache Hadoop cluster:

1. Download and install Apache HBase from `http://hbase.apache.org/`. Apache Nutch 4.1 and Apache Gora 0.2 recommend HBase 0.90.4 or the later versions of the 0.90.x branch.

2. Remove the `hadoop-core-*.jar` in the `$HBASE_HOME/lib`. Copy the `hadoop-core-*.jar` and the `commons-configuration*.jar` from your Hadoop deployment to the `$HBASE_HOME/lib` folder.

   ```
   > rm lib/hadoop-core-<version>.jar

   > cp ~/Software/hadoop-1.0.4/hadoop-core-1.0.4.jar ../lib/

   > cp ~/Software/hadoop-1.0.4/lib/commons-configuration-1.6.jar ..//
   lib/
   ```

3. Configure the `$HBASE_HOME/conf/hbase-site.xml`.

   ```
   <configuration>
     <property>
       <name>hbase.rootdir</name>
       <value>hdfs://xxx.xx.xx.xxx:9000/hbase</value>
     </property>
     <property>
       <name>hbase.cluster.distributed</name>
   ```

```
      <value>true</value>
    </property>
    <property>
      <name>hbase.zookeeper.quorum</name>
      <value>localhost</value>
    </property>
  </configuration>
```

4. Go to the `$HBASE_HOME` and start HBase.

   > bin/start-hbase.sh

5. Open the HBase UI at `http://localhost:60010` and monitor the HBase installation.

6. Start the HBase shell and execute the following commands to test the HBase deployment. If the preceding command fails, check the logs in the `$HBASE_HOME/logs` directory to identify the exact issue.

   > bin/hbase shell

   hbase(main):001:0> create 'test', 'cf'

   0 row(s) in 1.8630 seconds

   hbase(main):002:0> list 'test'

   TABLE

   test

   1 row(s) in 0.0180 seconds

> Hbase is very sensitive to the contents of the `/etc/hosts` file. Fixing the `/etc/host` file would solve most of the HBase deployment errors.

How it works...

The preceding steps configure and run the Apache HBase in the distributed mode. HBase distributed mode stores the actual data of the HBase tables in the HDFS, taking advantage of the distributed and fault tolerant nature of HDFS.

In order to run HBase in the distributed mode, we have to configure the HDFS NameNode and the path to store the HBase data using the `hbase.rootdir` property in the `hbase-site.xml`.

```
    <property>
      <name>hbase.rootdir</name>
      <value>hdfs://<namenode>:<port>/<path></value>
    </property>
```

We also have to set the `hbase.cluster.distributed` property to `true`.

```
<property>
  <name>hbase.cluster.distributed</name>
  <value>true</value>
</property>
```

See also

▶ The *Installing HBase* recipe of *Chapter 5*, *Hadoop Ecosystem*.

Whole web crawling with Apache Nutch using a Hadoop/HBase cluster

Crawling large amount of web documents can be done efficiently by utilizing the power of a MapReduce cluster.

Getting ready

We assume you already have your Hadoop (version 1.0.x) and HBase (version 0.90.x) cluster deployed. If not, refer to the *Deploying HBase on a Hadoop cluster* recipe of this chapter to configure and deploy an HBase cluster on a Hadoop cluster.

How to do it

The following steps show you how to use Apache Nutch with a Hadoop MapReduce cluster and a HBase data store to perform large-scale web crawling.

1. Add the `$HADOOP_HOME/bin` directory to the `PATH` environment variable of your machine.

   ```
   > export PATH=$PATH:$HADOOP_HOME/bin/
   ```

2. If you have already followed the *Indexing and searching web documents using Apache Solr* recipe, skip to the next step. If not, follow steps 2 to 6 of the recipe 3.

3. In case you have not downloaded Apache Nutch for the earlier recipes in this chapter, download Nutch from `http://nutch.apache.org` and extract it.

4. Add the following to the `nutch-site.xml` in the `$NUTCH_HOME/conf`. You can give any name to the value of the `http.agent.name` property, but that name should be given in the `http.robots.agents` property as well.

   ```
   <configuration>
   <property>
     <name>storage.data.store.class</name>
   ```

```
 <value>org.apache.gora.hbase.store.HBaseStore</value>
 <description>Default class for storing data</description>
</property>
<property>
 <name>http.agent.name</name>
 <value>NutchCrawler</value>
</property>
<property>
   <name>http.robots.agents</name>
   <value>NutchCrawler,*</value>
</property>
</configuration>
```

5. Uncomment the following in the `$NUTCH_HOME/ivy/ivy.xml file`:

    ```
    <dependency org="org.apache.gora" name="gora-hbase" rev="0.2"
    conf="*->default" />
    ```

6. Add the following to the `$NUTCH_HOME/conf/gora.properties` file to set the HBase storage as the default Gora data store:

    ```
    gora.datastore.default=org.apache.gora.hbase.store.HBaseStore
    ```

> You can restrict the domain names you wish to crawl by editing the following line in the `conf/regex-urlfiler.txt` file. Leave it unchanged for whole web crawling.
> ```
>     # accept anything else
>     +.
> ```

7. Execute the following commands in `$NUTCH_HOME` to build Nutch with HBase as the backend data storage:

    ```
    > ant clean
    ```

    ```
    > ant runtime
    ```

8. Create a directory in HDFS to upload the seed urls.

    ```
    > bin/hadoop dfs -mkdir urls
    ```

9. Create a text file with the seed URLs for the crawl. Upload the seed URLs file to the directory created in the above step.

    ```
    > bin/hadoop dfs -put seed.txt urls
    ```

> You can use the Open Directory project RDF dump (`http://rdf.dmoz.`
> `org/`) to create your seed URLs. Nutch provides a utility class to select a
> subset of URLs from the extracted DMOZ RDF data:
>
> `bin/nutch org.apache.nutch.tools.DmozParser content.`
> `rdf.u8 -subset 5000 > dmoz/urls`

10. Copy the `$NUTCH_HOME/runtime/deploy` directory to the JobTracker node of the
    Hadoop cluster.

11. Issue the following command from inside the copied deploy directory in the
    JobTracker node to inject the seed URLs to the Nutch database and to generate the
    initial fetch list.

    ```
    > bin/nutch inject urls
    ```

    ```
    > bin/nutch generate
    ```

12. Issue the following commands from inside the copied deploy directory in the
    JobTracker node:

    ```
    > bin/nutch fetch -all
    12/10/22 03:56:39 INFO fetcher.FetcherJob: FetcherJob: starting
    12/10/22 03:56:39 INFO fetcher.FetcherJob: FetcherJob: fetching
    all
    ......


    > bin/nutch parse -all
    12/10/22 03:48:51 INFO parse.ParserJob: ParserJob: starting
    ......


    12/10/22 03:50:44 INFO parse.ParserJob: ParserJob: success


    > bin/nutch updatedb
    12/10/22 03:53:10 INFO crawl.DbUpdaterJob: DbUpdaterJob: starting
    ....
    12/10/22 03:53:50 INFO crawl.DbUpdaterJob: DbUpdaterJob: done


    > bin/nutch generate -topN 10
    ```

```
12/10/22 03:51:09 INFO crawl.GeneratorJob: GeneratorJob: Selecting
best-scoring urls due for fetch.
12/10/22 03:51:09 INFO crawl.GeneratorJob: GeneratorJob: starting
....
12/10/22 03:51:46 INFO crawl.GeneratorJob: GeneratorJob: done
12/10/22 03:51:46 INFO crawl.GeneratorJob: GeneratorJob: generated
batch id: 1350892269-603479705
```

13. Repeat the commands in step 12 as many times as needed to crawl the desired number of pages or the desired depth.

14. Follow the *Indexing and searching fetched web documents using Apache Solr* recipe to index the fetched data using Apache Solr.

How it works

All the Nutch operations we used in this recipe, including fetching and parsing, are implemented as MapReduce programs. These MapReduce programs utilize the Hadoop cluster to perform the Nutch operations in a distributed manner and use the HBase to store the data across the HDFS cluster. You can monitor these MapReduce computations through the monitoring UI (`http://jobtracker_ip:50030`) of your Hadoop cluster.

Apache Nutch Ant build creates a Hadoop job file containing all the dependencies in the `$NUTCH_HOME/runtime/deploy` folder. The `bin/nutch` script uses this job file to submit the MapReduce computations to Hadoop.

See also

The *Intra-domain crawling using Apache Nutch* recipe of this chapter.

ElasticSearch for indexing and searching

ElasticSearch (`http://www.elasticsearch.org/`) is an Apache 2.0 licensed open source search solution built on top of Apache Lucene. ElasticSearch is a distributed, multi-tenant, and document-oriented search engine. ElasticSearch supports distributed deployments, by breaking down an index in to shards and by distributing the shards across the nodes in the cluster. While both ElasticSearch and Apach Solr use Apache Lucene as the core search engine, ElasticSearch aims to provide a more scalable and a distributed solution that is better suited for the cloud environments than Apache Solr.

Getting ready

Install Apache Nutch and crawl some web pages as per the *Whole web crawling with Apache Nutch using an existing Hadoop/HBase cluster* recipe or the *Configuring Apache HBase local mode as the backend data store for Apache Nutch* recipe. Make sure the backend Hbase (or HyperSQL) data store for Nutch is still available.

How to do it

The following steps show you how to index and search the data crawled by Nutch using ElasticSearch.

1. Download and extract ElasticSearch from `http://www.elasticsearch.org/ download/`.

2. Go to the extracted ElasticSearch directory and execute the following command to start the ElasticSearch server in the foreground.

   ```
   > bin/elasticsearch -f
   ```

3. Run the following command in a new console to verify your installation.

   ```
   > curl localhost:9200
   {
     "ok" : true,
     "status" : 200,
     "name" : "Outlaw",
     "version" : {
       "number" : "0.19.11",
       "snapshot_build" : false
     },
     "tagline" : "You Know, for Search"
   ```

4. Go to the `$NUTCH_HOME/runtime/deploy` (or `$NUTCH_HOME/runtime/local` in case you are running Nutch in the local mode) directory. Execute the following command to index the data crawled by Nutch in to the ElasticSearch server.

   ```
   > bin/nutch  elasticindex elasticsearch -all
   12/11/01 06:11:07 INFO elastic.ElasticIndexerJob: Starting
   …...
   ```

5. Issue the following command to perform a search:

   ```
   > curl -XGET 'http://localhost:9200/_search?q=hadoop'
   ....
   ```

```
{"took":3,"timed_out":false,
  "_shards":{"total":5,"successful":5,"failed":0},
  "hits":{"total":36,"max_score":0.44754887,
    "hits":[{"_index":"index","_type":"doc","_id": 100 30551  100
        30551 "org.apache.hadoop:http/","_score":0.44754887,
....
```

How it works

Similar to Apache Solr, ElasticSearch too is built using the Apache Lucene text search library. In the preceding steps we export the data crawled by Nutch in to an instance of ElasticSearch for indexing and searching purposes.

We add the -f switch to force the ElasticSearch to run in the foreground to make the development and testing process easier.

```
bin/elasticsearch -f
```

You can also install ElasticSearch as a service as well. Refer to http://www.elasticsearch.org/guide/reference/setup/installation.html for more details on installing ElasticSearch as a service.

We use the ElasticIndex job of Nutch to import the data crawled by Nutch into the ElasticSearch server. The usage of the elasticindex command is as follows:

```
bin/nutch  elasticindex  <elastic cluster name> \
    (<batchId> | -all | -reindex) [-crawlId <id>]
```

The elastic cluster name defaults to *elasticsearch*. You can change the cluster name by editing the cluster.name property in the config/elasticsearch.yml file. The cluster name is used for auto-discovery purposes and should be unique for each ElasticSearch deployment in a single network.

See also

▶   The *Indexing and searching web documents using Apache Solr* recipe of this chapter.

Generating the in-links graph for crawled web pages

The number of links to a particular web page from other pages, the number of in-links, is widely considered a good metric to measure the popularity or the importance of a web page. In fact, the number of in-links to a web page and the importance of the sources of those links have become integral components of most of the popular link analysis algorithms such as PageRank introduced by Google.

In this recipe, we are going to extract the in-links information from a set of web pages fetched by Apache Nutch and stored in Apache HBase backend data store. In our MapReduce program, we first retrieve the out-links information for the set of web pages stored in the Nutch HBase data store and then use that information to calculate the in-links graph for this set of web pages. The calculated in-link graph will contain only the link information from the fetched subset of the web graph only.

Getting ready

Follow the *Whole web crawling with Apache Nutch using an existing Hadoop/HBase cluster* or the *Configuring Apache HBase local mode as the backend data store for Apache Nutch* recipe and crawl a set of web pages using Apache Nutch to the backend HBase data store.

This recipe requires Apache Ant for building the source code.

How to do it

The following steps show you how to extract an out-links graph from the web pages stored in Nutch HBase data store and how to calculate the in-links graph using that extracted out-links graph.

1. Go to `$HBASE_HOME` and start the HBase shell.

   ```
   > bin/hbase shell
   ```

2. Create an HBase table with the name `linkdata` and a column family named `il`. Exit the HBase shell.

   ```
   hbase(main):002:0> create 'linkdata','il'

   0 row(s) in 1.8360 seconds

   hbase(main):002:0> quit
   ```

3. Unzip the source package for this chapter and compile it by executing `ant build` from the `Chapter 7` source directory.

4. Copy the `c7-samples.jar` file to `$HADOOP_HOME`. Copy the `$HBASE_HOME/hbase-*.jar` and `$HBASE_HOME/lib/zookeeper-*.jar` to `$HADOOP_HOME/lib`.

5. Run the Hadoop program by issuing the following command from `$HADOOP_HOME`.

   ```
   > bin/hadoop jar c7-samples.jar chapter7.InLinkGraphExtractor
   ```

6. Start the HBase shell and scan the `linkdata` table using the following command to check the output of the MapReduce program:

```
> bin/hbase shell
hbase(main):005:0> scan 'linkdata',{COLUMNS=>'il',LIMIT=>10}
ROW                              COLUMN+CELL
....
```

How it works

As we are going to use HBase to read input as well as to write the output, we will use the HBase TableMapper and TableReducer helper classes to implement our MapReduce application. We will configure the TableMapper and the TableReducer using the utility methods given in the TableMapReduceUtil class. The `Scan` object is used to specify the criteria to be used by the mapper when reading the input data from the HBase data store.

```
Configuration conf = HBaseConfiguration.create();
Job job = new Job(conf, "InLinkGraphExtractor");
job.setJarByClass(InLinkGraphExtractor.class);
Scan scan = new Scan();
scan.addFamily("ol".getBytes());
TableMapReduceUtil.initTableMapperJob("webpage", scan, ……);
TableMapReduceUtil.initTableReducerJob("linkdata",……);
```

The map implementation receives the HBase rows as the input records. In our implementation, each of the rows corresponds to a fetched web page. The input key to the Map function consists of the web page's URL and the value consists of the web pages linked from this particular web page. Map function emits a record for each of the linked web pages, where the key of a Map output record is the URL of the linked page and the value of a Map output record is the input key to the Map function (the URL of the current processing web page).

```
public void map(ImmutableBytesWritable sourceWebPage, Result
values,……) {
  List<KeyValue> results = values.list();
  for (KeyValue keyValue : results) {
    ImmutableBytesWritable outLink = new
        ImmutableBytesWritable(keyValue.getQualifier());
    try {
      context.write(outLink, sourceWebPage);
    } catch (InterruptedException e) {
      throw new IOException(e);
    }
  }
}
```

The reduce implementation receives a web page URL as the key and a list of web pages that contain links to that web page (provided in the key) as the values. The `reduce` function stores these data in to an HBase table.

```
public void reduce(ImmutableBytesWritable key,
  Iterable<ImmutableBytesWritable> values, ……{

Put put = new Put(key.get());
  for (ImmutableBytesWritable immutableBytesWritable :values)    {
put.add(Bytes.toBytes("il"),immutableBytesWritable.get(),
                                     Bytes.toBytes("link"));
  }
  context.write(key, put);
}
```

See also

▶ The *Running MapReduce jobs on HBase(table input/output)* recipe of *Chapter 5, Hadoop Ecosystem*.

8

Classifications, Recommendations, and Finding Relationships

In this chapter, we will cover:

- ▶ Content-based recommendations
- ▶ Hierarchical clustering
- ▶ Clustering a Amazon sales dataset
- ▶ Collaborative filtering-based recommendations
- ▶ Classification using Naive Bayes Classifier
- ▶ Assigning advertisements to keywords using the Adwords balance algorithm

Introduction

This chapter discusses how we can use Hadoop for more complex use cases such as classifying a dataset, making recommendations, or finding relationships between items.

A few instances of such scenarios are as follows:

- ▶ Recommending products to users either based on the similarities between the products (for example, if a user liked a book about history, he might like another book on the same subject) or based on the user behaviour patterns (for example, if two users are similar, they might like books that each to the other has read).
- ▶ Clustering a data set to identify similar entities. For example, identifying users with similar interests.
- ▶ Classifying data into several groups learning from the historical data.

In this recipe, we will apply these and other techniques using MapReduce. For recipes in this chapter, we will use the *Amazon product co-purchasing network metadata* dataset available from `http://snap.stanford.edu/data/amazon-meta.html`.

Content-based recommendations

Recommendations are to make suggestions to someone about things that might be of interest to him. For example, we would recommend a good book to a friend who has similar interests. We often find use cases for recommendations in online retail. For example, when you browse for a product, Amazon would suggest other products that were also bought by users who bought this item.

For example, online retail sites such as Amazon have a very large collection of items. Although books are found classified into several categories, often each category has too many to browse one after the other. Recommendations make the user's life easy by helping them find the best products for their tastes. As recommendations make a change of a high sale, online retailers are very much interested about recommendation algorithms.

There are many ways to do recommendations:

- ▶ **Content-based recommendations**: One could use information about the product to identify similar products. For example, you could use categories, content similarities, and so on to identify products that are similar and recommend them to the users who have already brought one.

- ▶ **Collaborative filtering**: The other option is to use user's behavior to identify similarities between products. For example, if the same user gave a high rating to the two products, we can argue that there is some similarity between those two products. We will look at an instance of this in the next recipe.

This recipe uses dataset collected from Amazon about products to make content-based recommendations. In the dataset, each product has a list of similar items provided pre-calculated by Amazon. In this recipe, we will use that data to make recommendations.

Getting ready

The following steps depict how to prepare for this recipe.

1. This recipe assumes that you have followed *Chapter 1, Getting Hadoop up and running in a Cluster*, and have installed Hadoop. We will use the HADOOP_HOME to refer to the Hadoop installation directory.

2. Start Hadoop by following the instructions in *Chapter 1, Getting Hadoop up and running in a Cluster*.

3. This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the *Writing the WordCount MapReduce sample, bundling it and running it using standalone Hadoop* recipe in *Chapter 1, Getting Hadoop up and running in a Cluster*.

4. We will use `HADOOP_HOME` to refer to the Hadoop installation directory.

How to do it...

The following steps describe how to run the content-based recommendation recipe.

1. Download the dataset from *Amazon product co-purchasing network metadata* available at `http://snap.stanford.edu/data/amazon-meta.html` and unzip it. We call this directory as `DATA_DIR`.

2. Upload the data to HDFS by running following commands from `HADOOP_HOME`. If the `/data` directory already exists, clean it up.

   `$ bin/hadoopdfs -mkdir /data`

   `$ bin/hadoopdfs -mkdir /data/input1`

   `$ bin/hadoopdfs -put <DATA_DIR>/amazon-meta.txt /data/input1`

3. Unzip the source code for Chapter 8 (`chapter8.zip`). We will call that folder as `CHAPTER_8_SRC`.

4. Change the `hadoop.home` property in the `CHAPTER_8_SRC/build.xml` file to point to your Hadoop installation directory.

5. Compile the source by running the `ant build` command from the `CHAPTER_8_SRC` directory.

6. Copy `build/lib/hadoop-cookbook-chapter8.jar` to your `HADOOP_HOME`.

7. Run the first Map reduce job through the following command from `HADOOP_HOME`.

   `$ bin/hadoopjar hadoop-cookbook-chapter8.jar chapter8.`
   `MostFrequentUserFinder/data/input1 /data/output1`

8. Read the results by running the following command:

   `$ bin/hadoopdfs -cat /data/output1/*`

9. You will see that the MapReduce job has extracted the purchase data from each customer, and the results will look like following:

```
customerID=A1002VY75YRZYF,review=ASIN=0375812253#title=Really
Useful Engines (Railway Series)#salesrank=623218#group=Book #ratin
g=4#similar=0434804622|0434804614|0434804630|0679894780|0375827439
|,review=ASIN=B000002BMD#title=EverythingMustGo#salesrank=77939#gr
oup=Music#rating=4#similar=B00000J5ZX|B000024J5H|B00005AWNW|B00002
5KKX|B0000008I2Z
```

10. Run the second Map reduce job through the following command from `HADOOP_HOME`.

    ```
    $ bin/hadoopjar hadoop-cookbook-chapter8.jar chapter8.
    ContentBasedRecommendation/data/output1 /data/output2
    ```

11. Read the results by running the following command:

    ```
    $ bin/hadoopdfs -cat /data/output2/*
    ```

You will see that it will print the results as follows. Each line of the result contains the customer ID and list of product recommendations for that customer.

```
A10003PM9DTGHQ  [0446611867, 0446613436, 0446608955, 0446606812,
0446691798, 0446611867, 0446613436, 0446608955, 0446606812,
0446691798]
```

How it works...

The following listing shows an entry for one product from the dataset. Here, each data entry includes an ID, title, categorization, similar items to this item, and information about users who has brought the item.

```
Id:   13
ASIN: 0313230269
title: Clockwork Worlds : Mechanized Environments in SF (Contributions
to the Study of Science Fiction and Fantasy)
group: Book
salesrank: 2895088
similar: 2  1559360968  1559361247
categories: 3
   |Books[283155]|Subjects[1000]|Literature & Fiction[17]|History &
Criticism[10204]|Criticism & Theory[10207]|General[10213]
   |Books[283155]|Subjects[1000]|Science Fiction & Fantasy[25]|Fantasy
[16190]|History & Criticism[16203]
   |Books[283155]|Subjects[1000]|Science Fiction & Fantasy[25]|Science
Fiction[16272]|History & Criticism[16288]
reviews: total: 2  downloaded: 2  avg rating: 5
    2002-8-5  customer: A14OJS0VWMOSWO  rating: 5  votes:   2
helpful:   1
    2003-3-21  customer: A2C27IQUH9N1Z  rating: 5  votes:   4
helpful:   4
```

We have written a new Hadoop data format to read and parse the Amazon product data, and the data format works similar to the format we have written in the *Simple Analytics using MapReduce* recipe in *Chapter 6, Analytics*. The source files `src/chapter8/AmazonDataReader.java` and `src/chapter8/AmazonDataFormat.java` contain the code for the Amazon data formatter.

Amazon data formatter will parse the dataset and emit the data about each Amazon product as key-value pairs to the map function. The data about each Amazon product is represented as string and the class `AmazonCustomer.java` class includes code to parse and write out the data about the Amazon customer.
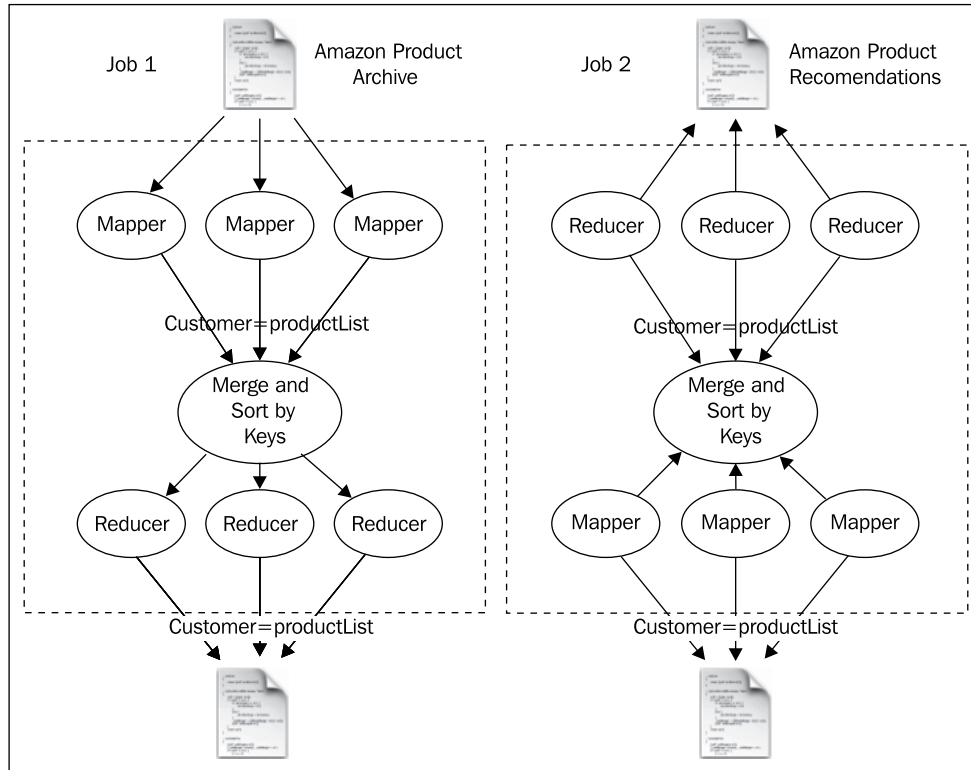
This recipe includes two MapReduce tasks. The source for those tasks can be found from `src/chapter8/MostFrequentUserFinder.java` and `src/chapter8/ContentBasedRecommendation.java`.

```
public void map(Object key, Text value, Context context)
  throws IOException, InterruptedException
{
  List<AmazonCustomer>customerList =
    AmazonCustomer. parseAItemLine(value.toString());
    for(AmazonCustomer customer:
    customerList){ context.write(new Text(customer.customerID),
      new Text(customer.toString()));
}
```

The map task of the first MapReduce job receives data about each product in the logfile as a different key-value pair. When the map task receives the product data, it emits the customer ID as the key and the product information as the value for each customer who has bought the product.

Hadoop collects all values for the key, and invokes the reducer once for each key. There will be a reducer task for each customer, and each reducer task will receive all products that have been brought by a customer. The reducer emits the list of items brought by each customer, thus building a customer profile. For limiting the size of the dataset, the reducer will not emit any customer who has brought less than five products.

```
public void reduce(Text key, Iterable<Text> values, Context context)
  throws IOException, InterruptedException
  {
    AmazonCustomer customer = new AmazonCustomer();
    customer.customerID = key.toString();
    for(Text value: values)
    {
      Set<ItemData>itemsBrought = newAmazonCustomer(
        value.toString()).itemsBrought;
      for(ItemDataitemData: itemsBrought)
      {
        customer.itemsBrought.add(itemData);
      }
    }
    if(customer.itemsBrought.size() > 5)
    {
      context.write( newIntWritable(customer.itemsBrought.size()),
        new Text(customer.toString()));
    }
  }
```

The second MapReduce job uses the data generated from the first MapReduce tasks to make recommendations for each customer. The map task receives data about each customer as the input, and the MapReduce tasks make recommendations using the following three steps:

1. Each product (item) data from Amazon includes similar items to that item. Given a customer, first the map task creates a list of all similar items for each item that customer has brought.

2. Then the map task removes any item from the list of similar items that has already brought by the same customer.

3. Then map task selects ten items as recommendations.

Here reducer only prints out the results.

```
public void map(Object key, Text value, Context context)
  throwsIOException, InterruptedException
{
  AmazonCustomeramazonCustomer = newAmazonCustomer(
    value.toString() .replaceAll("[0-9]+\\s+", ""));
  List<String>recemndations = new ArrayList<String>();
  for (ItemDataitemData : amazonCustomer.itemsBrought)
  {
    recemndations.addAll(itemData.similarItems);
  }
  for (ItemDataitemData : amazonCustomer.itemsBrought)
  {
  recemndations.remove(itemData.itemID);
  }
  ArrayList<String>finalRecemndations = newArrayList<String>();
  for (inti = 0; i<Math.min(10, recemndations.size());  i++)
  {
    finalRecemndations.add(recemndations.get(i));
  }
  context.write(new Text(amazonCustomer.customerID),
    new Text(finalRecemndations.toString()));
  }
public void reduce(Text key, Iterable<Text> values, Context
context)
  throws IOException, InterruptedException
{
  for(Text value: values)
  {
    context.write(key, value);
  }
}
```

There's more...

You can learn more about content-based recommendations in *Chapter 9*, *Recommendation Systems*, of *Mining of Massive Datasets*, *Anand Rajaraman and Jeffrey D. Ullman*. This book can be found from `http://infolab.stanford.edu/~ullman/mmds.html`.

Hierarchical clustering

Many operations such as recommendations and finding relationships use clustering as an integral component. Clustering groups a dataset into several groups using one or more measurements such that the items in the same cluster are rather similar and items in different clusters are more different. For example, given a set of living addresses of patients, clustering can group them into several groups where members of each group are close to each other so that doctors can visit them in most optimal manner.

There are many clustering algorithms; each has different costs and accuracy. Hierarchical clustering is one of the most basic and most accurate algorithms. It works as follows:

1. First, hierarchical clustering assigns each data point to its own cluster.
2. It calculates the distance between all cluster pairs and merges the two clusters that are closest to each other.
3. It repeats the process until only one cluster is left. At each repetition, the algorithm records each cluster pair it has merged. This merging history provides a tree that combines the clusters into larger and larger groups close to the root. Users may take a cut at some place in the tree based on the number of clusters they need.

However, hierarchical clustering has the complexity of $O(n^2 log(n))$. In other words, the algorithm would take about $Cn^2 log(n)$ time for input of size *n* and a constant *C*. Hence, often it is too complex to be used with a large dataset. However, it often serves as the basis for many other clustering algorithms.

In this recipe, we will use hierarchical clustering to cluster a sample drawn from the Amazon dataset. It is worth noting that it is not a MapReduce algorithm, but we will use its results in the MapReduce algorithm in the next recipe.

Getting ready

This recipe assumes that you have followed the first recipe and extracted the Amazon product data.

How to do it...

The following steps describe how to run the hierarchical clustering sample:

1. Copy the results of the first MapReduce job of the last recipe to the local machine.
   ```
   $ bin/hadoopdfs -get /data/output1/ product-sales.data
   ```

2. Run the following command to do hierarchical clustering:

   ```
   $ java –cphadoop-cookbook-chapter8.jar chapter8.
   HirachicalClusterProcessor product-sales.data
   ```

3. The algorithm will generate a file called `clusters.data` that includes clusters.

4. You can find the information about clusters from the `clusters.data` file created in the local directory, which will have the centroid of each cluster. Furthermore, the `clusters-raw.data` file includes all the points assigned to each cluster.

How it works...

You can find the source for the recipe from `src/chapter8/ HirachicalClusterProcessor.java`.

When executed, the code would read through the input data file and load data for 1000 Amazon customers randomly selected from the input file and perform hierarchical clustering on those customers.

Hierarchical clustering starts by placing each customer in its own cluster. Then it finds the two clusters that are close to each other and merges them into one cluster. Then it recalculates the distance between the new cluster and the old clusters, and repeats the process until it is left with a single cluster.

The `getDistance()` method of the `AmazonCustomer.java` class under `src/chapter8` shows how the distance between two Amazon clusters is calculated. It uses a variation of Jaccard distance. With the Jaccard distance, if two customers have brought the same item and have given similar reviews to those items, then Jaccard distance will be small. On the other hand, if they have given different reviews, the distance between them will be high.

```java
public double getDistance(ClusterablePoint other)
{
  double distance = 5;
  AmazonCustomer customer1 = (AmazonCustomer)this;
  AmazonCustomer customer2 = (AmazonCustomer)other;
  for(ItemData item1:customer1.itemsBrought)
  {
    for(ItemData item2:customer2.itemsBrought)
    {
      if(item1.equals(item2))
      {
        doubleratingDiff = Math.abs(item1.rating - item2.rating);
        if(ratingDiff< 5)
        {
        distance = distance - (5 - ratingDiff)/5;
        }
      }
```

```
        else
        {
          distance = distance + (5 - ratingDiff)/5;
        }
      }
    }
  }
  returnMath.min(10,Math.max(0.5, distance));
  }
```

A naive implementation of the algorithm will recalculate all distances between clusters every time two clusters are merged, and resulting algorithm will have *O(n³)* **computational complexity**. In other words, the algorithm will take *Cn³* amount of time to run with input of size *n* for some constant *C*. However, by remembering distances between clusters and only calculating distance from new clusters, the resulting implementation will have *O(n²log(n))* complexity. The following code listing shows the implementation of the algorithm.

```
public List<Cluster>doHirachicalClustering(List<ClusterablePoint>
  points)
{
  List<Cluster> clusters =
    new ArrayList<HirachicalClusterProcessor.Cluster>();
  for(ClusterablePointpoint:points)
  {
    clusters.add(new Cluster(point));
  }
  for(inti =0;i<clusters.size();i++)
  {
    for(int j =(i+1);j<clusters.size();j++)
      {
        ClusterPairclusterPair = newClusterPair(clusters.get(i),
        clusters.get(j));
        addNewPairs(clusterPair);
      }
   }
  while(clusters.size() > 1)
  {
    ClusterPairnextPair = null;
    doubleminDist = Double.MAX_VALUE;
    for(ClusterPair pair:
    pairsSortedByDistance)
    {
    if(pair.distance<minDist)
      {
        nextPair = pair;
```

```
        minDist = pair.distance;
      }
    }
   ClusterPair pair = nextPair;
  }
  removePairs(pair);
  Cluster newCluster = pair.merge();
  clusters.remove(pair.cluster1);
  clusters.remove(pair.cluster2);
  //recalcualte pairs for new cluster
  for(Cluster cluster: clusters)
  {
   ClusterPairnewclusterPair = newClusterPair(cluster, newCluster);
   addNewPairs(newclusterPair);
   }
  clusters.add(newCluster);
 }
  return clusters;
}
```

Finally, the program will output the centroid for each cluster. The centroid is the point within the cluster that has the minimal value of the sum of distances to all other points in the cluster.

There's more...

Among other alternative distance measures are Cosine distance, Edit distance, and Hamming distance. *Chapter 7*, *Clustering*, of the book *Mining of Massive Datasets*, *Anand Rajaraman* and *Jeffrey D. Ullman*, explains these distances in detail. Also you can learn more about hierarchical clustering from the same book. The book can be found from `http://infolab.stanford.edu/~ullman/mmds.html`.

Clustering an Amazon sales dataset

As explained in the earlier recipe, although very accurate, hierarchical clustering has a time complexity of $O(n^2 log(n))$, and therefore is not applicable to large datasets. For example, the Amazon data set we will use has about 0.2 million entries and it will need about 10 tera calculations (about 10^{13}).

This recipe describes how to apply the GRGPFClustering algorithm to Amazon sales dataset. We can find the distance between any two data points (products) using items they have co-purchased, but we do not know how to place those data points in some **multi-dimensional space** (2D or 3D space) with **orthogonal axes**. Therefore, we say that the data in a **non-Euclidian space**.

GRGPFClustering algorithm is a highly scalable algorithm applicable to a non-Euclidian space. It works by first taking a small **random sample** of the data and clustering it using an algorithm such as hierarchical clustering, and then using those clusters to cluster a large dataset without keeping all the data in memory.

Getting ready

The following steps describe how to ready to cluster the Amazon dataset.

1. This assumes that you have followed *Chapter 1*, *Getting Hadoop up and running in a Cluster* and have installed Hadoop. We will use the HADOOP_HOME to refer to the Hadoop installation directory.

2. Start Hadoop following the instructions in *Chapter 1*, *Getting Hadoop up and running in a Cluster*.

3. This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the Writing the *WordCount MapReduce sample, bundling it and running it using standalone Hadoop* recipe in *Chapter 1*, *Getting Hadoop up and running in a Cluster*.

4. This assumes that you have followed the *Content-based recommendations* recipe of this chapter and have extracted the Amazon product data.

How to do it...

The following steps describe how to get cluster the Amazon dataset.

1. Unzip the source code for Chapter 8 (chapter8.zip). We will call that folder CHAPTER_8_SRC.

2. Change the hadoop.home property in the build.xml file under CHAPTER_8_SRC to point to your Hadoop installation directory.

3. Compile the source by running the ant build command from the CHAPTER_8_SRC directory.

4. Copy build/lib/hadoop-cookbook-chapter8.jar to your HADOOP_HOME.

5. Run the MapReduce job using the following command from HADOOP_HOME. It will use the output generated by the first MapReduce task of the first recipe.

   $bin/hadoopjar hadoop-cookbook-chapter8.jar chapter8.
   GRGPFClustering /data/output1 /data/output3

6. Read the results by running the following command.

   $bin/hadoopdfs -cat /data/output3/*

You will see that it will print the results as following. Here the key indicates the cluster ID and the rest of the results show customer details of the customers assigned to the cluster.

```
customerID=A3S764AIU7PCLT,clusterID=0,
review=ASIN=0140296468#title=The New New Thing: A Silicon Valley Story
..
```

How it works...

As the figure depicts, this tasks has a MapReduce task. You can find the source for the recipe from `src/chapter8/GRGPFClustering.java`.
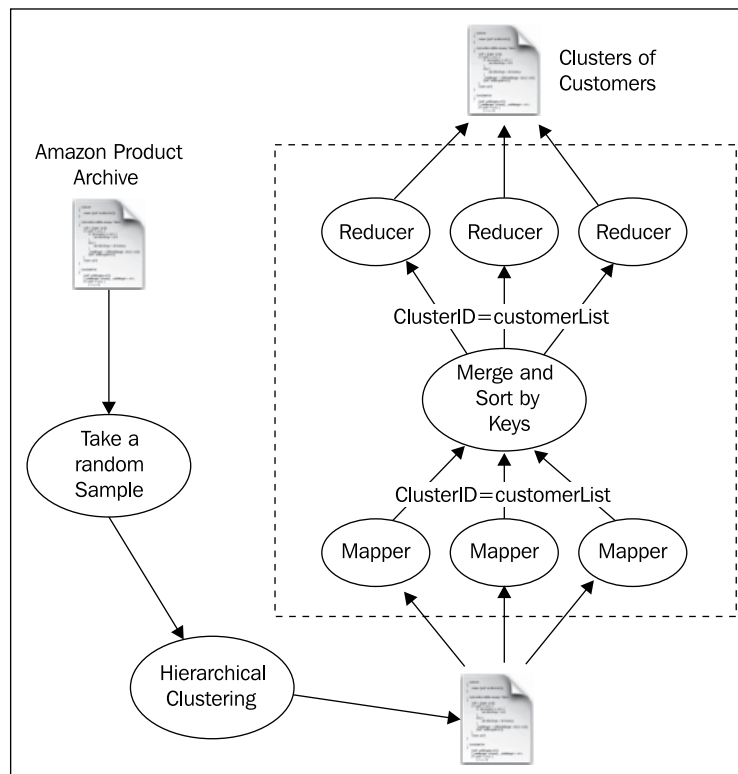
When initialized, the MapReduce job will load the information about the clusters calculated in the earlier recipe and use those clusters to assign the rest of the dataset to clusters.

```
public void map(Object key, Text value, Context context)
  throws IOException, InterruptedException
{
  AmazonCustomeramazonCustomer =
```

```
      new AmazonCustomer(value.toString() .replaceAll("[0-9]+\\s+",
""));
  doublemindistance = Double.MAX_VALUE;
  AmazonCustomerclosestCluster = null;
  for (AmazonCustomercentriod : clusterCentrodis)
  {
  double distance = amazonCustomer.getDistance(centriod);
  if (distance <mindistance)
  {
    mindistance = distance;
    closestCluster = centriod;
  }
  amazonCustomer.clusterID = closestCluster.clusterID;
  }
  context.write(new Text(closestCluster.clusterID),
    new Text(amazonCustomer.toString()));
}
```

The Map task receives each line in the logfile that contains information about Amazon customer as a different key-value pair. Then, the map task compares the information about the customer against each of the cluster's centroids, and assigns each customer to the cluster that is closest to that customer. Then, it emits the name of the cluster as the key and information about the customer as the value.

Then, Hadoop collects all values for the different keys (clusters) and invokes the reducer once for each cluster. Then each reducer emits the members of each cluster.

```
public void reduce(Text key, Iterable<Text> values,
  Context context) throws IOException, InterruptedException
{
  for (Text value : values)
  {
    context.write(key, value);
  }
}
```

The main method of the job works similar to the earlier recipes.

There's more...

It is possible to improve the results by recalculating the cluster centroids in the reducer, splitting any clusters that have customers that are too far apart from each others, and rerunning the GRGPF algorithm with new clusters. More information about this can be found from *Chapter 7*, *Clustering*, of the book *Mining of Massive Datasets*, *Anand Rajaraman* and *Jeffrey D. Ullman*. The book can be found from `http://infolab.stanford.edu/~ullman/mmds.html`.

Collaborative filtering-based recommendations

This recipe will use collaborative filtering to make recommendations for customers in the Amazon dataset. As described in the introduction, collaborative filtering uses sales activities about a given user that is common with other users to deduce the best product recommendations for the first user.

To implement collaborative filtering, we will cluster the users based on their behavior, and use items brought by members of a cluster to find recommendations of each member of the cluster. We will use the clusters calculated in the earlier recipe.

Getting ready

The following steps show how to prepare to run the collaborative filtering example:

1. This assumes that you have followed *Chapter 1, Getting Hadoop up and running in a Cluster* and have installed Hadoop. We will use the HADOOP_HOME to refer to Hadoop installation directory.

2. Start Hadoop by following the instructions in the first chapter.

3. This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the *Writing a WordCount MapReduce Sample, Bundling it and running it using standalone Hadoop* recipe from the first chapter.

4. This will use the results from the previous recipe of this chapter. Follow it if you have not done so already.

How to do it...

The following steps show how to prepare to run the collaborative filtering example:

1. Run the MapReduce job using the following command from HADOOP_HOME:

   $ bin/hadoopjar hadoop-cookbook-chapter8.jar chapter8. ClusterBasedRecommendation/data/output3 /data/output4

2. Read the results by running the following command.

   $bin/hadoopdfs -cat /data/output4/*

You will see that it will print the results as following. Here the key is the customer ID and the value is the list of recommendations for that customer.

```
A1JDFNG3KI9D1V  [ASIN=6300215539#title=The War of the Worlds#salesrank
=1#group=Video#rating=4#, ..]
```

How it works...
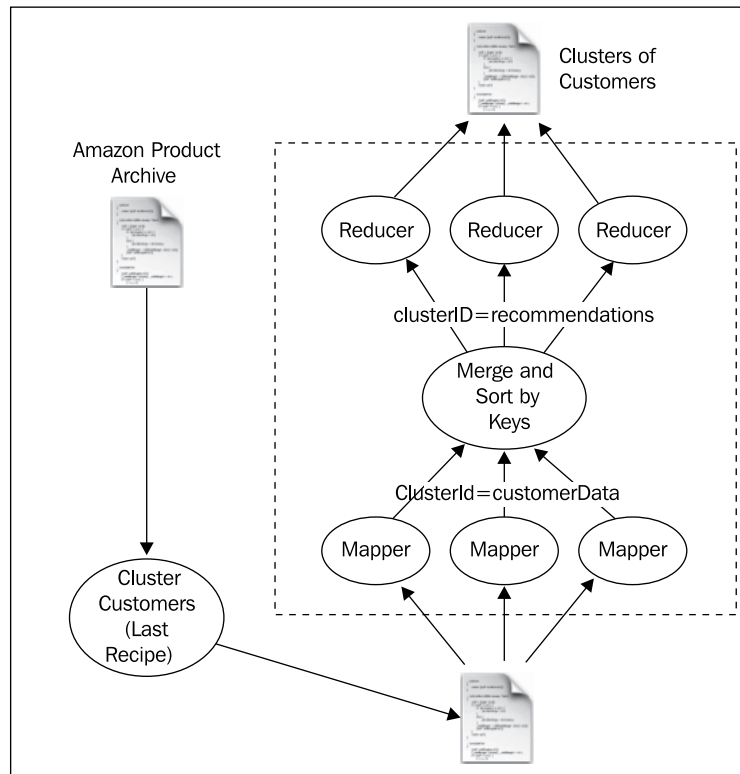
Collaborative filtering uses the behavior of the users to decide on the best recommendations for each user. For that process, the recipe will use the following steps:

1. Group customers into several groups such that similar customers are in the same group and different customers are in different groups.

2. For making recommendations for each customer, we have looked at the other members in the same group and used the items bought by those members assuming that similar users would like to buy similar products.

3. When there are many recommendations, we have used the Amazon sales rank to select the recommendations.

For grouping the customers, we can use clustering techniques used in the earlier recipes. As a measure of the distance between customers we have used the distance measure introduced in the second recipe of this chapter that uses customer co-purchase information to decide on the similarity between customers.

We have already clustered the customers to different groups in the earlier recipe. We would use those results to make recommendations.

You can find the source for the recipe from `src/chapter8/`
`ClusterBasedRecommendation.java`. The map task for the job will look like the following:

```
public void map(Object key, Text value, Context context)
  throws IOException, InterruptedException {
{
  AmazonCustomeramazonCustomer =
    newAmazonCustomer(value.toString() .replaceAll("[0-9]+\\s+", ""));
  context.write(new Text(amazonCustomer.clusterID),
    new Text(amazonCustomer.toString()));
}
```

The map task receives each line in the logfile as a different key-value pair. It parses the lines using regular expressions and emits cluster ID as the key and the customer information as the value.

Hadoop will group different customer information emitted against the same customer ID and call the reducer once for each customer ID. Then, the reducer walks through the customers assigned to this cluster and creates a list of items as potential recommendations sorted by Amazon sales rank. Then it makes final recommendations for a given user by removing any items that he has already brought.

```
public void reduce(Text key, Iterable<Text> values, Context context)
  throws IOException, InterruptedException
{
  List<AmazonCustomer>customerList =
      newArrayList<AmazonCustomer>();
  TreeSet<AmazonCustomer.SortableItemData> highestRated1000Items =
    newTreeSet<AmazonCustomer.SortableItemData>();
  for (Text value : values)
  {
    AmazonCustomer customer =
      newAmazonCustomer(value.toString());
    for (ItemDataitemData : customer.itemsBrought)
    {
      highestRated1000Items.add(
        customer.newSortableItemData(itemData));
      if (highestRated1000Items.size() > 1000)
      {
        highestRated1000Items.remove(highestRated1000Items.last());
      }
    }
    customerList.add(customer);
```

```
    }

    for (AmazonCustomeramazonCustomer : customerList)
    {
      List<ItemData>recemndationList =
        newArrayList<AmazonCustomer.ItemData>();
      for (SortableItemDatasortableItemData : highestRated1000Items)
      {
        if (!amazonCustomer.itemsBrought
         .contains(sortableItemData.itemData))
        {
          recemndationList.add(sortableItemData.itemData);
        }
      }
      ArrayList<ItemData>finalRecomendations =
        newArrayList<ItemData>();
      for (inti = 0; i< 10; i++)
      {
        finalRecomendations.add(recemndationList.get(i));
      }

      context.write(new Text(amazonCustomer.customerID),
        new Text(finalRecomendations.toString()));
    }
  }
```

The main method of the job will work similar to the earlier recipes.

Classification using Naive Bayes Classifier

A **classifier** assigns inputs into one of the *N* classes based on some properties (features) of inputs. Classifiers have widespread applications such as e-mail spam filtering, finding most promising products, selecting customers for closer interactions, and taking decisions in machine learning situations, and so on. Let us explore how to implement a classifier using a large dataset. For instance, a spam filter will assign each e-mail to one of the two clusters—spam mail or not a spam mail.

There are many classification algorithms. One of the simplest, but effective algorithms is Naive Bayesian classifier that uses the Bayes theorem. You can find more information about Bayesian classifier from `http://en.wikipedia.org/wiki/Naive_Bayes_classifier` and Bayes theorem from `http://betterexplained.com/articles/an-intuitive-and-short-explanation-of-bayes-theorem/`.

For this recipe, we will also focus on the Amazon purchase dataset as before. We will look at several features about a product such as number of reviews received, amount of positive ratings, and number of known similar items to identify a product as potential to be within the first 10,000 sales rank. We will use the Naive Bayesian classifier for classifications.

Getting ready

The following steps describe how to prepare to run Naive Bayesian example:

1. This assumes that you have followed *Chapter 1*, *Getting Hadoop up and running in a Cluster* and have installed Hadoop. We will use the HADOOP_HOME to refer to Hadoop installation directory.

2. Start Hadoop by following the instructions in the first chapter.

3. This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the *Writing the WordCount MapReduce sample, bundling it and running it using standalone Hadoop* recipe from *Chapter 1*, *Getting Hadoop up and running in a Cluster*.

How to do it...

The following steps describe how to run Naive Bayesian example.

1. Download the dataset from Amazon product co-purchasing network metadata, `http://snap.stanford.edu/data/amazon-meta.html` and unzip it. We will call this DATA_DIR.

2. Upload the data to HDFS by running the following commands from HADOOP_HOME. If the `/data` directory is already there, clean it up.

   `$ bin/hadoopdfs -mkdir /data`

   `$ bin/hadoopdfs -mkdir /data/input1`

   `$ bin/hadoopdfs -put <DATA_DIR>/amazon-meta.txt /data/input1`

3. Unzip the source code for chapter 8 (`chapter8.zip`). We will call that folder CHAPTER_8_SRC.

4. Change the `hadoop.home` property in the `CHAPTER_8_SRC/build.xml` file to point to your Hadoop installation directory.

5. Compile the source by running the `ant build` command from the CHAPTER_8_SRC directory.

6. Copy `build/lib/hadoop-cookbook-chapter8.jar` to your HADOOP_HOME.

7. Run the MapReduce job through the following command from HADOOP_HOME.

   `$ bin/hadoopjar hadoop-cookbook-chapter8.jar chapter8.`
   `NavieBayesProductClassifer/data/input1 /data/output5`

8. Read the results by running the following command.

   ```
   $ bin/hadoopdfs -cat /data/output5/*
   ```

9. You will see that it will print the results as following. You can use these values with Bayesian classifier to classify the inputs.

   ```
   postiveReviews>30        0.635593220338983
   reviewCount>60   0.62890625
   similarItemCount>150     0.5720620842572062
   ```

10. Verify the classifier using the following command.

    ```
    $ bin/hadoop-cp hadoop-cookbook-chapter8.jar chapter8.
    NavieBayesProductClassifer
    ```

How it works...

The goal of this recipe is to look at some properties of a product and predict whether it will fall under the first 10,000 products at Amazon by the sales rank. We call these properties *features*, and for this sample we will focus on the following three properties:

▸ The number of review counts for a given a product (`p.reviewCount`)

▸ The number of positive reviews for a given a product (`p.positiveReviews`)

▸ The number of similar items for a given a product (`p.similarItemCount`)

In the following discussion, we will write *P(p.salesrank<1000)* to mean that the probability that the given item *p* is within first 10,000 products and similar for other properties as well.

In this recipe, given a new product *p*, we want to find *P(p.salesrank< 1000)* based on statistical distribution features in the products. Furthermore, we need to use MapReduce for the calculations.

The first step is to understanding the equation for Naive Bayes Classifier. If A_p, B_p, and C_p are any three independent events (for example, A means *p.reviewCount> 60*) about a product *p*, and the following three a, b, and c are defined as follows, then we can write the equation for Naive Bayes Classifier.

a = P(Ap/ p.salesrank< 1000)

b = P(Bp/ p.salesrank< 1000)

c = P(Cp/ p.salesrank< 1000)

Here the slash suggests the conditional probability. For example, we read the first line as *a is the probability of event Ap occurring given that p.salesrank< 1000 has already occurred*.

Then using Bayes theorem we can write the following. The following equation provides the probability that the product will have a sales rank less than 1000 given three independent events A_p, B_p, and C_p.

P(p.salesrank< 1000/ A_p and B_p and C_p) = abc/(abc –(1-a).(1-b).(1-c)).

Now let us focus on the real calculation. As A_p, B_p, C_p, we will use following.

- ▸ **A_p**: This is the probability that given item has more than 60 reviews
- ▸ **B_p**: This is the probability that given item has more than 30 positive reviews
- ▸ **C_p**: This is the probability that given item has more than 150 similar items.

Then, we first run the MapReduce task to calculate the probabilities—*a=P(Ap/p. salesrank<1000)*, *b=P(Bp/p.salesrank<1000)*, and *c=P(Cp/p.salesrank<1000)*. Then we will use those with above formula to classify a given product. You can find the source for the classifier from `src/chapter8/NavieBayesProductClassifer.java`. The mapper function looks like the following:

```java
private static final Text TOTAL = new Text("total");
private static final Text RCOUNT_GT_60 =
  new Text("reviewCount>60");
private static final Text PREIVEWS_GT_30 =
  new Text("postiveReviews>30");
private static final Text SIMILAR_ITEMS_GT_150 =
  new Text("similarItemCount>150");

public void map(Object key, Text value, Context context)
  throwsIOException, InterruptedException
  {
    List<AmazonCustomer>customerList =
      AmazonCustomer.parseAItemLine(value.toString());
    intsalesRank = -1;
    intreviewCount = 0;
    intpostiveReviews = 0;
    intsimilarItemCount = 0;

    for (AmazonCustomer customer : customerList)
    {
      ItemDataitemData = customer.itemsBrought.iterator().next();
      reviewCount++;
      if (itemData.rating> 3)
      {
        postiveReviews++;
      }
      similarItemCount = similarItemCount +
```

```
      itemData.similarItems.size();
    if (salesRank == -1)
    {
      salesRank = itemData.salesrank;
    }
  }

  boolean isInFirst10k = (salesRank<= 10000);
  context.write(TOTAL, new BooleanWritable(isInFirst10k));
  if (reviewCount> 60)
  {
    context.write(RCOUNT_GT_60,
      newBooleanWritable(isInFirst10k));
  }
  if (postiveReviews> 30)
  {
    context.write(PREIVEWS_GT_30,
      newBooleanWritable(isInFirst10k));
  }
  if (similarItemCount> 150)
  {
    context.write(SIMILAR_ITEMS_GT_150,
      newBooleanWritable(isInFirst10k));
  }
}
```

The mapper function walks thorugh each product and for each, it evaluates the features. If the feature evaluates to be true, it emits the feature name as the key and notifies whether the product is within the first 10,000 products as the value.

MapReduce invokes the reducer once for each feature. Then each reduce job receives all values for which the feature is true, and it calculates the probability that given the feature is true, the product is within the first 10,000 products in the sales rank.

```
public void reduce(Text key, Iterable<BooleanWritable> values,
  Context context) throws IOException, InterruptedException
{
  int total = 0;
  int matches = 0;
  for (BooleanWritable value : values)
  {
    total++;
    if (value.get())
    {
      matches++;
```

```
    }
  }
  context.write(new Text(key),
    newDoubleWritable((double) matches / total));
}
```

Given a product, we will examine and decide following:

- ▸ Does it have more than 60 reviews?
- ▸ Does it have more than 30 positive reviews?
- ▸ Does it have more than 150 positive items?

We would use the above to decide what are the events A, B, C and we can calculate a, b, and c accordingly using P1, P2, and P3 calculated using MapReduce task. The following code implements this logic:

```
public static booleanclassifyItem(intsimilarItemCount,
  intreviewCount, intpostiveReviews)
{
  double reviewCountGT60 = 0.8;
  double postiveReviewsGT30 = 0.9;
  double similarItemCountGT150 = 0.7;
  double a , b, c;

  if (reviewCount> 60)
  {
    a = reviewCountGT60;
  }
  else
  {
    a= 1 - reviewCountGT60;
  }
  if (postiveReviews> 30)
  {
    b = postiveReviewsGT30;
  }
  else
  {
  b = 1- postiveReviewsGT30;
  }
  if (similarItemCount> 150)
    {
      c = similarItemCountGT150;
    }
    else
```

```
    {
      c = 1- similarItemCountGT150;
    }
    double p = a*b*c/ (a*b*c + (1-a)*(1-b)*(1-c));
    return p > 0.5;
}
```

When you run the classifier testing the logic, it will load the data generated by the MapReduce job and classify 1000 randomly selected products.

Assigning advertisements to keywords using the Adwords balance algorithm

Advertisements have become a major medium of revenue for the Web. It is a billion-dollar business, and the source of the most Google revenue. Further, it has made it possible for companies such as Google to run their main services free of cost, while collecting their revenue through advertisements.

Let us explore how we can implement a simple "Adwords" style algorithm using MapReduce.

Adwords lets people bid for keywords. For example, the advertiser "A" can bid for keywords "Hadoop Support" for 2$ and provided a maximum budget of 100$, and the advertiser "B" would bid for keywords "Hadoop Support" for 1.50$ and provided a maximum budget of 200$. When a user searches for a document with given keywords, the system will choose one or more advertisements among the bids for these keywords. Advertisers will pay only if a user clicks on the advertisement.

Adwords problem is to show advertisements such that it will maximize revenue. There are several factors in the play while designing such a solution:

> ▸ Only user clicks, not showing the advertisement, will get us money. Hence, we want to show advertisements that are more likely to be clicked often. We measure this as fraction of time an advertisement was clicked as oppose to how many times it was shown. We call this "**click-through rate**" for a keyword.

> ▸ We want to show people with large budgets, as those are likely be ones that are hard to spend as opposed to smaller budgets.

In this recipe, we will implement a simplified version of the Adwords balance algorithm that can be used in such situations. For simplicity, we will assume that advertisers only bid on single words. Also, as we cannot find a real bid dataset, we will generate a sample bid dataset.
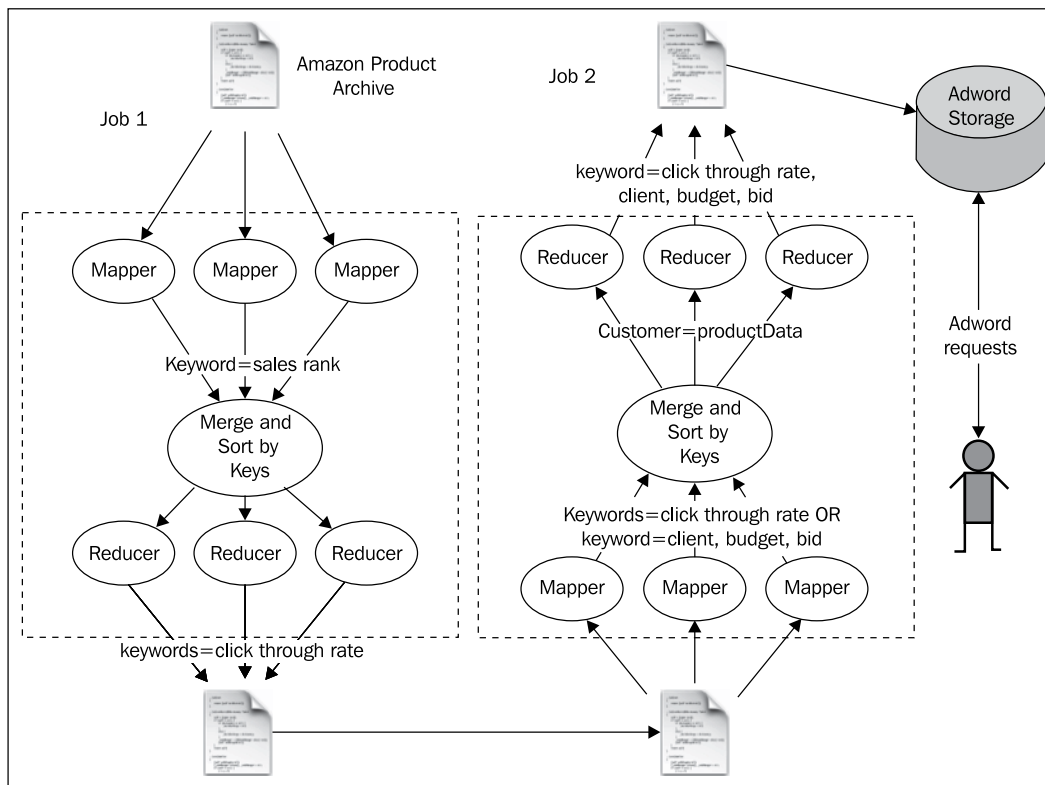
Assume that you are to support a keyword-based advertisement using the Amazon dataset. The recipe will work as follows:

▸ The first MapReduce task will approximate the click-through rate of the keyword using Amazon sales index. Here, we assume that the keywords that are found in the title of the products with higher sales rank will have a better click-through rate.

▸ Then we will run a Java task to generate a bid dataset.

▸ Then the second MapReduce task will group bids for the same product together and create an output that is appropriate to be used by advertisement assignment program.

▸ Finally, we will use an advertisement assignment program to assign keywords to advertisers. We would use Adword balance algorithm, which uses the following formula. The following formula assigns priority based on the fraction of unspent budget owned by each advertiser, bid value, and click-through rate.

*Measure = bid value * click-through rate * (1-e^(-1*current budget/ initial budget))*

Getting ready

The following steps describe how to prepare to run the Adwords sample:

1. This assumes that you have followed *Chapter 1, Getting Hadoop up and running in a Cluster* and have installed Hadoop. We will use `HADOOP_HOME` to refer to Hadoop installation directory.

2. Start Hadoop by following the instructions in *Chapter 1, Getting Hadoop up and running in a Cluster*.

3. This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the *Writing the WordCount MapReduce sample, bundling it and running it using standalone Hadoop* recipe from *Chapter 1, Getting Hadoop up and running in a Cluster*.

How to do it...

The following steps describe how to run the Adwords sample:

1. Download the dataset from Amazon product co-purchasing network metadata, `http://snap.stanford.edu/data/amazon-meta.html` and unzip it. We call this `DATA_DIR`.

2. Upload the data to HDFS by running following commands from `HADOOP_HOME`. If the `/data` directory is already there, clean it up. This dataset is large, and might take a long time if you try to run it with a single computer. You might want to only upload the first 50,000 lines or so of the dataset if you need the sample to run quickly.

   ```
   $ bin/hadoopdfs -mkdir /data
   ```

   ```
   $ bin/hadoopdfs -mkdir /data/input1
   ```

   ```
   $ bin/hadoopdfs -put <DATA_DIR>/amazon-meta.txt /data/input1
   ```

3. Unzip the source code for Chapter 8 (`chapter8.zip`). We will call that folder `CHAPTER_8_SRC`.

4. Change the `hadoop.home` property in the `CHAPTER_8_SRC/build.xml` file to point to your Hadoop installation directory.

5. Compile the source by running the `ant build` command from the `CHAPTER_8_SRC` directory.

6. Copy the `build/lib/hadoop-cookbook-chapter8.jar` to your `HADOOP_HOME`.

7. Run the Map reduce job through the following command from `HADOOP_HOME`.

   ```
   $ bin/hadoopjar hadoop-cookbook-chapter8.jar chapter8.adwords.
   ClickRateApproximator/data/input1 /data/output6
   ```

8. Download the results to your computer by running the following command:

   `$ bin/hadoopdfs -get /data/output6/part-r-00000clickrate.data`

9. You will see that file contains the results as following. You can use these values with Bayes classifier to classify the inputs.

   ```
   keyword:(Annals 74
   keyword:(Audio  153
   keyword:(BET    95
   keyword:(Beat   98
   keyword:(Beginners)    429
   keyword:(Beginning     110
   ```

10. Generate a bid dataset by running the following command from HADOOP_HOME. You can find the results in a `biddata.data` file.

    `$ java -cp build/lib/hadoop-cookbook-chapter8.jar chapter8.`
    `adwords.AdwordsBidGenerator clickrate.data`

11. Create a directory called `/data/input2` and upload the bid dataset and results from the earlier MapReduce task to the `/data/input2` directory of HDFS.

    `$ bin/hadoopdfs -put clickrate.data /data/input2`

    `$ bin/hadoopdfs -put biddata.data /data/input2`

12. Generate the data to be used by Adwords dataset by running the second MapReduce job.

    `$ bin/hadoopjar hadoop-cookbook-chapter8.jar chapter8.adwords.`
    `AdwordsBalanceAlgorithmDataGenerator/data/input2 /data/output7`

13. Download the results to your computer by running the following command:

    `$ bin/hadoopdfs -get /data/output7/part-r-00000adwords.data`

14. You will see that it will print the results as follows:

    `(Animated        client23,773.0,5.0,97.0|`

    `(Animated)       client33,310.0,8.0,90.0|`

    `(Annals          client76,443.0,13.0,74.0|`

    `client51,1951.0,4.0,74.0|`

    `(Beginners)      client86,210.0,6.0,429.0|`

    `client6,236.0,5.0,429.0|`

    `(Beginning       client31,23.0,10.0,110.0|`

15. Perform the matches for a random set of keywords by running the following command:

```
$ javajar hadoop-cookbook-chapter8.jar chapter8.adwords.
AdwordsAssigneradwords.data
```

How it works...

As we discussed in the *How to do it...* section, the recipe consists of two MapReduce tasks. You can find the source code for the first MapReduce task from `src/chapter8/adwords/ClickRateApproximator.java`.

The mapper function looks like the following. It parses the Amazon dataset using the Amazon data format, and for each word in each product title, it emits the word and the sales ranks of that product.

```
public void map(Object key, Text value, Context context)
{
  ItemDataitemData = null;
  List<AmazonCustomer>customerList =
    AmazonCustomer.parseAItemLine(value.toString());
  if(customerList.size() == 0)
  {
    return;
  }
  for (AmazonCustomer customer : customerList)
  {
    itemData = customer.itemsBrought.iterator().next();
    break;
  }


  String[] tokens = itemData.title.split("\\s");
  for(String token: tokens)
  {
    if(token.length() > 3)
    {
      context.write(new Text(token),
        new IntWritable(itemData.salesrank));
    }
  }
}
```

Then, Hadoop sorts the emitted key-value pairs by keys and invokes the reducer once for each key passing the values emitted against that key. As shown in the following code, the reducer calculates an approximation for the click rate using sales ranks emitted against the key.

```
public void reduce(Text key, Iterable<IntWritable> values,
  Context context) throws IOException, InterruptedException
{
  doubleclickrate = 0;
  for(IntWritableval: values)
  {
    if(val.get() > 1)
    {
      clickrate = clickrate + 1000/Math.log(val.get());
    }
    else
    {
      clickrate = clickrate + 1000;
    }
  }
  context.write(new Text("keyword:" +key.toString()),
    newIntWritable((int)clickrate));
}
```

There is no publicly available bid dataset. Therefore, we will generate a random bid data set for our recipe using AdwordsBidGenerator. It would read the keywords generated by the earlier recipe and generate a random bid dataset.

Then we will use the second MapReduce task to merge the bid data set with click-through rate and generate a dataset that has bids' information sorted against the keyword. You can find the source for the second MapReduce task from `src/chapter8/adwords/AdwordsBalanceAlgorithmDataGenerator.java`. The mapper function looks like the following:

```
public void map(Object key, Text value, Context context)
  throws IOException, InterruptedException
{
  String[] keyVal = value.toString().split("\\s");
  if (keyVal[0].startsWith("keyword:"))
  {
    context.write(
      new Text(keyVal[0].replace("keyword:", "")),
      new Text(keyVal[1]));
  }
  else if (keyVal[0].startsWith("client"))
  {
    List<String[]> bids = new ArrayList<String[]>();
```

```
        double budget = 0;
        String clientid = keyVal[0];
        String[] tokens = keyVal[1].split(",");
        for (String token : tokens)
        {
          String[] kp = token.split("=");
          if (kp[0].equals("budget"))
          {
            budget = Double.parseDouble(kp[1]);
          }
          else if (kp[0].equals("bid"))
          {
            String[] bidData = kp[1].split("\\|");
            bids.add(bidData);
          }
        }

        for (String[] bid : bids)
        {
          String keyword = bid[0];
          String bidValue = bid[1];
          context.write(new Text(keyword),
            new Text(new StringBuffer()
            .append(clientid).append(",")
          .append(budget).append(",")
          .append(bidValue).toString()));
        }
      }
    }
```

The mapper function reads both the bid data set and click-through rate datasets and emits both types of data against the keyword. Then, each reducer receives all bids and associated click-through data for each keyword. Then the reducer merges the data and emits a list of bids against each keyword.

```
public void reduce(Text key, Iterable<Text> values,
  Context context) throws IOException, InterruptedException
{
  String clientid = null;
  String budget = null;
```

```
    String bid = null;
    String clickRate = null;

    List<String>bids = new ArrayList<String>();
    for (Text val : values)
    {
      if (val.toString().indexOf(",") > 0)
      {
        bids.add(val.toString());
      }
      else
        {
          clickRate = val.toString();
        }
    }
    StringBufferbuf = new StringBuffer();
    for (String bidData : bids)
    {
      String[] vals = bidData.split(",");
      clientid = vals[0];
      budget = vals[1];
      bid = vals[2];
      buf.append(clientid).append(",")
        .append(budget).append(",")
        .append(Double.valueOf(bid)).append(",")
        .append(Math.max(1, Double.valueOf(clickRate)));
      buf.append("|");
    }
    if (bids.size() > 0)
    {
      context.write(key, new Text(buf.toString()));
    }
  }
```

Finally, the Adwords assigner loads the bids data and stores them against the keywords in the memory. Given a keyword, the Adwords assigner finds the bid that has maximum value for the following equation and selects a bid among all the bids for advertisement:

*Measure = bid value * click-through rate * (1-e^(-1*current budget/ initial budget))*

There's more...

The preceding recipe assumes that Adwords assigner can load all the data in the memory to make advertisements assignment decisions. However, if the dataset is big, we can partition the dataset among multiple computers by keywords (for example, assign keywords that start with "A-D" to the first computer and so on).

This recipe assumes that users only bid for single words. However, to support multiple keyword bids, we would need to combine the click-through rates, and the rest of the algorithm can proceed as before.

More information about online advertisement can be found from the book, *Mining of Massive Datasets*, by *Anand Rajaraman* and *Jeffrey D. Ullman*. This book can be found at `http://infolab.stanford.edu/~ullman/mmds.html`.

9
Mass Text Data Processing

In this chapter, we will cover:

- ▸ Data preprocessing (extract, clean, and format conversion) using Hadoop Streaming and Python
- ▸ Data de-duplication using Hadoop Streaming
- ▸ Loading large datasets to an Apache HBase data store using `importtsv` and `bulkload` tools
- ▸ Creating TF and TF-IDF vectors for the text data
- ▸ Clustering the text data
- ▸ Topic discovery using **Latent Dirichlet Allocation** (**LDA**)
- ▸ Document classification using Mahout Naive Bayes classifier

Introduction

Hadoop MapReduce together with the supportive set of projects makes for a good framework choice to process large text datasets and to perform ETL-type operations.

In this chapter, we'll be exploring how to use Hadoop Streaming to perform data preprocessing operations such as data extraction, format conversion, and de-duplication. We'll also use HBase as the data store to load the data and will explore mechanisms to perform large data loads to HBase with minimal overhead. Towards the end of the chapter, we'll look in at performing text analytics operations using the Apache Mahout algorithms.

Data preprocessing (extract, clean, and format conversion) using Hadoop Streaming and Python

Data preprocessing is an important and often required component in data analytics. Data preprocessing becomes even more important when consuming unstructured text data generated from multiple sources. Data preprocessing steps include operations such as cleaning the data, extracting important features from data, removing duplicate items from the datasets, converting data formats, and many more.

Hadoop MapReduce provides an ideal environment to perform these tasks in parallel with massive datasets. Apart from the ability to implement Java MapReduce programs, Pig, and Hive scripts to preprocess these data, Hadoop also provides several useful tools and features that we can utilize to perform these data preprocessing operations. One such feature is the support of different `InputFormat` classes, providing us with the ability to support proprietary data formats by implementing custom `InputFormat` classes. Another feature is the Hadoop Streaming feature, which allows us to use our favorite scripting languages to perform the actual data cleansing and extraction, while Hadoop parallelizes the computation to hundreds of compute and storage resources.

In this recipe, we are going to use Hadoop Streaming with a Python script-based mapper to perform data extraction and format conversion.

Getting ready

Install and deploy Hadoop MapReduce and HDFS. Export the `HADOOP_HOME` environment variable to point to your Hadoop installation root folder.

Install Python on your Hadoop compute nodes, if Python is not already installed.

How to do it...

The following steps show you how to clean and extract data from the `20news` dataset and store the data as a **tab-separated value** (**TSV**) file:

1. Download and extract the `20news` dataset from `http://qwone.com/~jason/20N ewsgroups/20news-19997.tar.gz`.

2. Upload the extracted data to the HDFS. In order to preserve the compute time and resources, you can use only a subset of the dataset; use the following command to upload the extracted data to the HDFS:

   ```
   >bin/hadoop fs -mkdir 20news-all
   >bin/Hadoop fs –put  <extracted_folder> 20news-all
   ```

3. Extract the resource package for this chapter and copy the `MailPreProcessor.py` Python script to `$HADOOP_HOME`.

4. Run the following Hadoop Streaming command:

```
>bin/hadoop jar \
  ../contrib/streaming/hadoop-streaming-VERSION.jar \
  -input 20news-all\
  -output 20news-cleaned\
  -mapper MailPreProcessor.py \
  -file MailPreProcessor.py
```

5. Inspect the results by using the following command:

```
>bin/hadoopd fs -cat 20news-cleaned/part-00000
```

How it works...

Hadoop uses the default `TextInputFormat` class as the input specification for the preceding computation. Usage of the `TextInputFormat` class generates a map task for each file in the input dataset and generates a map input record for each line. Hadoop Streaming provides the input to the map application through the standard input.

```
line =  sys.stdin.readline();
while line:
….
  if (doneHeaders):
    list.append( line )
  elif line.find( "Message-ID:" ) != -1:
    messageID = line[ len("Message-ID:"):]
  ….
  elif line == "":
    doneHeaders = True


line = sys.stdin.readline();
```

The previous Python code reads the input lines from the standard input until the end of file is reached. We parse the headers of the news group file till we encounter the empty line demarcating the headers from the message contents. The message content will be read into a list line by line.

```
value = ' '.join( list )
value = fromAddress + "\t" ……"\t" + value
print '%s\t%s' % (messageID, value)
```

The preceding code segment merges the message content to a single string and constructs the output value of the Streaming application as a tab-delimited set of selected headers followed by the message content. The output key value is the `Message-ID` header extracted from the input file. The output is written to the standard output by using a tab to delimit the key and the value.

There's more...

We can generate the output of the preceding computation in the Hadoop `SequenceFile` format by specifying `SequenceFileOutputFormat` as the `OutputFormat` class of the Streaming computations.

```
>bin/hadoop jar \
  ../contrib/streaming/hadoop-streaming-1.0.4.jar\
  -input 20news-all \
  -output 20news-seq \
  -mapper MailPreProcessor.py \
  -outputformat \
  org.apache.hadoop.mapred.SequenceFileOutputFormat \
  -file MailPreProcessor.py
```

It is a good practice to store the data as `SequenceFiles` after the first pass of the input data, as `SequenceFiles` take less space and support compression. You can use `bin/hadoopdfs -text <path to sequencefile>` to dump the contents of a `SequenceFile` format to text.

```
>bin/hadoop dfs –text 20news-seq/part-00000
```

However, for the preceding command to work, any writable classes that are used in the `SequenceFile` format should be available in the Hadoop classpath.

See also

▶ *Using Hadoop with legacy applications – Hadoop Streaming* in *Chapter 4, Developing Complex Hadoop MapReduce Applications*.

▶ *Adding support for new input data formats – implementing a custom InputFormat* in *Chapter 4, Developing Complex Hadoop MapReduce Applications*.

▶ More information on Hadoop Streaming can be found at `http://hadoop.apache.org/docs/r1.0.4/streaming.html`.

Data de-duplication using Hadoop Streaming

Often, the datasets contain duplicate items that need to be eliminated to ensure the accuracy of the results. In this recipe, we use Hadoop to remove the duplicate mail records in the `20news` dataset. These duplicate records are due to the user's cross-posting the same message to multiple news boards.

Getting ready

Install and deploy Hadoop MapReduce and HDFS. Export the `HADOOP_HOME` environment variable to point to your Hadoop installation root folder.

Install Python on your Hadoop compute nodes, if Python is not already installed.

How to do it...

The following steps show how to remove duplicate mails, due to cross-posting across the lists, from the `20news` dataset:

1. Download and extract the `20news` dataset from `http://qwone.com/~jason/20N ewsgroups/20news-19997.tar.gz`.

2. Upload the extracted data to the HDFS. In order to preserve the compute time and resources, you may use only a subset of the dataset:

   ```
   >bin/hadoop fs -mkdir 20news-all
   >bin/hadoop fs –put  <extracted_folder> 20news-all
   ```

3. We are going to use the `MailPreProcessor.py` Python script from the previous recipe, *Data extract, cleaning and format conversion using Hadoop Streaming* as the mapper. Extract the resource package for this chapter and copy the `MailPreProcessor.py` and the `MailPreProcessorReduce.py` Python scripts to the `$HADOOP_HOME` folder.

4. Execute the following command:

   ```
   >bin/hadoop jar \
     ../contrib/streaming/hadoop-streaming-1.0.4.jar \
     -input 20news-all\
     -output 20news-dedup\
     -mapper MailPreProcessor.py \
     -reducer MailPreProcessorReduce.py \
     -file MailPreProcessor.py\
     -file MailPreProcessorReduce.py
   ```

5. Inspect the results using the following command:

```
>bin/hadoop dfs –cat 20news-dedup/part-00000
```

How it works...

Mapper Python script outputs the message ID as the key. We use the message ID to identify the duplicated messages that are a result of cross-posting across different newsgroups.

Hadoop Streaming provides the `Reducer` input records of the each key group line by line to the Streaming reducer application through the standard input. However, Hadoop Streaming does not have a mechanism to distinguish when it starts to feed records of a new key to the process. The Streaming reducer applications need to keep track of the input key to identify new groups. Since we output the mapper results using the `MessageID` header as the key, the Reducer input gets grouped by `MessageID`. Any group with more than one value (that is, message) per MessageID contains duplicates.

```python
#!/usr/bin/env python
import sys;

currentKey = ""

for line in sys.stdin:
  line = line.strip()
  key, value = line.split('\t',1)
  if currentKey == key :
    continue
  print '%s\t%s' % (key, value)
```

In the previous script, we use only the first value (message) of the record group and discard the others, which are the duplicate

See also

- ▶ *Using Hadoop with legacy applications – Hadoop Streaming* from *Chapter 4, Developing Complex Hadoop MapReduce Applications*.

- ▶ *Data extract, cleaning, and format conversion using Hadoop Streaming* .

- ▶ More information on Hadoop Streaming can be found at `http://hadoop.apache.org/docs/r1.0.4/streaming.html`.

Loading large datasets to an Apache HBase data store using importtsv and bulkload tools

Apache HBase data store is very useful when storing large-scale data in a semi-structured manner, so that they can be used for further processing using Hadoop MapReduce programs or to provide a random access data storage for client applications. In this recipe, we are going to import a large text dataset to HBase using the `importtsv` and `bulkload` tools.

Getting ready

Install and deploy Hadoop MapReduce and HDFS. Export the `HADOOP_HOME` environment variable to point to your Hadoop installation root folder.

Install and deploy Apache HBase in the distributed mode. Refer to the *Deploying HBase on a Hadoop cluster* recipe in this chapter for more information. Export the `HBASE_HOME` environment variable to point to your HBase installation root folder.

Install Python on your Hadoop compute nodes, if Python is not already installed.

How to do it...

The following steps show you how to load the TSV-converted `20news` dataset into an HBase table:

1. Follow the *Data extract, cleaning, and format conversion using Hadoop Streaming and Python* recipe to perform the preprocessing of data for this recipe. We assume that the output of the following fourth step of that recipe is stored in a HDFS folder named `20news-cleaned`:

   ```
   >bin/hadoop jar \
       ../contrib/streaming/hadoop-streaming-VERSION.jar \
       -input 20news-all \
       -output 20news-cleaned \
       -mapper MailPreProcessor.py \
       -file MailPreProcessor.py
   ```

2. Go to `HBASE_HOME` and start the HBase Shell:

   ```
   >cd $HBASE_HOME
   >bin/hbase shell
   ```

3. Create a table named `20news-data` by executing the following command in the HBase Shell. Older versions of the `importtsv` (used in the next step) command can handle only a single column family. Hence, we are using only a single column family when creating the HBase table:

```
hbase(main):001:0> create '20news-data,'h'
```

4. Go to `HADOOP_HOME` and execute the following command to import the preprocessed data to the previously created HBase table:

```
> bin/hadoop jar \
  $HBASE_HOME/hbase-<VERSION>.jar importtsv \
  -Dimporttsv.columns=HBASE_ROW_KEY,h:from,h:group,h:subj,h:msg \
  20news-data 20news-cleaned
```

5. Go to the `HBASE_HOME`. Start the HBase Shell. Use the `count` and `scan` commands of the HBase Shell to verify contents of the table:

```
hbase(main):010:0> count '20news-data'
12xxx row(s) in 0.0250 seconds


hbase(main):010:0> scan '20news-data', {LIMIT => 10}
ROW                                   COLUMN+CELL
<1993Apr29.103624.1383@cronkite.ocis.te column=h:c1,
timestamp=1354028803355, value= katop@astro.ocis.temple.edu
(Chris Katopis)>
<1993Apr29.103624.1383@cronkite.ocis.te column=h:c2,
timestamp=1354028803355, value= sci.electronics
......
```

The following are the steps to load the `20news` dataset to an HBase table using the `bulkload` feature:

1. Follow steps 1 to 3, but create the table with a different name:

```
hbase(main):001:0> create '20news-bulk','h'
```

2. Go to `HADOOP_HOME`. Use the following command to generate an HBase `bulkload` datafile:

    ```
    >bin/hadoop jar \

      $HBASE_HOME/hbase-<VERSION>.jar importtsv\

      -Dimporttsv.columns=HBASE_ROW_KEY,h:from,h:group,h:subj,h:msg \

      -Dimporttsv.bulk.output=hbaseloaddir \

      20news-bulk-source20news-cleaned
    ```

3. List the files to verify that the `bulkload` datafiles are generated:

    ```
    >bin/hadoop fs -ls 20news-bulk-source

    ......

    drwxr-xr-x   - thilinasupergroup          0 2012-11-27 10:06 /
    user/thilina/20news-bulk-source/h


    >bin/hadoopfs -ls20news-bulk-source/h

    -rw-r--r--   1 thilinasupergroup      19110 2012-11-27 10:06 /
    user/thilina/20news-bulk-source/h/4796511868534757870
    ```

4. The following command loads the data to the HBase table by moving the output files to the correct location:

    ```
    >bin/hadoop jar $HBASE_HOME/hbase-<VERSION>.jar \

    completebulkload 20news-bulk-source 20news-bulk
    ......

    12/11/27 10:10:00 INFO mapreduce.LoadIncrementalHFiles: Trying
    to load hfile=hdfs://127.0.0.1:9000/user/thilina/20news-bulk-
    source/h/4796511868534757870 first= <1993Apr29.103624.1383@
    cronkite.ocis.temple.edu>last= <stephens.736002130@ngis>

    ......
    ```

5. Go to `HBASE_HOME`. Start the HBase Shell. Use the `count` and `scan` commands of the HBase Shell to verify the contents of the table:

    ```
    hbase(main):010:0> count 'datatsvbulk'

    hbase(main):010:0> scan 'datatsvbulk', {LIMIT => 10}
    ```

How it works...

The `MailPreProcessor.py` Python script extracts a selected set of data fields from the news board message and outputs them as a tab-separated dataset.

```
value = fromAddress + "\t" + newsgroup
+"\t" + subject +"\t" + value
print '%s\t%s' % (messageID, value)
```

We import the tab-separated dataset generated by the Streaming MapReduce computations to Hbase using the `importtsv` tool. The `importtsv` tool requires the data to have no other tab characters except for the tab characters separating the data fields. Hence, we remove any tab characters in the input data using the following snippet of the Python script:

```
line = line.strip()
line = re.sub('\t',' ',line)
```

The `importtsv` tool supports loading data to HBase directly using the `Put` operations as well as by generating the HBase internal HFiles. The following command loads the data to HBase directly using the `Put` operations. Our generated dataset contains a key and four fields in the values. We specify the data fields to table column name mapping for the dataset using the `-Dimporttsv.columns` parameter. This mapping consists of listing the respective table column names in the order of the tab-separated data fields in the input dataset:

```
>bin/hadoop jar \

  $HBASE_HOME/hbase-<VERSION>.jar importtsv \
  -Dimporttsv.columns=<data field to table column mappings> \
  <HBasetablename> <hdfs input directory>
```

We can use the following command to generate HBase HFiles for the dataset. These HFiles can be directly loaded to HBase, without going through the HBase APIs, thereby reducing the amount of CPU and network resources needed.

```
>bin/hadoop jar

  $HBASE_HOME/hbase-<VERSION>.jar importtsv \

  -Dimporttsv.columns=<filed to column mappings> \

  -Dimporttsv.bulk.output=<path for hfile output> \

  <HBasetablename> <hdfs input directory>
```

These generated HFiles can be loaded into HBase tables by simply moving the files to the right location. This is done by using the `completebulkload` command:

```
>bin/hadoop jar $HBASE_HOME/hbase-<VERSION>.jar \
completebulkload <path for hfiles> <table name>
```

There's more...

You can use the `importtsv` tool with datasets with other datafile separator characters as well by specifying the `-Dimporttsv.separator` parameter. The following is an example of using a comma as the separator character to import a comma-separated dataset into a HBase table.

```
>bin/hadoop jar \
  $HBASE_HOME/hbase-<VERSION>.jar importtsv \
  '-Dimporttsv.separator=,' \
  -Dimporttsv.columns=<data field to table column mappings> \
  <HBasetablename><hdfs input directory>
```

Look out for **Bad Lines** in the MapReduce job console output or in the Hadoop monitoring console. One reason is having unwanted delimiter characters. In the preceding Python script, we remove any extra tabs in the message; here is the message displayed in the job console:

12/11/27 00:38:10 INFO mapred.JobClient: ImportTsv

12/11/27 00:38:10 INFO mapred.JobClient: Bad Lines=2

Data de-duplication using HBase

HBase supports storing multiple versions of column values for each record. When querying, HBase returns the latest version of values, unless we specify a specific time period. This feature of HBase can be used to perform automatic de-duplication by making sure we use the same `RowKey` value for duplicate values. In our `20news` example, we use `MessageID` as the `RowKey` value for the records, thus ensuring that duplicate messages will appear as different versions of the same data record.

HBase allows us to configure the maximum or the minimum number of versions per column family. Setting maximum number of versions to a low value will reduce the data usage by discarding the older versions. Refer to `http://hbase.apache.org/book/schema.versions.html` for more information on setting the maximum or minimum number of versions.

See also

▶ *Installing HBase* in *Chapter 5, Hadoop Ecosystem*.

▶ *Running MapReduce jobs on HBASE(table input/output)* in *Chapter 5, Hadoop Ecosystem*.

▶ *Deploying HBase on a Hadoop cluster*.

Creating TF and TF-IDF vectors for the text data

Most of the text analysis data mining algorithms operate on vector data. We can use a vector space model to represent text data as a set of vectors. For an example, we can build a vector space model by taking the set of all terms that appear in the dataset and by assigning an index to each term in the term set. Number of terms in the term set is the dimensionality of the resulting vectors and each dimension of the vector corresponds to a term. For each document, the vector contains the number of occurrences of each term at the index location assigned to that particular term. This creates vector space model using term frequencies in each document, similar to the result of the computation we perform in the *Generating an inverted index using Hadoop MapReduce* recipe of *Chapter 7, Searching and Indexing*.

				basketball	Indiana	state						
Doc 1	0	0	1	3	5	0	2	0	85	0	{0,0,1,3,5,0,2,0,85,0,......}
Doc 2	0	2	5	0	1	1	0	10	70	0	{0,2,5,0,1,1,0,10,70,0,....}
Doc 3	0	0	0	1	0	5	0	2	56	1	{0,0,0,1,0,5,0,2,56,1,......}
....											{....................................}

The term frequencies and the resulting document vectors

However, creating vectors using the preceding term count model gives a lot of weight to the terms that occur frequently across many documents (for example, the, is, a, are, was, who, and so on), although these frequent terms have only a very minimal contribution when it comes to defining the meaning of a document. The **Term frequency-inverse document frequency** (**TF-IDF**) model solves this issue by utilizing the **inverted document frequencies** (**IDF**) to scale the **term frequencies** (**TF**). IDF is typically calculated by first counting the number of documents (DF) the term appears in, inversing it (1/DF), and normalizing it by multiplying with the number of documents and using the logarithm of the resultant value as shown roughly by the following equation:

$$TF\text{-}IDF_i = TF_i \times \log (N/DF_i)$$

In this recipe, we'll create TF-IDF vectors from a text dataset using a built-in utility tool of Apache Mahout.

Getting ready

Install and deploy Hadoop MapReduce and HDFS. Export the `HADOOP_HOME` environment variable to point to your Hadoop installation root folder.

Download and install Apache Mahout. Export the `MAHOUT_HOME` environment variable to point to your Mahout installation root folder. Refer to the *Installing Mahout* recipe of *Chapter 5*, *Hadoop Ecosystem*, for more information on installing Mahout.

How to do it...

The following steps show you how to build a vector model of the `20news` dataset:

1. Download and extract the `20news` dataset from `http://qwone.com/~jason/20N ewsgroups/20news-19997.tar.gz`.

2. Upload the extracted data to the HDFS:

   >bin/hadoop fs -mkdir 20news-all

   >bin/Hadoop fs –put <extracted_folder> 20news-all

3. Go to `MAHOUT_HOME`. Generate Hadoop sequence files from the uploaded text data:

   >bin/mahout seqdirectory -i 20news-all -o 20news-seq

4. Generate TF and TF-IDF sparse vector models from the text data in the sequence files:

   >bin/mahout seq2sparse –i 20news-seq -o 20news-vector

This launches a series of MapReduce computations, as shown in the following screenshot; wait for the completion of these computations:

5. Check the output folder by using the following command. The `tfidf-vectors` folder contains the TF-IDF model vectors, the `tf-vectors` folder contains the term count model vectors and the `dictionary.file-0` folder contains the term to term-index mapping.

```
>/bin/hadoop dfs -ls 20news-vector

Found 7 items

drwxr-xr-x   - usupergroup          0 2012-11-27 16:53 /user/
u/20news-vector /df-count

-rw-r--r--   1 usupergroup       7627 2012-11-27 16:51 /user/
u/20news-vector/dictionary.file-0

-rw-r--r--   1 usupergroup       8053 2012-11-27 16:53 /user/
u/20news-vector/frequency.file-0

drwxr-xr-x   - usupergroup          0 2012-11-27 16:52 /user/
u/20news-vector/tf-vectors

drwxr-xr-x   - usupergroup          0 2012-11-27 16:54 /user/
u/20news-vector/tfidf-vectors

drwxr-xr-x   - usupergroup          0 2012-11-27 16:50 /user/
u/20news-vector/tokenized-documents

drwxr-xr-x   - usupergroup          0 2012-11-27 16:51 /user/
u/20news-vector/wordcount
```

6. Optionally, you can also use the following command to dump the TF-IDF vectors as text. The key is the filename and the contents of the vectors are in the format `<term index>:<TF-IDF value>`:

```
>bin/mahout seqdumper -i 20news-vector/tfidf-vectors/part-r-00000

……

Key class: class org.apache.hadoop.io.Text Value Class: class org.
apache.mahout.math.VectorWritable

Key: /54492: Value: {225:3.374729871749878,400:1.5389964580535889,
321:1.0,324:2.386294364929199,326:2.386294364929199,315:1.0,144:2.
0986123085021973,11:1.0870113372802734,187:2.652313232421875,134:2
.386294364929199,132:2.0986123085021973,......}

……
```

How it works...

Hadoop sequence files store the data as binary key-value pairs and supports data compression. Mahout's `seqdirectory` command converts the text files into Hadoop `SequenceFile` by using the filename of the text file as the key and the contents of the text file as the value. The `seqdirectory` command stores all the text contents into a single `SequenceFile`. However, it's possible for us to specify a chuck size to control the actual storage of the `SequenceFile` data blocks in the HDFS. Following are a selected set of options for the `seqdirectory` command:

```
> bin/mahout seqdirectory –i <HDFS path to text files>

   -o <HDFS output directory for sequence file>

   -ow                     If present, overwrite the output directory

   -chunk<chunk size>      In MegaBytes.Defaults to 64mb

   -prefix<keyprefix>      The prefix to be prepended to the key
```

The `seq2sparse` command is an Apache Mahout tool that supports the generation of sparse vectors from `SequenceFiles` containing text data. It supports the generation of both TF as well as TF-IDF vector models. This command executes as a series of MapReduce computations. Following are a selected set of options for the `seq2sparse` command:

```
bin/mahout seq2sparse -i <HDFS path to the text sequence file>

  -o <HDFS output directory>

  -wt{tf|tfidf}

  -chunk <max dictionary chunksize inmb to keep in memory>

  --minSupport<minimum support>

  --minDF<minimum document frequency>

  --maxDFPercent<MAX PERCENTAGE OF DOCS FOR DF
```

`minSupport` is the minimum frequency for the word to be considered as a feature. `minDF` is the minimum number of documents the word needs to be in. `maxDFPercent` is the maximum value of the expression (document frequency of a word/total number of documents) in order for that word to be considered as a good feature in the document. This helps remove high frequency features such as stop words.

You can use the Mahout `seqdumper` command to dump the contents of a `SequenceFile` format that uses the Mahout `Writable` data types as plain text:

```
bin/mahout seqdumper -i <HDFS path to the sequence file>
   -o <output directory>
   --count              Output only the number of key value pairs.
   --numItems           Max number of key value pairs to output
   --facets             Output the value counts per key.
```

See also

> ▸ *Generating an inverted index using Hadoop MapReduce* in *Chapter 7*, *Searching and Indexing*.

> ▸ Mahout documentation on creating vectors from text data at `https://cwiki.apache.org/confluence/display/MAHOUT/ Creating+Vectors+from+Text`.

Clustering the text data

Clustering plays an integral role in data mining computations. Clustering groups together similar items of a dataset by using one or more features of the data items based on the use-case. Document clustering is used in many text mining operations such as document organization, topic identification, information presentation, and so on. Document clustering shares many of the mechanisms and algorithms with traditional data clustering mechanisms. However, document clustering has its unique challenges when it comes to determining the features to use for clustering and when building vector space models to represent the text documents.

The *Running K-Means with Mahout* recipe of *Chapter 5*, *Hadoop Ecosystem*, focuses on using Mahout K-Means clustering from Java code to cluster a statistics data. The *Hierarchical clustering* and *Clustering an Amazon sales dataset* recipes of *Chapter 8*, *Classifications, Recommendations, and Finding Relationships*, focuses on using clustering to identify customers with similar interests. These three recipes provide a more in-depth understanding of using clustering algorithms in general. This recipe focuses on exploring two of the several clustering algorithms available in Apache Mahout for document clustering.

Getting ready

Install and deploy Hadoop MapReduce and HDFS. Export the HADOOP_HOME environment variable to point to your Hadoop installation root folder.

Download and install Apache Mahout. Export the MAHOUT_HOME environment variable to point to your Mahout installation root folder. Refer to the *Installing Mahout* recipe of *Chapter 5*, *Hadoop Ecosystem*, for more information on installing Mahout.

How to do it...

The following steps use the Apache Mahout K-Means clustering algorithm to cluster the 20news dataset:

1. Follow the *Creating TF and TF-IDF vectors for the text data* recipe in this chapter and generate TF-IDF vectors for the 20news dataset. We assume that the TF-IDF vectors are in the 20news-vector/tfidf-vectors folder of the HDFS.

2. Go to the MAHOUT_HOME.

3. Execute the following command to execute the K-Means clustering computation:

```
>bin/mahout kmeans \

  --input 20news-vector/tfidf-vectors \

  --clusters 20news-seed/clusters \

  --output 20news-km-clusters \

  --distanceMeasure \

  org.apache.mahout.common.distance.
    SquaredEuclideanDistanceMeasure-k 10 --maxIter 20 --clustering
```

4. Execute the following command to convert the clusters to text:

```
>bin/mahout clusterdump \

  -i20news-km-clusters/clusters-*-final\

  -o 20news-clusters-dump \

  -d 20news-vector/dictionary.file-0 \

  -dt sequencefile \

  --pointsDir 20news-km-clusters/clusteredPoints


>cat 20news-clusters-dump
```

The following steps use the Apache Mahout MinHash clustering algorithm to cluster the `20news` dataset:

1. Execute the following command to run MinHash clustering on an already vectorised `20news` data:

   ```
   >bin/mahout minhash \
     --input 20news-vector/tfidf-vectors \
     --output minhashout
   ```

2. Go to `HADOOP_HOME` and execute the following command to inspect the MinHash clustering results:

   ```
   >bin/hadoop dfs -cat minhashout/part*
   ```

How it works...

The following is the usage of the Mahout K-Means algorithm:

```
>bin/mahout kmeans
  --input <tfidf vector input>
  --clusters <seed clusters>
  --output <HDFS path for output>
  --distanceMeasure<distance measure>
  -k <number of clusters>
  --maxIter<maximum number of iterations>
  --clustering
```

Mahout will generate random seed clusters when an empty HDFS folder path is given to the `--clusters` option. Mahout supports several different distance calculation methods such as Euclidean, Cosine, Manhattan, and so on.

Following is the usage of the Mahout `clusterdump` command:

```
>bin/mahout clusterdump
  -i <HDFS path to clusters>
  -o <local path for text output>
  -d <dictionary mapping for the vector data points>
  -dt <dictionary file type (sequencefile or text)>
  --pointsDir <directory containing the input vectors to
                clusters mapping>
```

Following is the usage of the Mahout MinHash clustering algorithm:

```
>bin/mahout minhash

    --input  <tfidf vector input>

    --output <HDFS path for output>
```

See also

▸ *Running K-Means with Mahout* in *Chapter 5*, *Hadoop Ecosystem*.

Topic discovery using Latent Dirichlet Allocation (LDA)

We can use **Latent Dirichlet Allocation** to cluster a given set of words into topics and a set of documents to combinations of topics. LDA is useful when identifying the meaning of a document or a word based on the context, not solely depending on the number of words or the exact words. LDA can be used to identify the intent and to resolve ambiguous words in systems such as a search engine. Some other example use-cases of LDA are identifying influential Twitter users for particular topics and Twahpic (`http://twahpic.cloudapp.net`) application uses LDA to identify topics used on Twitter.

LDA uses the TF vector space model instead of the TF-IDFmodel, as it needs to consider the co-occurrence and correlation of words.

Getting ready

Install and deploy Hadoop MapReduce and HDFS. Export the `HADOOP_HOME` environment variable to point to your Hadoop installation root folder.

Download and install Apache Mahout. Export the `MAHOUT_HOME` environment variable to point to your Mahout installation root folder. Refer to the *Installing Mahout* recipe of *Chapter 5*, *Hadoop Ecosystem*, for more information on installing Mahout.

How to do it...

The following steps show you how to run Mahout LDA algorithm on a subset of the `20news` dataset:

1. Download and extract the `20news` dataset from `http://qwone.com/~jason/20Newsgroups/20news-19997.tar.gz`.

2. Upload the extracted data to the HDFS:

```
>bin/hadoop fs -mkdir 20news-all
>bin/hadoop fs –put  <extracted_folder> 20news-all
```

3. Go to the MAHOUT_HOME. Generate sequence files from the uploaded text data:

```
>bin/mahout seqdirectory -i 20news-all -o 20news-seq
```

4. Generate sparse vector from the text data in the sequence files:

```
> bin/mahout seq2sparse \
   –i 20news-seq  -o 20news-tf \
   –wt tf \
   -a org.apache.lucene.analysis.WhitespaceAnalyzer
```

5. Convert the TF vectors from SequenceFile<Text, VectorWritable> to SequenceFile<IntWritable,Text>:

```
>bin/mahout rowid -i 20news-tf/tf-vectors -o 20news-tf-int
```

6. Run the following command to perform the LDA computation:

```
> bin/mahout cvb \
   -i 20news-tf-int/matrix -o lda-out \
   -k 10  -x 20 \
   -dict 20news-tf/dictionary.file-0 \
   –dt lda-topics \
   –mt lda-topic-model
```

7. Dump and inspect the results of the LDA computation:

```
>bin/mahout seqdumper -i lda-topics/part-m-00000
Input Path: lda-topics5/part-m-00000
Key class: class org.apache.hadoop.io.IntWritable Value Class:
class org.apache.mahout.math.VectorWritable
Key: 0: Value: {0:0.12492744375758073,1:0.03875953927132082,2:0.12
28639250669511,3:0.15074522974495433,4:0.10512715697420276,5:0.101
30565323653766,6:0.061169131590630275,7:0.14501579630233746,8:0.07
872957132697946,9:0.07135655272850545}
.....
```

8. Join the output vectors with the dictionary mapping of term to term indexes:

```
>bin/mahoutvectordump -i lda-topics/part-m-00000 --dictionary
20news-tf/dictionary.file-0 --vectorSize 10  -dt sequencefile

......


{"Fluxgate:0.12492744375758073,&:0.03875953927132082,(140.220.1.1
):0.1228639250669511,(Babak:0.15074522974495433,(Bill:0.105127156
97420276,(Gerrit:0.10130565323653766,(Michael:0.06116913159063027
5,(Scott:0.14501579630233746,(Usenet:0.07872957132697946,(continu
ed):0.07135655272850545}
{"Fluxgate:0.13130952097888746,&:0.05207587369196414,(140.220.1.1
):0.12533225607394424,(Babak:0.08607740024552457,(Bill:0.20218284
543514245,(Gerrit:0.07318295757631627,(Michael:0.0876688824220103
9,(Scott:0.08858421220476514,(Usenet:0.09201906604666685,(continu
ed):0.06156698532477829}

.......
```

How it works...

Mahout CVB version of LDA implements the Collapse Variable Bayesian inference algorithm using an iterative MapReduce approach:

```
>bin/mahout cvb -i 20news-tf-int/matrix -o lda-out -k 10  -x 20  -dict
20news-tf/dictionary.file-0 -dt lda-topics -mt lda-topic-model
```

The `-i` parameter provides the input path, while the `-o` parameter provides the path to store the output. `-k` specifies the number of topics to learn and `-x` specifies the maximum number of iterations for the computation. `-dict` points to the dictionary containing the mapping of terms to term-indexes. Path given in the `-dt` parameter stores the training topic distribution. Path given in `-mt` is used as a temporary location to store the intermediate models.

All the command-line options of the `cvb` command can be queried by invoking the `help` option as follows:

```
> bin/mahout  cvb  --help
```

Setting the number of topics to a very small value brings out very high-level topics. Large number of topics gives more descriptive topics, but takes longer to process. `maxDFPercentoption` can be used to remove common words, thereby speeding up the processing.

See also

▶ Y. W. Teh, D. Newman, and M. Welling's article, *A Collapsed Variational Bayesian Inference Algorithm for Latent Dirichlet Allocation*, in NIPS, volume 19, 2006 at `http://www.gatsby.ucl.ac.uk/~ywteh/research/inference/nips2006.pdf`.

Document classification using Mahout Naive Bayes classifier

Classification assigns documents or data items to an already known set of classes with already known properties. Document classification or categorization is used when we need to assign documents to one or more categories. This is a frequent use-case in information retrieval as well as library science.

The *Classification using Naive Bayes classifier* recipe in *Chapter 8*, *Classifications, Recommendations, and Finding Relationships*, provides a more detailed description about classification use-cases and gives you an overview of using the Naive Bayes classifier algorithm. The recipe focuses on highlighting the classification support in Apache Mahout for text documents.

Getting ready

Install and deploy Hadoop MapReduce and HDFS. Export the `HADOOP_HOME` environment variable to point to your Hadoop installation root folder.

Download and install Apache Mahout. Export the `MAHOUT_HOME` environment variable to point to your Mahout installation root folder. Refer to the *Installing Mahout* recipe of *Chapter 5*, *Hadoop Ecosystem*, for more information on installing Mahout.

How to do it...

The following steps use the Apache Mahout Naive Bayes algorithm to cluster the `20news` dataset:

1. Follow the *Creating TF and TF-IDF vectors for the text data* recipe in this chapter and generate TF-IDF vectors for the `20news` dataset. We assume that the TF-IDF vectors are in the `20news-vector/tfidf-vectors` folder of the HDFS.

2. Go to the `MAHOUT_HOME`.

3. Split the data to training and test datasets:

    ```
    >bin/mahout split \
        -i 20news-vectors/tfidf-vectors \
        --training Output/20news-train-vectors \
        --test Output/20news-test-vectors  \
        --randomSelectionPct 40 --overwrite --sequenceFiles
    ```

4. Train the Naive Bayes model:

    ```
    >bin/mahout trainnb \
    -i 20news-train-vectors -el \
        -o  model \
        -li labelindex
    ```

5. Test the classification on the test dataset:

    ```
    >bin/mahout testnb \
        -i 20news-train-vectors\
        -m model \
        -l labelindex \
    -o 20news-testing
    ```

How it works...

The Mahout `split` command can be used to split a dataset to a training dataset and a test dataset. The Mahout `split` command works with text datasets as well as with Hadoop `SequenceFile` datasets. Following is the usage of the Mahout `split` command. You can use the `--help` option with the `split` command to print out all the options:

```
>bin/mahout split \
  -i <input data directory> \
--trainingOutput<HDFS path to store the training dataset> \
  --testOutput<HDFS path to store the test dataset>  \
 --randomSelectionPct<percentage to be selected as test data> \
--sequenceFiles
```

The `sequenceFiles` option specifies that the input dataset is in Hadoop `SequenceFiles` format.

Following is the usage of the Mahout Naive Bayes classifier training command. The `-el` option informs Mahout to extract the labels from the input dataset:

```
>bin/mahout trainnb \
-i <HDFS path to the training dataset> \
-el \
  -o <HDFS path to store the trained classifier model> \
-li <Path to store the label index> \
```

Following is the usage of the Mahout Naive Bayes classifier testing command:

```
>bin/mahout testnb \
    -i <HDFS path to the test dataset>
    -m <HDFS path to the classifier model>\
    -l <Path to the label index> \
    -o <path to store the test result>
```

See also

▶ *Classification using Naive Bayes classifier* in *Chapter 8, Classifications, Recommendations, and Finding Relationships*.

▶ The book, *Mahout in Action*, at `http://www.amazon.com/Mahout-Action-Sean-Owen/dp/1935182684`

10

Cloud Deployments: Using Hadoop on Clouds

In this chapter, we will cover:

- ▶ Running Hadoop MapReduce computations using Amazon Elastic MapReduce (EMR)
- ▶ Saving money using Amazon EC2 Spot Instances to execute EMR job flows
- ▶ Executing a Pig script using EMR
- ▶ Executing a Hive script using EMR
- ▶ Creating an Amazon EMR job flow using the Command Line Interface
- ▶ Deploying an Apache HBase Cluster on Amazon EC2 cloud using EMR
- ▶ Using EMR Bootstrap actions to configure VMs for the Amazon EMR jobs
- ▶ Using Apache Whirr to deploy an Apache Hadoop cluster in a cloud environment
- ▶ Using Apache Whirr to deploy an Apache HBase cluster in a cloud environment

Introduction

Computing clouds provide on-demand, horizontal, scalable computing resources with no upfront capital investment, making them an ideal environment to perform occasional large -scale Hadoop computations. In this chapter, we will explore several mechanisms to deploy and execute Hadoop MapReduce and Hadoop-related computations on cloud environments.

This chapter discusses how to use **Amazon Elastic MapReduce** (**EMR**), the hosted Hadoop infrastructure, to execute traditional MapReduce computations as well as Pig and Hive computations on the Amazon EC2 cloud infrastructure. This chapter also presents how to provision an HBase cluster using Amazon EMR and how to back up and restore the data belonging to an EMR HBase cluster. We will also use Apache Whirr, a cloud neutral library for deploying services on cloud environments, to provision Apache Hadoop and Apache HBase clusters on cloud environments.

Running Hadoop MapReduce computations using Amazon Elastic MapReduce (EMR)

Amazon Elastic MapReduce (EMR) provides on-demand managed Hadoop clusters in the **Amazon Web Services (AWS)** cloud to perform your Hadoop MapReduce computations. EMR uses **Amazon Elastic Compute Cloud (EC2)** instances as the compute resources. EMR supports reading input data from Amazon **Simple Storage Service (S3)** and storing of the output data in Amazon S3 as well. EMR makes our life easier by taking care of the provisioning of cloud instances, configuring the Hadoop cluster and the execution of our MapReduce computational flows.

In this recipe, we are going to run the WordCount MapReduce sample (Refer to the *Writing the WordCount MapReduce sample, bundling it and running it using standalone Hadoop* recipe from *Chapter 1, Getting Hadoop up and running in a Cluster*) in the Amazon EC2 cloud using Amazon Elastic MapReduce.

Getting ready

Build the required `c10-samples.jar` by running the Ant build in the code samples for this chapter.

How to do it...

The steps for executing WordCount MapReduce application on Amazon Elastic MapReduce are as follows:

1. Sign up for an AWS account by visiting `http://aws.amazon.com`.

2. Open the Amazon S3 monitoring console at `https://console.aws.amazon.com/s3` and sign in.

3. Create a **S3 bucket** to upload the input data by clicking on **Create Bucket**. Provide a unique name for your bucket. Let's assume the name of the bucket as `wc-input-data`. You can find more information on creating a S3 bucket in `http://docs.amazonwebservices.com/AmazonS3/latest/gsg/CreatingABucket.html`. There exist several third-party desktop clients for the Amazon S3. You can use one of those clients to manage your data in S3 as well.

4. Upload your input data to the above-created bucket by selecting the bucket and clicking on **Upload**. The input data for the WordCount sample should be one or more text files.

5. Create a S3 bucket to upload the JAR file needed for our MapReduce computation. Let's assume the name of the bucket as `sample-jars`. Upload the `C10Samples.jar` file to the newly created bucket.

6. Create a S3 bucket to store the output data of the computation. Let's assume the name of this bucket as `ws-output-data`. Create another S3 bucket to store the logs of the computation. Let's assume the name of this bucket as `c10-logs`.

> S3 bucket namespace is shared globally by all users. Hence, using the example bucket names given in this recipe might not work for you. In such scenarios, you should give your own custom names for the buckets and substitute those names in the subsequent steps of this recipe.

7. Open the Amazon EMR console at `https://console.aws.amazon.com/` `elasticmapreduce`. Click on the **Create New Job Flow** button to create a new EMR MapReduce job flow. Provide a name for your job flow. Select **Run your own application** option under **Create a Job Flow**. Select the **Custom Jar** option from the drop-down menu below that. Click on **Continue**.

8. Specify the S3 location of the `c10-samples.jar` in the **Jar Location** textbox of the next tab (the **Specify Parameters** tab). You should specify the location of the JAR in the format `bucket_name/jar_name`. In the **JAR Arguments** textbox, enter `chapter1.WordCount` followed by the bucket location where you uploaded the input data and the output path. The output path should not exist and we use a directory (`wc-output-data/out1`) inside the output bucket you created in Step 6 as the output path. You should specify the locations using the format, `s3n://` `bucket_name/path`. Click on **Continue**.

9. Leave the default options and click on **Continue** in the next tab, **Configure EC2 Instances**. The default options use two EC2 `m1.small` instances for the Hadoop slave nodes and one EC2 `m1.small` instance for the Hadoop master node.

10. In the **Advanced Options** tab, enter the path of S3 bucket you created above for the logs in the **Amazon S3 Log Path** textbox. Select **Yes** for the **Enable Debugging**. Click on **Continue**.

11. Click on **Continue** in the **Bootstrap Options**. Review your job flow in the **Review** tab and click on **Create Job Flow** to launch instances and to run the MapReduce computation.

> Amazon will charge you for the compute and storage resources you use by clicking on **Create Job Flow** in step 11. Refer to the *Saving money using Amazon EC2 Spot Instances for EMR* recipe below to find out how you can save money by using Amazon EC2 Spot instances.

12. Click on **Refresh** in the EMR console to monitor the progress of your MapReduce job. Select your job flow entry and click on **Debug** to view the logs and to debug the computation. As EMR uploads the logfiles periodically, you might have to wait and refresh to access the logfiles. Check the output of the computation in the output data bucket using the AWS S3 console.

See also

▶ *Writing the WordCount MapReduce sample, bundling it and running it using standalone Hadoop* and *Running WordCount program in a distributed cluster environment* recipes from *Chapter 1, Getting Hadoop up and running in a Cluster*.

Saving money by using Amazon EC2 Spot Instances to execute EMR job flows

Amazon **EC2 Spot Instances** allow us to purchase underutilized EC2 compute resources at a significant discount. The prices of Spot Instances change depending on the demand. We can submit bids for the Spot Instances and we receive the requested compute instances, if our bid exceeds the current Spot Instance price. Amazon bills these instances based on the actual Spot Instance price, which can be lower than your bid. Amazon will terminate your instances, if the Spot Instance price exceeds your bid. However, Amazon do not charged for partial spot instance hours if Amazon terminated your instances. You can find more information on Amazon EC2 Spot Instances on `http://aws.amazon.com/ec2/spot-instances/`.

Amazon EMR supports using Spot Instances both as master as well as worker compute instances. Spot Instances are ideal to execute non-time critical computations such as batch jobs.

How to do it...

The following steps show you how to use Amazon EC2 Spot Instances with Amazon Elastic MapReduce to execute the WordCount MapReduce application.

1. Follow the steps 1 to 8 of the *Running Hadoop MapReduce computations using Amazon ElasticMapReduce (EMR)* recipe.

2. Configure your EMR job flow to use Spot Instances in the **Configure EC2 Instances** tab. (See Step 9 of the *Running Hadoop MapReduce computations using Amazon ElasticMapReduce (EMR)* recipe).

3. In the **Configure EC2 Instances** tab, select the **Request Spot Instances** checkboxes next to the **Instance Type** drop-down boxes under **Master and Core Instance Group** and **Core Instance Group**.

4. Specify your bid price in the Spot Bid Price textboxes. You can find the Spot Instance pricing history in the Spot Requests window of the Amazon EC2 console (`https://console.aws.amazon.com/ec2`).

5. Follow the steps 10 to 12 of the *Running Hadoop MapReduce computations using Amazon ElasticMapReduce (EMR)* recipe.

There's more...

You can also run the EMR computations on a combination of traditional EC2 on-demand instances and EC2 Spot instances, safe guarding your computation against possible Spot instance terminations.

As Amazon bills the Spot Instances using the current spot price irrespective of your bid price, it is a good practice not to set the Spot Instance price too low to avoid the risk of frequent terminations.

See also

▸ The *Running Hadoop MapReduce computations using Amazon Elastic MapReduce (EMR)* recipe from this chapter.

Executing a Pig script using EMR

Amazon EMR supports executing Pig scripts on the data stored in S3. For more details on Pig, refer to the *Installing Pig* and *Running your first Pig command* recipes in *Chapter 5, Hadoop Ecosystem*.

In this recipe, we are going to execute the Pig script sample from the *Running your first Pig commands* recipe using Amazon EMR. This sample will use the Human Development Report data (http://hdr.undp.org/en/statistics/data/) to print names of countries that have a GNI value greater than 2000 of gross national income per capita (GNI) sorted by GNI.

How to do it...

The following steps show you how to use a Pig script with Amazon Elastic MapReduce to process a dataset stored on Amazon S3.

1. Use the Amazon S3 console to create a bucket in S3 to upload the input data. Upload the `resources/hdi-data.csv` file in the source package associated with this chapter to the newly created bucket. You can also use an existing bucket or a directory inside a bucket as well. We assume the S3 path for the uploaded file as `c10-input-data/hdi-data.csv`.

2. Modify the Pig script from the *Running your first Pig commands* recipe of *Chapter 5, Hadoop Ecosystem*, to run it using EMR. Add a `STORE` command to save the result in the filesystem. Parameterize the `LOAD` command of the Pig script by adding `$INPUT` as the input file and the store command by adding `$OUTPUT` as the output directory. The modified Pig script is available in the `resources/countryFilter-EMR.pig` file of the resources associated with this chapter:

```
A = LOAD ''$INPUT'' using PigStorage('','')  AS
(id:int, country:chararray, hdi:float, lifeex:int,
mysch:int, eysch:int, gni:int);
B = FILTER A BY gni > 2000;
C = ORDER B BY gni;
STORE C into ''$OUTPUT'';
```

3. Use the Amazon S3 console to create a bucket in S3 to upload the Pig script. Upload the `resources/countryFilter-EMR.pig` script to the newly created bucket. You can also use an existing bucket or a directory inside a bucket as well. We assume the S3 path for the uploaded file as `c10-resources/countryFilter-EMR.pig`.

4. Open the Amazon EMR console at `https://console.aws.amazon.com/elasticmapreduce`. Click on the **Create New Job Flow** button to create a new EMR MapReduce job flow. Provide a name for your job flow. Select **Run your own application** option under **Create a Job Flow**. Select **Pig Program** option from the drop-down menu below that. Click on **Continue**.

5. Specify the S3 location of the Pig script in the **Script Location** textbox of the next tab (the **Specify Parameters** tab). You should specify the location of the script in the format `bucket_name/file_name`. Specify the S3 location of the uploaded input data file In the **Input Location** textbox. In the **Output Location** textbox, specify a S3 location to store the output. The output path should not exist and we use a directory (`c10-out/out4`) inside the output bucket as the output path. You should specify the locations using the format, `s3n://bucket_name/path`. Click on **Continue**.

6. Configure the EC2 instances for the job flow and configure the log paths for the MapReduce computations in the next two tabs. Click on **Continue** on the **Bootstrap Options** screen. Review your job flow in the **Review** tab and click on **Create Job Flow** to launch instances and to execute the Pig script. Refer to the steps 9, 10, and 11 of the *Running Hadoop MapReduce computations using Amazon ElasticMapReduce (EMR)* recipe for more details.

> Amazon will charge you for the compute and storage resources you use by clicking **Create Job Flow** in the step 11. Refer to the *Saving money by using EC2 Spot Instances* recipe to find out how you can save money by using Amazon EC2 Spot instances.

7. Click on **Refresh** in the EMR console to monitor the progress of your MapReduce job. Select your job flow entry and click on **Debug** to view the logs and to debug the computation. As EMR uploads the logfiles periodically; you might have to wait and refresh to access the logfiles. Check the output of the computation in the output data bucket using the AWS S3 console.

There's more...

Amazon EMR allows to us to use Apache Pig in the interactive mode as well.

Starting a Pig interactive session

Let's look at the steps to start a Pig interactive session:

1. Open the Amazon EMR console at `https://console.aws.amazon.com/elasticmapreduce`. Click on the **Create New Job Flow** button to create a new EMR MapReduce job flow. Provide a name for your job flow. Select the **Run your own application** option under **Create a Job Flow**. Select the **Pig Program** option from the drop-down menu below that. Click on **Continue**.

2. In order to start an interactive Pig session, select the **Start an Interactive Pig Session** option of the **Specify Parameters** tab. Click on **Continue**.

3. Configure the EC2 instances for the job flow in the **Configure EC2 Instances** tab. Click on **Continue**.

4. You must select a key pair from the **Amazon EC2 Key Pair** drop-down box in the **Advanced Options** tab. In case you do not have a usable Amazon EC2 key pair, log in to the Amazon EC2 console and create a new key pair.

5. Click on **Continue** on the **Bootstrap Options** screen. Review your job flow in the **Review** tab and click on **Create Job Flow** to launch instances.

6. After the cluster is provisioned, go to the Amazon EMR console (`https://console.aws.amazon.com/elasticmapreduce`). Select the current job flow to view more information about the job flow. Retrieve the **Master Public DNS Name** value from the information pane. (If you need more information about this step, please refer to step 6 of the *Deploying an Apache HBase Cluster on Amazon EC2 cloud using EMR* recipe).

7. Use the master public DNS name and the key file of the Amazon EC2 key pair you specified in step 4 to SSH in to the master node of the cluster:

   ```
   > ssh -i <path-to-the-key-file> hadoop@<master-public-DNS>
   ```

8. Start the Pig interactive grunt shell in the master node and issue your Pig commands.

See also

▸ The Running your first Pig commands recipe in *Chapter 5*, *Hadoop Ecosystem*.

Executing a Hive script using EMR

Amazon EMR supports executing Hive queries on the data stored in S3. For more details on Hive, refer to the *Installing Hive, Running SQL-style query with Hive* and *Performing a join with Hive* recipes in *Chapter 5*, *Hadoop Ecosystem*.

In this recipe, we are going to execute the Hive queries from the *Running SQL style Query with Hive* recipe using Amazon EMR. This sample will use the Human Development Report data (`http://hdr.undp.org/en/statistics/data/`) to print names of countries that have a GNI value greater than 2000$ of gross national income per capita (GNI) sorted by GNI.

How to do it...

The following steps show you how to use a Hive script with Amazon Elastic MapReduce to query a data set stored on Amazon S3.

1. Use the Amazon S3 console to create a bucket in S3 to upload the input data. Upload the `resources/hdi-data.csv` file in the source package associate with this chapter to the newly created bucket. You can also use an existing bucket or a directory inside a bucket as well. We assume the S3 path for the uploaded file as `c10-input-data/hdi-data.csv`.

2. Create a Hive batch script using the queries in the *Running SQL-style query with Hive* recipe of *Chapter 5*, *Hadoop Ecosystem*. Create a Hive table to store the result of the `Select` query. The Hive batch script is available in the `resources/countryFilter-EMR.hive` file of the resources associated with this chapter.

    ```
    CREATE TABLE HDI(
      id INT, country STRING, hdi FLOAT, lifeex INT, mysch INT, eysch
    INT, gni INT
      )
      ROW FORMAT DELIMITED
      FIELDS TERMINATED BY '',''
      STORED AS TEXTFILE
      LOCATION ''s3://c10-input-data/hdi-data.csv'';

    CREATE EXTERNAL TABLE output_countries(
        country STRING, gni INT
        )
        ROW FORMAT DELIMITED
        FIELDS TERMINATED BY '',''
        STORED AS TEXTFILE
        LOCATION ''${OUTPUT}/countries''
      ;

    INSERT OVERWRITE TABLE output_countries
      SELECT
        country, gni
      FROM
        HDI
      WHERE
        gni > 2000;
    ```

3. Use the Amazon S3 console to create a bucket in S3 to upload the Hive script. Upload the `resources/countryFilter-EMR.hive` script to the newly created bucket. You can also use an existing bucket or a directory inside a bucket as well. We assume the S3 path for the uploaded file as `c10-resources/countryFilter-EMR.hive`.

4. Open the Amazon EMR console at `https://console.aws.amazon.com/elasticmapreduce`. Click on the **Create New Job Flow** button to create a new EMR MapReduce job flow. Provide a name for your job flow. Select the **Run your own application** option under the **Create a Job Flow**. Select the **Hive Program** option from the drop down menu below that. Click on **Continue**.

5. Specify the S3 location of the hive script in the **Script Location** textbox of the next tab (**Specify Parameters** tab). You should specify the location of the script in the format `bucket_name/file_name`. Specify the S3 location of the uploaded input data file In the **Input Location** textbox. In the **Output Location** textbox, specify a S3 location to store the output. The output path should not exist and we use a directory (`c10-out/hive-out-1`) inside the output bucket as the output path. You should specify the input and output locations using the format, `s3n://bucket_name/path`. Click on **Continue**.

6. Configure the EC2 instances for the job flow and configure the log paths for the MapReduce computations in the next two tabs. Click on **Continue** on the **Bootstrap Options** screen. Review your job flow in the **Review** tab and click on **Create Job Flow** to launch instances and to execute the Pig script. Refer to steps 9, 10, and 11 of the *Running Hadoop MapReduce computations using Amazon ElasticMapReduce (EMR)* recipe for more details.

> Amazon will charge you for the compute and storage resources you use by clicking on **Create Job Flow** in step 11. Refer to the *Saving money by using EC2 Spot Instances* recipe to find out how you can save money by using Amazon EC2 Spot Instances.

7. Click on **Refresh** in the EMR console to monitor the progress of your MapReduce job. Select your job flow entry and click **Debug** to view the logs and to debug the computation. As EMR uploads the log files periodically; you might have to wait and refresh to access the logfiles. Check the output of the computation in the output data bucket using the AWS S3 console.

There's more...

Amazon EMR also allows to us to use Hive in the interactive mode as well.

Starting a Hive interactive session

Let's look at the steps to start a Hive interactive session:

1. Open the Amazon EMR console at `https://console.aws.amazon.com/ elasticmapreduce`. Click on the **Create New Job Flow** button to create a new EMR MapReduce **job flow**. Provide a name for your job flow. Select the **Run your own application** option under the **Create a Job Flow**. Select the **Hive Program** option from the drop-down menu below that. Click on **Continue**.

2. In order to start an interactive Hive session, select the **Start an Interactive Hive Session** option of the **Specify Parameters** tab. Click on **Continue**.

3. Configure the EC2 instances for the job flow in the **Configure EC2 Instances** tab. Click on **Continue**.

4. You must select a key pair from the **Amazon EC2 Key Pair** drop-down box in the **Advanced Options** tab. In case you do not have a usable Amazon EC2 key pair, log in to the Amazon EC2 console and create a new key pair.

5. Click on **Continue** in **Bootstrap Options**. Review your job flow in the **Review** tab and click on **Create Job Flow** to launch instances.

6. After the cluster is provisioned, go to the Amazon EMR console (`https://console. aws.amazon.com/elasticmapreduce`). Select the current job flow to view more information about the job flow. Retrieve the **Master Public DNS Name** from the information pane. (If you need more information about this step, please refer to step 6 of the *Deploying an Apache HBase Cluster on Amazon EC2 cloud using EMR* recipe).

7. Use the master public DNS name and the key file of the Amazon EC2 key pair you specified in step 4 to SSH in to the master node of the cluster.

    > ssh -i <path-to-the-key-file> hadoop@<master-public-DNS>

8. Start the Hive shell in the master node and issue your Hive queries.

See also

▸   The *Running SQL style Query with Hive* recipe of *Chapter 5*, *Hadoop Ecosystem*.

Creating an Amazon EMR job flow using the Command Line Interface

Amazon also provides a Ruby-based **Command Line Interface** (**CLI**) for EMR. The EMR Command Line Interface supports creating job flows with multiple steps as well.

This recipe creates a job flow using the EMR CLI to execute the WordCount sample from the *Running Hadoop MapReduce computations using Amazon ElasticMapReduce (EMR)* recipe of this chapter.

How to do it...

The following steps show you how to create an EMR job flow using the EMR command line interface:

1. Install Ruby 1.8 in your machine. You can verify the version of your Ruby installation by using the following command:

    ```
    > ruby –v
    ruby 1.8……
    ```

2. Create a new directory. Download the EMR Ruby CLI from `http://aws.amazon.com/developertools/2264` and unzip it to the newly created directory.

3. Create an Amazon EC2 key pair by logging in to the AWS EC2 console (`https://console.aws.amazon.com/ec2`). To create a key pair, log in to the EC2 dashboard, select a region and click on **Key Pairs** under the **Network and Security** menu. Click on the **Create Key Pair** button in the **Key Pairs** window and provide a name for the new key pair. Download and save the private key file (PEM format) in a safe location.

    > Make sure to set the appropriate file access permissions for the downloaded private key file.

4. Save the following JSON snippet in to a file named `credentials.json` in the directory of the extracted EMR CLI. Fill the fields using the credentials of your AWS account. A sample `credentials.json` file is available in the `resources/emr-cli` folder of the resource bundle available for this chapter.

    -   You can retrieve your **AWS Access Keys** from the AWS console (`http://console.aws.amazon.com`) by clicking on **Security Credentials** in the context menu that appears by clicking your AWS username in the upper-right corner of the console. You can also retrieve the AWS Access Keys by clicking on the **Security Credentials** web page link in the AWS **My Account** portal as well.

❑ Provide the name of your Key Pair (created in step 3) as the value of the key pair property.

❑ Provide the path of the saved private key file as the value of the key-pair file property.

❑ Create a S3 bucket to store the logs of the computation. Provide the S3 bucket name as the value of the `log_uri` property to store the logging and the debugging information. We assume the S3 bucket name for logging as `c10-logs`.

❑ You can use either us-east-1, us-west-2, us-west-1, eu-west-1, ap-northeast-1, ap-southeast-1, or sa-east-1 as the AWS region.

```
{
"access_id": "[Your AWS Access Key ID]",
"private_key": "[Your AWS Secret Access Key]",
"keypair": "[Your key pair name]",
"key-pair-file": "[The path and name of your PEM file]",
"log_uri": "s3n://c10-logs/",
"region": "us-east-1"
}
```

> You can skip to step 8, if you have completed the steps 2 to 6 of the _Running Hadoop MapReduce computations using Amazon ElasticMapReduce (EMR)_ recipe on this chapter.

5. Create a bucket to upload the input data by clicking on **Create Bucket in the** Amazon S3 monitoring console (`https://console.aws.amazon.com/s3`). Provide a unique name for your bucket. Upload your input data to the newly-created bucket by selecting the bucket and clicking on **Upload**. The input data for the WordCount sample should be one or more text files.

6. Create a S3 bucket to upload the JAR file needed for our MapReduce computation. Upload the `c10-samples.jar` to the newly created bucket.

7. Create a S3 bucket to store the output data of the computation.

8. Create a job flow by executing the following command inside the directory of the unzipped CLI. Replace the paths of the JAR file, input data location and the output data location with the locations you used in steps 5, 6, and 7.

```
> ./elastic-mapreduce --create --name "Hello EMR CLI" \
--jar s3n://[S3 jar file bucket]/c10-samples.jar \
--arg chapter1.WordCount \
--arg s3n://[S3 input data path] \
--arg s3n://[S3 output data path]
```

The preceding commands will create a job flow and display the job flow ID.

```
Created job flow x-xxxxxx
```

9. You can use the following command to view the description of your job flow. Replace `<job-flow-id>` using the job flow ID displayed in step 8.

```
>./elastic-mapreduce --describe <job-flow-id>
{
  "JobFlows": [
    {
      "SupportedProducts": [],
………
```

10. You can use the following command to list and to check the status of your job flows. You can also check the status and debug your job flow using the Amazon EMR Console (`https://console.aws.amazon.com/elasticmapreduce`) as well.

```
>./elastic-mapreduce --list

x-xxxxxxx     STARTING                    Hello EMR CLI
     PENDING        Example Jar Step
……..
```

11. Once the job flow is completed, check the result of the computation in the output data location using the S3 console.

```
>./elastic-mapreduce --list

x-xxxxxx    COMPLETED    ec2-xxx.amazonaws.com     Hello EMR CLI
     COMPLETED       Example Jar Step
```

There's more...

You can use EC2 spot instances with your job flows to reduce the cost of your computations. Add a bid price to your request by adding the following commands to your job flow `create` command:

```
>./elastic-mapreduce --create --name …. \
.........
--instance-group master --instance-type m1.small \
--instance-count 1 --bid-price 0.01 \
--instance-group core   --instance-type m1.small \
--instance-count 2  --bid-price 0.01
```

Refer to the *Saving money by using Amazon EC2 Spot Instances* to execute EMR job flows recipe in this chapter for more details on Amazon Spot Instances.

See also

▶ The *Running Hadoop MapReduce computations using Amazon Elastic MapReduce (EMR)* recipe of this chapter.

Deploying an Apache HBase Cluster on Amazon EC2 cloud using EMR

We can use Amazon Elastic MapReduce to start an Apache HBase cluster on the Amazon infrastructure to store large quantities of data in column oriented data store. We can use the data stored on Amazon EMR HBase clusters as input and output of EMR MapReduce computations as well. We can incrementally back up the data stored in Amazon EMR HBase clusters to Amazon S3 for data persistency. We can also start an EMR HBase cluster by restoring the data from a previous S3 backup.

In this recipe, we start an Apache HBase cluster on Amazon EC2 cloud using Amazon EMR; perform several simple operations on the newly created HBase cluster and backup the HBase data in to Amazon S3 before shutting down the cluster. Then we start a new HBase cluster restoring the HBase data backups from the original HBase cluster.

Getting ready

You should have the Amazon EMR **Command Line Interface** (**CLI**) installed and configured to manually back up HBase data. Refer to the *Creating an Amazon EMR job flow using the Command Line Interface* recipe in this chapter for more information on installing and configuring the EMR CLI.

How to do it...

The following steps show how to deploy an Apache HBase cluster on Amazon EC2 using Amazon EMR:

1. Create a S3 bucket to store the HBase backups. We assume the S3 bucket for the HBase data backups as `c10-data`.

2. Open the Amazon EMR console at `https://console.aws.amazon.com/elasticmapreduce`. Click on the **Create New Job Flow** button to create a new EMR MapReduce job flow. Provide a name for your job flow. Select the **Run your own application** option under the **Create a Job Flow**. Select the **HBase** option from the drop-down menu below that. Click on **Continue**.

3. Configure your Apache HBase cluster in the **Specify Parameters** tab. Select **No** for the **Restore from Backup** option. Select **Yes** for the **Schedule Regular Backups** and **Consistent Backup** options. Specify **Backup Frequency** for automatic schedules incremental data backups and provide a path inside the Blob we created in step 1 as the **Backup Location**. Click on **Continue**.

4. Select a key pair in the **Amazon EC2 Key Pair** drop-down box. Make sure you have the private key for the selected EC2 key pair downloaded in your computer.

> If you do not have a usable key pair, go to the EC2 console (`https://console.aws.amazon.com/ec2`) to create a key pair. To create a key pair, log in to the EC2 dashboard, select a region and click on **Key Pairs** under the **Network and Security** menu. Click on the **Create Key Pair** button in the **Key Pairs** window and provide a name for the new key pair. Download and save the private key file (PEM format) in to a safe location.

5. Configure the EC2 instances for the job flow and configure the log paths for the MapReduce computations in the next two tabs. Note that Amazon EMR does not support the use of EC2 Small and Medium instances with HBase clusters. Click on **Continue** in **Bootstrap Options**. Review your job flow in the **Review** tab and click on **Create Job Flow** to launch instances and to create your Apache HBase cluster.

> Amazon will charge you for the compute and storage resources you use by clicking **Create Job Flow** in the above step. Refer to the *Saving money by using EC2 Spot Instances* recipe to find out how you can save money by using Amazon EC2 Spot Instances.

The following steps show you how to connect to the master node of the deployed HBase cluster to start the HBase shell.

1. Go to the Amazon EMR console (`https://console.aws.amazon.com/elasticmapreduce`). Select the job flow for the HBase cluster to view more information about the job flow.

2. Retrieve the **Master Public DNS Name** value from the information pane.

3. Use the Master Public DNS Name and the EC2 PEM-based key (selected in step 4) to connect to the master node of the HBase cluster.

    ```
    > ssh -i ec2.pem hadoop@ec2-184-72-138-2.compute-1.amazonaws.com
    ```

4. Start the HBase shell using the `hbase shell` command. Create the table named `test` in your HBase installation and insert a sample entry to the table using the `put` command. Use the `scan` command to view the contents of the table.

```
> hbase shell

.........


hbase(main):001:0> create ''test'',''cf''
0 row(s) in 2.5800 seconds


hbase(main):002:0> put ''test'',''row1'',''cf:a'',''value1''
0 row(s) in 0.1570 seconds


hbase(main):003:0> scan ''test''
ROW                   COLUMN+CELL
 row1                 column=cf:a, timestamp=1347261400477,
value=value1
1 row(s) in 0.0440 seconds


hbase(main):004:0> quit
```

The following step will back up the data stored in an Amazon EMR HBase cluster.

5. Execute the following command using the Amazon EMR CLI to manually backup the data stored in an EMR HBase cluster. Retrieve the job flow name (`j-FDMXCBZP9P85`) from the EMR console. Replace `<job_flow_name>` using the retrieved job flow name. Change the backup directory path (`s3://c10-data/hbase2`) according to your backup data blob.

```
> ./elastic-mapreduce --jobflow <job_flow_name> --hbase-backup
--backup-dir s3://c10-data/hbase-manual
```

6. Select the job flow in the EMR console and click on **Terminate**.

Now, we will start a new Amazon EMR HBase cluster by restoring data from a backup.

7. Create a new job flow by clicking on **Create New Job Flow** button in the EMR console. Provide a name for your job flow. Select the **Run your own application** option under **Create a Job Flow**. Select the **HBase** option from the drop-down menu below that. Click on **Continue**.

8. Configure EMR HBase cluster to restore data from the previous data backup in the **Specify Parameters** tab. Select **Yes** for the **Restore from Backup** option and provide the backup directory path you used in step 9 in the **Backup Location** textbox. Select **Yes** for the **Schedule Regular Backups** and **Consistent Backup** options. Specify **Backup Frequency** for automatic schedules incremental data backups and provide a path inside the Blob we created in step 1 as the **Backup Location**. Click on **Continue**.

9. Repeat steps 4, 5, 6, and 7.

10. Start the HBase shell by logging to the master node of the new HBase cluster. Use the `list` command to list the set tables in HBase and the `scan test` command to view the contents of the test table.

```
> hbase shell

.........

hbase(main):001:0> list

TABLE

test

1 row(s) in 1.4870 seconds

hbase(main):002:0> scan ''test''

ROW                    COLUMN+CELL

 row1                  column=cf:a, timestamp=1347318118294,
value=value1

1 row(s) in 0.2030 seconds
```

11. Terminate your job flow using the EMR console, by selecting the job flow and clicking on the **Terminate** button.

See also

▶ The *Installing HBase* recipe of *Chapter 5, Hadoop Ecosystem* and Using Apache Whirr to deploy an Apache HBase cluster in a cloud environment recipe on this chapter.

Using EMR Bootstrap actions to configure VMs for the Amazon EMR jobs

EMR Bootstrap actions provide us a mechanism to configure the EC2 instances before running our MapReduce computations. The examples of Bootstrap actions include providing custom configuration for Hadoop, installing of any dependent software, distributing a common dataset, and so on. Amazon provides a set of predefined Bootstrap actions as well as allows us to write our own custom Bootstrap actions as well. EMR runs the Bootstrap actions in each instance before the Hadoop is started.

In this recipe, we are going to use a stop words list to filter out the common words from our WordCount sample. We download the stop words list to the workers using a custom Bootstrap action.

How to do it...

The following steps show you how to download a file to all the EC2 instances of an EMR computation using a Bootstrap script.

1. Save the following script to a file named `download-stopwords.sh`. Upload the file to a Blob container in the Amazon S3. This custom Bootstrap file downloads a stop words list to each instance and copy it to a pre-designated directory inside the instance.

```
#!/bin/bash
set -e
wget http://www.textfixer.com/resources/common-english-words-
with-contractions.txt
mkdir -p /home/Hadoop/stopwords
mv common-english-words-with-contractions.txt   /home/Hadoop/
stopwords
```

2. Complete steps 1 to 10 of the *Running Hadoop MapReduce computations using Amazon ElasticMapReduce (EMR)* recipe in this chapter.

3. Select the **Configure your Boostrap Actions** option in the **Bootstrap Options** tab. Select **Custom Action** in the **Action Type** drop-down box. Give a name to your action in the **Name** textbox and provide the S3 path of the location where you uploaded the `download-stopwords.sh` in the **Amazon S3 Location** textbox. Click on **Continue**.

4. Review your job flow in the **Review** tab and click on **Create Job Flow** to launch instances and to run the MapReduce computation.

5. Click on **Refresh** in the EMR console to monitor the progress of your MapReduce job. Select your job flow entry and click on **Debug** to view the logs and to debug the computation.

There's more...

Amazon provides us with the following predefined Bootstrap actions:

▸ `configure-daemons`: This allows us to set Java Virtual Machine (JVM) options for the Hadoop daemons such as the heap size and garbage collections behaviour.

▸ `configure-hadoop`: This allows us to modify the Hadoop configuration settings. We can either upload a Hadoop configuration XML or we can specify individual configuration options as key-value pairs.

- ▸ `memory-intensive`: This configures the Hadoop cluster for memory-intensive workloads.

- ▸ `run-if`: Run a Bootstrap action based on a property of an instance. This action can be used in scenarios where we want to run a command only in the Hadoop master node.

You can also create shutdown actions by writing scripts to a designated directory in the instance. Shutdown actions are executed after the job flow is terminated.

Refer to `http://docs.amazonwebservices.com/ElasticMapReduce/latest/DeveloperGuide/Bootstrap.html` for more information.

Using Apache Whirr to deploy an Apache Hadoop cluster in a cloud environment

Apache Whirr provides a set of cloud vendor neutral set of libraries to provision services on the cloud resources. Apache Whirr supports provisioning, installing, and configuring of Hadoop clusters in several cloud environments. In addition to Hadoop, Apache Whirr also supports provisioning of Apache Cassandra, Apache ZooKeeper, Apache HBase, Valdemort (key-value storage), and Apache Hama clusters on the cloud environments.

In this recipe, we are going to start a Hadoop cluster on Amazon EC2 cloud using Apache Whirr and run the WordCount MapReduce sample (*Writing the WordCount MapReduce sample, bundling it and running it using standalone Hadoop* recipe from *Chapter 1, Getting Hadoop up and running in a Cluster*) program on that cluster.

How to do it...

The following are the steps to deploy a Hadoop cluster on Amazon EC2 cloud using Apache Whirr and to execute the WordCount MapReduce sample on the deployed cluster.

1. Download and unzip the Apache Whirr binary distribution from `http://whirr.apache.org/`.

2. Run the following command from the extracted directory to verify your Whirr installation.

   ```
   >bin/whirr version
   Apache Whirr 0.8.0
   jclouds 1.5.0-beta.10
   ```

3. Create a directory in your home directory named `.whirr`. Copy the `conf/credentials.sample` file in the Whirr directory to the newly created directory.

    >mkdir ~/.whirr

    >cp conf/credentials.sample ~/.whirr/credentials

4. Add your AWS credentials to the `~/.whirr/credentials` file by editing it as below. You can retrieve your AWS Access Keys from the AWS console (`http://console.aws.amazon.com`) by clicking on the **Security Credentials** in the context menu that appears by clicking your AWS username in the upper-right corner of the console. A sample `credentials` file is provide in the `resources/whirr` folder of the resources for this chapter.

    ```
    # Set cloud provider connection details
    PROVIDER=aws-ec2
    IDENTITY=<AWS Access Key ID>
    CREDENTIAL=<AWS Secret Access Key>
    ```

5. Generate a `rsa` key pair using the following command. This key pair is not the same as your AWS key pair.

    >ssh-keygen -t rsa -P ''''

6. Copy the following to a file named `hadoop.properties`. If you provided a custom name for your key-pair in the preceding step, change the `whirr.private-key-file` and the `whirr.public-key-file` property values to the paths of the private key and the public key you generated. A sample `hadoop.properties` file is provided in the `resources/whirr` directory of the chapter resources.

> `whirr.aws-ec2-spot-price` is an optional property that allows us to use cheaper EC2 Spot Instances. You can delete that property to use EC2 traditional on-demand instances.

```
whirr.cluster-name=whirrhadoopcluster
whirr.instance-templates=1 hadoop-jobtracker+hadoop-namenode,2
hadoop-datanode+hadoop-tasktracker
whirr.provider=aws-ec2
whirr.private-key-file=${sys:user.home}/.ssh/id_rsa
whirr.public-key-file=${sys:user.home}/.ssh/id_rsa.pub
whirr.hadoop.version=1.0.2
whirr.aws-ec2-spot-price=0.08
```

7. Execute the following command in the whirr directory to launch your Hadoop cluster on EC2.

    >bin/whirr launch-cluster --config hadoop.properties

8. The traffic from the outside to the provisioned EC2 Hadoop cluster is routed through the master node. Whirr generates a script that we can use to start this proxy, under a subdirectory named after your Hadoop cluster inside the `~/.whirr` directory. Run this in a new terminal. It will take few minutes for whirr to start the cluster and to generate this script.

```
>cd ~/.whirr/whirrhadoopcluster/
>hadoop-proxy.sh
```

9. You can open the Hadoop web based monitoring console in your local machine by configuring this proxy in your web browser.

10. Whirr generates a `hadoop-site.xml` for your cluster in the `~/.whirr/<your cluster name>` directory. You can use it to issue Hadoop commands from your local machine to your Hadoop cluster on EC2. Export the path of the generated `hadoop-conf.xml` file to an environmental variable named `HADOOP_CONF_DIR`. To execute the Hadoop commands, you should add the `$HADOOP_HOME/bin` directory to your path or you should issue the commands from the `$HADOOP_HOME/bin` directory.

```
>export HADOOP_CONF_DIR=~/.whirr/whirrhadoopcluster/
>hadoop fs -ls /
```

11. Create a directory named `wc-input-data` in HDFS and upload a text data set to that directory.

```
>hadoop fs -mkdir wc-input-data
>hadoop fs -put sample.txt wc-input-data
```

12. In this step, we run the Hadoop WordCount sample in the Hadoop cluster we started in Amazon EC2.

```
>hadoop jar ~/workspace/HadoopBookChap10/c10-samples.jar chapter1.
WordCount wc-input-data wc-out
```

13. View the results of the WordCount computation by executing the following commands:

```
>hadoop fs -ls wc-out
Found 3 items
-rw-r--r--   3 thilina supergroup          0 2012-09-05 15:40 /
user/thilina/wc-out/_SUCCESS
drwxrwxrwx   - thilina supergroup          0 2012-09-05 15:39 /
user/thilina/wc-out/_logs
-rw-r--r--   3 thilina supergroup      19908 2012-09-05 15:40 /
user/thilina/wc-out/part-r-00000

>hadoop fs -cat wc-out/part-* | more
```

14. Issue the following command to shut down the Hadoop cluster. Make sure to download any important data before shutting down the cluster, as the data will be permanently lost after shutting down the cluster.

```
>bin/whirr destroy-cluster --config hadoop.properties
```

How it works...

This section describes the properties we used in the `hadoop.properties` file.

```
whirr.cluster-name=whirrhadoopcluster
```

The preceding property provides a name for the cluster. The instances of the cluster will be tagged using this name.

```
whirr.instance-templates=1 hadoop-jobtracker+hadoop-namenode,1 hadoop-
datanode+hadoop-tasktracker
```

The preceding property specifies the number of instances to be used for each set of roles and the type of roles for the instances. In the above example, one EC2 small instance is used with roles hadoop-jobtracker and the `hadoop-namenode`. Another two EC2 small instances are used with roles `hadoop-datanode` and `hadoop-tasktracker` in each instance.

```
whirr.provider=aws-ec2
```

We use the Whirr Amazon EC2 provider to provision our cluster.

```
whirr.private-key-file=${sys:user.home}/.ssh/id_rsa
whirr.public-key-file=${sys:user.home}/.ssh/id_rsa.pub
```

The preceding two properties point to the paths of the private key and the public key you provide for the cluster.

```
whirr.hadoop.version=1.0.2
```

We specify a custom Hadoop version using the preceding property. By default, Whirr 0.8 provisions a Hadoop 0.20.x cluster.

```
whirr.aws-ec2-spot-price=0.08
```

The preceding property specifies a bid price for the Amazon EC2 Spot Instances. Specifying this property triggers Whirr to use EC2 spot instances for the cluster. If the bid price is not met, Apache Whirr spot instance requests time out after 20 minutes. Refer to the *Saving money by using Amazon EC2 Spot Instances to execute EMR job flows* recipe for more details.

More details on Whirr configuration can be found on `http://whirr.apache.org/docs/0.6.0/configuration-guide.html`.

See also

▸ The *Using Apache Whirr to deploy an Apache HBase cluster in a cloud environment* and *Saving money by using Amazon EC2 Spot Instances to execute EMR job flows* recipes of this chapter.

Using Apache Whirr to deploy an Apache HBase cluster in a cloud environment

Apache Whirr provides a cloud vendor neutral set of libraries to access the cloud resources. In this recipe, we deploy an Apache HBase cluster on Amazon EC2 cloud using Apache Whirr.

Getting ready

Follow steps 1 to 5 of the *Using Apache Whirr to deploy an Apache Hadoop cluster in a cloud environment* recipe.

How to do it...

The following are the steps to deploy a HBase cluster on Amazon EC2 cloud using Apache Whirr.

1. Copy the following to a file named `hbase.properties`. If you provided a customs name for your key-pair in step 5 of the *Using Apache Whirr to deploy an Apache Hadoop cluster in a cloud environment* recipe, change the `whirr.private-key-file` and the `whirr.public-key-file` property values to the paths of the private key and the public key you generated. A sample `hbase.properties` file is provided in the `resources/whirr` directory of the chapter resources.

   ```
   whirr.cluster-name=whirrhbase
   whirr.instance-templates=1 zookeeper+hadoop-namenode+hadoop-
   jobtracker+hbase-master,2 hadoop-datanode+hadoop-
   tasktracker+hbase-regionserver
   whirr.provider=aws-ec2
   whirr.private-key-file=${sys:user.home}/.ssh/id_rsa
   whirr.public-key-file=${sys:user.home}/.ssh/id_rsa.pub
   ```

2. Execute the following command in the Whirr home directory to launch your HBase cluster on EC2. After provisioning the cluster, HBase prints out the commands that we can use to log in to the cluster instances. Note them down for the next steps.

   ```
   >bin/whirr launch-cluster --config hbase.properties
   ………
   ```

```
You can log into instances using the following ssh commands:

''ssh -i ~/.ssh/id_rsa -o "UserKnownHostsFile /dev/null" -o
StrictHostKeyChecking=no thilina@174.129.92.98''

''ssh -i ~//.ssh/id_rsa -o "UserKnownHostsFile /dev/null" -o
StrictHostKeyChecking=no thilina@50.16.158.59''
```

> The traffic from outside to the provisioned EC2 HBase cluster needs to be routed through the master node. Whirr generates a script that we can use to start a proxy for this purpose. The script can be found in a subdirectory named after your HBase cluster inside the `~/.whirr` directory. It will take few minutes for Whirr to provision the cluster and to generate this script. Execute this script in a new terminal to start the proxy.
>
> **>cd ~/.whirr/whirrhadoopcluster/**
> **>hbase-proxy.sh**
>
> Whirr also generates `hbase-site.xml` for your cluster in the `~/.whirr/<your cluster name>` directory, which we can use in combination with the above proxy to connect to the HBase cluster from the local client machine. However, currently a Whirr bug (`https://issues.apache.org/jira/browse/WHIRR-383`) prevents us from accessing HBase shell from our local client machine. Hence in this recipe, we directly log in to the master node of the HBase cluster.

3. Log in to an instance of your cluster using a command you note down in step 2.

   **>ssh -i ~/.ssh/id_rsa -o "UserKnownHostsFile /dev/null" -o
   StrictHostKeyChecking=no xxxx@xxx.xxx.xx.xxx**

4. Go to the `/usr/local/hbase-<your-version>` directory in the instance or add the `/usr/local/hbase-<your-version> /bin` to the `PATH` variable of the instance.

   >cd /usr/local/hbase-0.90.3

5. Start the HBase shell. Execute the following commands to test your HBase installation.

   ```
   >bin/hbase shell
   HBase Shell; .....
   Version 0.90.3, r1100350, Sat May  7 13:31:12 PDT 2011

   hbase(main):001:0> create ''test'',''cf''
   0 row(s) in 5.9160 seconds

   hbase(main):007:0> put ''test'',''row1'',''cf:a'',''value1''
   ```

```
0 row(s) in 0.6190 seconds


hbase(main):008:0> scan ''test''
ROW                       COLUMN+CELL
 row1                     column=cf:a, timestamp=1346893759876,
value=value1
1 row(s) in 0.0430 seconds


hbase(main):009:0> quit
```

6. Issue the following command to shut down the Hadoop cluster. Make sure to download any important data before shutting down the cluster, as the data will be permanently lost after shutting down the cluster.

```
>bin/whirr destroy-cluster --config hadoop.properties
```

How it works...

This section describes the `whirr.instance-templates` property we used in the `hbase.properties` file. Refer to the *Using Apache Whirr to deploy an Apache Hadoop cluster in a cloud environment* recipe for descriptions of the other properties.

```
whirr.instance-templates=1 zookeeper+hadoop-namenode+hadoop-
jobtracker+hbase-master,2 hadoop-datanode+hadoop-  tasktracker+hbase-
regionserver
```

This property specifies the number of instances to be used for each set of roles and the type of roles for the instances. In the preceding example, one EC2 small instance is used with roles `hbase-master`, `zookeeper`, `hadoop-jobtracker`, and the `hadoop-namenode`. Another two EC2 small instances are used with roles `hbase-regionserver`, `hadoop-datanode`, and `hadoop-tasktracker` in each instance.

More details on Whirr configuration can be found on `http://whirr.apache.org/docs/0.6.0/configuration-guide.html`.

See also

▶ The *Installing HBase* recipe of *Chapter 5, Hadoop Ecosystem* and the *Deploying an Apache HBase Cluster on Amazon EC2 cloud using EMR* and the *Using Apache Whirr to deploy an Apache Hadoop cluster in a cloud environment* recipes in this chapter.

Index

D

data
emitting, from mapper 83-86
grouping, MapReduce used 140-142
data de-duplication
Hadoop streaming, used 227, 228
HBase, used 233
Dataflow language 120
data mining algorithm 129
DataNodes
about 6
adding 31
decommissioning 33, 34
data preprocessing 224
datasets
joining, MapReduce used 159-164
debug scripts
about 57
writing 58
decommissioning process
about 34
working 33
DFSIO
about 30
used, for benchmarking 30
distributed cache 60
DistributedCache. *See* **Hadoop
DistributedCache**
distributed mode, Hadoop installation 6
document classification
about 244
Naive Bayes Classifier, used 244, 246

E

EC2 console
URL 264
ElasticSearch
about 185
download link 186
URL 185
used, for indexing and searching data 186,
187
using 187
working 187

EMR

used, for deploying Apache HBase Cluster on
Amazon EC2 cloud 263-268
used, for executing Hive script 256-258
used, for executing Pig script 253-255
EMR Bootstrap actions
configure-daemons 269
configure-hadoop 269
memory-intensive 270
run-if 270
used, for configuring VMs for EMR
jobs 268-270
EMR CLI
used, for creating EMR job flow 260-262
EMR job flows
creating, CLI used 260-262
executing, Amazon EC2 Spot Instances
used 252
exclude file 33

F

failure percentages
setting 60, 61
fair scheduler 62
fault tolerance 56, 57
FIFO scheduler 62
file replication factor
setting 36
FileSystem.create(filePath) method 40
FileSystem.Create() method 40
FileSystem object 42
configuring 41
frequency distribution
about 143
calculating, MapReduce used 143, 144
Fuse-DFS project
mounting 46, 47
URL 48
working 48

G

getDistance() method 199
getFileBlockLocations() function 42
getGeoLocation() method 96
getInputSplit() method 168

Thank you for buying
Hadoop MapReduce Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.
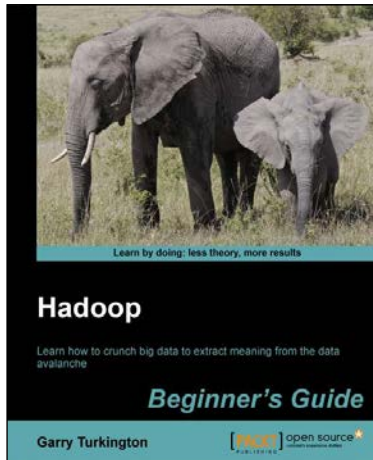
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
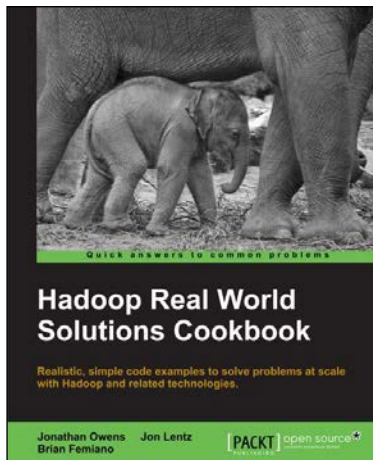
Hadoop Beginner's Guide

ISBN: 978-1-84951-730-0   Paperback: 340 pages

Learn how to crunch big data to extract meaning from the data avalanche

1. Learn tools and techniques that let you approach big data with relish and not fear

2. Shows how to build a complete infrastructure to handle your needs as your data grows

3. Hands-on examples in each chapter give the big picture while also giving direct experience

Hadoop Real World Solutions Cookbook

ISBN: 978-1-84951-912-0   Paperback: 325 pages

Realistic, simple code examples to solve problems at scale with Hadoop and related technologies

1. Solutions to common problems when working in the Hadoop environment

2. Recipes for (un)loading data, analytics, and troubleshooting

3. In depth code examples demonstrating various analytic models, analytic solutions, and common best practices

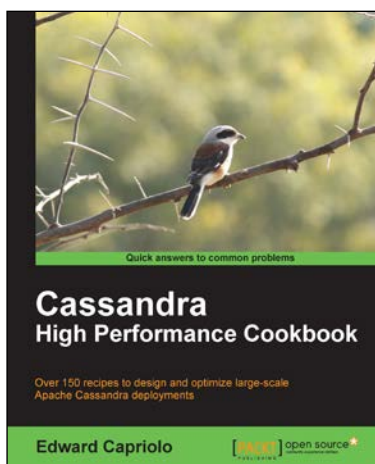Please check **www.PacktPub.com** for information on our titles

HBase Administration Cookbook

ISBN: 978-1-84951-714-0   Paperback: 332 pages

Master HBase configuration and administration for optimum database performance

1. Move large amounts of data into HBase and learn how to manage it efficiently

2. Set up HBase on the cloud, get it ready for production, and run it smoothly with high performance

3. Maximize the ability of HBase with the Hadoop eco-system including HDFS, MapReduce, Zookeeper, and Hive

Cassandra High Performance Cookbook

ISBN: 978-1-84951-512-2   Paperback: 310 pages

Over 150 recipes to design and optimize large-scale Apache Cassandra deployments

1. Get the best out of Cassandra using this efficient recipe bank

2. Configure and tune Cassandra components to enhance performance

3. Deploy Cassandra in various environments and monitor its performance

4. Well illustrated, step-by-step recipes to make all tasks look easy!

Please check **www.PacktPub.com** for information on our titles