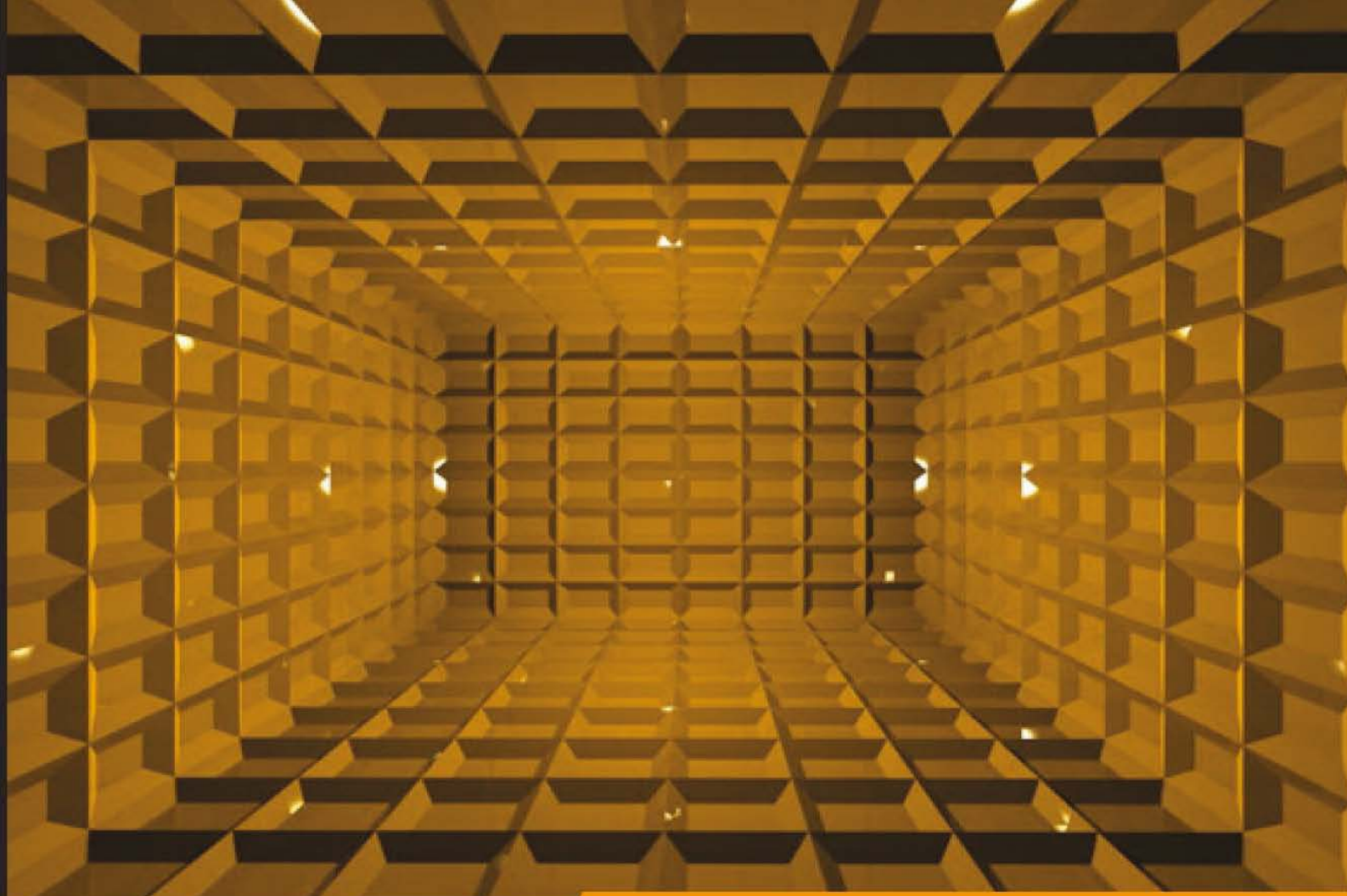


INSTANT

Short | Fast | Focused

# MapReduce Patterns – Hadoop Essentials How-to

Practical recipes to write your own MapReduce solution patterns for Hadoop programs

Srinath Perera

[PACKT] PUBLISHING

# Instant MapReduce Patterns – Hadoop Essentials How-to

Practical recipes to write your own MapReduce solution patterns for Hadoop programs

**Srinath Perera**

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

# Instant MapReduce Patterns – Hadoop Essentials How-to



First published: May 2013

Production Reference: 1160513

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-770-9

www.packtpub.com

# Credits

**Author**
Srinath Perera

**Reviewer**
Skanda Bhargav

**Acquisition Editor**
Kartikey Pandey

**Commissioning Editors**
Meeta Rajani
Llewellyn Rozario

**Technical Editor**
Worrell Lewis

**Project Coordinator**
Amey Sawant

**Proofreader**
Amy Guest

**Graphics**
Valentina D'silva

**Production Coordinator**
Prachali Bhiwandkar

**Cover Work**
Prachali Bhiwandkar

**Cover Image**
Nitesh Thakur

# About the Author

**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. He 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 degree from the University of Moratuwa, Sri Lanka.

He has authored many technical and peer reviewed research articles, and more details can be found on 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 like Hadoop and Cassandra daily. He also teaches a parallel programming graduate class at University of Moratuwa, which is primarily based on Hadoop.

---

I would like to thank my wife Miyuru, my son Dimath, and my parents, whose never-ending support keeps me going. I would also like to thank Sanjiva from WSO2 who encourages us to make our mark even though projects like these are not in the job description. Finally, I would like to thank my colleges at WSO2 for the ideas and companionship that has shaped the book in many ways.

---

# About the Reviewer

**Skanda Bhargav** is an Engineering graduate from VTU, Belgaum in Karnataka, India. He did his majors in Computer Science Engineering. He is currently employed with a MNC based out of Bangalore. Skanda is a Cloudera-certified developer in Apache Hadoop. His interests are Big Data and Hadoop.

---

I would like to thank my family for their immense support and faith in me throughout my learning stage. My friends have brought the confidence in me to a level that makes me bring out the best out of myself. I am happy that God has blessed me with such wonderful people around me without which this work might not have been the success that it is today.

---

# 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

Although there are many resources available on the Web for Hadoop, most stop at the surface or provide a solution for a specific problem. *Instant MapReduce Patterns – Hadoop Essentials How-to* is a concise introduction to Hadoop and programming with MapReduce. It is aimed to get you started and give an overall feel to programming with Hadoop so that you will have a solid foundation to dig deep into each type of MapReduce problem, as needed.

## What this book covers

*Writing a word count application using Java (Simple)* describes how to write a word count program using Java, without MapReduce. We will use this to compare and contrast against the MapReduce model.

*Writing a word count application with MapReduce and running it (Simple)* explains how to write the word count using MapReduce and how to run it using the Hadoop local mode.

*Installing Hadoop in a distributed setup and running a word count application (Simple)* describes how to install Hadoop in a distributed setup and run the above Wordcount job in a distributed setup.

*Writing a formatter (Intermediate)* explains how to write a Hadoop data formatter to read the Amazon data format as a record instead of reading data line by line.

*Analytics – drawing a frequency distribution with MapReduce (Intermediate)* describes how to process Amazon data with MapReduce, generate data for a histogram, and plot it using gnuplot.

*Relational operations – join two datasets with MapReduce (Advanced)* describes how to join two datasets using MapReduce.

*Set operations with MapReduce (Intermediate)* describes how to process Amazon data and perform the set difference with MapReduce. Further, it will discuss how other set operations can also be implemented using similar methods.

*Cross correlation with MapReduce (Intermediate)* explains how to use MapReduce to count the number of times two items occur together (cross correlation).

*Simple search with MapReduce (Intermediate)* describes how to process Amazon data and implement a simple search using an inverted index.

*Simple graph operations with MapReduce (Advanced)* describes how to perform a graph traversal using MapReduce.

*Kmeans with MapReduce (Advanced)* describes how to cluster a dataset using the Kmeans algorithm. Clustering groups the data into several groups such that items in each group are similar and items in different groups are different according to some distance measure.

# What you need for this book

To try out this book, you need access to a Linux or Mac computer with JDK 1.6 installed.

# Who this book is for

For big data enthusiasts and would-be Hadoop programmers. The book for Java programmers who either have not worked with Hadoop at all, or who know Hadoop and MapReduce bit, but are not sure how to deepen their understanding.

# 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: " Verify the installation by listing processes through the `ps | grep java` command."

A block of code is set as follows:

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

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
job.setInputFormatClass(ItemSalesDataFormat.class);
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

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

```
>bin/hadoopdfs -mkdir /data/
>bin/hadoopdfs -mkdir /data/amazon-dataset
>bin/hadoopdfs -put <SAMPLE_DIR>/amazon-meta.txt /data/amazon-dataset/
>bin/hadoopdfs -ls /data/amazon-dataset
```

**New terms** and **important words** are shown in bold.

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 sendan e-mail to `feedback@packtpub.com`, and mention the book title viathe 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 theexample 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/submit-errata`, selecting your book, clicking on the **erratasubmissionform** 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.

# Instant MapReduce Patterns – Hadoop Essentials How-to

Welcome to *Instant Mapreduce Patterns – Hadoop Essentials How-to*. This book provides an introduction to Hadoop and discusses several Hadoop-based analysis implementations with Hadoop. It is intended to be a concise "hands-on" Hadoop guide for beginners.

Historically, data processing was completely done using database technologies. Most of the data had a well-defined structure and was often stored in databases. When handling such data, relational databases were the most common store choice. Those, datasets were small enough to be stored and queried using relational databases.

However, the datasets started to grow in size. Soon, high-tech companies like Google found many large datasets that were not amenable to databases. For example, Google was crawling and indexing the entire Internet, which soon reached terabytes and then petabytes. Google developed a new programming model called MapReduce to handle large-scale data analysis, and later they introduced the model through their seminal paper *MapReduce: Simplified Data Processing on Large Clusters*.

Hadoop, the Java-based open source project, is an implementation of the MapReduce programming model. It enables users to only write the processing logic, and MapReduce frameworks such as Hadoop can execute the logic while handling distributed aspects such as job scheduling, data movements, and failures transparently from the users.

Hadoop has become the de facto MapReduce implementation for Java. A wide spectrum of users from students to large enterprises use Hadoop to solve their data processing problems, and MapReduce has become one of the most sought after skill in the job market.

This book is an effort to provide a concise introduction to MapReduce and different problems you can solve using MapReduce. There are many resources on how to get started with Hadoop and run a word count example, which is the "Hello World" equivalent in the MapReduce world. However, there is not much resource that provides a concise introduction to solving different types of problems using MapReduce. This book tries to address that gap.

The first three recipes of the book focus on writing a simple MapReduce program and running it using Hadoop. The next recipe explains how to write a custom formatter that can be used to parse a complicated data structure from the input files. The next recipe explains how to use MapReduce to calculate basic analytics and how to use GNU plot to plot the results. This is one of the common use case of Hadoop.

The rest of the recipes cover different classes of problems that can be solved with MapReduce, and provide an example of the solution pattern common to that class. They cover the problem classes: set operations, cross correlation, search, graph and relational operations, and similarity clustering.

Throughout this book, we will use the public dataset on the Amazon sales data collected by Stanford University. Dataset provides information about books and users who have brought those books. An example data record is shows as follows:

```
Id:   3
ASIN: 0486287785
title: World War II Allied Fighter Planes Trading Cards
group: Book
salesrank: 1270652
similar: 0
categories: 1
   |Books[283155]|Subjects[1000]|Home & Garden[48]|Crafts &
Hobbies[5126]|General[5144]
reviews: total: 1  downloaded: 1  avg rating: 5
    2003-7-10  cutomer: A3IDGASRQAW8B2  rating: 5  votes:   2
helpful:   2
```

The dataset is available at `http://snap.stanford.edu/data/#amazon`. It is about 1 gigabyte in size. Unless you have access to a large Hadoop cluster, it is recommended to use smaller subsets of the same dataset available with the sample directory while running the samples.

# Writing a word count application using Java (Simple)

This recipe demonstrates how to write an analytics tasks with Hadoop using basic Java constructs. It further discusses challenges of running applications that work on many machines and motivates the need for MapReduce like frameworks.

It will describe how to count the number of occurrences of words in a file.

## Getting ready

This recipe assumes you have a computer that has Java installed and the `JAVA_HOME` environment variable points to your Java installation. Download the code for the book and unzip them to a directory. We will refer to the unzipped directory as `SAMPLE_DIR`.

## How to do it...

1. Copy the dataset from `hadoop-microbook.jar` to `HADOOP_HOME`.
2. Run the word count program by running the following command from `HADOOP_HOME`:

   ```
   $ java -cp hadoop-microbook.jar microbook.wordcount.JavaWordCount
   SAMPLE_DIR/amazon-meta.txt results.txt
   ```

3. Program will run and write the word count of the input file to a file called `results.txt`. You will see that it will print the following as the result:

   ```
   B00007ELF7=1

   Vincent[412370]=2

   35681=1
   ```

## How it works...

You can find the source code for the recipe at `src/microbook/JavaWordCount.java`. The code will read the file line by line, tokenize each line, and count the number of occurrences of each word.

```
BufferedReaderbr = new BufferedReader(
newFileReader(args[0]));
String line = br.readLine();
while (line != null) {
    StringTokenizertokenizer = new StringTokenizer(line);
    while(tokenizer.hasMoreTokens()){
        String token = tokenizer.nextToken();
        if(tokenMap.containsKey(token)){
            Integer value = (Integer)tokenMap.get(token);
            tokenMap.put(token, value+1);
        }else{
            tokenMap.put(token, new Integer(1));
        }
    }
    line = br.readLine();
```

```
}

Writer writer = new BufferedWriter(
    new FileWriter("results.txt"));

for(Entry<String, Integer> entry: tokenMap.entrySet()){
    writer.write(entry.getKey() + "= "+ entry.getValue());
}
```

This program can only use one computer for processing. For a reasonable size dataset, this is acceptable. However, for a large dataset, it will take too much time. Also, this solution keeps all the data in memory, and with a large dataset, the program is likely to run out of memory. To avoid that, the program will have to move some of the data to disk as the available free memory becomes limited, which will further slow down the program.

We solve problems involving large datasets using many computers where we can parallel process the dataset using those computers. However, writing a program that processes a dataset in a distributed setup is a heavy undertaking. The challenges of such a program are shown as follows:

- The distributed program has to find available machines and allocate work to those machines.
- The program has to transfer data between machines using message passing or a shared filesystem. Such a framework needs to be integrated, configured, and maintained.
- The program has to detect any failures and take corrective action.
- The program has to make sure all nodes are given, roughly, the same amount of work, thus making sure resources are optimally used.
- The program has to detect the end of the execution, collect all the results, and transfer them to the final location.

Although it is possible to write such a program, it is a waste to write such programs again and again. MapReduce-based frameworks like Hadoop lets users write only the processing logic, and the frameworks can take care of complexities of a distributed execution.

# Writing a word count application with MapReduce and running it (Simple)

The first recipe explained how to implement the word count application without MapReduce, and limitations of the implementation. This recipe explains how to implement a word counting application with MapReduce and explains how it works.

## Getting ready

1. This recipe assumes that you have access to a computer that has **Java Development Kit** (**JDK**) installed and the `JAVA_HOME` variable configured.
2. Download a Hadoop distribution 1.1.x from `http://hadoop.apache.org/releases.html` page.
3. Unzip the distribution; we will call this directory `HADOOP_HOME`. Now you can run Hadoop jobs in local mode.
4. Download the sample code for the book and download the data files as described in the first recipe. We call that directory as `DATA_DIR`.

**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.

## How to do it...

1. Copy the `hadoop-microbook.jar` file from `SAMPLE_DIR` to `HADOOP_HOME`.
2. Run the MapReduce job through the following command from `HADOOP_HOME`:

```
$bin/hadoop -cp hadoop-microbook.jar  microbook.wordcount.
WordCount amazon-meta.txt wordcount-output1
```

3. Your can find the results from output directory.
4. It will print the results as follows:

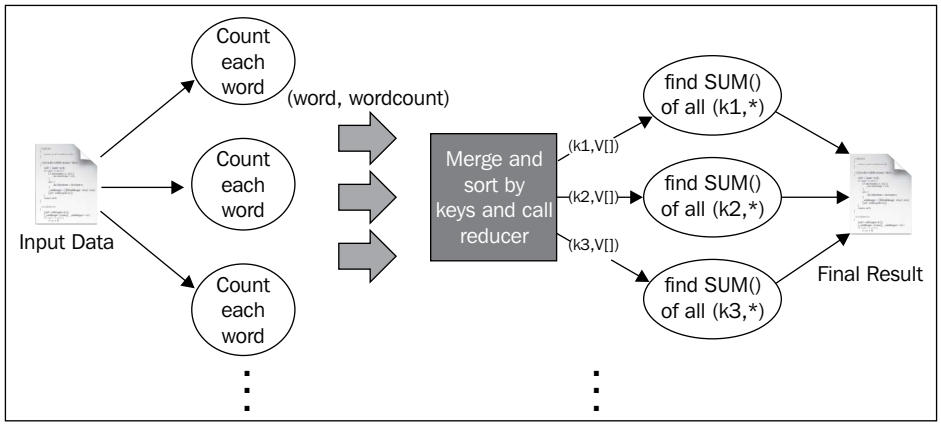
```
B00007ELF7=1
Vincent[412370]=2
35681=1
```

## How it works...

You can find the source code for the recipe at `src/microbook/wordcount/WordCount.java`.

The word count job accepts an input directory, a mapper function, and a reducer function as inputs. We use the mapper function to process the data in parallel, and we use the reducer function to collect results of the mapper and produce the final results. Mapper sends its results to reducer using a key-value based model. Let us walk through a MapReduce execution in detail.

The following diagram depicts the MapReduce job execution, and the following code listing shows the mapper and reducer functions:




When you run the MapReduce job, Hadoop first reads the input files from the input directory line by line. Then Hadoop invokes the mapper once for each line passing the line as the argument. Subsequently, each mapper parses the line, and extracts words included in the line it received as the input. After processing, the mapper sends the word count to the reducer by emitting the word and word count as name value pairs.

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

Hadoop collects all name value pairs emitted from the mapper functions, and sorts them by the key. Here the key is the word and value is the number of occurrences of the word. Then it invokes the reducer once for each key passing all the values emitted against the same key as arguments. The reducer calculates the sum of those values and emits them against the key. Hadoop collects results from all reducers and writes them to the output file.

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

```
        result.set(sum);
        context.write(key, result);
    }
```

The following code shows the main method that will invoke the job. It configures mapper, reducer, input and output formats, and input and output directories. Here, input and output of mapper and reducer match the values configured with `setOutputKeyClass(..)`, `setOutputValueClass(..)`, `job.setMapOutputKeyClass(..)`, and `job.setMapOutputValueClass(..)`:

```
    JobConfconf = new JobConf();
    String[] otherArgs =
        new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
        System.err.println("Usage: <in><out>");
        System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(WordcountMapper.class);
    job.setReducerClass(WordcountReducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.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);
```

In the model, the map function is used to process data in parallel and distribute them to the reducers, and we use the reduce function to collect the results together.

## There's more...

Since we run this program in a local Hadoop installation, it will completely run in a single machine. The next recipe explains how to run it within a distributed Hadoop cluster.

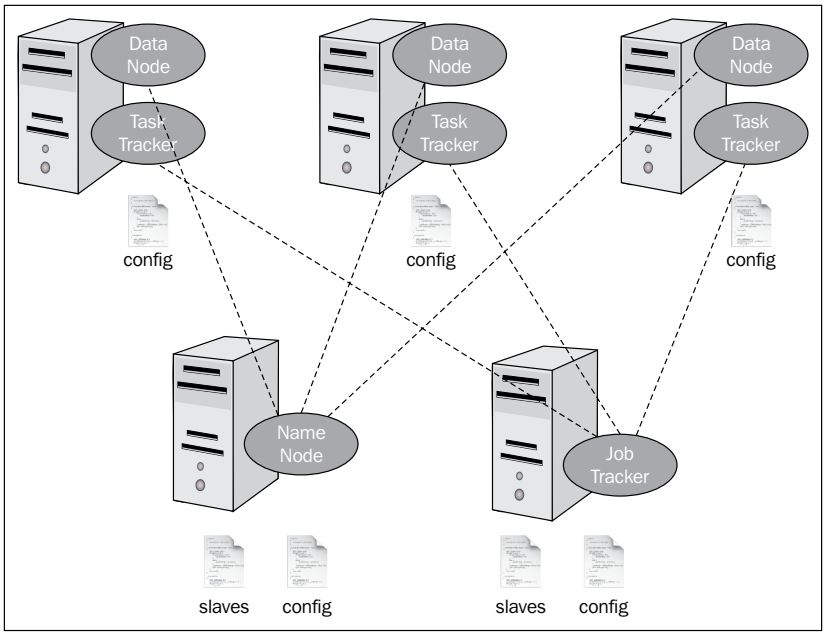
# Installing Hadoop in a distributed setup and running a word count application (Simple)

The following figure shows a typical Hadoop deployment. A Hadoop deployment consists of a single name node, multiple data nodes, a single job tracker, and multiple task trackers. Let us look at each type of node.

The **name node** and **data nodes** provide the HDFS filesystem where data nodes hold the actual data and the name node holds information about which file is in which data node. A user, who wants to read a file, first talks to the name node, finds where the file is located, and then talks to data nodes to access the file.

Similarly, the **job tracker** keeps track of MapReduce jobs and schedules the individual map and reduces tasks in the Task Trackers. Users submit the jobs to the Job Tracker, which runs them in the Task Trackers. However, it is possible to run all these types of servers in a single node or in multiple nodes.

This recipe explains how to set up your own Hadoop cluster. For the setup, we need to configure job trackers and task trackers and then point to the task trackers in the `HADOOP_HOME/conf/slaves` file of the job tracker. When we start the job tracker, it will start the task tracker nodes. Let us look at the deployment in detail:




## Getting ready

1. You need at least one Linux or Mac OS X machine for the setup. 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 and the other nodes as slave nodes. You will run the HDFS name node and job tracker in the master node. If you are using a single machine, use it as both the master node as well as the slave node.
2. Install Java in all machines that will be used to set up Hadoop.

## How to do it...

1. In each machine, create a directory for Hadoop data, which we will call `HADOOP_DATA_DIR`. Then, let us create three subdirectories `HADOOP_DATA/data`, `HADOOP_DATA/local`, `HADOOP_DATA/name`.
2. Set up the SSH key to enable SSH from master nodes to slave nodes. Check that you can SSH to the localhost and to all other nodes without a passphrase by running the following command.

   ```
   >ssh localhost (or sshIPaddress)
   ```

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

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

4. Then move the `~/.ssh/id_dsa.pub` file to all the nodes in the cluster. Add the SSH keys to the `~/.ssh/authorized_keys` file in each node by running the following command:

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

5. Then you can log in with the following command:

   ```
   >ssh localhost
   ```

6. Unzip the Hadoop distribution at the same location in all machines using the following command:

   ```
   >tar -zxvf hadoop-1.0.0.tar.gz.
   ```

7. In all machines, edit the `HADOOP_HOME/conf/hadoop-env.sh` file by uncommenting the line with `JAVA_HOME` and to point to your local Java installation. For example, if Java is in `/opt/jdk1.6`, change the line to `export JAVA_HOME=/opt/jdk1.6`.
8. Place the IP address of the node used as the master (for running job tracker and name node) in `HADOOP_HOME/conf/masters` in a single line. If you are doing a single node deployment, leave the current value of `localhost` as it is.

   ```
   209.126.198.72
   ```

9. 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
   ```

10. Inside each node's `HADOOP_HOME/conf` directory, add the following to the `core-site.xml`, `hdfs-site.xml`, and `mapred-site.xml` files. Before adding the configurations, replace `MASTER_NODE` with the IP of the master node and `HADOOP_DATA_DIR` with the directory you created in step 1.

11. Add the URL of the name node to `HADOOP_HOME/conf/core-site.xml` as follows:

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

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

```
<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>
```

13. The MapReduce local directory is the location used by Hadoop to store temporary files. Also add job tracker location to `HADOOP_HOME/conf/mapred-site.xml`. The Hadoop client will use this job tracker when submitting jobs. The final property sets the maximum map tasks per node. You should set this same as the amount of cores (CPU) in the machine.

```
<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>
```

14. Format a new HDFS filesystem by running the following command from the Hadoop name node (aka master node).

```
>run bin/hadoopnamenode –format
...
/Users/srinath/playground/hadoop-book/hadoop-temp/dfs/name has
been successfully formatted.
```

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

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

16. Verify the installation by listing processes through the `ps | grep java` command. The master node will list three processes: name node, data node, job tracker, and task tracker and the salves will have a data node and task tracker.
17. Browse the Web-based monitoring pages for the name node and job tracker, NameNode – `http://MASTER_NODE:50070/` and JobTracker – `http://MASTER_NODE:50030/`.
18. You can find the log files in `${HADOOP_HOME}/logs`.
19. Make sure the HDFS setup is OK by listing the files using HDFS command line.

```
bin/hadoopdfs -ls /
Found 2 items
drwxr-xr-x   - srinathsupergroup     0 2012-04-09 08:47 /Users
drwxr-xr-x   - srinathsupergroup     0 2012-04-09 08:47 /tmp
```

20. Download the weblog dataset from `http://snap.stanford.edu/data/bigdata/amazon/amazon-meta.txt.gz` and unzip it. We call this `DATA_DIR`. The dataset will be about 1 gigabyte, and if you want your executions to finish faster, you can only use a subset of the dataset.
21. Copy the `hadoop-microbook.jar` file from `SAMPLE_DIR` to `HADOOP_HOME`.
22. If you have not already done so, let us upload the amazon dataset to the HDFS filesystem using following commands:

```
>bin/hadoopdfs -mkdir /data/
>bin/hadoopdfs -mkdir /data/amazon-dataset
>bin/hadoopdfs -put <SAMPLE_DIR>/amazon-meta.txt /data/amazon-
dataset/
>bin/hadoopdfs -ls /data/amazon-dataset
```

23. Run the MapReduce job through the following command from `HADOOP_HOME`:

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.wrodcount.
WordCount /data/amazon-dataset /data/wordcount-doutput
```

24. Your can find the results of the MapReduce job from the output directory. Use the following command to list the content:

```
$ bin/hadoop jar hadoop-microbook.jar dfs –ls /data/wordcount-
doutput
```

## How it works...

As described in the introduction to the chapter, Hadoop installation consists of HDFS nodes, a job tracker, and worker nodes. When we start the name node, it finds salves through `HADOOP_HOME/salves` file and uses SSH to start the data nodes in the remote server. Also when we start the job tracker, it finds salves through the `HADOOP_HOME/salves` file and starts the task trackers.

When we run the MapReduce job, the client finds the job tracker from the configuration and submits the jobs to the job tracker. The clients wait for the execution to finish and keep receiving standard out and prints it to the console as long as the job is running.

# Writing a formatter (Intermediate)

By default, when you run a MapReduce job, it will read the input file line by line and feed each line into the map function. For most cases, this works well. However, sometimes one data record is contained within multiple lines. For example, as explained in the introduction, our dataset has a record format that spans multiple lines. In such cases, it is complicated to write a MapReduce job that puts those lines together and processes them.

The good news is that Hadoop lets you override the way it is reading and writing files, letting you take control of that step. We can do that by adding a new formatter. This recipe explains how to write a new formatter.

You can find the code for the formatter from `src/microbook/ItemSalesDataFormat.java`. The recipe will read the records from the dataset using the formatter, and count the words in the titles of the books.

## Getting ready

1. This assumes that you have installed Hadoop and started it. Refer to the *Writing a word count application using Java (Simple)* and *Installing Hadoop in a distributed setup and running a word count application (Simple)* recipes for more information. We will use the `HADOOP_HOME` to refer to the Hadoop installation directory.

2. This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the *Writing a word count application with MapReduce and running it (Simple)* recipe.
3. Download the sample code for the chapter and copy the data files as described in the *Writing a word count application with MapReduce and running it (Simple)* recipe.

## How to do it...

1. If you have not already done so, let us upload the amazon dataset to the HDFS filesystem using the following commands:

```
>bin/hadoopdfs -mkdir /data/

>bin/hadoopdfs -mkdir /data/amazon-dataset

>bin/hadoopdfs -put <SAMPLE_DIR>/amazon-meta.txt /data/amazon-dataset/

>bin/hadoopdfs -ls /data/amazon-dataset
```

2. Copy the `hadoop-microbook.jar` file from `SAMPLE_DIR` to `HADOOP_HOME`.
3. Run the MapReduce job through the following command from `HADOOP_HOME`:

```
>bin/hadoop jar hadoop-microbook.jar  microbook.format.
TitleWordCount /data/amazon-dataset /data/titlewordcount-output
```

4. You can find the result from output directory using the following command:

```
>bin/Hadoop dfs -cat /data/titlewordcount-output/*
```

   You will see that it has counted the words in the book titles.

## How it works...

In this recipe, we ran a MapReduce job that uses a custom formatter to parse the dataset. We enabled the formatter by adding the following highlighted line to the main program.

```
JobConfconf = new JobConf();
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(TitleWordCount.class);
job.setMapperClass(WordcountMapper.class);
job.setReducerClass(WordcountReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
job.setInputFormatClass(ItemSalesDataFormat.class);

FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

The following code listing shows the formatter:

```
public class ItemSalesDataFormat
    extends FileInputFormat<Text, Text>{
    private ItemSalesDataReadersaleDataReader = null;

    public RecordReader<Text, Text>createRecordReader(
        InputSplitinputSplit, TaskAttemptContext attempt) {
        saleDataReader = new ItemSalesDataReader();
        saleDataReader.initialize(inputSplit, attempt);
        return saleDataReader;
    }

}
```

The formatter creates a record reader, and the record reader will do the bulk of the real work. When we run the Hadoop job, it will find the formatter, create a new record reader passing each file, read records from record readers, and pass those records to the map tasks.

The following code listing shows the record reader:

```
public class ItemSalesDataReader
extendsRecordReader<Text, Text> {

  public void initialize(InputSplitinputSplit,
  TaskAttemptContext attempt) {
     //open the file
  }

  public boolean nextKeyValue(){
```

```
          //parse the file until end of first record
        }

    public Text getCurrentKey(){ ... }

    public Text getCurrentValue(){ ... }

    public float getProgress(){ ..   }

    public void close() throws IOException {
    //close the file
    }
  }
```

Hadoop will invoke the `initialize(..)` method passing the input file and call other methods until there are keys to be read. The implementation will read the next record when `nextKeyValue()` is invoked, and return results when the other methods are called.

Mapper and reducer look similar to the versions used in the second recipe except for the fact that mapper will read the title from the record it receives and only use the title when counting words. You can find the code for mapper and reducer at `src/microbook/wordcount/TitleWordCount.java`.

## There's more...

Hadoop also supports output formatters, which is enabled in a similar manner, and it will return a `RecordWriter` instead of the reader. You can find more information at `http://www.infoq.com/articles/HadoopOutputFormat` or from the freely available article of the *Hadoop MapReduce Cookbook*, *Srinath Perera* and *Thilina Gunarathne*, *Packt Publishing* at `http://www.packtpub.com/article/advanced-hadoop-mapreduce-administration`.

Hadoop has several other input output formats such as `ComposableInputFormat`, `CompositeInputFormat`, `DBInputFormat`, `DBOutputFormat`, `IndexUpdateOutputFormat`, `MapFileOutputFormat`, `MultipleOutputFormat`, `MultipleSequenceFileOutputFormat`, `MultipleTextOutputFormat`, `NullOutputFormat`, `SequenceFileAsBinaryOutputFormat`, `SequenceFileOutputFormat`, `TeraOutputFormat`, and `TextOutputFormat`. In most cases, you might be able to use one of these instead of writing a new one.

# Analytics – drawing a frequency distribution with MapReduce (Intermediate)

Often, we use Hadoop to calculate **analytics**, which are basic statistics about data. In such cases, we walk through the data using Hadoop and calculate interesting statistics about the data. Some of the common analytics are show as follows:

- Calculating statistical properties like minimum, maximum, mean, median, standard deviation, and so on of a dataset. For a dataset, generally there are multiple dimensions (for example, when processing HTTP access logs, names of the web page, the size of the web page, access time, and so on, are few of the dimensions). We can measure the previously mentioned properties by using one or more dimensions. For example, we can group the data into multiple groups and calculate the mean value in each case.
- Frequency distributions histogram counts the number of occurrences of each item in the dataset, sorts these frequencies, and plots different items as X axis and frequency as Y axis.
- Finding a correlation between two dimensions (for example, correlation between access count and the file size of web accesses).
- Hypothesis testing: To verify or disprove a hypothesis using a given dataset.

However, Hadoop will only generate numbers. Although the numbers contain all the information, we humans are very bad at figuring out overall trends by just looking at numbers. On the other hand, 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 will explain how to use MapReduce to calculate frequency distribution of the number of items brought by each customer. Then we will use gnuplot, a free and powerful, plotting program to plot results from the Hadoop job.

## Getting ready

1. This recipe assumes that you have access to a computer that has Java installed and the `JAVA_HOME` variable configured.
2. Download a Hadoop distribution 1.1.x from `http://hadoop.apache.org/releases.html` page.

3. Unzip the distribution, we will call this directory `HADOOP_HOME`.
4. Download the sample code for the chapter and copy the data files as described in the *Writing a word count application using Java (Simple)* recipe.

## How to do it...

1. If you have not already done so, let us upload the amazon dataset to the HDFS filesystem using the following commands:

```
>bin/hadoopdfs -mkdir /data/

>bin/hadoopdfs -mkdir /data/amazon-dataset

>bin/hadoopdfs -put <SAMPLE_DIR>/amazon-meta.txt /data/amazon-dataset/

>bin/hadoopdfs -ls /data/amazon-dataset
```

2. Copy the `hadoop-microbook.jar` file from `SAMPLE_DIR` to `HADOOP_HOME`.
3. Run the first MapReduce job to calculate the buying frequency. To do that run the following command from `HADOOP_HOME`:

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.frequency.
BuyingFrequencyAnalyzer/data/amazon-dataset /data/frequency-
output1
```

4. Use the following command to run the second MapReduce job to sort the results of the first MapReduce job:

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.frequency.
SimpleResultSorter /data/frequency-output1 frequency-output2
```

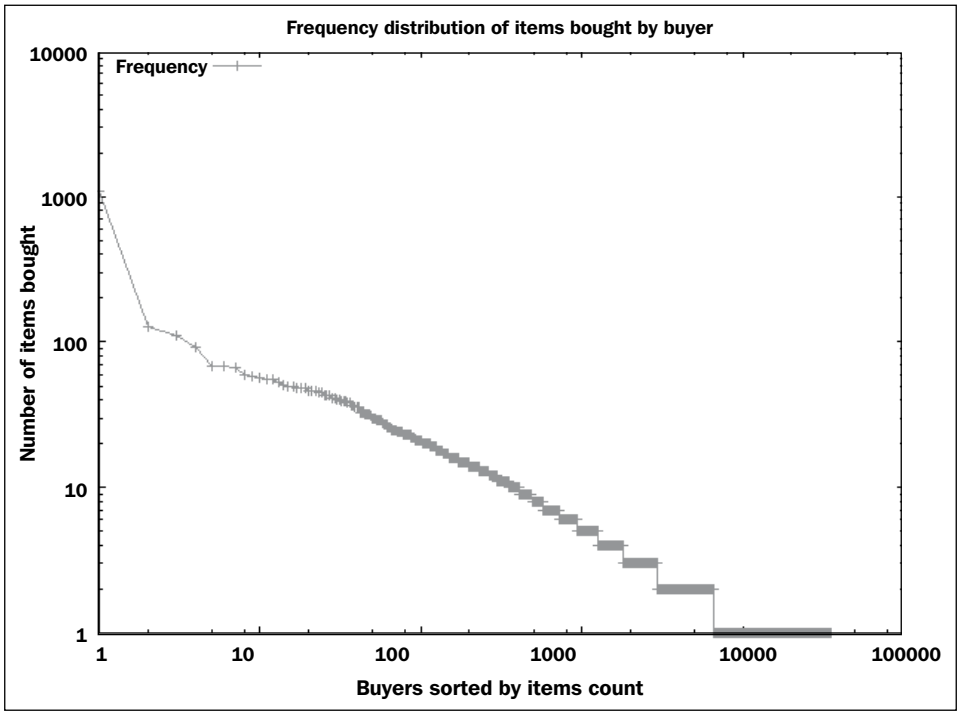
5. You can find the results from the output directory. Copy results to `HADOOP_HOME` using the following command:

```
$ bin/Hadoop dfs -get /data/frequency-output2/part-r-00000 1.data
```

6. Copy all the `*.plot` files from `SAMPLE_DIR` to `HADOOP_HOME`.
7. Generate the plot by running the following command from `HADOOP_HOME`.

```
$gnuplot buyfreq.plot
```

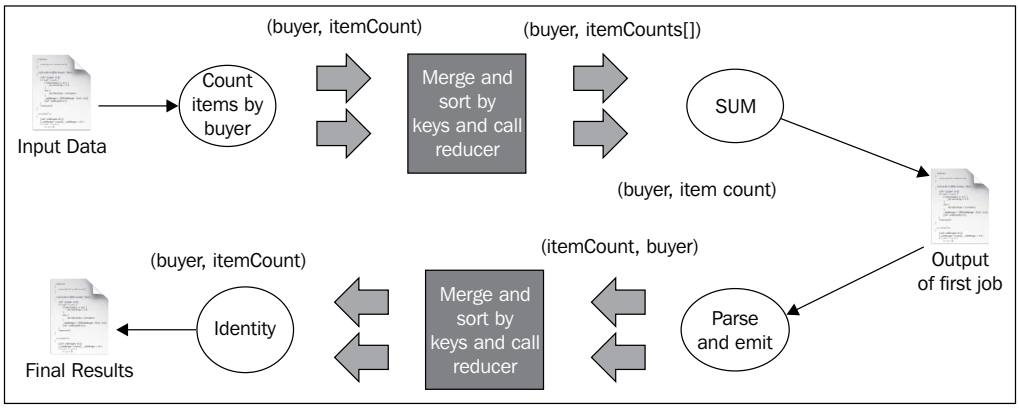
8. It will generate a file called `buyfreq.png`, which will look like the following:




As the figure depicts, few buyers have brought a very large number of items. The distribution is much steeper than normal distribution, and often follows what we call a Power Law distribution. This is an example that analytics and plotting results would give us insight into, underlying patterns in the dataset.

## How it works...

You can find the mapper and reducer code at `src/microbook/frequency/BuyingFrequencyAnalyzer.java`.




This figure shows the execution of two MapReduce jobs. Also the following code listing shows the map function and the reduce function of the first job:

```
public void map(Object key, Text value, Context context
                ) throwsIOException, InterruptedException {
    List<BuyerRecord> records =
        BuyerRecord.parseAItemLine(value.toString());
    for(BuyerRecord record: records){
        context.write(new Text(record.customerID),
           new IntWritable(record.itemsBrought.size()));
    }
}

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

As shown by the figure, Hadoop will read the input file from the input folder and read records using the custom formatter we introduced in the *Writing a formatter (Intermediate)* recipe. It invokes the mapper once per each record, passing the record as input.

The mapper extracts the customer ID and the number of items the customer has brought, and emits the customer ID as the key and number of items as the value.

Then, Hadoop sorts the key-value pairs by the key and invokes a reducer once for each key passing all values for that key as inputs to the reducer. Each reducer sums up all item counts for each customer ID and emits the customer ID as the key and the count as the value in the results.

Then the second job sorted the results. It reads output of the first job as the result and passes each line as argument to the map function. The map function extracts the customer ID and the number of items from the line and emits the number of items as the key and the customer ID as the value. Hadoop will sort the key-value pairs by the key, thus sorting them by the number of items, and invokes the reducer once per key in the same order. Therefore, the reducer prints them out in the same order essentially sorting the dataset.

Since we have generated the results, let us look at the plotting. You can find the source for the gnuplot file from `buyfreq.plot`. The source for the plot will look like the following:

```
set terminal png
set output "buyfreq.png"

set title "Frequency Distribution of Items brought by Buyer";
setylabel "Number of Items Brought";
setxlabel "Buyers Sorted by Items count";
set key left top
set log y
set log x

plot "1.data" using 2 title "Frequency" with linespoints
```

Here the first two lines define the output format. This example uses `png`, but gnuplot supports many other terminals such as `screen`, `pdf`, and `eps`. The next four lines define the axis labels and the title, and the next two lines define the scale of each axis, and this plot uses log scale for both.

The last line defines the plot. Here, it is asking gnuplot to read the data from the `1.data` file, and to 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.

Here if you want to plot one column against another, for example data from column 1 against column 2, you should write `using 1:2` instead of `using 2`.

## There's more...

We can use a similar method to calculate the most types of analytics and plot the results. Refer to the freely available article of *Hadoop MapReduce Cookbook*, *Srinath Perera* and *Thilina Gunarathne*, *Packt Publishing* at `http://www.packtpub.com/article/advanced-hadoop-mapreduce-administration` for more information.

# Relational operations – join two datasets with MapReduce (Advanced)

Before MapReduce, relational operations like filter, join, sorting, and grouping were the primary operations used for processing large datasets. MapReduce can very easily support operations like filter and sorting. For more information, refer to *2.3.3 Relational-Algebra Operations* of the free available book *Mining of Massive Datasets*, *Anand Rajaraman* and *Jeffrey D. Ullman*, *Cambridge University Press*, *2011*.

This recipe explains how to use MapReduce to join two datasets. It will join 100 customers who have bought most items against the dataset that provides items bought by each customer and produce a list of items brought by the 100 most-frequent customers as output.

## Getting ready

1. This assumes that you have installed Hadoop and started it. Refer to the *Writing a word count application using Java (Simple)* and *Installing Hadoop in a distributed setup and running a word count application (Simple)* recipes for more information. We will use `HADOOP_HOME` to refer to the Hadoop installation directory.
2. This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the *Writing a word count application with MapReduce and running it (Simple)* recipe.
3. Download the sample code for the chapter and download the data files as described in the *Writing a word count application with MapReduce and running it (Simple)* recipe. Select a subset of data from the Amazon dataset if you are running this with few computers. You can find the smaller dataset with sample directory.
4. This sample uses the data created from earlier recipes. If you have not already run it, please run it first.

## How to do it...

1. Upload the amazon dataset to the HDFS filesystem using the following commands, if not already done so:

```
> bin/hadoop dfs -mkdir /data/

> bin/hadoop dfs -mkdir /data/amazon-dataset

> bin/hadoop dfs -put <SAMPLE_DIR>/amazon-meta.txt /data/amazon-
dataset/
```

2. Copy the `hadoop-microbook.jar` file from `SAMPLE_DIR` to `HADOOP_HOME`.

3. Run the following MapReduce job to create the dataset that provides items brought by customers. To do that run the following command from `HADOOP_HOME`:

   ```
   $ bin/hadoop jar hadoop-microbook.jar  microbook.join.
   Customer2ItemCalculater /data/amazon-dataset /data/join-output1
   ```

4. Copy the output of MapReduce job and output of the earlier recipe to the input directory. Note that the names of the files must be `mostFrequentBuyers.data` and `itemsByCustomer.data`.

   ```
   >bin/hadoop dfs -mkdir /data/join-input

   > bin/hadoop dfs -cp /data/join-output1/part-r-00000 /data/join-
   input/itemsByCustomer.data

   > bin/hadoop dfs -cp /data/frequency-output1/part-r-00000 /data/
   join-input/mostFrequentBuyers.data
   ```

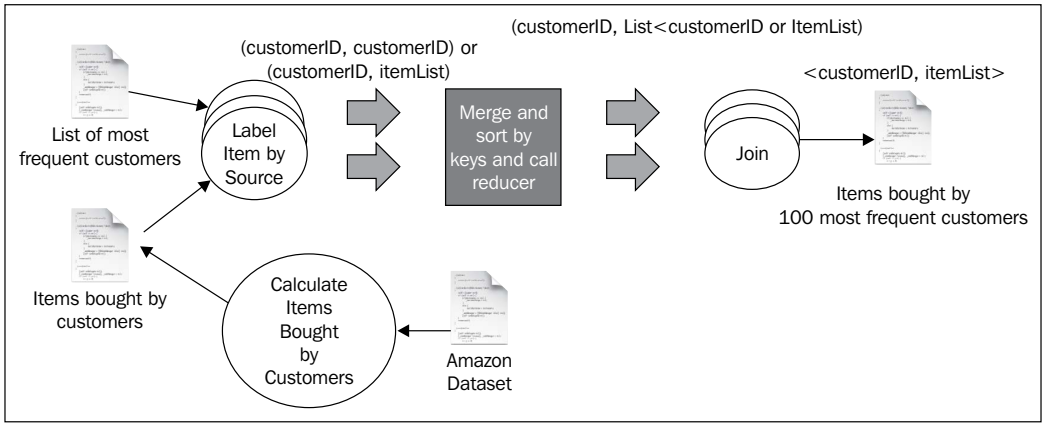
5. Run the second MapReduce job. To do that run the following command from `HADOOP_HOME`:

   ```
   $ bin/hadoop jar hadoop-microbook.jar  microbook.join.
   BuyerRecordJoinJob /data/join-input /data/join-output2
   ```

6. Your can find the results from the output directory, `/data/join-output2`.

## How it works...

You can find the mapper and reducer code at `src/microbook/join/BuyerRecordJoinJob.java`.




The first MapReduce job emits the items brought against the customer ID. The mapper emits customer ID as the key and item IDs as values. The reducer receives customer IDs as keys and item IDs emitted against that customer ID as values. It emits key and value without any changes.

We join the two datasets using the customer IDs. Here we put files for both sets into the same input directory. Hadoop will read the input files from the input folder and read records from the file. It invokes the mapper once per each record passing the record as input.

When the mapper receives an input, we find out which line belongs to which dataset by getting the filename using `InputSplit` available through the Hadoop context. For the list of frequent customers, we emit customer ID as both key and the value and for the other dataset, we emit customer ID as the key and list of items as the value.

```
public void map(Object key, Text value, Context context){
    String currentFile =  ((FileSplit)context
         .getInputSplit()).getPath().getName();
    Matcher matcher = parsingPattern
         .matcher(value.toString());
    if (matcher.find()) {
        String propName = matcher.group(1);
        String propValue = matcher.group(2);
        if(currentFile.contains("itemsByCustomer.data")){
            context.write(new Text(propName),
               new Text(propValue));
        }else
               if(currentFile.equals("mostFrequentBuyers.data")){
    context.write(new Text(propName),
                   new Text(propValue));
         }else{
             throw new IOException("Unexpected file "
                + currentFile);
         }
    }
}
```

Hadoop will sort the key-value pairs by the key and invokes the reducer once for each unique key passing the list of values as the second argument. The reducer inspects the list of values, and if the values also contain the customer ID, it emits customer ID as the key and list of items as the value.

```
public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException,
        InterruptedException {
    boolean isPresent = false;
    String itemList = null;
    for (Text val : values) {
        if(val.toString().equals(key.toString())){
                isPresent = true;
        }else{
```

```
                itemList = val.toString();
            }
        }
        if(isPresent && itemList != null){
            context.write(key, new Text(itemList));
        }
    }
```

## There's more...

Here the main idea is to send the information needed to be joined to the same reducer using the same key at the mapper and performing the joining logic at the reducer. The same idea can be used to join any kind of dataset.

# Set operations with MapReduce (Intermediate)

Set operations are a useful tool we use when we want to understand a dataset. This recipe will explain how to use MapReduce to perform a set operation on a large dataset. The following MapReduce job calculates the set difference between the customers who have bought the items that have an amazon sales rank less than 100 and most frequent customers which we calculated in the earlier recipe.

## Getting ready

1. This assumes that you have installed Hadoop and started it. Refer to the *Writing a word count application using Java (Simple)* and *Installing Hadoop in a distributed setup and running a word count application (Simple)* recipes for more information. We will use `HADOOP_HOME` to refer to the Hadoop installation directory.
2. This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the *Writing a word count application with MapReduce and running it (Simple)* recipe.
3. Download the sample code for the chapter and download the data files as described in the *Writing a word count application with MapReduce and running it (Simple)* recipe. Select a subset of data from the Amazon dataset if you are running this with few computers. You can find the smaller dataset with the sample directory.
4. This sample uses the data created from earlier recipes. If you have not already run it, please run it first.

## How to do it...

1. If you have not already done so, let us upload the amazon dataset to the HDFS filesystem using the following commands:

```
> bin/hadoop dfs -mkdir /data/

> bin/hadoop dfs -mkdir /data/amazon-dataset

> bin/hadoop dfs -put <SMAPLE_DIR>/amazon-meta.txt /data/amazon-dataset

> bin/Hadoop dfs –mkdir /data/set-input
```

2. Copy the output from earlier recipes to the output directory.

```
>bin/hadoop dfs -cp
    /data/frequency-output1/part-r-00000
    /data/set-input/mostFrequentBuyers.data
```

3. Copy the `hadoop-microbook.jar` file from `SAMPLE_DIR` to `HADOOP_HOME`.
4. Run the first MapReduce job. To do that run the following command from `HADOOP_HOME`:

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.set.
FindCustomersBroughtFirst100Items /data/amazon-dataset /data/set-
output1
```

5. Copy the output of the MapReduce job and output of the earlier recipe to the input directory.

```
> bin/hadoop dfs -cp /data/set-output1/part-r-00000  /data/set-
input/first100ItemBuyers.data
```

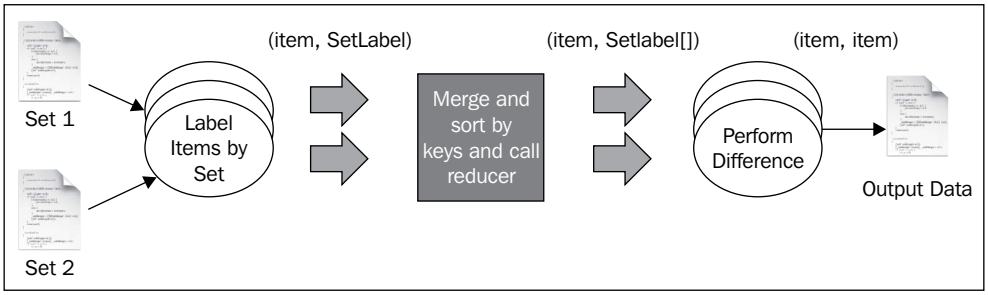
6. Run the second MapReduce job. To do that run the following command from `HADOOP_HOME`:

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.set.
BuyersSetDifference /data/set-input /data/set-output2
```

7. You can find the results from the output directory at `/data/set-output2`.

## How it works...

You can find the mapper and reducer code at `src/microbook/BuyersSetDifference.java`.




We define the set difference between the two sets S1 and S2, written as S1-S2, as the items that are in set S1 but not in set S2.

To perform set difference, we label each element at the mapper with the set it came from. Then send the search to a reducer, which emits an item only if it is in the first set, but not in the second set. The preceding figure shows the execution of the MapReduce job. Also the following code listing shows the map function and the reduce function.

Let us look at the execution in detail.

Here we put files for both sets into the same input directory. Hadoop will read the input files from the input folder and read records from each file. It invokes the mapper once per each record passing the record as input.

When the mapper receives an input, we finds out which line belongs to which set by getting the filename using `InputSplit` available through the Hadoop context. Then we emit elements in the set as the key and the set name (1 or 2) as the value.

```
public void map(Object key, Text value, Context context) {
    String currentFile =  ((FileSplit)context.getInputSplit()).
    getPath().getName();

    Matcher matcher =
    parsingPattern.matcher(value.toString());
    if (matcher.find()) {
        String propName = matcher.group(1);
        String propValue = matcher.group(2);
        if(currentFile.equals("first100ItemBuyers.data")){
            context.write(new Text(propName),
            new IntWritable(1));
}else{ if(currentFile.equals("mostFrequentBuyers.data")){
```

```
            int count = Integer.parseInt(propValue);
            if(count > 100){
                context.write(new Text(propName),
new IntWritable(2));
            }
        }else{
            throw new IOException("Unexpected file "
+ currentFile);
        }
    } else {
        System.out.println(currentFile
+ ":Unprocessed Line " + value);
    }
}
```

Hadoop will sort the key-value pairs by the key and invoke the reducer once for each unique key, passing the list of values as the second argument. The reducer inspects the list of values, which contain the name of sets the values comes from, and then emits the key only if the given value is in the first set, but not in the second.

```
public void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException,
        InterruptedException {
    boolean has1 = false;
    boolean has2 = false;
    System.out.print(key + "=");
    for (IntWritable val : values) {
        switch(val.get()){
            case 1:
                has1 = true;
                break;
            case 2:
                has2 = true;
                break;
        }
        System.out.println(val);
    }
    if(has1 && !has2){
        context.write(key, new IntWritable(1));
    }
}
```

## There's more...

We can use MapReduce to implement most set operations by labeling elements against the sets they came from using a similar method and changing the reducer logic to emit only relevant elements. For example, we can implement the set intersection by changing the reducer to emit only elements that have both sets as values.

# Cross correlation with MapReduce (Intermediate)

Cross correlation detects the number of times two things occur together. For example, in the Amazon dataset, if two buyers have bought the same item, we say that they are cross correlated. Through cross correlation, we count the number of times two customers have bought a same item.

## Getting ready

1. This assumes that you have installed Hadoop and started it. *Writing a word count application using Java (Simple)* and *Installing Hadoop in a distributed setup and running a word count application (Simple)* recipes for more information. We will use the `HADOOP_HOME` to refer to the Hadoop installation directory.
2. This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the *Writing a word count application with MapReduce and running it (Simple)* recipe.
3. Download the sample code for the chapter and download the data files as described in the *Writing a word count application with MapReduce and running it (Simple)* recipe. Select a subset of data from the Amazon dataset if you are running this with few computers. You can find the smaller dataset with the sample directory.

## How to do it...

1. Upload the Amazon dataset to the HDFS filesystem using the following commands from `HADOOP_HOME`, if you have not already done so:

```
> bin/hadoop dfs -mkdir /data/

> bin/hadoop dfs -mkdir /data/amazon-dataset

> bin/hadoop dfs -put <DATA_DIR>/amazon-meta.txt /data/amazon-dataset/

> bin/hadoop dfs -ls /data/amazon-dataset
```

2. Copy the `hadoop-microbook.jar` file from `SAMPLE_DIR` to `HADOOP_HOME`

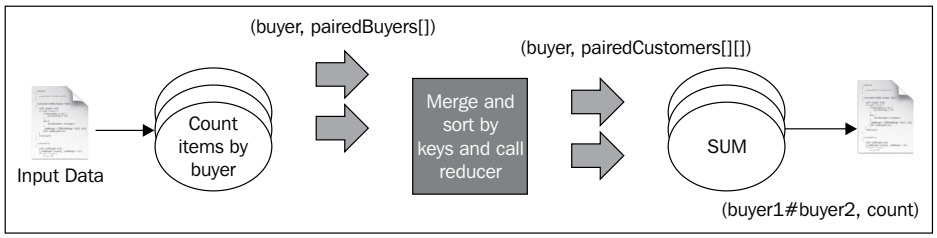
3. Run the MapReduce job to calculate the buying frequency using the following command from `HADOOP_HOME`:

   ```
   $ bin/hadoop jar hadoop-microbook.jar  microbook.crosscorrelation.
   CustomerCorrleationFinder /data/amazon-dataset /data/cor-output1
   ```

4. You can find the results from the output directory `/data/cor-output1`.

## How it works...

You can find the mapper and reducer code at `src/microbook/Crosscorrelation/CustomerCorrleationFinder.java`.




The preceding figure shows the execution of the MapReduce job. Also the following code listing shows the map function and the reduce function of the first job:

```
public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
    List<BuyerRecord> records =
        BuyerRecord.parseAItemLine(value.toString());
    List<String> customers = new ArrayList<String>();

    for(BuyerRecord record: records){
        customers.add(record.customerID);
    }

    for(int i =0;i< records.size();i++){
        StringBuffer buf = new StringBuffer();
        int index = 0;
        for(String customer:customers){
            if(index != i){
                buf.append(customer).append(",");
            }
        }
        context.write(new Text(customers.get(i)),
        new Text(buf.toString()));
    }
}
```

As shown by the figure, Hadoop will read the input file from the input folder and read records using the custom formatter we introduced in the *Write a formatter (Intermediate)* recipe. It invokes the mapper once per each record passing the record as input.

The map function reads the record of a date item and extracts the sales data from the data item. Buyers in the sales data have a cross correlation with each other because they have bought the same item.

Most trivial implementations of cross correlation will emit each pair of buyers that have a cross correlation from the map, and count the number of occurrences at the reduce function after the MapReduce step has grouped the same buyers together.

However, this would generate more than the square of the number of different buyers, and for a large dataset, this can be a very large number. Therefore, we will use a more optimized version, which limits the number of keys.

Instead the mapper emits the buyer as the key and emits all other buyers, paired with that mapper, as keys.

```
public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException,
        InterruptedException {
    Set<String> customerSet = new HashSet<String>();
    for(Text text: values){
        String[] split = text.toString().split(",");
        for(String token:split){
            customerSet.add(token);
        }
    }

    StringBuffer buf = new StringBuffer();
    for(String customer: customerSet){
        if(customer.compareTo(key.toString()) < 0){
            buf.append(customer).append(",");
        }
    }

    buf.append("|").append(Integer.MAX_VALUE)
        .append("|").append(SimilarItemsFinder.Color.White);
    context.write(new Text(key), new Text(buf.toString()));
}
```

MapReduce will sort the key-value pairs by the key and invoke the reducer once for each unique key passing the list of values emitted against that key as the input.

The reducer, then, calculates the pairs and counts how many times each pair has occurred. Given two buyers B1 and B2, we can emit B1, B2 or B2, B1 as pairs, thus generating duplicate data. We avoid that by only emitting a pair when B1 is lexicographically less than B2.

## There's more...

Cross correlation is one of the hard problems for MapReduce as it generates large amount of pairs. It generally works with only a smaller-size dataset.

# Simple search with MapReduce (Intermediate)

Text search is one of the first use cases for MapReduce, and according to Google, they built MapReduce as the programming model for text processing related to their search platform.

Search is generally implemented with an inverted index. An inverted index is a mapping of words to the data items that includes that word. Given a search query, we find all documents that have the words in the query. One of the complexities of web search is that there are too many results and we only need to show important queries. However, ranking the documents based on their importance is out of the scope of this discussion.

This recipe explains how to build a simple inverted index based search using MapReduce.

## Getting ready

1. This assumes that you have installed Hadoop and started it. *Writing a word count application using Java (Simple)* and *Installing Hadoop in a distributed setup and running a word count application (Simple)* recipes for more information. We will use `HADOOP_HOME` to refer to the Hadoop installation directory.
2. This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the *Writing a word count application with MapReduce and running it (Simple)* recipe.
3. Download the sample code for the chapter and download the data files as described in the *Writing a word count application with MapReduce and running it (Simple)* recipe. Select a subset of data from the Amazon dataset if you are running this with few computers. You can find the smaller dataset with sample directory.

## How to do it...

1. If you have not already done so, let us upload the Amazon dataset to the HDFS filesystem using the following commands from `HADOOP_HOME`:

   ```
   > bin/hadoop dfs -mkdir /data/
   > bin/hadoop dfs -mkdir /data/amazon-dataset
   > bin/hadoop dfs -put <DATA_DIR>/amazon-meta.txt /data/amazon-dataset/
   ```

2. Copy the `hadoop-microbook.jar` file from `SAMPLE_DIR` to `HADOOP_HOME`.
3. Run the MapReduce job to calculate the buying frequency. To do that run the following command from `HADOOP_HOME`:

   ```
   $ bin/hadoop jar hadoop-microbook.jar  microbook.search.
   TitleInvertedIndexGenerator /data/amazon-dataset /data/search-
   output
   ```

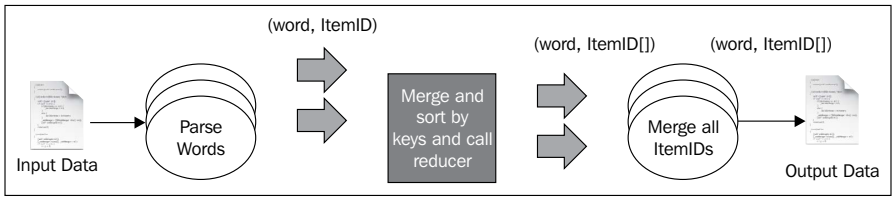
4. Your can find the results from the output directory, `/data/search-output`.
5. Run the following commands to download the results file from the server and to search for the word `Cycling` using the index built by the MapReduce job. It should print the items that have the word `Cycling` in the title.

   ```
   $ bin/hadoop dfs -get /data/search-output/part-r-00000
   invetedIndex.data


   $ java –cp hadoop-microbook.jar microbook.search.
   IndexBasedTitleSearch invetedIndex.data Cycling
   ```

## How it works...

You can find the mapper and reducer code at `src/microbook/search/TitleInvertedIndexGenerator.java`.




The preceding figure shows the execution of two MapReduce job. Also the following code listing shows the map function and the reduce function of the first job.

As shown by the figure, Hadoop will read the input file from the input folder and read records using the custom formatter we introduced in the *Write a formatter (Intermediate)* recipe. It invokes the mapper once per each record passing the record as input.

The map function reads the title of the item from the record, tokenizes it, and emits each word in the title as the key and the item ID as the value.

```
public void map(Object key, Text value, Context context) {
    List<BuyerRecord> records =
BuyerRecord.parseAItemLine(value.toString());
    for (BuyerRecord record : records) {
        for(ItemData item: record.itemsBrought){
            StringTokenizer itr =
new StringTokenizer(item.title);
            while (itr.hasMoreTokens()) {
                String token =
itr.nextToken().replaceAll("[^A-z0-9]", "");
                if(token.length() > 0){
                    context.write(new Text(token),
                    new Text(
                    pad(String.valueOf(item.salesrank))
                    + "#" +item.itemID));
                }
            }
        }
    }
}
```

MapReduce will sort the key-value pairs by the key and invoke the reducer once for each unique key, passing the list of values emitted against that key as the input.

Each reducer will receive a word as the key and list of item IDs as the values, and it will emit them as it is. The output is an inverted index.

```
public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException,
        InterruptedException {
    TreeSet<String> set = new TreeSet<String>();
    for (Text valtemp : values) {
        set.add(valtemp.toString());
    }

    StringBuffer buf = new StringBuffer();
    for (String val : set) {
        buf.append(val).append(",");
    }
    context.write(key, new Text(buf.toString()));
}
```

The following listing gives the code for the search program. The search program loads the inverted index to the memory, and when we search for a word, it will find the item IDs against that word, and list them.

```
String line = br.readLine();
while (line != null) {
    Matcher matcher = parsingPattern.matcher(line);
    if (matcher.find()) {
        String key = matcher.group(1);
        String value = matcher.group(2);

        String[] tokens = value.split(",");
        invertedIndex.put(key, tokens);
        line = br.readLine();
    }
}

String searchQuery = "Cycling";
String[] tokens = invertedIndex.get(searchQuery);
if (tokens != null) {
    for (String token : tokens) {
        System.out.println(Arrays.toString(token.split("#")));
        System.out.println(token.split("#")[1]);
    }
}
```

## There's more...

We use indexes to quickly find data from a large dataset. The same pattern is very useful for building indexes to support fast searches.

# Simple graph operations with MapReduce (Advanced)

Graphs are another type of data that we often encounter. One of the primary use cases for graphs is social networking; people want to search graphs for interesting patterns. This recipe explains how to perform a simple graph operation, graph traversal, using MapReduce.

This recipe uses the results from the *Cross correlation with MapReduce (Intermediate)* recipe. Each buyer is a node, and if two buyers have bought the same item, there is an edge between these nodes.

A sample input is shown as follows:

```
AR1T36GLLAFFX  A26TSW6AI59ZCV,A39LRCAB9G8F21,ABT9YLRGT4ISP|Gray
```

Here the first token is node, and the comma-separated values are lists of nodes to which the first node has an edge. The last value is the color of the node. This is a construct we use for the graph traversal algorithm.

Given a buyer (a node), this recipe walks though the graph and calculates the distance from the given node to all other nodes.

This recipe and the next recipe belong to a class called iterative MapReduce where we cannot solve the problem by processing data once. Iterative MapReduce processes the data many times using a MapReduce job until we have calculated the distance from the given node to all other nodes.

## Getting ready

1. This assumes that you have installed Hadoop and started it. *Writing a word count application using Java (Simple)* and *Installing Hadoop in a distributed setup and running a word count application (Simple)* recipes for more information. We will use `HADOOP_HOME` to refer to the Hadoop a word count application (Simple) installation directory.
2. This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the *Writing a word count application with MapReduce and running it (Simple)* recipe.
3. Download the sample code for the chapter and download the data files as described in the *Writing a word count application with MapReduce and running it (Simple)* recipe. Select a subset of data from the Amazon dataset if you are running this with few computers. You can find the smaller dataset with sample directory.

## How to do it...

1. Change directory to `HADOOP_HOME` and copy the `hadoop-microbook.jar` file from `SAMPLE_DIR` to `HADOOP_HOME`.
2. Upload the Amazon dataset to the HDFS filesystem using the following commands from `HADOOP_HOME`, if you have not already done so:

```
> bin/hadoop dfs -mkdir /data/

> bin/hadoop dfs -mkdir /data/amazon-dataset

> bin/hadoop dfs -put <DATA_DIR>/amazon-meta.txt /data/amazon-
dataset/

> bin/hadoop dfs -ls /data/amazon-dataset
```

3. Run the following command to generate the graph:

```
> bin/hadoop jar hadoop-microbook.jar  microbook.graph.
GraphGenerator /data/amazon-dataset /data/graph-output1
```

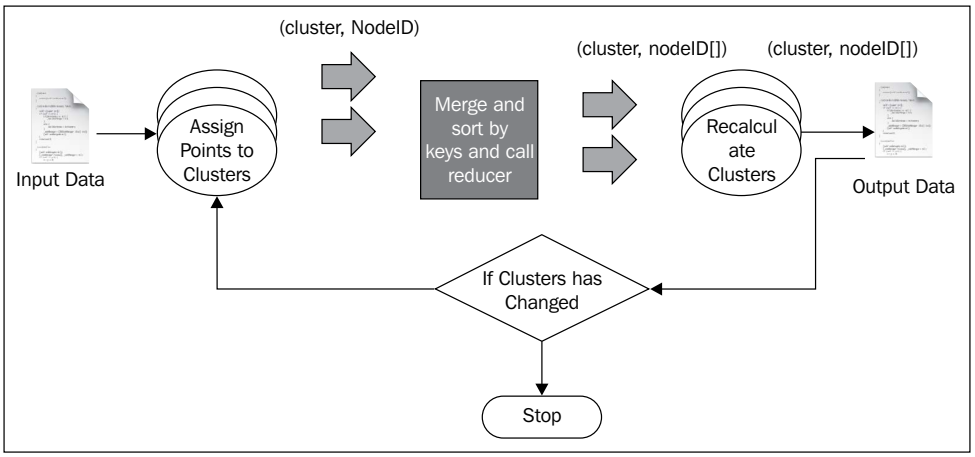
4. Run the following command to run MapReduce job to calculate the graph distance:

```
$ bin/hadoop jar hadoop-microbook.jar  microbook.graph.
SimilarItemsFinder /data/graph-output1 /data/graph-output2
```

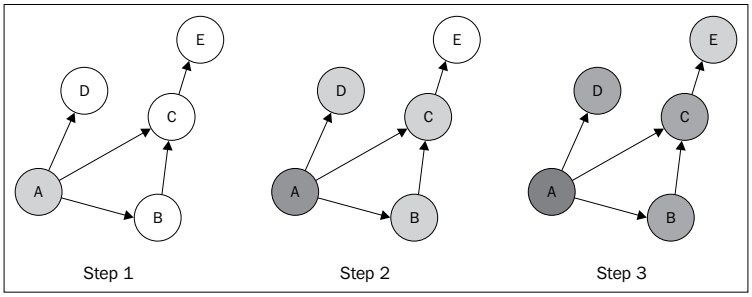
5. You can find the results at `/data/graph-output2`. It will have results for all iterations, and you should look at the last iteration.

## How it works...

You can find the mapper and reducer code at `src/microbook/SimilarItemsFinder.java`.




The preceding figure shows the execution of two MapReduce job and the driver code. The driver code repeats the map reduce job until the graph traversal is complete.




The algorithm operates by coloring the graph nodes. Each node is colored white at the start, except for the node where we start the traversal, which is marked gray. When we generate the graph, the code will mark that node as gray. If you need to change the starting node, you can do so by editing the graph.

As shown in the figure, at each step, the MapReduce job processes the nodes that are marked gray and calculates the distance to the nodes that are connected to the gray nodes via an edge, and updates the distance. Furthermore, the algorithm will also mark those adjacent nodes as gray if their current color is white. Finally, after visiting and marking all its children gray, we set the node color as black. At the next step, we will visit those nodes marked with the color gray. It continues this until we have visited all the nodes.

Also the following code listing shows the map function and the reduce function of the MapReduce job.

```
public void map(Object key, Text value, Context context){
    Matcher matcher = parsingPattern.matcher(value.toString());
    if (matcher.find()) {
        String id = matcher.group(1);
        String val = matcher.group(2);

        GNode node = new GNode(id, val);
        if(node.color == Color.Gray){
            node.color = Color.Black;
            context.write(new Text(id),
            new Text(node.toString()));
            for(String e: node.edges){
                GNode nNode = new GNode(e, (String[])null);
                nNode.minDistance = node.minDistance+1;
                nNode.color = Color.Red;
                context.write(new Text(e),
                new Text(nNode.toString()));
            }
        }else{
            context.write(new Text(id), new Text(val));
        }
    } else {
        System.out.println("Unprocessed Line " + value);
    }
}
```

As shown by the figure, Hadoop will read the input file from the input folder and read records using the custom formatter we introduced in the *Write a formatter (Intermediate)* recipe. It invokes the mapper once per each record passing the record as input.

Each record includes the node. If the node is not colored gray, the mapper will emit the node without any change using the node ID as the key.

If the node is colored gray, the mapper explores all the edges connected to the node, updates the distance to be the current node distance +1. Then it emits the node ID as the key and distance as the value to the reducer.

```
public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException,
        InterruptedException {
    GNode originalNode =  null;
    boolean hasRedNodes = false;
    int minDistance = Integer.MAX_VALUE;
    for(Text val: values){
        GNode node = new GNode(key.toString(),val.toString());
        if(node.color == Color.Black ||
node.color == Color.White){
            originalNode = node;
        }else if(node.color == Color.Red){
            hasRedNodes = true;
        }
        if(minDistance > node.minDistance){
            minDistance = node.minDistance;
        }
    }
    if(originalNode != null){
        originalNode.minDistance = minDistance;
        if(originalNode.color == Color.White && hasRedNodes){
            originalNode.color = Color.Gray;
        }
        context.write(key, new Text(originalNode.toString()));
    }
}
```

MapReduce will sort the key-value pairs by the key and invoke the reducer once for each unique key passing the list of values emitted against that key as the input.

Each reducer will receive a key-value pairs information about nodes and distances as calculated by the mapper when it encounters the node. The reducer updates the distance in the node if the distance updates are less than the current distance of the node. Then, it emits the node ID as the key and node information as the value.

The driver repeats the process until all the nodes are marked black and the distance is updated. When starting, we will have only one node marked as gray and all others as white. At each execution, the MapReduce job will mark the nodes connected to the first node as gray and update the distances. It will mark the visited node as black.

We continue this until all nodes are marked as black and have updated distances.

## There's more...

Users can use the iterative MapReduce-based solution we discussed in this recipe with many graph algorithms such as graph search.

# Kmeans with MapReduce (Advanced)

When we try to find or calculate interesting information from large datasets, often we need to calculate more complicated algorithms than the algorithms we discussed so far. There are many such algorithms available (for example clustering, collaborative filtering, and data mining algorithms). This recipe will implement one such algorithm called Kmeans that belongs to clustering algorithms.

Let us assume that the Amazon dataset includes customer locations. Since that information is not available, we will create a dataset by picking random values from IP addresses to the latitude and longitude dataset available from `http://www.infochimps.com/datasets/united-states-ip-address-to-geolocation-data`.

If we can group the customers by geo location, we can provide more specialized and localized services. In this recipe, we will implement the Kmeans clustering algorithm using MapReduce and use that identify the clusters based on geo location of customers.

A clustering algorithm groups a dataset into several groups called clusters such that data points within the same cluster are much closer to each other than data points between two different clusters. In this case, we will represent the cluster using the center of it's data points.

## Getting ready

1. This assumes that you have installed Hadoop and started it. *Writing a word count application using Java (Simple)* and *Installing Hadoop in a distributed setup and running a word count application (Simple)* recipes for more information. We will use `HADOOP_HOME` to refer to the Hadoop installation directory.
2. This recipe assumes you are aware of how Hadoop processing works. If you have not already done so, you should follow the *Writing a word count application with MapReduce and running it (Simple)* recipe.
3. Download the sample code for the chapter and download the data files from `http://www.infochimps.com/datasets/united-states-ip-address-to-geolocation-data`.

## How to do it...

1. Unzip the geo-location dataset to a directory of your choice. We will call this `GEO_DATA_DIR`.
2. Change the directory to `HADOOP_HOME` and copy the `hadoop-microbook.jar` file from `SAMPLE_DIR` to `HADOOP_HOME`.
3. Generate the sample dataset and initial clusters by running the following command. It will generate a file called `customer-geo.data`.

   ```
   > java –cp hadoop-microbook.jar microbook.kmean.GenerateGeoDataset
   GEO_DATA_DIR/ip_blocks_us_geo.tsv customer-geo.data
   ```

4. Upload the dataset to the HDFS filesystem.

   ```
   > bin/hadoop dfs -mkdir /data/

   > bin/hadoop dfs -mkdir /data/kmeans/

   > bin/hadoop dfs -mkdir /data/kmeans-input/

   > bin/hadoop dfs -put HADOOP_HOME/customer-geo.data /data/kmeans-
   input/
   ```

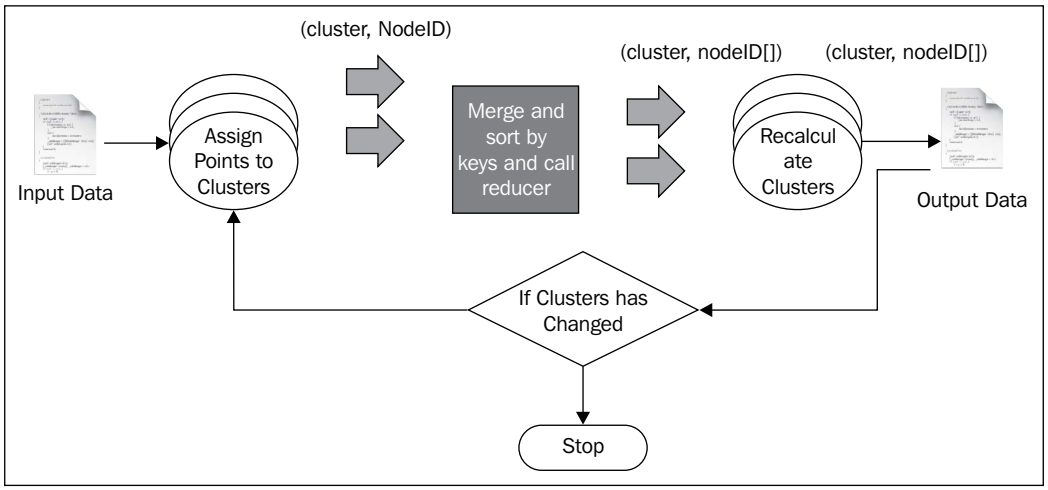
5. Run the MapReduce job to calculate the clusters. To do that run the following command from `HADOOP_HOME`. Here, `5` stands for the number of iterations and `10` stands for number of clusters.

   ```
   $ bin/hadoop jar hadoop-microbook.jar  microbook.kmean.
   KmeanCluster /data/kmeans-input/ /data/kmeans-output 5 10
   ```

6. The execution will finish and print the final clusters to the console, and you can also find the results from the output directory, `/data/kmeans-output`.

## How it works...

You can find the mapper and reducer code from `src/microbook/KmeanCluster.java`. This class includes the map function, reduce function, and the driver program.




When started, the driver program generates 10 random clusters, and writes them to a file in the HDFS filesystem. Then, it invokes the MapReduce job once for each iteration.

The preceding figure shows the execution of two MapReduce jobs. This recipe belongs to the iterative MapReduce style where we iteratively run the MapReduce program until the results converge.

When the MapReduce job is invoked, Hadoop invokes the setup method of mapper class, where the mapper loads the current clusters into memory by reading them from the HDFS filesystem.

As shown by the figure, Hadoop will read the input file from the input folder and read records using the custom formatter, that we introduced in the *Write a formatter (Intermediate)* recipe. It invokes the mapper once per each record passing the record as input.

When the mapper is invoked, it parses and extracts the location from the input, finds the cluster that is nearest to the location, and emits that cluster ID as the key and the location as the value. The following code listing shows the map function:

```
public void map(Object key, Text value, Context context) {
    Matcher matcher = parsingPattern.matcher(value.toString());
    if (matcher.find()) {
        String propName = matcher.group(1);
        String propValue = matcher.group(2);
        String[] tokens = propValue.split(",");
        double lat = Double.parseDouble(tokens[0]);
        double lon = Double.parseDouble(tokens[1]);
```

```
            int minCentroidIndex = -1;
            double minDistance = Double.MAX_VALUE;
            int index = 0;
            for(Double[] point: centriodList){
                double distance =
    Math.sqrt(Math.pow(point[0] -lat, 2)
    + Math.pow(point[1] -lon, 2));
                if(distance < minDistance){
                    minDistance = distance;
                    minCentroidIndex = index;
                }
                index++;
            }

            Double[] centriod = centriodList.get(minCentroidIndex);
            String centriodAsStr = centriod[0] + "," + centriod[1];
            String point = lat +"," + lon;
            context.write(new Text(centriodAsStr), new Text(point));
        }
    }
```

MapReduce will sort the key-value pairs by the key and invoke the reducer once for each unique key passing the list of values emitted against that key as the input.

The reducer receives a cluster ID as the key and the list of all locations that are emitted against that cluster ID. Using these, the reducer recalculates the cluster as the mean of all the locations in that cluster and updates the HDFS location with the cluster information. The following code listing shows the reducer function:

```
public void reduce(Text key, Iterable<Text> values,
Context context)
{
    context.write(key, key);
    //recalcualte clusters
    double totLat = 0;
    double totLon = 0;
    int count = 0;

    for(Text text: values){
        String[] tokens = text.toString().split(",");
        double lat = Double.parseDouble(tokens[0]);
        double lon = Double.parseDouble(tokens[1]);
        totLat = totLat + lat;
        totLon = totLon + lon;
```

```
            count++;
        }

        String centroid = (totLat/count) + "," + (totLon/count);

        //print them out
        for(Text token: values){
            context.write(new Text(token), new Text(centroid));

        }
        FileSystem fs =FileSystem.get(context.getConfiguration());

        BufferedWriter bw = new BufferedWriter(
    new OutputStreamWriter(fs.create(new Path("/data/kmeans/clusters.
    data"), true)));
        bw.write(centroid);bw.write("\n");
        bw.close();

    }
```

The driver program continues above per each iteration until input cluster and output clusters for a MapReduce job are the same.

The algorithm starts with random cluster points. At each step, it assigns locations to cluster points, and at the reduced phase it adjusts each cluster point to be the mean of the locations assigned to each cluster. At each iteration, the clusters move until the clusters are the best clusters for the dataset. We stop when clusters stop changing in the iteration.

## There's more...

One limitation of the Kmeans algorithm is that we need to know the number of clusters in the dataset. There are many other clustering algorithms. You can find more information about these algorithms from the *Chapter 7* of the freely available book *Mining of Massive Datasets*, *Anand Rajaraman* and *Jeffrey D. Ullman*, *Cambridge University Press*, *2011*.

Thank you for buying

# Instant MapReduce Patterns – Hadoop Essentials How-to

## 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.

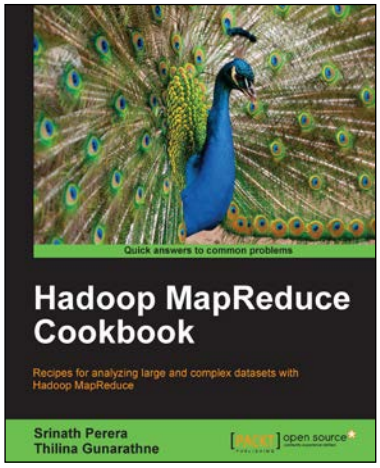
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

## 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.

[PACKT] PUBLISHING



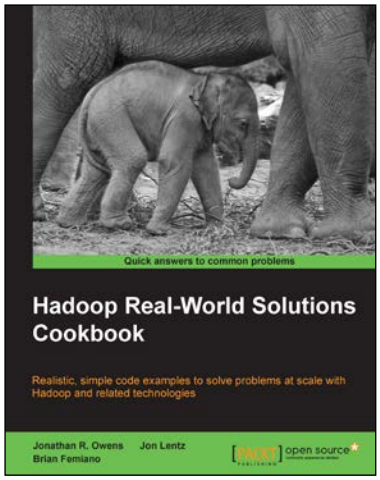

## Hadoop MapReduce Cookbook

ISBN: 978-1-84951-728-7 Paperback: 300 pages

Recipes for analyzing large and complex datasets with Hadoop MapReduce

1. Learn to process large and complex data sets, starting simply, then diving in deep
2. Solve complex big data problems such as classifications, finding relationships, online marketing and recommendations
3. More than 50 Hadoop MapReduce recipes, presented in a simple and straightforward manner, with step-by-step instructions and real world examples

---



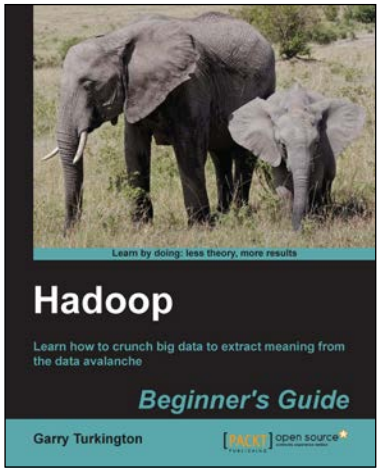

## Hadoop Real-World Solutions Cookbook

ISBN: 9781849519120 Paperback: 316 pages

Realistic, simple code examples to solve problems at scale with Hadoop and related tecnologies

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

[PACKT] PUBLISHING



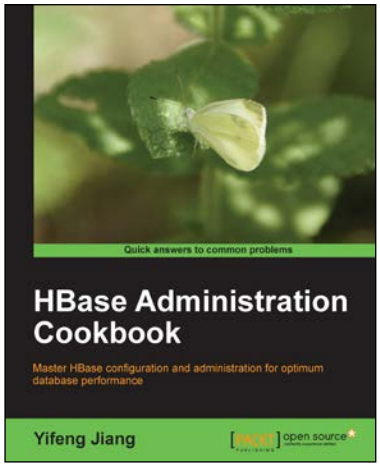

## Hadoop Beginner's Guide

ISBN: 978-1-84951-730-0 Paperback: 398 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

---




## 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