

Hadoop: Data Processing and Modelling

Unlock the power of your data with Hadoop 2.X ecosystem and its data warehousing techniques across large data sets





Packt>

LEARNING PATH

Hadoop: Data Processing and Modelling

Unlock the power of your data with Hadoop 2.X ecosystem and its data warehousing techniques across large data sets

A course in three modules

Packt>
BIRMINGHAM - MUMBAI

Hadoop: Data Processing and Modelling

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: August 2016

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

ISBN: 978-1-78712-516-2

www.packtpub.com

Credits

Authors

Garry Turkington

Tanmay Deshpande

Sandeep Karanth

Reviewers

David Gruzman

Muthusamy Manigandan

Vidyasagar N V

Shashwat Shriparv

Shiva Achari

Pavan Kumar Polineni

Uchit Vyas

Yohan Wadia

Content Development Editor

Rashmi Suvarna

Graphics

Kirk D'Phena

Production Coordinator

Shantanu N. Zagade

Preface

A number of organizations are focusing on big data processing, particularly with Hadoop. This course will help you understand how Hadoop, as an ecosystem, helps us store, process, and analyze data. Hadoop is synonymous with Big Data processing. Its simple programming model, "code once and deploy at any scale" paradigm, and an ever-growing ecosystem make Hadoop an inclusive platform for programmers with different levels of expertise and breadth of knowledge. A team of machines are interconnected via a very fast network and provide better scaling and elasticity, but that is not enough.

These clusters have to be programmed. A greater number of machines, just like a team of human beings, require more coordination and synchronization. The higher the number of machines, the greater the possibility of failures in the cluster. How do we handle synchronization and fault tolerance in a simple way easing the burden on the programmer? The answer is systems such as Hadoop. Today, it is the number-one sought after job skill in the data sciences space. To handle and analyze Big Data, Hadoop has become the go-to tool. Hadoop 2.x is spreading its wings to cover a variety of application paradigms and solve a wider range of data problems. It is rapidly becoming a general-purpose cluster platform for all data processing needs, and will soon become a mandatory skill for every engineer across verticals. Explore the power of Hadoop ecosystem to be able to tackle real-world scenarios and build.

This course covers optimizations and advanced features of MapReduce, Pig, and Hive.

Along with Hadoop 2.x and illustrates how it can be used to extend the capabilities of Hadoop.

When you finish this course, you will be able to tackle the real-world scenarios and become a big data expert using the tools and the knowledge based on the various step-by-step tutorials and recipes.

What this learning path covers

Hadoop beginners Guide

This module is here to help you make sense of Hadoop and use it to solve your big data problems. It's a really exciting time to work with data processing technologies such as Hadoop. The ability to apply complex analytics to large data sets—once the monopoly of large corporations and government agencies—is now possible through free open source software (OSS). This module removes the mystery from Hadoop, presenting Hadoop and related technologies with a focus on building working systems and getting the job done, using cloud services to do so when it makes sense. From basic concepts and initial setup through developing applications and keeping the system running as the data grows, the module gives the understanding needed to effectively use Hadoop to solve real world problems. Starting with the basics of installing and configuring Hadoop, the module explains how to develop applications, maintain the system, and how to use additional products to integrate with other systems. In addition to examples on Hadoop clusters on Ubuntu uses of cloud services such as Amazon, EC2 and Elastic MapReduce are covered.

Hadoop Real World Solutions Cookbook, 2nd edition

Big Data is the need the day. Many organizations are producing huge amounts of data every day. With the advancement of Hadoop-like tools, it has become easier for everyone to solve Big Data problems with great efficiency and at a very low cost. When you are handling such a massive amount of data, even a small mistake can cost you dearly in terms of performance and storage. It's very important to learn the best practices of handling such tools before you start building an enterprise Big Data Warehouse, which will be greatly advantageous in making your project successful.

This module gives readers insights into learning and mastering big data via recipes. The module not only clarifies most big data tools in the market but also provides best practices for using them. The module provides recipes that are based on the latest versions of Apache Hadoop 2.X, YARN, Hive, Pig, Sqoop, Flume, Apache Spark, Mahout and many more such ecosystem tools. This real-world-solution cookbook is packed with handy recipes you can apply to your own everyday issues. Each chapter provides in-depth recipes that can be referenced easily. This module provides detailed practices on the latest technologies such as YARN and Apache Spark. Readers will be able to consider themselves as big data experts on completion of this module.

Mastering Hadoop

This era of Big Data has similar changes in businesses as well. Almost everything in a business is logged. Every action taken by a user on the page of an e-commerce page is recorded to improve quality of service and every item bought by the user are recorded to cross-sell or up-sell other items. Businesses want to understand the DNA of their customers and try to infer it by pinching out every possible data they can get about these customers. Businesses are not worried about the format of the data. They are ready to accept speech, images, natural language text, or structured data. These data points are used to drive business decisions and personalize experiences for the user. The more data, the higher the degree of personalization and better the experience for the user.

Hadoop is synonymous with Big Data processing. Its simple programming model, "code once and deploy at any scale" paradigm, and an ever-growing ecosystem makes Hadoop an all-encompassing platform for programmers with different levels of expertise. This module explores the industry guidelines to optimize MapReduce jobs and higher-level abstractions such as Pig and Hive in Hadoop 2.0. Then, it dives deep into Hadoop 2.0 specific features such as YARN and HDFS Federation. This module is a step-by-step guide that focuses on advanced Hadoop concepts and aims to take your Hadoop knowledge and skill set to the next level. The data processing flow dictates the order of the concepts in each chapter, and each chapter is illustrated with code fragments or schematic diagrams.

This module is a guide focusing on advanced concepts and features in Hadoop.

Foundations of every concept are explained with code fragments or schematic illustrations. The data processing flow dictates the order of the concepts in each chapter

What you need for this learning path

In the simplest case, a single Linux-based machine will give you a platform to explore almost all the exercises in this course. We assume you have a recent distribution of Ubuntu, but as long as you have command-line Linux familiarity any modern distribution will suffice. Some of the examples in later chapters really need multiple machines to see things working, so you will require access to at least four such hosts. Virtual machines are completely acceptable; they're not ideal for production but are fine for learning and exploration. Since we also explore Amazon Web Services in this course, you can run all the examples on EC2 instances, and we will look at some other more Hadoop-specific uses of AWS throughout the modules. AWS services are usable by anyone, but you will need a credit card to sign up!

To get started with this hands-on recipe-driven module, you should have a laptop/desktop with any OS, such as Windows, Linux, or Mac. It's good to have an IDE, such as Eclipse or IntelliJ, and of course, you need a lot of enthusiasm to learn.

The following software suites are required to try out the examples in the module:

- Java Development Kit (JDK 1.7 or later): This is free software from Oracle that provides a JRE (Java Runtime Environment) and additional tools for developers. It can be downloaded from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.
- The IDE for editing Java code: IntelliJ IDEA is the IDE that has been used to develop the examples. Any other IDE of your choice can also be used. The community edition of the IntelliJ IDE can be downloaded from `https://www.jetbrains.com/idea/download/`.
- Maven: Maven is a build tool that has been used to build the samples in the course. Maven can be used to automatically pull-build dependencies and specify configurations via XML files. The code samples in the chapters can be built into a JAR using two simple Maven commands:

  ```
  mvn compile
  mvn assembly:single
  ```

These commands compile the code into a JAR file. These commands create a consolidated JAR with the program along with all its dependencies. It is important to change the mainClass references in the pom.xml to the driver class name when building the consolidated JAR file.

Hadoop-related consolidated JAR files can be run using the command:

```
hadoop jar <jar file> args
```

This command directly picks the driver program from the mainClass that was specified in the pom.xml. Maven can be downloaded and installed from `http://maven.apache.org/download.cgi`. The Maven XML template file used to build the samples in this course is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>MasteringHadoop</groupId>
  <artifactId>MasteringHadoop</artifactId>
  <version>1.0-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
```

```
        <version>3.0</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      <plugin>
        <version>3.1</version>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <archive>
            <manifest>
              <mainClass>MasteringHadoop.MasteringHadoopTest</
mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
<plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <archive>
            <manifest>
              <mainClass>MasteringHadoop.MasteringHadoopTest</
mainClass>
            </manifest>
          </archive>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
    </plugins>
    <pluginManagement>
      <plugins>
        <!--This plugin's configuration is used to store Eclipse
m2e settings
                only. It has no influence on the Maven build
itself. -->
        <plugin>
          <groupId>org.eclipse.m2e</groupId>
          <artifactId>lifecycle-mapping</artifactId>
          <version>1.0.0</version>
```

```
            <configuration>
              <lifecycleMappingMetadata>
                <pluginExecutions>
                  <pluginExecution>
                    <pluginExecutionFilter>
                      <groupId>org.apache.maven.plugins</groupId>
                      <artifactId>maven-dependency-plugin</
artifactId>
                      <versionRange>[2.1,)</versionRange>
                      <goals>
                        <goal>copy-dependencies</goal>
                      </goals>
                    </pluginExecutionFilter>
                    <action>
                      <ignore />
                    </action>
                  </pluginExecution>
                </pluginExecutions>
              </lifecycleMappingMetadata>
            </configuration>
</plugin>
      </plugins>
    </pluginManagement>
  </build>
  <dependencies>
    <!-- Specify dependencies in this section -->
  </dependencies>
</project>
```

- Hadoop 2.2.0 : Apache Hadoop is required to try out the examples in general. Appendix , Hadoop for Microsoft Windows, has the details on Hadoop's single-node installation on a Microsoft Windows machine. The steps are similar and easier for other operating systems such as Linux or Mac, and they can be found at http://hadoop.apache.org/docs/r2.2.0/hadoop-project-dist/hadoop-common/SingleNodeSetup.html

Who this learning path is for

We assume you are reading this course because you want to know more about Hadoop at a hands-on level; the key audience is those with software development experience but no prior exposure to Hadoop or similar big data technologies.

For developers who want to know how to write MapReduce applications, we assume you are comfortable writing Java programs and are familiar with the Unix command-line interface. We

will also show you a few programs in Ruby, but these are usually only to demonstrate language independence, and you don't need to be a Ruby expert.

For architects and system administrators, the course also provides significant value in explaining how Hadoop works, its place in the broader architecture, and how it can be managed operationally. Some of the more involved techniques in Chapter 4, Developing MapReduce Programs, and Chapter 5, Advanced MapReduce Techniques, are probably of less direct interest to this audience.

This course is for Java developers, who know scripting, wanting a career shift to Hadoop - Big Data segment of the IT industry. So if you are a novice in Hadoop or an expert this course will make you reach the most advanced level in Hadoop 2.X.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the course's title in the subject of your message.

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

Customer support

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

Downloading the example code

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

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.

6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at `https://github.com/PacktPublishing/Data-Science-with-Hadoop/tree/master`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

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 modules—maybe a mistake in the text or the code—we would be grateful if you could 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 **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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

If you have a problem with any aspect of this course, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Module 1: Hadoop Beginner's Guide

Chapter 1: What It's All About 3
- Big data processing 4
- Cloud computing with Amazon Web Services 16
- Summary 20

Chapter 2: Getting Hadoop Up and Running 21
- Hadoop on a local Ubuntu host 21
- Using Elastic MapReduce 41
- Comparison of local versus EMR Hadoop 51
- Summary 52

Chapter 3: Understanding MapReduce 53
- Key/value pairs 53
- The Hadoop Java API for MapReduce 56
- Writing MapReduce programs 60
- Walking through a run of WordCount 71
- Hadoop-specific data types 79
- Input/output 84
- Summary 88

Chapter 4: Developing MapReduce Programs 89
- Using languages other than Java with Hadoop 90
- Analyzing a large dataset 94
- Counters, status, and other output 113
- Summary 122

Chapter 5: Advanced MapReduce Techniques 123
Simple, advanced, and in-between 123
Joins 124
Graph algorithms 133
Using language-independent data structures 147
Summary 162

Chapter 6: When Things Break 163
Failure 163
Summary 198

Chapter 7: Keeping Things Running 201
A note on EMR 202
Hadoop configuration properties 202
Setting up a cluster 205
Cluster access control 216
Managing the NameNode 220
Managing HDFS 226
MapReduce management 227
Scaling 231
Summary 232

Chapter 8: A Relational View on Data with Hive 233
Overview of Hive 233
Setting up Hive 234
Using Hive 237
Hive on Amazon Web Services 266
Summary 274

Chapter 9: Working with Relational Databases 275
Common data paths 275
Setting up MySQL 277
Getting data into Hadoop 283
Getting data out of Hadoop 299
AWS considerations 309
Summary 310

Chapter 10: Data Collection with Flume 311
A note about AWS 311
Data data everywhere... 312
Introducing Apache Flume 315
The bigger picture 339
Summary 341

Chapter 11: Where to Go Next 343
What we did and didn't cover in this book 343
Upcoming Hadoop changes 344
Alternative distributions 345
Other Apache projects 348
Other programming abstractions 350
AWS resources 351
Sources of information 352
Summary 353
Appendix: Pop Quiz Answers 355

Module 2: Hadoop Real-World Solutions Cookbook, Second Edition

Chapter 1: Getting Started with Hadoop 2.X 359
Introduction 359
Installing a single-node Hadoop Cluster 360
Installing a multi-node Hadoop cluster 367
Adding new nodes to existing Hadoop clusters 371
Executing the balancer command for uniform data distribution 372
Entering and exiting from the safe mode in a Hadoop cluster 375
Decommissioning DataNodes 376
Performing benchmarking on a Hadoop cluster 377
Chapter 2: Exploring HDFS 381
Introduction 381
Loading data from a local machine to HDFS 382
Exporting HDFS data to a local machine 383
Changing the replication factor of an existing file in HDFS 384
Setting the HDFS block size for all the files in a cluster 385
Setting the HDFS block size for a specific file in a cluster 386
Enabling transparent encryption for HDFS 387
Importing data from another Hadoop cluster 389
Recycling deleted data from trash to HDFS 390
Saving compressed data in HDFS 392
Chapter 3: Mastering Map Reduce Programs 395
Introduction 395
Writing the Map Reduce program in Java to analyze web log data 396
Executing the Map Reduce program in a Hadoop cluster 400

Adding support for a new writable data type in Hadoop 401
Implementing a user-defined counter in a Map Reduce program 405
Map Reduce program to find the top X 409
Map Reduce program to find distinct values 413
Map Reduce program to partition data using a custom partitioner 416
Writing Map Reduce results to multiple output files 419
Performing Reduce side Joins using Map Reduce 421
Unit testing the Map Reduce code using MRUnit 427

Chapter 4: Data Analysis Using Hive, Pig, and Hbase 431
Introduction 432
Storing and processing Hive data in a sequential file format 433
Storing and processing Hive data in the RC file format 435
Storing and processing Hive data in the ORC file format 437
Storing and processing Hive data in the Parquet file format 439
Performing FILTER By queries in Pig 441
Performing Group By queries in Pig 442
Performing Order By queries in Pig 444
Performing JOINS in Pig 445
Writing a user-defined function in Pig 448
Analyzing web log data using Pig 452
Performing the Hbase operation in CLI 455
Performing Hbase operations in Java 461
Executing the MapReduce programming with an Hbase Table 463

Chapter 5: Advanced Data Analysis Using Hive 467
Introduction 467
Processing JSON data in Hive using JSON SerDe 468
Processing XML data in Hive using XML SerDe 471
Processing Hive data in the Avro format 474
Writing a user-defined function in Hive 476
Performing table joins in Hive 479
Executing map side joins in Hive 482
Performing context Ngram in Hive 485
Call Data Record Analytics using Hive 487
Twitter sentiment analysis using Hive 489
Implementing Change Data Capture using Hive 493
Multiple table inserting using Hive 497

Chapter 6: Data Import/Export Using Sqoop and Flume 501
Introduction 501
Importing data from RDMBS to HDFS using Sqoop 502
Exporting data from HDFS to RDBMS 506

Using query operator in Sqoop import 509
Importing data using Sqoop in compressed format 511
Performing Atomic export using Sqoop 512
Importing data into Hive tables using Sqoop 514
Importing data into HDFS from Mainframes 515
Incremental import using Sqoop 516
Creating and executing Sqoop job 519
Importing data from RDBMS to Hbase using Sqoop 522
Importing Twitter data into HDFS using Flume 526
Importing data from Kafka into HDFS using Flume 531
Importing web logs data into HDFS using Flume 533

Chapter 7: Automation of Hadoop Tasks Using Oozie 535
Introduction 535
Implementing a Sqoop action job using Oozie 536
Implementing a Map Reduce action job using Oozie 538
Implementing a Java action job using Oozie 540
Implementing a Hive action job using Oozie 542
Implementing a Pig action job using Oozie 544
Implementing an e-mail action job using Oozie 546
Executing parallel jobs using Oozie (fork) 548
Scheduling a job in Oozie 550

Chapter 8: Machine Learning and Predictive Analytics Using Mahout and R 553
Introduction 553
Setting up the Mahout development environment 554
Creating an item-based recommendation engine using Mahout 556
Creating a user-based recommendation engine using Mahout 559
Using Predictive analytics on Bank Data using Mahout 562
Clustering text data using K-Means 565
Performing population Data Analytics using R 566
Performing Twitter Sentiment Analytics using R 570
Performing Predictive Analytics using R 575

Chapter 9: Integration with Apache Spark 579
Introduction 579
Running Spark standalone 580
Running Spark on YARN 582
Performing Olympics Athletes analytics using the Spark Shell 583
Creating Twitter trending topics using Spark Streaming 587
Twitter trending topics using Spark streaming 591
Analyzing Parquet files using Spark 595

Analyzing JSON data using Spark 597
Processing graphs using Graph X 600
Conducting predictive analytics using Spark MLib 603

Chapter 10: Hadoop Use Cases 607
Introduction 607
Call Data Record analytics 607
Web log analytics 612
Sensitive data masking and encryption using Hadoop 618

Module 3: Mastering Hadoop

Chapter 1: Hadoop 2.X 625
The inception of Hadoop 626
The evolution of Hadoop 627
Hadoop 2.X 631
Hadoop distributions 637
Summary 644

Chapter 2: Advanced MapReduce 645
MapReduce input 646
The Map task 658
The Reduce task 664
MapReduce output 666
MapReduce job counters 667
Handling data joins 669
Summary 682

Chapter 3: Advanced Pig 683
Pig versus SQL 684
Different modes of execution 685
Complex data types in Pig 686
Compiling Pig scripts 687
Development and debugging aids 690
The advanced Pig operators 691
User-defined functions 699
Pig performance optimizations 709
Best practices 715
Summary 718

Chapter 4: Advanced Hive 719
The Hive architecture 720
Data types 722
File formats 723
The data model 725
Hive query optimizers 732
Advanced DML 733
UDF, UDAF, and UDTF 737
Summary 741

Chapter 5: Serialization and Hadoop I/O 743
Data serialization in Hadoop 743
Avro serialization 749
File formats 759
Compression 766
Summary 769

Chapter 6: YARN – Bringing Other Paradigms to Hadoop 771
The YARN architecture 772
Developing YARN applications 776
Monitoring YARN 789
Job scheduling in YARN 795
YARN commands 802
Summary 804

Chapter 7: Storm on YARN – Low Latency Processing in Hadoop 805
Batch processing versus streaming 806
Apache Storm 807
Storm on YARN 821
Summary 832

Chapter 8: Hadoop on the Cloud 833
Cloud computing characteristics 833
Hadoop on the cloud 835
Amazon Elastic MapReduce (EMR) 836
Summary 851

Chapter 9: HDFS Replacements 853
HDFS – advantages and drawbacks 854
Amazon AWS S3 854
Implementing a filesystem in Hadoop 857
Implementing an S3 native filesystem in Hadoop 858
Summary 870

Chapter 10: HDFS Federation 871
Limitations of the older HDFS architecture 872
Architecture of HDFS Federation 873
HDFS high availability 878
HDFS block placement 882
Summary 885
Chapter 11: Hadoop Security 887
The security pillars 888
Authentication in Hadoop 889
Authorization in Hadoop 893
Data confidentiality in Hadoop 900
Audit logging in Hadoop 907
Summary 909
Chapter 12: Analytics Using Hadoop 911
Data analytics workflow 912
Machine learning 913
Apache Mahout 916
Document analysis using Hadoop and Mahout 918
RHadoop 931
Summary 932
Appendix: Hadoop for Microsoft Windows 933
Deploying Hadoop on Microsoft Windows 934
Summary 952
Bibliography 953

Module 1

Hadoop Beginner's Guide

Learn how to crunch big data to extract meaning from the data avalanche

1
What It's All About

This book is about Hadoop, an open source framework for large-scale data processing. Before we get into the details of the technology and its use in later chapters, it is important to spend a little time exploring the trends that led to Hadoop's creation and its enormous success.

Hadoop was not created in a vacuum; instead, it exists due to the explosion in the amount of data being created and consumed and a shift that sees this data deluge arrive at small startups and not just huge multinationals. At the same time, other trends have changed how software and systems are deployed, using cloud resources alongside or even in preference to more traditional infrastructures.

This chapter will explore some of these trends and explain in detail the specific problems Hadoop seeks to solve and the drivers that shaped its design.

In the rest of this chapter we shall:

- Learn about the big data revolution
- Understand what Hadoop is and how it can extract value from data
- Look into cloud computing and understand what Amazon Web Services provides
- See how powerful the combination of big data processing and cloud computing can be
- Get an overview of the topics covered in the rest of this book

So let's get on with it!

Big data processing

Look around at the technology we have today, and it's easy to come to the conclusion that it's all about data. As consumers, we have an increasing appetite for rich media, both in terms of the movies we watch and the pictures and videos we create and upload. We also, often without thinking, leave a trail of data across the Web as we perform the actions of our daily lives.

Not only is the amount of data being generated increasing, but the rate of increase is also accelerating. From emails to Facebook posts, from purchase histories to web links, there are large data sets growing everywhere. The challenge is in extracting from this data the most valuable aspects; sometimes this means particular data elements, and at other times, the focus is instead on identifying trends and relationships between pieces of data.

There's a subtle change occurring behind the scenes that is all about using data in more and more meaningful ways. Large companies have realized the value in data for some time and have been using it to improve the services they provide to their customers, that is, us. Consider how Google displays advertisements relevant to our web surfing, or how Amazon or Netflix recommend new products or titles that often match well to our tastes and interests.

The value of data

These corporations wouldn't invest in large-scale data processing if it didn't provide a meaningful return on the investment or a competitive advantage. There are several main aspects to big data that should be appreciated:

- Some questions only give value when asked of sufficiently large data sets. Recommending a movie based on the preferences of another person is, in the absence of other factors, unlikely to be very accurate. Increase the number of people to a hundred and the chances increase slightly. Use the viewing history of ten million other people and the chances of detecting patterns that can be used to give relevant recommendations improve dramatically.
- Big data tools often enable the processing of data on a larger scale and at a lower cost than previous solutions. As a consequence, it is often possible to perform data processing tasks that were previously prohibitively expensive.
- The cost of large-scale data processing isn't just about financial expense; latency is also a critical factor. A system may be able to process as much data as is thrown at it, but if the average processing time is measured in weeks, it is likely not useful. Big data tools allow data volumes to be increased while keeping processing time under control, usually by matching the increased data volume with additional hardware.

- Previous assumptions of what a database should look like or how its data should be structured may need to be revisited to meet the needs of the biggest data problems.
- In combination with the preceding points, sufficiently large data sets and flexible tools allow previously unimagined questions to be answered.

Historically for the few and not the many

The examples discussed in the previous section have generally been seen in the form of innovations of large search engines and online companies. This is a continuation of a much older trend wherein processing large data sets was an expensive and complex undertaking, out of the reach of small- or medium-sized organizations.

Similarly, the broader approach of data mining has been around for a very long time but has never really been a practical tool outside the largest corporations and government agencies.

This situation may have been regrettable but most smaller organizations were not at a disadvantage as they rarely had access to the volume of data requiring such an investment.

The increase in data is not limited to the big players anymore, however; many small and medium companies—not to mention some individuals—find themselves gathering larger and larger amounts of data that they suspect may have some value they want to unlock.

Before understanding how this can be achieved, it is important to appreciate some of these broader historical trends that have laid the foundations for systems such as Hadoop today.

Classic data processing systems

The fundamental reason that big data mining systems were rare and expensive is that scaling a system to process large data sets is very difficult; as we will see, it has traditionally been limited to the processing power that can be built into a single computer.

There are however two broad approaches to scaling a system as the size of the data increases, generally referred to as scale-up and scale-out.

Scale-up

In most enterprises, data processing has typically been performed on impressively large computers with impressively larger price tags. As the size of the data grows, the approach is to move to a bigger server or storage array. Through an effective architecture—even today, as we'll describe later in this chapter—the cost of such hardware could easily be measured in hundreds of thousands or in millions of dollars.

The advantage of simple scale-up is that the architecture does not significantly change through the growth. Though larger components are used, the basic relationship (for example, database server and storage array) stays the same. For applications such as commercial database engines, the software handles the complexities of utilizing the available hardware, but in theory, increased scale is achieved by migrating the same software onto larger and larger servers. Note though that the difficulty of moving software onto more and more processors is never trivial; in addition, there are practical limits on just how big a single host can be, so at some point, scale-up cannot be extended any further.

The promise of a single architecture at any scale is also unrealistic. Designing a scale-up system to handle data sets of sizes such as 1 terabyte, 100 terabyte, and 1 petabyte may conceptually apply larger versions of the same components, but the complexity of their connectivity may vary from cheap commodity through custom hardware as the scale increases.

Early approaches to scale-out

Instead of growing a system onto larger and larger hardware, the scale-out approach spreads the processing onto more and more machines. If the data set doubles, simply use two servers instead of a single double-sized one. If it doubles again, move to four hosts.

The obvious benefit of this approach is that purchase costs remain much lower than for scale-up. Server hardware costs tend to increase sharply when one seeks to purchase larger machines, and though a single host may cost $5,000, one with ten times the processing power may cost a hundred times as much. The downside is that we need to develop strategies for splitting our data processing across a fleet of servers and the tools historically used for this purpose have proven to be complex.

As a consequence, deploying a scale-out solution has required significant engineering effort; the system developer often needs to handcraft the mechanisms for data partitioning and reassembly, not to mention the logic to schedule the work across the cluster and handle individual machine failures.

Limiting factors

These traditional approaches to scale-up and scale-out have not been widely adopted outside large enterprises, government, and academia. The purchase costs are often high, as is the effort to develop and manage the systems. These factors alone put them out of the reach of many smaller businesses. In addition, the approaches themselves have had several weaknesses that have become apparent over time:

- As scale-out systems get large, or as scale-up systems deal with multiple CPUs, the difficulties caused by the complexity of the concurrency in the systems have become significant. Effectively utilizing multiple hosts or CPUs is a very difficult task, and implementing the necessary strategy to maintain efficiency throughout execution of the desired workloads can entail enormous effort.

- Hardware advances—often couched in terms of Moore's law—have begun to highlight discrepancies in system capability. CPU power has grown much faster than network or disk speeds have; once CPU cycles were the most valuable resource in the system, but today, that no longer holds. Whereas a modern CPU may be able to execute millions of times as many operations as a CPU 20 years ago would, memory and hard disk speeds have only increased by factors of thousands or even hundreds. It is quite easy to build a modern system with so much CPU power that the storage system simply cannot feed it data fast enough to keep the CPUs busy.

A different approach

From the preceding scenarios there are a number of techniques that have been used successfully to ease the pain in scaling data processing systems to the large scales required by big data.

All roads lead to scale-out

As just hinted, taking a scale-up approach to scaling is not an open-ended tactic. There is a limit to the size of individual servers that can be purchased from mainstream hardware suppliers, and even more niche players can't offer an arbitrarily large server. At some point, the workload will increase beyond the capacity of the single, monolithic scale-up server, so then what? The unfortunate answer is that the best approach is to have two large servers instead of one. Then, later, three, four, and so on. Or, in other words, the natural tendency of scale-up architecture is—in extreme cases—to add a scale-out strategy to the mix. Though this gives some of the benefits of both approaches, it also compounds the costs and weaknesses; instead of very expensive hardware or the need to manually develop the cross-cluster logic, this hybrid architecture requires both.

As a consequence of this end-game tendency and the general cost profile of scale-up architectures, they are rarely used in the big data processing field and scale-out architectures are the de facto standard.



If your problem space involves data workloads with strong internal cross-references and a need for transactional integrity, big iron scale-up relational databases are still likely to be a great option.

Share nothing

Anyone with children will have spent considerable time teaching the little ones that it's good to share. This principle does not extend into data processing systems, and this idea applies to both data and hardware.

The conceptual view of a scale-out architecture in particular shows individual hosts, each processing a subset of the overall data set to produce its portion of the final result. Reality is rarely so straightforward. Instead, hosts may need to communicate between each other, or some pieces of data may be required by multiple hosts. These additional dependencies create opportunities for the system to be negatively affected in two ways: bottlenecks and increased risk of failure.

If a piece of data or individual server is required by every calculation in the system, there is a likelihood of contention and delays as the competing clients access the common data or host. If, for example, in a system with 25 hosts there is a single host that must be accessed by all the rest, the overall system performance will be bounded by the capabilities of this key host.

Worse still, if this "hot" server or storage system holding the key data fails, the entire workload will collapse in a heap. Earlier cluster solutions often demonstrated this risk; even though the workload was processed across a farm of servers, they often used a shared storage system to hold all the data.

Instead of sharing resources, the individual components of a system should be as independent as possible, allowing each to proceed regardless of whether others are tied up in complex work or are experiencing failures.

Expect failure

Implicit in the preceding tenets is that more hardware will be thrown at the problem with as much independence as possible. This is only achievable if the system is built with an expectation that individual components will fail, often regularly and with inconvenient timing.



You'll often hear terms such as "five nines" (referring to 99.999 percent uptime or availability). Though this is absolute best-in-class availability, it is important to realize that the overall reliability of a system comprised of many such devices can vary greatly depending on whether the system can tolerate individual component failures.

Assume a server with 99 percent reliability and a system that requires five such hosts to function. The system availability is 0.99*0.99*0.99*0.99*0.99 which equates to 95 percent availability. But if the individual servers are only rated at 95 percent, the system reliability drops to a mere 76 percent.

Instead, if you build a system that only needs one of the five hosts to be functional at any given time, the system availability is well into five nines territory. Thinking about system uptime in relation to the criticality of each component can help focus on just what the system availability is likely to be.



If figures such as 99 percent availability seem a little abstract to you, consider it in terms of how much downtime that would mean in a given time period. For example, 99 percent availability equates to a downtime of just over 3.5 days a year or 7 hours a month. Still sound as good as 99 percent?

This approach of embracing failure is often one of the most difficult aspects of big data systems for newcomers to fully appreciate. This is also where the approach diverges most strongly from scale-up architectures. One of the main reasons for the high cost of large scale-up servers is the amount of effort that goes into mitigating the impact of component failures. Even low-end servers may have redundant power supplies, but in a big iron box, you will see CPUs mounted on cards that connect across multiple backplanes to banks of memory and storage systems. Big iron vendors have often gone to extremes to show how resilient their systems are by doing everything from pulling out parts of the server while it's running to actually shooting a gun at it. But if the system is built in such a way that instead of treating every failure as a crisis to be mitigated it is reduced to irrelevance, a very different architecture emerges.

Smart software, dumb hardware

If we wish to see a cluster of hardware used in as flexible a way as possible, providing hosting to multiple parallel workflows, the answer is to push the smarts into the software and away from the hardware.

In this model, the hardware is treated as a set of resources, and the responsibility for allocating hardware to a particular workload is given to the software layer. This allows hardware to be generic and hence both easier and less expensive to acquire, and the functionality to efficiently use the hardware moves to the software, where the knowledge about effectively performing this task resides.

Move processing, not data

Imagine you have a very large data set, say, 1000 terabytes (that is, 1 petabyte), and you need to perform a set of four operations on every piece of data in the data set. Let's look at different ways of implementing a system to solve this problem.

A traditional big iron scale-up solution would see a massive server attached to an equally impressive storage system, almost certainly using technologies such as fibre channel to maximize storage bandwidth. The system will perform the task but will become I/O-bound; even high-end storage switches have a limit on how fast data can be delivered to the host.

Alternatively, the processing approach of previous cluster technologies would perhaps see a cluster of 1,000 machines, each with 1 terabyte of data divided into four quadrants, with each responsible for performing one of the operations. The cluster management software would then coordinate the movement of the data around the cluster to ensure each piece receives all four processing steps. As each piece of data can have one step performed on the host on which it resides, it will need to stream the data to the other three quadrants, so we are in effect consuming 3 petabytes of network bandwidth to perform the processing.

Remembering that processing power has increased faster than networking or disk technologies, so are these really the best ways to address the problem? Recent experience suggests the answer is no and that an alternative approach is to avoid moving the data and instead move the processing. Use a cluster as just mentioned, but don't segment it into quadrants; instead, have each of the thousand nodes perform all four processing stages on the locally held data. If you're lucky, you'll only have to stream the data from the disk once and the only things travelling across the network will be program binaries and status reports, both of which are dwarfed by the actual data set in question.

If a 1,000-node cluster sounds ridiculously large, think of some modern server form factors being utilized for big data solutions. These see single hosts with as many as twelve 1- or 2-terabyte disks in each. Because modern processors have multiple cores it is possible to build a 50-node cluster with a petabyte of storage and still have a CPU core dedicated to process the data stream coming off each individual disk.

Build applications, not infrastructure

When thinking of the scenario in the previous section, many people will focus on the questions of data movement and processing. But, anyone who has ever built such a system will know that less obvious elements such as job scheduling, error handling, and coordination are where much of the magic truly lies.

If we had to implement the mechanisms for determining where to execute processing, performing the processing, and combining all the subresults into the overall result, we wouldn't have gained much from the older model. There, we needed to explicitly manage data partitioning; we'd just be exchanging one difficult problem with another.

This touches on the most recent trend, which we'll highlight here: a system that handles most of the cluster mechanics transparently and allows the developer to think in terms of the business problem. Frameworks that provide well-defined interfaces that abstract all this complexity—smart software—upon which business domain-specific applications can be built give the best combination of developer and system efficiency.

Hadoop

The thoughtful (or perhaps suspicious) reader will not be surprised to learn that the preceding approaches are all key aspects of Hadoop. But we still haven't actually answered the question about exactly what Hadoop is.

Thanks, Google

It all started with Google, which in 2003 and 2004 released two academic papers describing Google technology: the **Google File System** (**GFS**) (`http://research.google.com/archive/gfs.html`) and MapReduce (`http://research.google.com/archive/mapreduce.html`). The two together provided a platform for processing data on a very large scale in a highly efficient manner.

Thanks, Doug

At the same time, Doug Cutting was working on the Nutch open source web search engine. He had been working on elements within the system that resonated strongly once the Google GFS and MapReduce papers were published. Doug started work on the implementations of these Google systems, and Hadoop was soon born, firstly as a subproject of Lucene and soon was its own top-level project within the Apache open source foundation. At its core, therefore, Hadoop is an open source platform that provides implementations of both the MapReduce and GFS technologies and allows the processing of very large data sets across clusters of low-cost commodity hardware.

Thanks, Yahoo

Yahoo hired Doug Cutting in 2006 and quickly became one of the most prominent supporters of the Hadoop project. In addition to often publicizing some of the largest Hadoop deployments in the world, Yahoo has allowed Doug and other engineers to contribute to Hadoop while still under its employ; it has contributed some of its own internally developed Hadoop improvements and extensions. Though Doug has now moved on to Cloudera (another prominent startup supporting the Hadoop community) and much of the Yahoo's Hadoop team has been spun off into a startup called Hortonworks, Yahoo remains a major Hadoop contributor.

Parts of Hadoop

The top-level Hadoop project has many component subprojects, several of which we'll discuss in this book, but the two main ones are **Hadoop Distributed File System** (**HDFS**) and MapReduce. These are direct implementations of Google's own GFS and MapReduce. We'll discuss both in much greater detail, but for now, it's best to think of HDFS and MapReduce as a pair of complementary yet distinct technologies.

HDFS is a filesystem that can store very large data sets by scaling out across a cluster of hosts. It has specific design and performance characteristics; in particular, it is optimized for throughput instead of latency, and it achieves high availability through replication instead of redundancy.

MapReduce is a data processing paradigm that takes a specification of how the data will be input and output from its two stages (called map and reduce) and then applies this across arbitrarily large data sets. MapReduce integrates tightly with HDFS, ensuring that wherever possible, MapReduce tasks run directly on the HDFS nodes that hold the required data.

Common building blocks

Both HDFS and MapReduce exhibit several of the architectural principles described in the previous section. In particular:

- Both are designed to run on clusters of commodity (that is, low-to-medium specification) servers
- Both scale their capacity by adding more servers (scale-out)
- Both have mechanisms for identifying and working around failures
- Both provide many of their services transparently, allowing the user to concentrate on the problem at hand
- Both have an architecture where a software cluster sits on the physical servers and controls all aspects of system execution

HDFS

HDFS is a filesystem unlike most you may have encountered before. It is not a POSIX-compliant filesystem, which basically means it does not provide the same guarantees as a regular filesystem. It is also a distributed filesystem, meaning that it spreads storage across multiple nodes; lack of such an efficient distributed filesystem was a limiting factor in some historical technologies. The key features are:

- HDFS stores files in blocks typically at least 64 MB in size, much larger than the 4-32 KB seen in most filesystems.
- HDFS is optimized for throughput over latency; it is very efficient at streaming read requests for large files but poor at seek requests for many small ones.
- HDFS is optimized for workloads that are generally of the write-once and read-many type.
- Each storage node runs a process called a DataNode that manages the blocks on that host, and these are coordinated by a master NameNode process running on a separate host.

- Instead of handling disk failures by having physical redundancies in disk arrays or similar strategies, HDFS uses replication. Each of the blocks comprising a file is stored on multiple nodes within the cluster, and the HDFS NameNode constantly monitors reports sent by each DataNode to ensure that failures have not dropped any block below the desired replication factor. If this does happen, it schedules the addition of another copy within the cluster.

MapReduce

Though MapReduce as a technology is relatively new, it builds upon much of the fundamental work from both mathematics and computer science, particularly approaches that look to express operations that would then be applied to each element in a set of data. Indeed the individual concepts of functions called `map` and `reduce` come straight from functional programming languages where they were applied to lists of input data.

Another key underlying concept is that of "divide and conquer", where a single problem is broken into multiple individual subtasks. This approach becomes even more powerful when the subtasks are executed in parallel; in a perfect case, a task that takes 1000 minutes could be processed in 1 minute by 1,000 parallel subtasks.

MapReduce is a processing paradigm that builds upon these principles; it provides a series of transformations from a source to a result data set. In the simplest case, the input data is fed to the `map` function and the resultant temporary data to a `reduce` function. The developer only defines the data transformations; Hadoop's MapReduce job manages the process of how to apply these transformations to the data across the cluster in parallel. Though the underlying ideas may not be novel, a major strength of Hadoop is in how it has brought these principles together into an accessible and well-engineered platform.

Unlike traditional relational databases that require structured data with well-defined schemas, MapReduce and Hadoop work best on semi-structured or unstructured data. Instead of data conforming to rigid schemas, the requirement is instead that the data be provided to the `map` function as a series of key value pairs. The output of the `map` function is a set of other key value pairs, and the `reduce` function performs aggregation to collect the final set of results.

Hadoop provides a standard specification (that is, interface) for the `map` and `reduce` functions, and implementations of these are often referred to as **mappers** and **reducers**. A typical MapReduce job will comprise of a number of mappers and reducers, and it is not unusual for several of these to be extremely simple. The developer focuses on expressing the transformation between source and result data sets, and the Hadoop framework manages all aspects of job execution, parallelization, and coordination.

This last point is possibly the most important aspect of Hadoop. The platform takes responsibility for every aspect of executing the processing across the data. After the user defines the key criteria for the job, everything else becomes the responsibility of the system. Critically, from the perspective of the size of data, the same MapReduce job can be applied to data sets of any size hosted on clusters of any size. If the data is 1 gigabyte in size and on a single host, Hadoop will schedule the processing accordingly. Even if the data is 1 petabyte in size and hosted across one thousand machines, it still does likewise, determining how best to utilize all the hosts to perform the work most efficiently. From the user's perspective, the actual size of the data and cluster are transparent, and apart from affecting the time taken to process the job, they do not change how the user interacts with Hadoop.

Better together

It is possible to appreciate the individual merits of HDFS and MapReduce, but they are even more powerful when combined. HDFS can be used without MapReduce, as it is intrinsically a large-scale data storage platform. Though MapReduce can read data from non-HDFS sources, the nature of its processing aligns so well with HDFS that using the two together is by far the most common use case.

When a MapReduce job is executed, Hadoop needs to decide where to execute the code most efficiently to process the data set. If the MapReduce-cluster hosts all pull their data from a single storage host or an array, it largely doesn't matter as the storage system is a shared resource that will cause contention. But if the storage system is HDFS, it allows MapReduce to execute data processing on the node holding the data of interest, building on the principle of it being less expensive to move data processing than the data itself.

The most common deployment model for Hadoop sees the HDFS and MapReduce clusters deployed on the same set of servers. Each host that contains data and the HDFS component to manage it also hosts a MapReduce component that can schedule and execute data processing. When a job is submitted to Hadoop, it can use an optimization process as much as possible to schedule data on the hosts where the data resides, minimizing network traffic and maximizing performance.

Think back to our earlier example of how to process a four-step task on 1 petabyte of data spread across one thousand servers. The MapReduce model would (in a somewhat simplified and idealized way) perform the processing in a `map` function on each piece of data on a host where the data resides in HDFS and then reuse the cluster in the `reduce` function to collect the individual results into the final result set.

A part of the challenge with Hadoop is in breaking down the overall problem into the best combination of `map` and `reduce` functions. The preceding approach would only work if the four-stage processing chain could be applied independently to each data element in turn. As we'll see in later chapters, the answer is sometimes to use multiple MapReduce jobs where the output of one is the input to the next.

Common architecture

Both HDFS and MapReduce are, as mentioned, software clusters that display common characteristics:

- Each follows an architecture where a cluster of worker nodes is managed by a special master/coordinator node
- The master in each case (NameNode for HDFS and JobTracker for MapReduce) monitors the health of the cluster and handle failures, either by moving data blocks around or by rescheduling failed work
- Processes on each server (DataNode for HDFS and TaskTracker for MapReduce) are responsible for performing work on the physical host, receiving instructions from the NameNode or JobTracker, and reporting health/progress status back to it

As a minor terminology point, we will generally use the terms **host** or **server** to refer to the physical hardware hosting Hadoop's various components. The term **node** will refer to the software component comprising a part of the cluster.

What it is and isn't good for

As with any tool, it's important to understand when Hadoop is a good fit for the problem in question. Much of this book will highlight its strengths, based on the previous broad overview on processing large data volumes, but it's important to also start appreciating at an early stage where it isn't the best choice.

The architecture choices made within Hadoop enable it to be the flexible and scalable data processing platform it is today. But, as with most architecture or design choices, there are consequences that must be understood. Primary amongst these is the fact that Hadoop is a batch processing system. When you execute a job across a large data set, the framework will churn away until the final results are ready. With a large cluster, answers across even huge data sets can be generated relatively quickly, but the fact remains that the answers are not generated fast enough to service impatient users. Consequently, Hadoop alone is not well suited to low-latency queries such as those received on a website, a real-time system, or a similar problem domain.

When Hadoop is running jobs on large data sets, the overhead of setting up the job, determining which tasks are run on each node, and all the other housekeeping activities that are required is a trivial part of the overall execution time. But, for jobs on small data sets, there is an execution overhead that means even simple MapReduce jobs may take a minimum of 10 seconds.



Another member of the broader Hadoop family is **HBase**, an open-source implementation of another Google technology. This provides a (non-relational) database atop Hadoop that uses various means to allow it to serve low-latency queries.

But haven't Google and Yahoo both been among the strongest proponents of this method of computation, and aren't they all about such websites where response time is critical? The answer is yes, and it highlights an important aspect of how to incorporate Hadoop into any organization or activity or use it in conjunction with other technologies in a way that exploits the strengths of each. In a paper (`http://research.google.com/archive/googlecluster.html`), Google sketches how they utilized MapReduce at the time; after a web crawler retrieved updated webpage data, MapReduce processed the huge data set, and from this, produced the web index that a fleet of MySQL servers used to service end-user search requests.

Cloud computing with Amazon Web Services

The other technology area we'll explore in this book is cloud computing, in the form of several offerings from Amazon Web Services. But first, we need to cut through some hype and buzzwords that surround this thing called cloud computing.

Too many clouds

Cloud computing has become an overused term, arguably to the point that its overuse risks it being rendered meaningless. In this book, therefore, let's be clear what we mean—and care about—when using the term. There are two main aspects to this: a new architecture option and a different approach to cost.

A third way

We've talked about scale-up and scale-out as the options for scaling data processing systems. But our discussion thus far has taken for granted that the physical hardware that makes either option a reality will be purchased, owned, hosted, and managed by the organization doing the system development. The cloud computing we care about adds a third approach; put your application into the cloud and let the provider deal with the scaling problem.

It's not always that simple, of course. But for many cloud services, the model truly is this revolutionary. You develop the software according to some published guidelines or interface and then deploy it onto the cloud platform and allow it to scale the service based on the demand, for a cost of course. But given the costs usually involved in making scaling systems, this is often a compelling proposition.

Different types of costs

This approach to cloud computing also changes how system hardware is paid for. By offloading infrastructure costs, all users benefit from the economies of scale achieved by the cloud provider by building their platforms up to a size capable of hosting thousands or millions of clients. As a user, not only do you get someone else to worry about difficult engineering problems, such as scaling, but you pay for capacity as it's needed and you don't have to size the system based on the largest possible workloads. Instead, you gain the benefit of elasticity and use more or fewer resources as your workload demands.

An example helps illustrate this. Many companies' financial groups run end-of-month workloads to generate tax and payroll data, and often, much larger data crunching occurs at year end. If you were tasked with designing such a system, how much hardware would you buy? If you only buy enough to handle the day-to-day workload, the system may struggle at month end and may likely be in real trouble when the end-of-year processing rolls around. If you scale for the end-of-month workloads, the system will have idle capacity for most of the year and possibly still be in trouble performing the end-of-year processing. If you size for the end-of-year workload, the system will have significant capacity sitting idle for the rest of the year. And considering the purchase cost of hardware in addition to the hosting and running costs—a server's electricity usage may account for a large majority of its lifetime costs—you are basically wasting huge amounts of money.

The service-on-demand aspects of cloud computing allow you to start your application on a small hardware footprint and then scale it up and down as the year progresses. With a pay-for-use model, your costs follow your utilization and you have the capacity to process your workloads without having to buy enough hardware to handle the peaks.

A more subtle aspect of this model is that this greatly reduces the costs of entry for an organization to launch an online service. We all know that a new hot service that fails to meet demand and suffers performance problems will find it hard to recover momentum and user interest. For example, say in the year 2000, an organization wanting to have a successful launch needed to put in place, on launch day, enough capacity to meet the massive surge of user traffic they hoped for but did n't know for sure to expect. When taking costs of physical location into consideration, it would have been easy to spend millions on a product launch.

Today, with cloud computing, the initial infrastructure cost could literally be as low as a few tens or hundreds of dollars a month and that would only increase when—and if—the traffic demanded.

AWS – infrastructure on demand from Amazon

Amazon Web Services (**AWS**) is a set of such cloud computing services offered by Amazon. We will be using several of these services in this book.

Elastic Compute Cloud (EC2)

Amazon's **Elastic Compute Cloud** (**EC2**), found at `http://aws.amazon.com/ec2/`, is basically a server on demand. After registering with AWS and EC2, credit card details are all that's required to gain access to a dedicated virtual machine, it's easy to run a variety of operating systems including Windows and many variants of Linux on our server.

Need more servers? Start more. Need more powerful servers? Change to one of the higher specification (and cost) types offered. Along with this, EC2 offers a suite of complimentary services, including load balancers, static IP addresses, high-performance additional virtual disk drives, and many more.

Simple Storage Service (S3)

Amazon's **Simple Storage Service** (**S3**), found at `http://aws.amazon.com/s3/`, is a storage service that provides a simple key/value storage model. Using web, command-line, or programmatic interfaces to create objects, which can be everything from text files to images to MP3s, you can store and retrieve your data based on a hierarchical model. You create buckets in this model that contain objects. Each bucket has a unique identifier, and within each bucket, every object is uniquely named. This simple strategy enables an extremely powerful service for which Amazon takes complete responsibility (for service scaling, in addition to reliability and availability of data).

Elastic MapReduce (EMR)

Amazon's **Elastic MapReduce** (**EMR**), found at `http://aws.amazon.com/elasticmapreduce/`, is basically Hadoop in the cloud and builds atop both EC2 and S3. Once again, using any of the multiple interfaces (web console, CLI, or API), a Hadoop workflow is defined with attributes such as the number of Hadoop hosts required and the location of the source data. The Hadoop code implementing the MapReduce jobs is provided and the virtual go button is pressed.

In its most impressive mode, EMR can pull source data from S3, process it on a Hadoop cluster it creates on EC2, push the results back into S3, and terminate the Hadoop cluster and the EC2 virtual machines hosting it. Naturally, each of these services has a cost (usually on per GB stored and server time usage basis), but the ability to access such powerful data processing capabilities with no need for dedicated hardware is a powerful one.

What this book covers

In this book we will be learning how to write MapReduce programs to do some serious data crunching and how to run them on both locally managed and AWS-hosted Hadoop clusters.

Not only will we be looking at Hadoop as an engine for performing MapReduce processing, but we'll also explore how a Hadoop capability can fit into the rest of an organization's infrastructure and systems. We'll look at some of the common points of integration, such as getting data between Hadoop and a relational database and also how to make Hadoop look more like such a relational database.

A dual approach

In this book we will not be limiting our discussion to EMR or Hadoop hosted on Amazon EC2; we will be discussing both the building and the management of local Hadoop clusters (on Ubuntu Linux) in addition to showing how to push the processing into the cloud via EMR.

The reason for this is twofold: firstly, though EMR makes Hadoop much more accessible, there are aspects of the technology that only become apparent when manually administering the cluster. Though it is also possible to use EMR in a more manual mode, we'll generally use a local cluster for such explorations. Secondly, though it isn't necessarily an either/or decision, many organizations use a mixture of in-house and cloud-hosted capacities, sometimes due to a concern of over reliance on a single external provider, but practically speaking, it's often convenient to do development and small-scale tests on local capacity then deploy at production scale into the cloud.

In some of the latter chapters, where we discuss additional products that integrate with Hadoop, we'll only give examples of local clusters as there is no difference in how the products work regardless of where they are deployed.

Summary

We learned a lot in this chapter about big data, Hadoop, and cloud computing.

Specifically, we covered the emergence of big data and how changes in the approach to data processing and system architecture bring within the reach of almost any organization techniques that were previously prohibitively expensive.

We also looked at the history of Hadoop and how it builds upon many of these trends to provide a flexible and powerful data processing platform that can scale to massive volumes. We also looked at how cloud computing provides another system architecture approach, one which exchanges large up-front costs and direct physical responsibility for a pay-as-you-go model and a reliance on the cloud provider for hardware provision, management and scaling. We also saw what Amazon Web Services is and how its Elastic MapReduce service utilizes other AWS services to provide Hadoop in the cloud.

We also discussed the aim of this book and its approach to exploration on both locally-managed and AWS-hosted Hadoop clusters.

Now that we've covered the basics and know where this technology is coming from and what its benefits are, we need to get our hands dirty and get things running, which is what we'll do in *Chapter 2*, *Getting Hadoop Up and Running*.

2
Getting Hadoop Up and Running

Now that we have explored the opportunities and challenges presented by large-scale data processing and why Hadoop is a compelling choice, it's time to get things set up and running.

In this chapter, we will do the following:

- Learn how to install and run Hadoop on a local Ubuntu host
- Run some example Hadoop programs and get familiar with the system
- Set up the accounts required to use Amazon Web Services products such as EMR
- Create an on-demand Hadoop cluster on Elastic MapReduce
- Explore the key differences between a local and hosted Hadoop cluster

Hadoop on a local Ubuntu host

For our exploration of Hadoop outside the cloud, we shall give examples using one or more Ubuntu hosts. A single machine (be it a physical computer or a virtual machine) will be sufficient to run all the parts of Hadoop and explore MapReduce. However, production clusters will most likely involve many more machines, so having even a development Hadoop cluster deployed on multiple hosts will be good experience. However, for getting started, a single host will suffice.

Nothing we discuss will be unique to Ubuntu, and Hadoop should run on any Linux distribution. Obviously, you may have to alter how the environment is configured if you use a distribution other than Ubuntu, but the differences should be slight.

Other operating systems

Hadoop does run well on other platforms. Windows and Mac OS X are popular choices for developers. Windows is supported only as a development platform and Mac OS X is not formally supported at all.

If you choose to use such a platform, the general situation will be similar to other Linux distributions; all aspects of how to work with Hadoop will be the same on both platforms but you will need use the operating system-specific mechanisms for setting up environment variables and similar tasks. The Hadoop FAQs contain some information on alternative platforms and should be your first port of call if you are considering such an approach. The Hadoop FAQs can be found at `http://wiki.apache.org/hadoop/FAQ`.

Time for action – checking the prerequisites

Hadoop is written in Java, so you will need a recent **Java Development Kit** (**JDK**) installed on the Ubuntu host. Perform the following steps to check the prerequisites:

1. First, check what's already available by opening up a terminal and typing the following:

```
$ javac
$ java -version
```

2. If either of these commands gives a `no such file or directory` or similar error, or if the latter mentions "Open JDK", it's likely you need to download the full JDK. Grab this from the Oracle download page at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`; you should get the latest release.

3. Once Java is installed, add the `JDK/bin` directory to your path and set the `JAVA_HOME` environment variable with commands such as the following, modified for your specific Java version:

```
$ export JAVA_HOME=/opt/jdk1.6.0_24
$ export PATH=$JAVA_HOME/bin:${PATH}
```

What just happened?

These steps ensure the right version of Java is installed and available from the command line without having to use lengthy pathnames to refer to the install location.

Remember that the preceding commands only affect the currently running shell and the settings will be lost after you log out, close the shell, or reboot. To ensure the same setup is always available, you can add these to the startup files for your shell of choice, within the `.bash_profile` file for the BASH shell or the `.cshrc` file for TCSH, for example.

An alternative favored by me is to put all required configuration settings into a standalone file and then explicitly call this from the command line; for example:

```
$ source Hadoop_config.sh
```

This technique allows you to keep multiple setup files in the same account without making the shell startup overly complex; not to mention, the required configurations for several applications may actually be incompatible. Just remember to begin by loading the file at the start of each session!

Setting up Hadoop

One of the most confusing aspects of Hadoop to a newcomer is its various components, projects, sub-projects, and their interrelationships. The fact that these have evolved over time hasn't made the task of understanding it all any easier. For now, though, go to `http://hadoop.apache.org` and you'll see that there are three prominent projects mentioned:

- Common
- HDFS
- MapReduce

The last two of these should be familiar from the explanation in *Chapter 1*, *What It's All About*, and common projects comprise a set of libraries and tools that help the Hadoop product work in the real world. For now, the important thing is that the standard Hadoop distribution bundles the latest versions all of three of these projects and the combination is what you need to get going.

A note on versions

Hadoop underwent a major change in the transition from the 0.19 to the 0.20 versions, most notably with a migration to a set of new APIs used to develop MapReduce applications. We will be primarily using the new APIs in this book, though we do include a few examples of the older API in later chapters as not of all the existing features have been ported to the new API.

Hadoop versioning also became complicated when the 0.20 branch was renamed to 1.0. The 0.22 and 0.23 branches remained, and in fact included features not included in the 1.0 branch. At the time of this writing, things were becoming clearer with 1.1 and 2.0 branches being used for future development releases. As most existing systems and third-party tools are built against the 0.20 branch, we will use Hadoop 1.0 for the examples in this book.

Time for action – downloading Hadoop

Carry out the following steps to download Hadoop:

1. Go to the Hadoop download page at `http://hadoop.apache.org/common/releases.html` and retrieve the latest stable version of the 1.0.x branch; at the time of this writing, it was 1.0.4.
2. You'll be asked to select a local mirror; after that you need to download the file with a name such as `hadoop-1.0.4-bin.tar.gz`.
3. Copy this file to the directory where you want Hadoop to be installed (for example, `/usr/local`), using the following command:

   ```
   $ cp Hadoop-1.0.4.bin.tar.gz /usr/local
   ```

4. Decompress the file by using the following command:

   ```
   $ tar –xf hadoop-1.0.4-bin.tar.gz
   ```

5. Add a convenient symlink to the Hadoop installation directory.

   ```
   $ ln -s /usr/local/hadoop-1.0.4 /opt/hadoop
   ```

6. Now you need to add the Hadoop binary directory to your path and set the `HADOOP_HOME` environment variable, just as we did earlier with Java.

   ```
   $ export HADOOP_HOME=/usr/local/Hadoop
   $ export PATH=$HADOOP_HOME/bin:$PATH
   ```

7. Go into the `conf` directory within the Hadoop installation and edit the `Hadoop-env.sh` file. Search for `JAVA_HOME` and uncomment the line, modifying the location to point to your JDK installation, as mentioned earlier.

What just happened?

These steps ensure that Hadoop is installed and available from the command line. By setting the path and configuration variables, we can use the Hadoop command-line tool. The modification to the Hadoop configuration file is the only required change to the setup needed to integrate with your host settings.

As mentioned earlier, you should put the export commands in your shell startup file or a standalone-configuration script that you specify at the start of the session.

Don't worry about some of the details here; we'll cover Hadoop setup and use later.

Time for action – setting up SSH

Carry out the following steps to set up SSH:

1. Create a new OpenSSL key pair with the following commands:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hadoop/.ssh/id_rsa):
Created directory '/home/hadoop/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/hadoop/.ssh/id_rsa.
Your public key has been saved in /home/hadoop/.ssh/id_rsa.pub.
…
```

2. Copy the new public key to the list of authorized keys by using the following command:

```
$ cp .ssh/id _rsa.pub  .ssh/authorized_keys
```

3. Connect to the local host.

```
$ ssh localhost
The authenticity of host 'localhost (127.0.0.1)' can't be
established.
RSA key fingerprint is b6:0c:bd:57:32:b6:66:7c:33:7b:62:92:61:fd:c
a:2a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known
hosts.
```

4. Confirm that the password-less SSH is working.

```
$ ssh localhost
$ ssh localhost
```

What just happened?

Because Hadoop requires communication between multiple processes on one or more machines, we need to ensure that the user we are using for Hadoop can connect to each required host without needing a password. We do this by creating a **Secure Shell** (**SSH**) key pair that has an empty passphrase. We use the `ssh-keygen` command to start this process and accept the offered defaults.

Once we create the key pair, we need to add the new public key to the stored list of trusted keys; this means that when trying to connect to this machine, the public key will be trusted. After doing so, we use the `ssh` command to connect to the local machine and should expect to get a warning about trusting the host certificate as just shown. After confirming this, we should then be able to connect without further passwords or prompts.



Note that when we move later to use a fully distributed cluster, we will need to ensure that the Hadoop user account has the same key set up on every host in the cluster.

Configuring and running Hadoop

So far this has all been pretty straightforward, just downloading and system administration. Now we can deal with Hadoop directly. Finally! We'll run a quick example to show Hadoop in action. There is additional configuration and set up to be performed, but this next step will help give confidence that things are installed and configured correctly so far.

Time for action – using Hadoop to calculate Pi

We will now use a sample Hadoop program to calculate the value of Pi. Right now, this is primarily to validate the installation and to show how quickly you can get a MapReduce job to execute. Assuming the `HADOOP_HOME/bin` directory is in your path, type the following commands:

```
$ Hadoop jar hadoop/hadoop-examples-1.0.4.jar  pi 4 1000
Number of Maps  = 4
Samples per Map = 1000
Wrote input for Map #0
Wrote input for Map #1
Wrote input for Map #2
Wrote input for Map #3
Starting Job
12/10/26 22:56:11 INFO jvm.JvmMetrics: Initializing JVM Metrics
with processName=JobTracker, sessionId=
12/10/26 22:56:11 INFO mapred.FileInputFormat: Total input paths
to process : 4
12/10/26 22:56:12 INFO mapred.JobClient: Running job: job_
local_0001
```

```
12/10/26 22:56:12 INFO mapred.FileInputFormat: Total input paths
to process : 4
12/10/26 22:56:12 INFO mapred.MapTask: numReduceTasks: 1


…
12/10/26 22:56:14 INFO mapred.JobClient:  map 100% reduce 100%
12/10/26 22:56:14 INFO mapred.JobClient: Job complete: job_
local_0001
12/10/26 22:56:14 INFO mapred.JobClient: Counters: 13
12/10/26 22:56:14 INFO mapred.JobClient:   FileSystemCounters
…
Job Finished in 2.904 seconds
Estimated value of Pi is 3.14000000000000000000
$
```

What just happened?

There's a lot of information here; even more so when you get the full output on your screen. For now, let's unpack the fundamentals and not worry about much of Hadoop's status output until later in the book. The first thing to clarify is some terminology; each Hadoop program runs as a job that creates multiple tasks to do its work.

Looking at the output, we see it is broadly split into three sections:

- The start up of the job
- The status as the job executes
- The output of the job

In our case, we can see the job creates four tasks to calculate Pi, and the overall job result will be the combination of these subresults. This pattern should sound familiar to the one we came across in *Chapter 1*, *What It's All About*; the model is used to split a larger job into smaller pieces and then bring together the results.

The majority of the output will appear as the job is being executed and provide status messages showing progress. On successful completion, the job will print out a number of counters and other statistics. The preceding example is actually unusual in that it is rare to see the result of a MapReduce job displayed on the console. This is not a limitation of Hadoop, but rather a consequence of the fact that jobs that process large data sets usually produce a significant amount of output data that isn't well suited to a simple echoing on the screen.

Congratulations on your first successful MapReduce job!

Three modes

In our desire to get something running on Hadoop, we sidestepped an important issue: in which mode should we run Hadoop? There are three possibilities that alter where the various Hadoop components execute. Recall that HDFS comprises a single NameNode that acts as the cluster coordinator and is the master for one or more DataNodes that store the data. For MapReduce, the JobTracker is the cluster master and it coordinates the work executed by one or more TaskTracker processes. The Hadoop modes deploy these components as follows:

- **Local standalone mode**: This is the default mode if, as in the preceding Pi example, you don't configure anything else. In this mode, all the components of Hadoop, such as NameNode, DataNode, JobTracker, and TaskTracker, run in a single Java process.
- **Pseudo-distributed mode**: In this mode, a separate JVM is spawned for each of the Hadoop components and they communicate across network sockets, effectively giving a fully functioning minicluster on a single host.
- **Fully distributed mode**: In this mode, Hadoop is spread across multiple machines, some of which will be general-purpose workers and others will be dedicated hosts for components, such as NameNode and JobTracker.

Each mode has its benefits and drawbacks. Fully distributed mode is obviously the only one that can scale Hadoop across a cluster of machines, but it requires more configuration work, not to mention the cluster of machines. Local, or standalone, mode is the easiest to set up, but you interact with it in a different manner than you would with the fully distributed mode. In this book, we shall generally prefer the pseudo-distributed mode even when using examples on a single host, as everything done in the pseudo-distributed mode is almost identical to how it works on a much larger cluster.

Time for action – configuring the pseudo-distributed mode

Take a look in the `conf` directory within the Hadoop distribution. There are many configuration files, but the ones we need to modify are `core-site.xml`, `hdfs-site.xml` and `mapred-site.xml`.

1. Modify `core-site.xml` to look like the following code:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
<name>fs.default.name</name>
```

```
<value>hdfs://localhost:9000</value>
</property>
</configuration>
```

2. Modify `hdfs-site.xml` to look like the following code:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
</configuration>
```

3. Modify `mapred-site.xml` to look like the following code:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
<name>mapred.job.tracker</name>
<value>localhost:9001</value>
</property>
</configuration>
```

What just happened?

The first thing to note is the general format of these configuration files. They are obviously XML and contain multiple property specifications within a single configuration element.

The property specifications always contain name and value elements with the possibility for optional comments not shown in the preceding code.

We set three configuration variables here:

- The `dfs.default.name` variable holds the location of the NameNode and is required by both HDFS and MapReduce components, which explains why it's in `core-site.xml` and not `hdfs-site.xml`.

- The `dfs.replication` variable specifies how many times each HDFS block should be replicated. Recall from *Chapter 1*, *What It's All About*, that HDFS handles failures by ensuring each block of filesystem data is replicated to a number of different hosts, usually 3. As we only have a single host and one DataNode in the pseudo-distributed mode, we change this value to `1`.
- The `mapred.job.tracker` variable holds the location of the JobTracker just like `dfs.default.name` holds the location of the NameNode. Because only MapReduce components need know this location, it is in `mapred-site.xml`.



You are free, of course, to change the port numbers used, though 9000 and 9001 are common conventions in Hadoop.

The network addresses for the NameNode and the JobTracker specify the ports on which the actual system requests should be directed. These are not user-facing locations, so don't bother pointing your web browser at them. There are web interfaces that we will look at shortly.

Configuring the base directory and formatting the filesystem

If the pseudo-distributed or fully distributed mode is chosen, there are two steps that need to be performed before we start our first Hadoop cluster.

1. Set the base directory where Hadoop files will be stored.
2. Format the HDFS filesystem.



To be precise, we don't need to change the default directory; but, as seen later, it's a good thing to think about it now.

Time for action – changing the base HDFS directory

Let's first set the base directory that specifies the location on the local filesystem under which Hadoop will keep all its data. Carry out the following steps:

1. Create a directory into which Hadoop will store its data:

```
$ mkdir /var/lib/hadoop
```

2. Ensure the directory is writeable by any user:

```
$ chmod 777 /var/lib/hadoop
```

3. Modify `core-site.xml` once again to add the following property:

```
<property>
<name>hadoop.tmp.dir</name>
<value>/var/lib/hadoop</value>
</property>
```

What just happened?

As we will be storing data in Hadoop and all the various components are running on our local host, this data will need to be stored on our local filesystem somewhere. Regardless of the mode, Hadoop by default uses the `hadoop.tmp.dir` property as the base directory under which all files and data are written.

MapReduce, for example, uses a `/mapred` directory under this base directory; HDFS uses `/dfs`. The danger is that the default value of `hadoop.tmp.dir` is `/tmp` and some Linux distributions delete the contents of `/tmp` on each reboot. So it's safer to explicitly state where the data is to be held.

Time for action – formatting the NameNode

Before starting Hadoop in either pseudo-distributed or fully distributed mode for the first time, we need to format the HDFS filesystem that it will use. Type the following:

```
$  hadoop namenode -format
```

The output of this should look like the following:

```
$ hadoop namenode -format
12/10/26 22:45:25 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = vm193/10.0.0.193
STARTUP_MSG:   args = [-format]
…
12/10/26 22:45:25 INFO namenode.FSNamesystem: fsOwner=hadoop,hadoop
12/10/26 22:45:25 INFO namenode.FSNamesystem: supergroup=supergroup
12/10/26 22:45:25 INFO namenode.FSNamesystem: isPermissionEnabled=true
12/10/26 22:45:25 INFO common.Storage: Image file of size 96 saved in 0
seconds.
```

```
12/10/26 22:45:25 INFO common.Storage: Storage directory /var/lib/hadoop-
hadoop/dfs/name has been successfully formatted.
12/10/26 22:45:26 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at vm193/10.0.0.193
$
```

What just happened?

This is not a very exciting output because the step is only an enabler for our future use of HDFS. However, it does help us think of HDFS as a filesystem; just like any new storage device on any operating system, we need to format the device before we can use it. The same is true for HDFS; initially there is a default location for the filesystem data but no actual data for the equivalents of filesystem indexes.



Do this every time!

If your experience with Hadoop has been similar to the one I have had, there will be a series of simple mistakes that are frequently made when setting up new installations. It is very easy to forget about the formatting of the NameNode and then get a cascade of failure messages when the first Hadoop activity is tried.

But do it only once!

The command to format the NameNode can be executed multiple times, but in doing so all existing filesystem data will be destroyed. It can only be executed when the Hadoop cluster is shut down and sometimes you will want to do it but in most other cases it is a quick way to irrevocably delete every piece of data on HDFS; it does take much longer on large clusters. So be careful!

Starting and using Hadoop

After all that configuration and setup, let's now start our cluster and actually do something with it.

Time for action – starting Hadoop

Unlike the local mode of Hadoop, where all the components run only for the lifetime of the submitted job, with the pseudo-distributed or fully distributed mode of Hadoop, the cluster components exist as long-running processes. Before we use HDFS or MapReduce, we need to start up the needed components. Type the following commands; the output should look as shown next, where the commands are included on the lines prefixed by `$`:

1. Type in the first command:

```
$ start-dfs.sh
starting namenode, logging to /home/hadoop/hadoop/bin/../logs/
hadoop-hadoop-namenode-vm193.out
localhost: starting datanode, logging to /home/hadoop/hadoop/
bin/../logs/hadoop-hadoop-datanode-vm193.out
localhost: starting secondarynamenode, logging to /home/hadoop/
hadoop/bin/../logs/hadoop-hadoop-secondarynamenode-vm193.out
```

2. Type in the second command:

```
$ jps
9550 DataNode
9687 Jps
9638 SecondaryNameNode
9471 NameNode
```

3. Type in the third command:

```
$ hadoop dfs -ls /
Found 2 items
drwxr-xr-x   - hadoop supergroup          0 2012-10-26 23:03 /tmp
drwxr-xr-x   - hadoop supergroup          0 2012-10-26 23:06 /user
```

4. Type in the fourth command:

```
$ start-mapred.sh
starting jobtracker, logging to /home/hadoop/hadoop/bin/../logs/
hadoop-hadoop-jobtracker-vm193.out
localhost: starting tasktracker, logging to /home/hadoop/hadoop/
bin/../logs/hadoop-hadoop-tasktracker-vm193.out
```

5. Type in the fifth command:

```
$ jps
9550 DataNode
9877 TaskTracker
9638 SecondaryNameNode
9471 NameNode
9798 JobTracker
9913 Jps
```

What just happened?

The `start-dfs.sh` command, as the name suggests, starts the components necessary for HDFS. This is the NameNode to manage the filesystem and a single DataNode to hold data. The SecondaryNameNode is an availability aid that we'll discuss in a later chapter.

After starting these components, we use the JDK's `jps` utility to see which Java processes are running, and, as the output looks good, we then use Hadoop's `dfs` utility to list the root of the HDFS filesystem.

After this, we use `start-mapred.sh` to start the MapReduce components—this time the JobTracker and a single TaskTracker—and then use `jps` again to verify the result.

There is also a combined `start-all.sh` file that we'll use at a later stage, but in the early days it's useful to do a two-stage start up to more easily verify the cluster configuration.

Time for action – using HDFS

As the preceding example shows, there is a familiar-looking interface to HDFS that allows us to use commands similar to those in Unix to manipulate files and directories on the filesystem. Let's try it out by typing the following commands:

Type in the following commands:

```
$ hadoop -mkdir /user
$ hadoop -mkdir /user/hadoop
$ hadoop fs -ls /user
Found 1 items
drwxr-xr-x   - hadoop supergroup          0 2012-10-26 23:09 /user/Hadoop
$ echo "This is a test." >> test.txt
$ cat test.txt
This is a test.
$ hadoop dfs -copyFromLocal test.txt  .
$ hadoop dfs -ls
Found 1 items
-rw-r--r--   1 hadoop supergroup         16 2012-10-26 23:19/user/hadoop/
test.txt
$ hadoop dfs -cat test.txt
This is a test.
$ rm test.txt
$ hadoop dfs -cat test.txt
```

```
This is a test.
$ hadoop fs -copyToLocal test.txt
$ cat test.txt
This is a test.
```

What just happened?

This example shows the use of the `fs` subcommand to the Hadoop utility. Note that both `dfs` and `fs` commands are equivalent). Like most filesystems, Hadoop has the concept of a home directory for each user. These home directories are stored under the `/user` directory on HDFS and, before we go further, we create our home directory if it does not already exist.

We then create a simple text file on the local filesystem and copy it to HDFS by using the `copyFromLocal` command and then check its existence and contents by using the `-ls` and `-cat` utilities. As can be seen, the user home directory is aliased to `.` because, in Unix, `-ls` commands with no path specified are assumed to refer to that location and relative paths (not starting with `/`) will start there.

We then deleted the file from the local filesystem, copied it back from HDFS by using the `-copyToLocal` command, and checked its contents using the local `cat` utility.



Mixing HDFS and local filesystem commands, as in the preceding example, is a powerful combination, and it's very easy to execute on HDFS commands that were intended for the local filesystem and vice versa. So be careful, especially when deleting.

There are other HDFS manipulation commands; try `Hadoop fs -help` for a detailed list.

Time for action – WordCount, the Hello World of MapReduce

Many applications, over time, acquire a canonical example that no beginner's guide should be without. For Hadoop, this is WordCount – an example bundled with Hadoop that counts the frequency of words in an input text file.

1. First execute the following commands:

```
$ hadoop dfs -mkdir data
$ hadoop dfs -cp test.txt data
$ hadoop dfs -ls data
Found 1 items
-rw-r--r--   1 hadoop supergroup         16 2012-10-26 23:20 /
user/hadoop/data/test.txt
```

2. Now execute these commands:

```
$ Hadoop Hadoop/hadoop-examples-1.0.4.jar  wordcount data out
12/10/26 23:22:49 INFO input.FileInputFormat: Total input paths to
process : 1
12/10/26 23:22:50 INFO mapred.JobClient: Running job:
job_201210262315_0002
12/10/26 23:22:51 INFO mapred.JobClient:  map 0% reduce 0%
12/10/26 23:23:03 INFO mapred.JobClient:  map 100% reduce 0%
12/10/26 23:23:15 INFO mapred.JobClient:  map 100% reduce 100%
12/10/26 23:23:17 INFO mapred.JobClient: Job complete:
job_201210262315_0002
12/10/26 23:23:17 INFO mapred.JobClient: Counters: 17
12/10/26 23:23:17 INFO mapred.JobClient:   Job Counters
12/10/26 23:23:17 INFO mapred.JobClient:     Launched reduce
tasks=1
12/10/26 23:23:17 INFO mapred.JobClient:     Launched map tasks=1
12/10/26 23:23:17 INFO mapred.JobClient:     Data-local map
tasks=1
12/10/26 23:23:17 INFO mapred.JobClient:   FileSystemCounters
12/10/26 23:23:17 INFO mapred.JobClient:     FILE_BYTES_READ=46
12/10/26 23:23:17 INFO mapred.JobClient:     HDFS_BYTES_READ=16
12/10/26 23:23:17 INFO mapred.JobClient:     FILE_BYTES_
WRITTEN=124
12/10/26 23:23:17 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=24
12/10/26 23:23:17 INFO mapred.JobClient:   Map-Reduce Framework
12/10/26 23:23:17 INFO mapred.JobClient:     Reduce input groups=4
12/10/26 23:23:17 INFO mapred.JobClient:     Combine output
records=4
12/10/26 23:23:17 INFO mapred.JobClient:     Map input records=1
12/10/26 23:23:17 INFO mapred.JobClient:     Reduce shuffle
bytes=46
12/10/26 23:23:17 INFO mapred.JobClient:     Reduce output
records=4
12/10/26 23:23:17 INFO mapred.JobClient:     Spilled Records=8
12/10/26 23:23:17 INFO mapred.JobClient:     Map output bytes=32
12/10/26 23:23:17 INFO mapred.JobClient:     Combine input
records=4
12/10/26 23:23:17 INFO mapred.JobClient:     Map output records=4
12/10/26 23:23:17 INFO mapred.JobClient:     Reduce input
records=4
```

3. Execute the following command:

```
$ hadoop fs -ls out
Found 2 items
drwxr-xr-x   - hadoop supergroup          0 2012-10-26 23:22 /
user/hadoop/out/_logs
-rw-r--r--   1 hadoop supergroup         24 2012-10-26 23:23 /
user/hadoop/out/part-r-00000
```

4. Now execute this command:

```
$ hadoop fs -cat out/part-0-00000
This  1
a  1
is  1
test.  1
```

What just happened?

We did three things here, as follows:

- Moved the previously created text file into a new directory on HDFS
- Ran the example WordCount job specifying this new directory and a non-existent output directory as arguments
- Used the `fs` utility to examine the output of the MapReduce job

As we said earlier, the pseudo-distributed mode has more Java processes, so it may seem curious that the job output is significantly shorter than for the standalone Pi. The reason is that the local standalone mode prints information about each individual task execution to the screen, whereas in the other modes this information is written only to logfiles on the running hosts.

The output directory is created by Hadoop itself and the actual result files follow the part-*nnnnn* convention illustrated here; though given our setup, there is only one result file. We use the `fs -cat` command to examine the file, and the results are as expected.



If you specify an existing directory as the output source for a Hadoop job, it will fail to run and will throw an exception complaining of an already existing directory. If you want Hadoop to store the output to a directory, it must not exist. Treat this as a safety mechanism that stops Hadoop from writing over previous valuable job runs and something you will forget to ascertain frequently. If you are confident, you can override this behavior, as we will see later.

The Pi and WordCount programs are only some of the examples that ship with Hadoop. Here is how to get a list of them all. See if you can figure some of them out.

```
$ hadoop jar hadoop/hadoop-examples-1.0.4.jar
```

Have a go hero – WordCount on a larger body of text

Running a complex framework like Hadoop utilizing five discrete Java processes to count the words in a single-line text file is not terribly impressive. The power comes from the fact that we can use exactly the same program to run WordCount on a larger file, or even a massive corpus of text spread across a multinode Hadoop cluster. If we had such a setup, we would execute exactly the same commands as we just did by running the program and simply specifying the location of the directories for the source and output data.

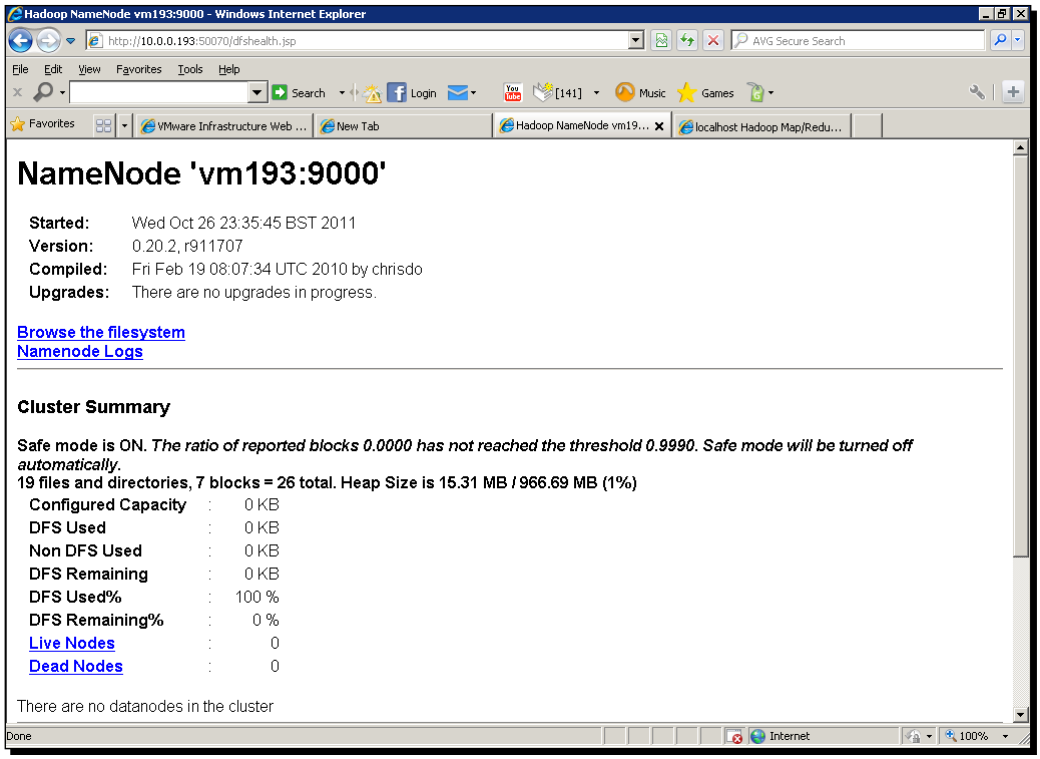
Find a large online text file—Project Gutenberg at `http://www.gutenberg.org` is a good starting point—and run WordCount on it by copying it onto the HDFS and executing the WordCount example. The output may not be as you expect because, in a large body of text, issues of dirty data, punctuation, and formatting will need to be addressed. Think about how WordCount could be improved; we'll study how to expand it into a more complex processing chain in the next chapter.

Monitoring Hadoop from the browser

So far, we have been relying on command-line tools and direct command output to see what our system is doing. Hadoop provides two web interfaces that you should become familiar with, one for HDFS and the other for MapReduce. Both are useful in pseudo-distributed mode and are critical tools when you have a fully distributed setup.

The HDFS web UI

Point your web browser to port 50030 on the host running Hadoop. By default, the web interface should be available from both the local host and any other machine that has network access. Here is an example screenshot:

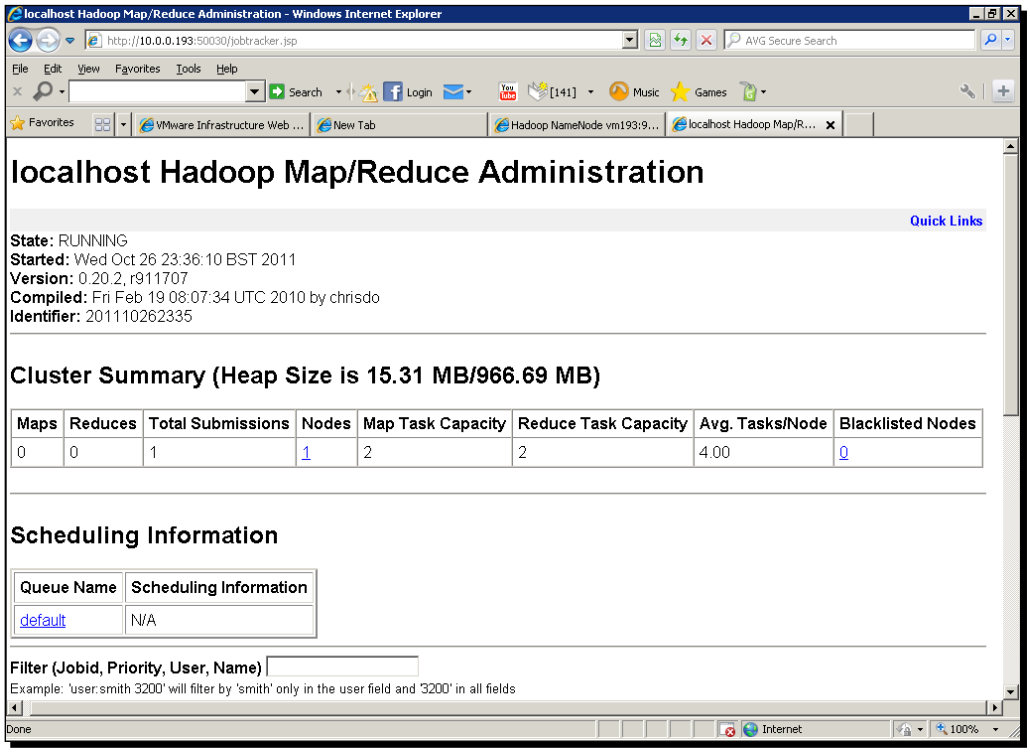


There is a lot going on here, but the immediately critical data tells us the number of nodes in the cluster, the filesystem size, used space, and links to drill down for more info and even browse the filesystem.

Spend a little time playing with this interface; it needs to become familiar. With a multinode cluster, the information about live and dead nodes plus the detailed information on their status history will be critical to debugging cluster problems.

The MapReduce web UI

The JobTracker UI is available on port 50070 by default, and the same access rules stated earlier apply. Here is an example screenshot:



This is more complex than the HDFS interface! Along with a similar count of the number of live/dead nodes, there is a history of the number of jobs executed since startup and a breakdown of their individual task counts.

The list of executing and historical jobs is a doorway to much more information; for every job, we can access the history of every task attempt on every node and access logs for detailed information. We now expose one of the most painful parts of working with any distributed system: debugging. It can be really hard.

Imagine you have a cluster of 100 machines trying to process a massive data set where the full job requires each host to execute hundreds of map and reduce tasks. If the job starts running very slowly or explicitly fails, it is not always obvious where the problem lies. Looking at the MapReduce web UI will likely be the first port of call because it provides such a rich starting point to investigate the health of running and historical jobs.

Using Elastic MapReduce

We will now turn to Hadoop in the cloud, the Elastic MapReduce service offered by Amazon Web Services. There are multiple ways to access EMR, but for now we will focus on the provided web console to contrast a full point-and-click approach to Hadoop with the previous command-line-driven examples.

Setting up an account in Amazon Web Services

Before using Elastic MapReduce, we need to set up an Amazon Web Services account and register it with the necessary services.

Creating an AWS account

Amazon has integrated their general accounts with AWS, meaning that if you already have an account for any of the Amazon retail websites, this is the only account you will need to use AWS services.

Note that AWS services have a cost; you will need an active credit card associated with the account to which charges can be made.

If you require a new Amazon account, go to `http://aws.amazon.com`, select **create a new AWS account**, and follow the prompts. Amazon has added a free tier for some services, so you may find that in the early days of testing and exploration you are keeping many of your activities within the non-charged tier. The scope of the free tier has been expanding, so make sure you know for what you will and won't be charged.

Signing up for the necessary services

Once you have an Amazon account, you will need to register it for use with the required AWS services, that is, **Simple Storage Service** (**S3**), **Elastic Compute Cloud** (**EC2**), and **Elastic MapReduce** (**EMR**). There is no cost for simply signing up to any AWS service; the process just makes the service available to your account.

Go to the S3, EC2, and EMR pages linked from `http://aws.amazon.com` and click on the **Sign up** button on each page; then follow the prompts.



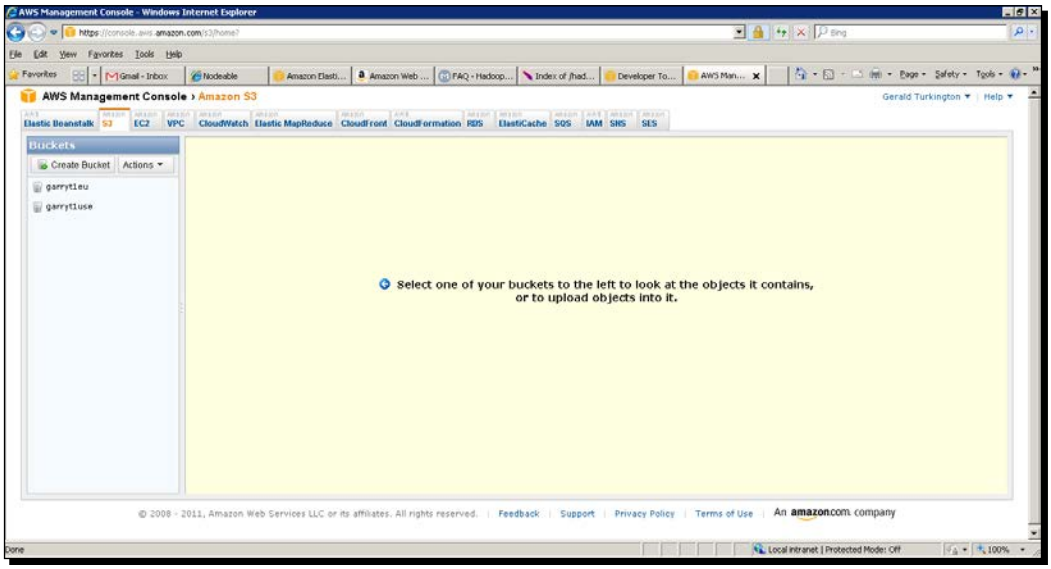
Caution! This costs real money!

Before going any further, it is critical to understand that use of AWS services will incur charges that will appear on the credit card associated with your Amazon account. Most of the charges are quite small and increase with the amount of infrastructure consumed; storing 10 GB of data in S3 costs 10 times more than for 1 GB, and running 20 EC2 instances costs 20 times as much as a single one. There are tiered cost models, so the actual costs tend to have smaller marginal increases at higher levels. But you should read carefully through the pricing sections for each service before using any of them. Note also that currently data transfer out of AWS services, such as EC2 and S3, is chargeable but data transfer between services is not. This means it is often most cost-effective to carefully design your use of AWS to keep data within AWS through as much of the data processing as possible.

Time for action – WordCount on EMR using the management console

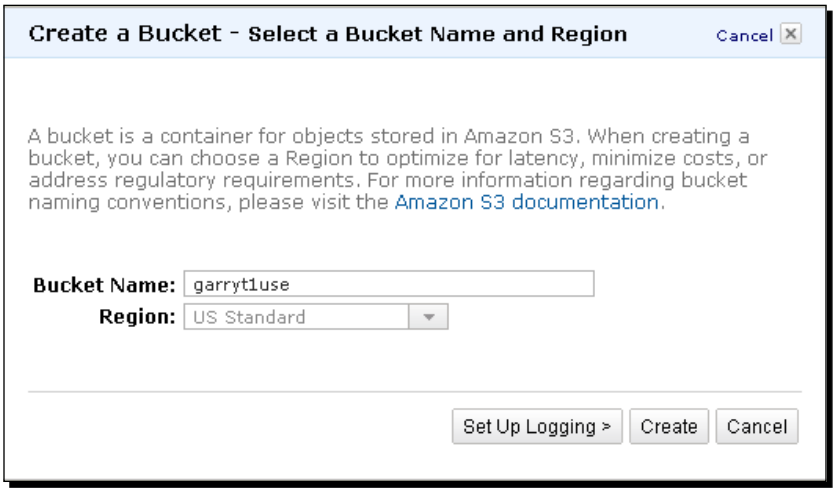
Let's jump straight into an example on EMR using some provided example code. Carry out the following steps:

1. Browse to `http://aws.amazon.com`, go to **Developers** | **AWS Management Console**, and then click on the **Sign in to the AWS Console** button. The default view should look like the following screenshot. If it does not, click on **Amazon S3** from within the console.

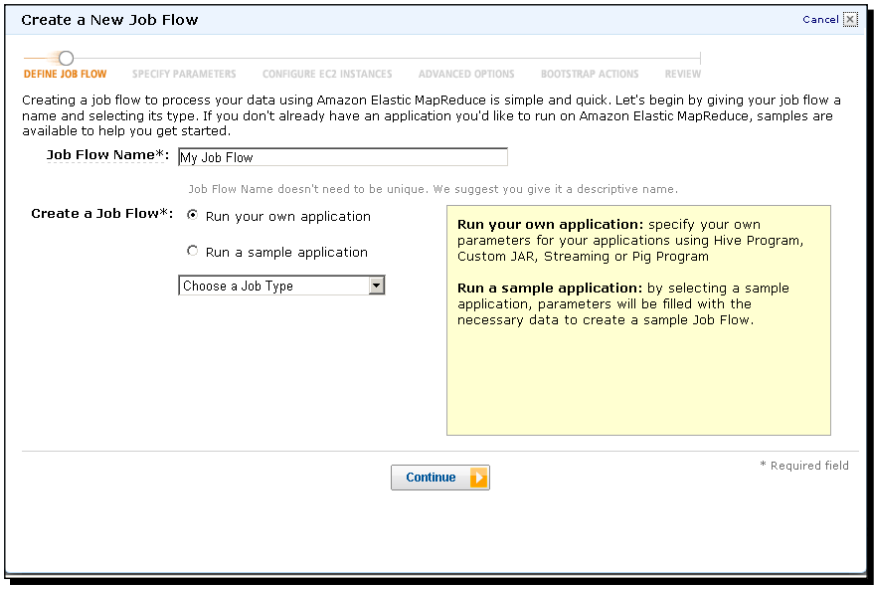


2. As shown in the preceding screenshot, click on the **Create bucket** button and enter a name for the new bucket. Bucket names must be globally unique across all AWS users, so do not expect obvious bucket names such as `mybucket` or `s3test` to be available.

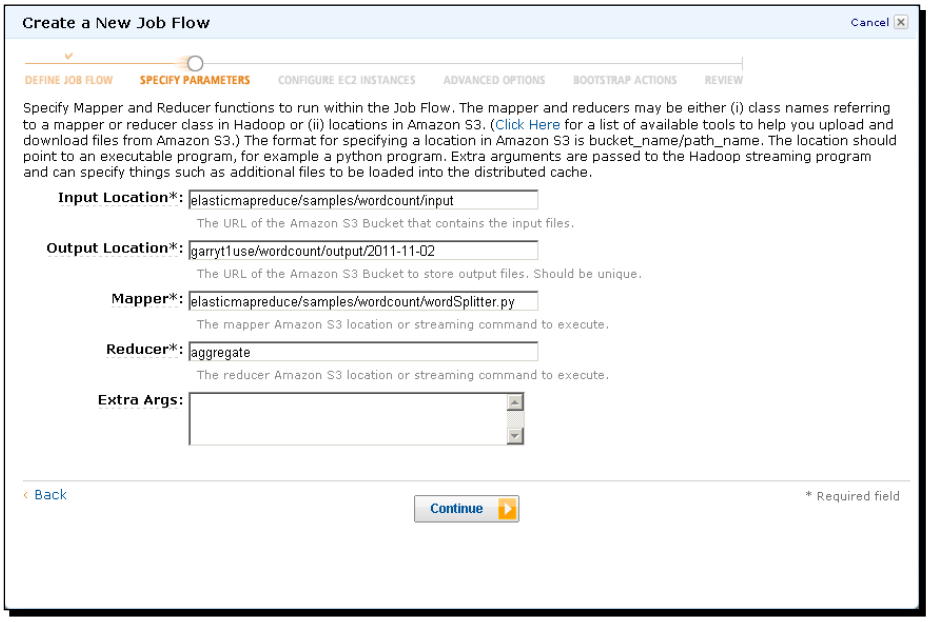
3. Click on the **Region** drop-down menu and select the geographic area nearest to you.



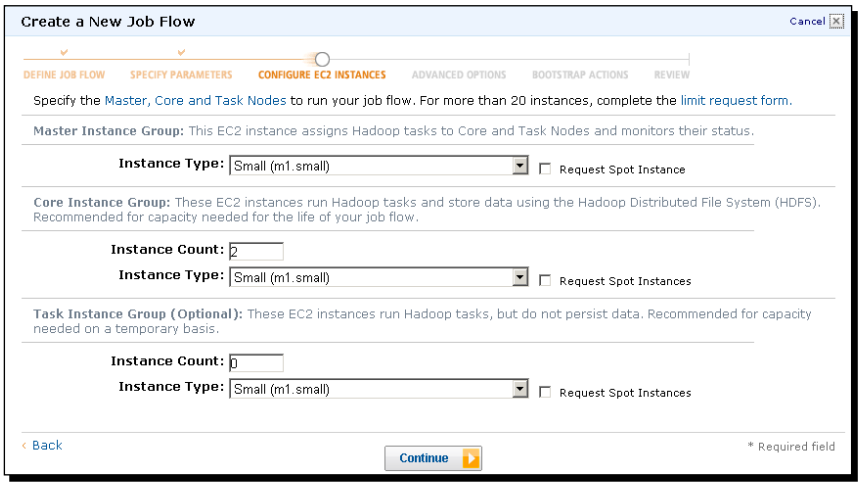
4. Click on the **Elastic MapReduce** link and click on the **Create a new Job Flow** button. You should see a screen like the following screenshot:



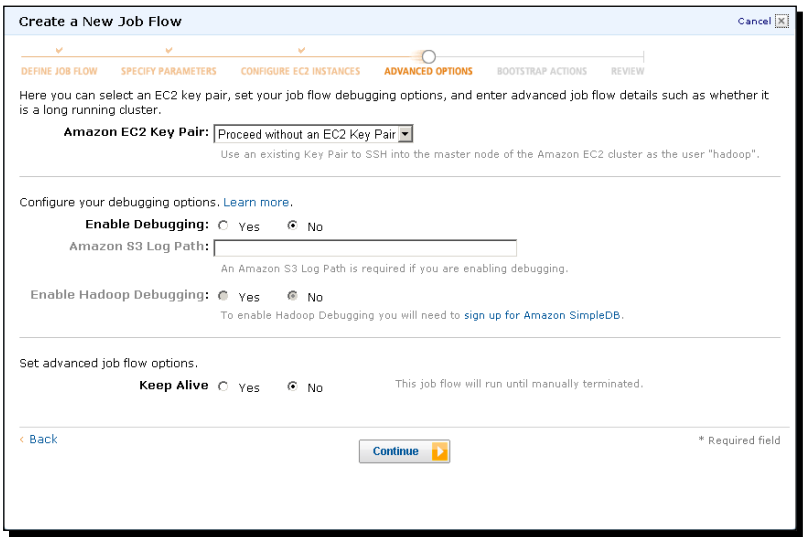
5. You should now see a screen like the preceding screenshot. Select the **Run a sample application** radio button and the **Word Count (Streaming)** menu item from the sample application drop-down box and click on the **Continue** button.



6. The next screen, shown in the preceding screenshot, allows us to specify the location of the output produced by running the job. In the edit box for the output location, enter the name of the bucket created in step 1 (`garryt1use` is the bucket we are using here); then click on the **Continue** button.



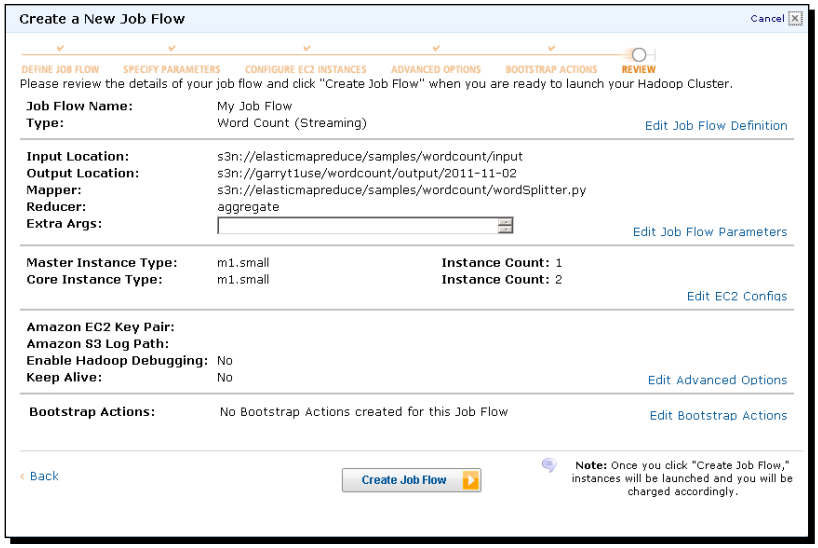
7. The next screenshot shows the page where we can modify the number and size of the virtual hosts utilized by our job. Confirm that the instance type for each combo box is **Small (m1.small)**, and the number of nodes for the Core group is **2** and for the Task group it is **0**. Then click on the **Continue** button.



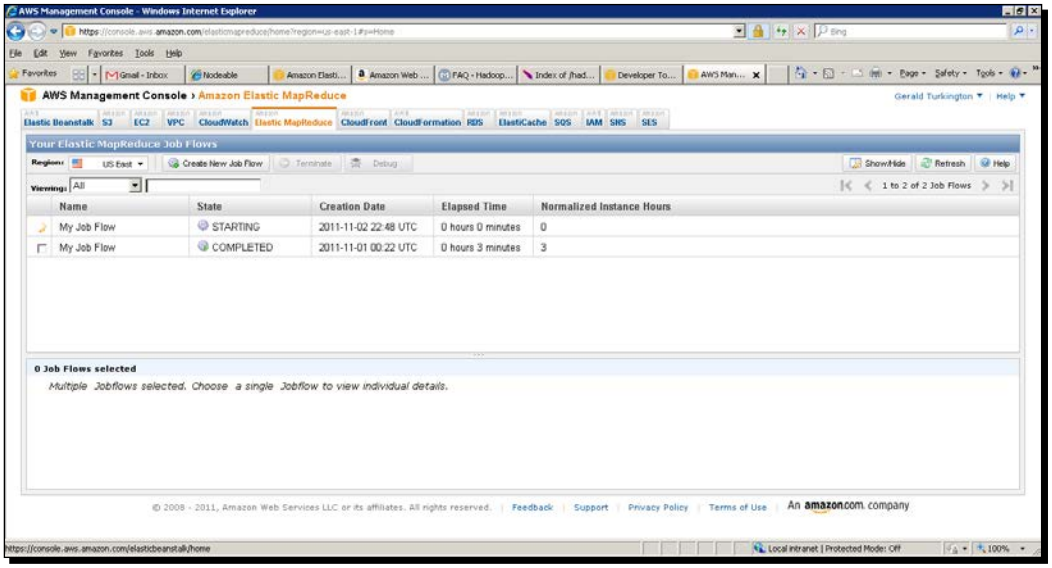
8. This next screenshot involves options we will not be using in this example. For the **Amazon EC2 key pair** field, select the **Proceed without key pair** menu item and click on the **No** radio button for the **Enable Debugging** field. Ensure that the **Keep Alive** radio button is set to **No** and click on the **Continue** button.



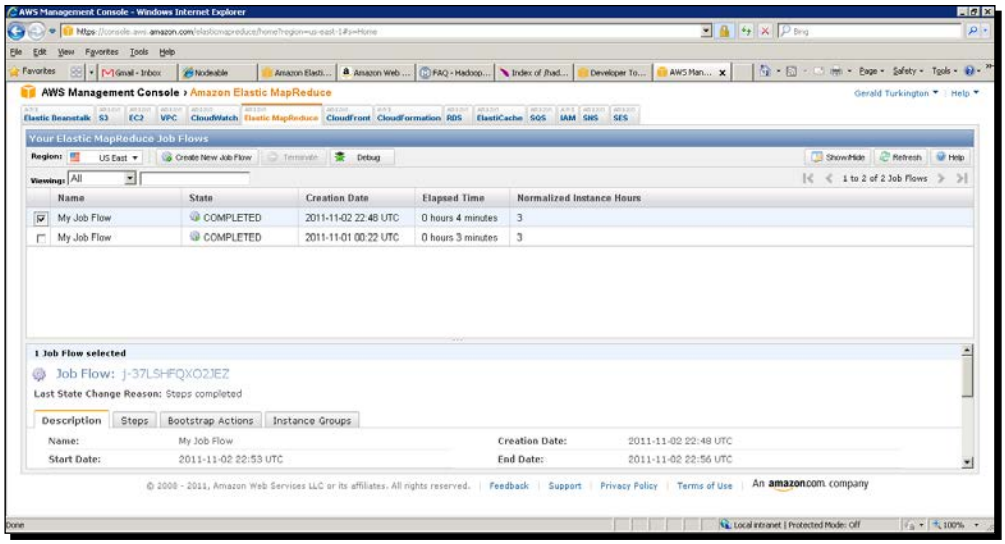
9. The next screen, shown in the preceding screenshot, is one we will not be doing much with right now. Confirm that the **Proceed with no Bootstrap Actions** radio button is selected and click on the **Continue** button.



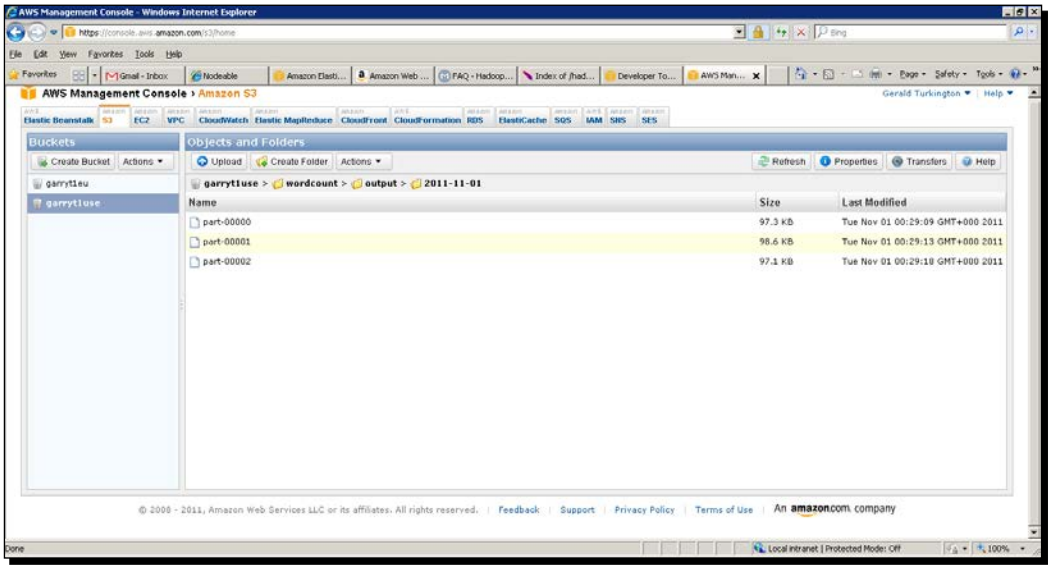
10. Confirm the job flow specifications are as expected and click on the **Create Job Flow** button. Then click on the **View my Job Flows** and **check status** buttons. This will give a list of your job flows; you can filter to show only running or completed jobs. The default is to show all, as in the example shown in the following screenshot:



11. Occasionally hit the **Refresh** button until the status of the listed job, **Running** or **Starting**, changes to **Complete**; then click its checkbox to see details of the job flow, as shown in the following screenshot:



12. Click the **S3** tab and select the bucket you created for the output location. You will see it has a single entry called **wordcount**, which is a directory. Right-click on that and select **Open**. Then do the same until you see a list of actual files following the familiar Hadoop part-*nnnnn* naming scheme, as shown in the following screenshot:



Right click on **part-00000** and open it. It should look something like this:

```
a               14716
aa              52
aakar           3
aargau          3
abad            3
abandoned       46
abandonment     6
abate           9
abauj           3
abbassid        4
abbes           3
abbl            3
…
```

Does this type of output look familiar?

What just happened?

The first step deals with S3, and not EMR. S3 is a scalable storage service that allows you to store files (called objects) within containers called buckets, and to access objects by their bucket and object key (that is, name). The model is analogous to the usage of a filesystem, and though there are underlying differences, they are unlikely to be important within this book.

S3 is where you will place the MapReduce programs and source data you want to process in EMR, and where the output and logs of EMR Hadoop jobs will be stored. There is a plethora of third-party tools to access S3, but here we are using the AWS management console, a browser interface to most AWS services.

Though we suggested you choose the nearest geographic region for S3, this is not required; non-US locations will typically give better latency for customers located nearer to them, but they also tend to have a slightly higher cost. The decision of where to host your data and applications is one you need to make after considering all these factors.

After creating the S3 bucket, we moved to the EMR console and created a new job flow. This term is used within EMR to refer to a data processing task. As we will see, this can be a one-time deal where the underlying Hadoop cluster is created and destroyed on demand or it can be a long-running cluster on which multiple jobs are executed.

We left the default job flow name and then selected the use of an example application, in this case, the Python implementation of WordCount. The term Hadoop Streaming refers to a mechanism allowing scripting languages to be used to write map and reduce tasks, but the functionality is the same as the Java WordCount we used earlier.

The form to specify the job flow requires a location for the source data, program, map and reduce classes, and a desired location for the output data. For the example we just saw, most of the fields were prepopulated; and, as can be seen, there are clear similarities to what was required when running local Hadoop from the command line.

By not selecting the **Keep Alive** option, we chose a Hadoop cluster that would be created specifically to execute this job, and destroyed afterwards. Such a cluster will have a longer startup time but will minimize costs. If you choose to keep the job flow alive, you will see additional jobs executed more quickly as you don't have to wait for the cluster to start up. But you will be charged for the underlying EC2 resources until you explicitly terminate the job flow.

After confirming, we do not need to add any additional bootstrap options; we selected the number and types of hosts we wanted to deploy into our Hadoop cluster. EMR distinguishes between three different groups of hosts:

- **Master group**: This is a controlling node hosting the NameNode and the JobTracker. There is only 1 of these.
- **Core group**: These are nodes running both HDFS DataNodes and MapReduce TaskTrackers. The number of hosts is configurable.
- **Task group**: These hosts don't hold HDFS data but do run TaskTrackers and can provide more processing horsepower. The number of hosts is configurable.

The type of host refers to different classes of hardware capability, the details of which can be found on the EC2 page. Larger hosts are more powerful but have a higher cost. Currently, by default, the total number of hosts in a job flow must be 20 or less, though Amazon has a simple form to request higher limits.

After confirming, all is as expected—we launch the job flow and monitor it on the console until the status changes to **COMPLETED**. At this point, we go back to S3, look inside the bucket we specified as the output destination, and examine the output of our WordCount job, which should look very similar to the output of a local Hadoop WordCount.

An obvious question is where did the source data come from? This was one of the prepopulated fields in the job flow specification we saw during the creation process. For nonpersistent job flows, the most common model is for the source data to be read from a specified S3 source location and the resulting data written to the specified result S3 bucket.

That is it! The AWS management console allows fine-grained control of services such as S3 and EMR from the browser. Armed with nothing more than a browser and a credit card, we can launch Hadoop jobs to crunch data without ever having to worry about any of the mechanics around installing, running, or managing Hadoop.

Have a go hero – other EMR sample applications

EMR provides several other sample applications. Why not try some of them as well?

Other ways of using EMR

Although a powerful and impressive tool, the AWS management console is not always how we want to access S3 and run EMR jobs. As with all AWS services, there are both programmatic and command-line tools to use the services.

AWS credentials

Before using either programmatic or command-line tools, however, we need to look at how an account holder authenticates for AWS to make such requests. As these are chargeable services, we really do not want anyone else to make requests on our behalf. Note that as we logged directly into the AWS management console with our AWS account in the preceding example, we did not have to worry about this.

Each AWS account has several identifiers that are used when accessing the various services:

- **Account ID**: Each AWS account has a numeric ID.
- **Access key**: Each account has an associated access key that is used to identify the account making the request.
- **Secret access key**: The partner to the access key is the secret access key. The access key is not a secret and could be exposed in service requests, but the secret access key is what you use to validate yourself as the account owner.
- **Key pairs**: These are the key pairs used to log in to EC2 hosts. It is possible to either generate public/private key pairs within EC2 or to import externally generated keys into the system.

If this sounds confusing, it's because it is. At least at first. When using a tool to access an AWS service, however, there's usually a single up-front step of adding the right credentials to a configured file, and then everything just works. However, if you do decide to explore programmatic or command-line tools, it will be worth a little time investment to read the documentation for each service to understand how its security works.

The EMR command-line tools

In this book, we will not do anything with S3 and EMR that cannot be done from the AWS management console. However, when working with operational workloads, looking to integrate into other workflows, or automating service access, a browser-based tool is not appropriate, regardless of how powerful it is. Using the direct programmatic interfaces to a service provides the most granular control but requires the most effort.

Amazon provides for many services a group of command-line tools that provide a useful way of automating access to AWS services that minimizes the amount of required development. The Elastic MapReduce command-line tools, linked from the main EMR page, are worth a look if you want a more CLI-based interface to EMR but don't want to write custom code just yet.

The AWS ecosystem

Each AWS service also has a plethora of third-party tools, services, and libraries that can provide different ways of accessing the service, provide additional functionality, or offer new utility programs. Check out the developer tools hub at `http://aws.amazon.com/developertools`, as a starting point.

Comparison of local versus EMR Hadoop

After our first experience of both a local Hadoop cluster and its equivalent in EMR, this is a good point at which we can consider the differences of the two approaches.

As may be apparent, the key differences are not really about capability; if all we want is an environment to run MapReduce jobs, either approach is completely suited. Instead, the distinguishing characteristics revolve around a topic we touched on in *Chapter 1*, *What It's All About*, that being whether you prefer a cost model that involves upfront infrastructure costs and ongoing maintenance effort over one with a pay-as-you-go model with a lower maintenance burden along with rapid and conceptually infinite scalability. Other than the cost decisions, there are a few things to keep in mind:

- EMR supports specific versions of Hadoop and has a policy of upgrading over time. If you have a need for a specific version, in particular if you need the latest and greatest versions immediately after release, then the lag before these are live on EMR may be unacceptable.
- You can start up a persistent EMR job flow and treat it much as you would a local Hadoop cluster, logging into the hosting nodes and tweaking their configuration. If you find yourself doing this, its worth asking if that level of control is really needed and, if so, is it stopping you getting all the cost model benefits of a move to EMR?
- If it does come down to a cost consideration, remember to factor in all the hidden costs of a local cluster that are often forgotten. Think about the costs of power, space, cooling, and facilities. Not to mention the administration overhead, which can be nontrivial if things start breaking in the early hours of the morning.

Summary

We covered a lot of ground in this chapter, in regards to getting a Hadoop cluster up and running and executing MapReduce programs on it.

Specifically, we covered the prerequisites for running Hadoop on local Ubuntu hosts. We also saw how to install and configure a local Hadoop cluster in either standalone or pseudo-distributed modes. Then, we looked at how to access the HDFS filesystem and submit MapReduce jobs. We then moved on and learned what accounts are needed to access Elastic MapReduce and other AWS services.

We saw how to browse and create S3 buckets and objects using the AWS management console, and also how to create a job flow and use it to execute a MapReduce job on an EMR-hosted Hadoop cluster. We also discussed other ways of accessing AWS services and studied the differences between local and EMR-hosted Hadoop.

Now that we have learned about running Hadoop locally or on EMR, we are ready to start writing our own MapReduce programs, which is the topic of the next chapter.

3
Understanding MapReduce

The previous two chapters have discussed the problems that Hadoop allows us to solve, and gave some hands-on experience of running example MapReduce jobs. With this foundation, we will now go a little deeper.

In this chapter we will be:

- Understanding how key/value pairs are the basis of Hadoop tasks
- Learning the various stages of a MapReduce job
- Examining the workings of the map, reduce, and optional combined stages in detail
- Looking at the Java API for Hadoop and use it to develop some simple MapReduce jobs
- Learning about Hadoop input and output

Key/value pairs

Since *Chapter 1*, *What It's All About*, we have been talking about operations that process and provide the output in terms of key/value pairs without explaining why. It is time to address that.

What it mean

Firstly, we will clarify just what we mean by key/value pairs by highlighting similar concepts in the Java standard library. The `java.util.Map` interface is the parent of commonly used classes such as `HashMap` and (through some library backward reengineering) even the original `Hashtable`.

For any Java `Map` object, its contents are a set of mappings from a given key of a specified type to a related value of a potentially different type. A `HashMap` object could, for example, contain mappings from a person's name (`String`) to his or her birthday (`Date`).

In the context of Hadoop, we are referring to data that also comprises keys that relate to associated values. This data is stored in such a way that the various values in the data set can be sorted and rearranged across a set of keys. If we are using key/value data, it will make sense to ask questions such as the following:

- Does a given key have a mapping in the data set?
- What are the values associated with a given key?
- What is the complete set of keys?

Think back to WordCount from the previous chapter. We will go into it in more detail shortly, but the output of the program is clearly a set of key/value relationships; for each word (the key), there is a count (the value) of its number of occurrences. Think about this simple example and some important features of key/value data will become apparent, as follows:

- Keys must be unique but values need not be
- Each value must be associated with a key, but a key could have no values (though not in this particular example)
- Careful definition of the key is important; deciding on whether or not the counts are applied with case sensitivity will give different results



Note that we need to define carefully what we mean by keys being unique here. This does not mean the key occurs only once; in our data set we may see a key occur numerous times and, as we shall see, the MapReduce model has a stage where all values associated with each key are collected together. The uniqueness of keys guarantees that if we collect together every value seen for any given key, the result will be an association from a single instance of the key to every value mapped in such a way, and none will be omitted.

Why key/value data?

Using key/value data as the foundation of MapReduce operations allows for a powerful programming model that is surprisingly widely applicable, as can be seen by the adoption of Hadoop and MapReduce across a wide variety of industries and problem scenarios. Much data is either intrinsically key/value in nature or can be represented in such a way. It is a simple model with broad applicability and semantics straightforward enough that programs defined in terms of it can be applied by a framework like Hadoop.

Of course, the data model itself is not the only thing that makes Hadoop useful; its real power lies in how it uses the techniques of parallel execution, and divide and conquer discussed in *Chapter 1*, *What It's All About*. We can have a large number of hosts on which we can store and execute data and even use a framework that manages the division of the larger task into smaller chunks, and the combination of partial results into the overall answer. But we need this framework to provide us with a way of expressing our problems that doesn't require us to be an expert in the execution mechanics; we want to express the transformations required on our data and then let the framework do the rest. MapReduce, with its key/value interface, provides such a level of abstraction, whereby the programmer only has to specify these transformations and Hadoop handles the complex process of applying this to arbitrarily large data sets.

Some real-world examples

To become less abstract, let's think of some real-world data that is key/value pair:

- An address book relates a name (key) to contact information (value)
- A bank account uses an account number (key) to associate with the account details (value)
- The index of a book relates a word (key) to the pages on which it occurs (value)
- On a computer filesystem, filenames (keys) allow access to any sort of data, such as text, images, and sound (values)

These examples are intentionally broad in scope, to help and encourage you to think that key/value data is not some very constrained model used only in high-end data mining but a very common model that is all around us.

We would not be having this discussion if this was not important to Hadoop. The bottom line is that if the data can be expressed as key/value pairs, it can be processed by MapReduce.

MapReduce as a series of key/value transformations

You may have come across MapReduce described in terms of key/value transformations, in particular the intimidating one looking like this:

```
{K1,V1} -> {K2, List<V2>} -> {K3,V3}
```

We are now in a position to understand what this means:

- The input to the `map` method of a MapReduce job is a series of key/value pairs that we'll call `K1` and `V1`.

- The output of the `map` method (and hence input to the `reduce` method) is a series of keys and an associated list of values that are called `K2` and `V2`. Note that each mapper simply outputs a series of individual key/value outputs; these are combined into a key and list of values in the `shuffle` method.
- The final output of the MapReduce job is another series of key/value pairs, called `K3` and `V3`.

These sets of key/value pairs don't have to be different; it would be quite possible to input, say, names and contact details and output the same, with perhaps some intermediary format used in collating the information. Keep this three-stage model in mind as we explore the Java API for MapReduce next. We will first walk through the main parts of the API you will need and then do a systematic examination of the execution of a MapReduce job.

Pop quiz – key/value pairs

Q1. The concept of key/value pairs is…

1. Something created by and specific to Hadoop.
2. A way of expressing relationships we often see but don't think of as such.
3. An academic concept from computer science.

Q2. Are username/password combinations an example of key/value data?

1. Yes, it's a clear case of one value being associated to the other.
2. No, the password is more of an attribute of the username, there's no index-type relationship.
3. We'd not usually think of them as such, but Hadoop could still process a series of username/password combinations as key/value pairs.

The Hadoop Java API for MapReduce

Hadoop underwent a major API change in its 0.20 release, which is the primary interface in the 1.0 version we use in this book. Though the prior API was certainly functional, the community felt it was unwieldy and unnecessarily complex in some regards.

The new API, sometimes generally referred to as context objects, for reasons we'll see later, is the future of Java's MapReduce development; and as such we will use it wherever possible in this book. Note that caveat: there are parts of the pre-0.20 MapReduce libraries that have not been ported to the new API, so we will use the old interfaces when we need to examine any of these.

The 0.20 MapReduce Java API

The 0.20 and above versions of MapReduce API have most of the key classes and interfaces either in the `org.apache.hadoop.mapreduce` package or its subpackages.

In most cases, the implementation of a MapReduce job will provide job-specific subclasses of the `Mapper` and `Reducer` base classes found in this package.



We'll stick to the commonly used `K1` / `K2` / `K3` / and so on terminology, though more recently the Hadoop API has, in places, used terms such as `KEYIN`/`VALUEIN` and `KEYOUT`/`VALUEOUT` instead. For now, we will stick with `K1` / `K2` / `K3` as it helps understand the end-to-end data flow.

The Mapper class

This is a cut-down view of the base `Mapper` class provided by Hadoop. For our own mapper implementations, we will subclass this base class and override the specified method as follows:

```
class Mapper<K1, V1, K2, V2>
{
      void map(K1 key, V1 value Mapper.Context context)
            throws IOException, InterruptedException
{..}
}
```

Although the use of Java generics can make this look a little opaque at first, there is actually not that much going on. The class is defined in terms of the key/value input and output types, and then the `map` method takes an input key/value pair in its parameters. The other parameter is an instance of the `Context` class that provides various mechanisms to communicate with the Hadoop framework, one of which is to output the results of a `map` or `reduce` method.



Notice that the `map` method only refers to a single instance of `K1` and `V1` key/value pairs. This is a critical aspect of the MapReduce paradigm in which you write classes that process single records and the framework is responsible for all the work required to turn an enormous data set into a stream of key/value pairs. You will never have to write `map` or `reduce` classes that try to deal with the full data set. Hadoop also provides mechanisms through its `InputFormat` and `OutputFormat` classes that provide implementations of common file formats and likewise remove the need of having to write file parsers for any but custom file types.

There are three additional methods that sometimes may be required to be overridden.

```
protected void setup( Mapper.Context context)
      throws IOException, Interrupted Exception
```

This method is called once before any key/value pairs are presented to the `map` method. The default implementation does nothing.

```
protected void cleanup( Mapper.Context context)
      throws IOException, Interrupted Exception
```

This method is called once after all key/value pairs have been presented to the `map` method. The default implementation does nothing.

```
protected void run( Mapper.Context context)
      throws IOException, Interrupted Exception
```

This method controls the overall flow of task processing within a JVM. The default implementation calls the `setup` method once before repeatedly calling the `map` method for each key/value pair in the split, and then finally calls the `cleanup` method.



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.

The Reducer class

The `Reducer` base class works very similarly to the `Mapper` class, and usually requires only subclasses to override a single `reduce` method. Here is the cut-down class definition:

```
public class Reducer<K2, V2, K3, V3>
{
void reduce(K1 key, Iterable<V2> values,
      Reducer.Context context)
        throws IOException, InterruptedException
{..}
}
```

Again, notice the class definition in terms of the broader data flow (the `reduce` method accepts `K2`/`V2` as input and provides `K3`/`V3` as output) while the actual reduce method takes only a single key and its associated list of values. The `Context` object is again the mechanism to output the result of the method.

This class also has the `setup`, `run`, and `cleanup` methods with similar default implementations as with the `Mapper` class that can optionally be overridden:

```
protected void setup( Reduce.Context context)
throws IOException, InterruptedException
```

This method is called once before any key/lists of values are presented to the `reduce` method. The default implementation does nothing.

```
protected void cleanup( Reducer.Context context)
throws IOException, InterruptedException
```

This method is called once after all key/lists of values have been presented to the `reduce` method. The default implementation does nothing.

```
protected void run( Reducer.Context context)
throws IOException, InterruptedException
```

This method controls the overall flow of processing the task within JVM. The default implementation calls the `setup` method before repeatedly calling the `reduce` method for as many key/values provided to the `Reducer` class, and then finally calls the `cleanup` method.

The Driver class

Although our mapper and reducer implementations are all we need to perform the MapReduce job, there is one more piece of code required: the driver that communicates with the Hadoop framework and specifies the configuration elements needed to run a MapReduce job. This involves aspects such as telling Hadoop which `Mapper` and `Reducer` classes to use, where to find the input data and in what format, and where to place the output data and how to format it. There is an additional variety of other configuration options that can be set and which we will see throughout this book.

There is no default parent Driver class as a subclass; the driver logic usually exists in the main method of the class written to encapsulate a MapReduce job. Take a look at the following code snippet as an example driver. Don't worry about how each line works, though you should be able to work out generally what each is doing:

```
public class ExampleDriver
{
...
public static void main(String[] args) throws Exception
{
// Create a Configuration object that is used to set other options
    Configuration conf = new Configuration() ;
// Create the object representing the job
Job job = new Job(conf, "ExampleJob") ;
// Set the name of the main class in the job jarfile
    job.setJarByClass(ExampleDriver.class) ;
// Set the mapper class
    job.setMapperClass(ExampleMapper.class) ;
```

```
// Set the reducer class
    job.setReducerClass(ExampleReducer.class) ;
// Set the types for the final output key and value
    job.setOutputKeyClass(Text.class) ;
    job.setOutputValueClass(IntWritable.class) ;
// Set input and output file paths
FileInputFormat.addInputPath(job, new Path(args[0])) ;
FileOutputFormat.setOutputPath(job, new Path(args[1]))
// Execute the job and wait for it to complete
 System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}}
```

Given our previous talk of jobs, it is not surprising that much of the setup involves operations on a `Job` object. This includes setting the job name and specifying which classes are to be used for the mapper and reducer implementations.

Certain input/output configurations are set and, finally, the arguments passed to the main method are used to specify the input and output locations for the job. This is a very common model that you will see often.

There are a number of default values for configuration options, and we are implicitly using some of them in the preceding class. Most notably, we don't say anything about the file format of the input files or how the output files are to be written. These are defined through the `InputFormat` and `OutputFormat` classes mentioned earlier; we will explore them in detail later. The default input and output formats are text files that suit our WordCount example. There are multiple ways of expressing the format within text files in addition to particularly optimized binary formats.

A common model for less complex MapReduce jobs is to have the `Mapper` and `Reducer` classes as inner classes within the driver. This allows everything to be kept in a single file, which simplifies the code distribution.

Writing MapReduce programs

We have been using and talking about WordCount for quite some time now; let's actually write an implementation, compile, and run it, and then explore some modifications.

Time for action – setting up the classpath

To compile any Hadoop-related code, we will need to refer to the standard Hadoop-bundled classes.

Add the `Hadoop-1.0.4.core.jar` file from the distribution to the Java classpath as follows:

```
$ export CLASSPATH=.:${HADOOP_HOME}/Hadoop-1.0.4.core.jar:${CLASSPATH}
```

What just happened?

This adds the `Hadoop-1.0.4.core.jar` file explicitly to the classpath alongside the current directory and the previous contents of the CLASSPATH environment variable.

Once again, it would be good to put this in your shell startup file or a standalone file to be sourced.



We will later need to also have many of the supplied third-party libraries that come with Hadoop on our classpath, and there is a shortcut to do this. For now, the explicit addition of the core JAR file will suffice.

Time for action – implementing WordCount

We have seen the use of the WordCount example program in *Chapter 2*, *Getting Hadoop Up and Running*. Now we will explore our own Java implementation by performing the following steps:

1. Enter the following code into the `WordCount1.java` file:

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

```
public class WordCount1
{


    public static class WordCountMapper
    extends Mapper<Object, Text, Text, IntWritable>
{

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

        public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {
            String[] words = value.toString().split(" ") ;

            for (String str: words)
            {
                word.set(str);
                context.write(word, one);
            }
        }
    }

    public static class WordCountReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
        public void reduce(Text key, Iterable<IntWritable> values,
            Context context
            ) throws IOException, InterruptedException {
                int total = 0;
            for (IntWritable val : values) {
                total++ ;
            }
            context.write(key, new IntWritable(total));
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "word count");
        job.setJarByClass(WordCount1.class);
```

```
            job.setMapperClass(WordCountMapper.class);
            job.setReducerClass(WordCountReducer.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);
            FileInputFormat.addInputPath(job, new Path(args[0]));
            FileOutputFormat.setOutputPath(job, new Path(args[1]));
            System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
    }
```

2. Now compile it by executing the following command:

```
$ javac WordCount1.java
```

What just happened?

This is our first complete MapReduce job. Look at the structure and you should recognize the elements we have previously discussed: the overall `Job` class with the driver configuration in its main method and the `Mapper` and `Reducer` implementations defined as inner classes.

We'll do a more detailed walkthrough of the mechanics of MapReduce in the next section, but for now let's look at the preceding code and think of how it realizes the key/value transformations we talked about earlier.

The input to the `Mapper` class is arguably the hardest to understand, as the key is not actually used. The job specifies TextInputFormat as the format of the input data and, by default, this delivers to the mapper data where the key is the line number in the file and the value is the text of that line. In reality, you may never actually see a mapper that uses that line number key, but it is provided.

The mapper is executed once for each line of text in the input source and every time it takes the line and breaks it into words. It then uses the `Context` object to output (more commonly known as emitting) each new key/value of the form `<word, 1>`. These are our `K2`/`V2` values.

We said before that the input to the reducer is a key and a corresponding list of values, and there is some magic that happens between the `map` and `reduce` methods to collect together the values for each key that facilitates this, which we'll not describe right now. Hadoop executes the reducer once for each key and the preceding reducer implementation simply counts the numbers in the `Iterable` object and gives output for each word in the form of `<word, count>`. This is our `K3`/`V3` values.

Take a look at the signatures of our `mapper` and `reducer` classes: the `WordCountMapper` class gives `IntWritable` and `Text` as input and gives `Text` and `IntWritable` as output. The `WordCountReducer` class gives `Text` and `IntWritable` both as input and output. This is again quite a common pattern, where the `map` method performs an inversion on the key and values, and instead emits a series of data pairs on which the reducer performs aggregation.

The driver is more meaningful here, as we have real values for the parameters. We use arguments passed to the class to specify the input and output locations.

Time for action – building a JAR file

Before we run our job in Hadoop, we must collect the required class files into a single JAR file that we will submit to the system.

Create a JAR file from the generated class files.

```
$ jar cvf wc1.jar WordCount1*class
```

What just happened?

We must always package our class files into a JAR file before submitting to Hadoop, be it local or on Elastic MapReduce.



Be careful with the JAR command and file paths. If you include in a JAR file class the files from a subdirectory, the class may not be stored with the path you expect. This is especially common when using a catch-all classes directory where all source data gets compiled. It may be useful to write a script to change into the directory, convert the required files into JAR files, and move the JAR files to the required location.

Time for action – running WordCount on a local Hadoop cluster

Now we have generated the class files and collected them into a JAR file, we can run the application by performing the following steps:

1. Submit the new JAR file to Hadoop for execution.

   ```
   $ hadoop jar wc1.jar WordCount1 test.txt output
   ```

2. If successful, you should see the output being very similar to the one we obtained when we ran the Hadoop-provided sample WordCount in the previous chapter. Check the output file; it should be as follows:

```
$ Hadoop fs -cat output/part-r-00000
This 1
yes 1
a 1
is 2
test 1
this 1
```

What just happened?

This is the first time we have used the Hadoop JAR command with our own code. There are four arguments:

1. The name of the JAR file.
2. The name of the driver class within the JAR file.
3. The location, on HDFS, of the input file (a relative reference to the `/user/Hadoop home` folder, in this case).
4. The desired location of the output folder (again, a relative path).



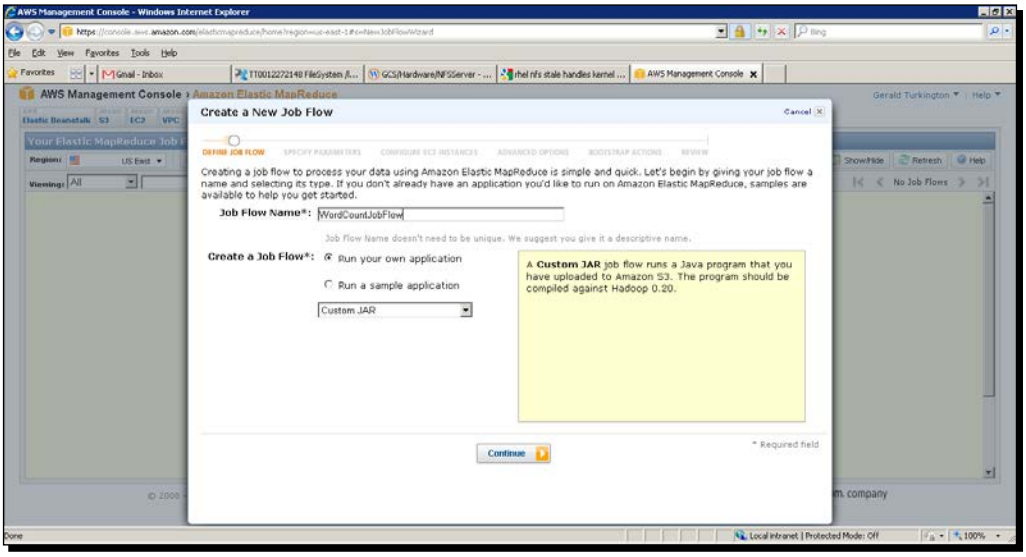
The name of the driver class is only required if a main class has not (as in this case) been specified within the JAR file manifest.

Time for action – running WordCount on EMR

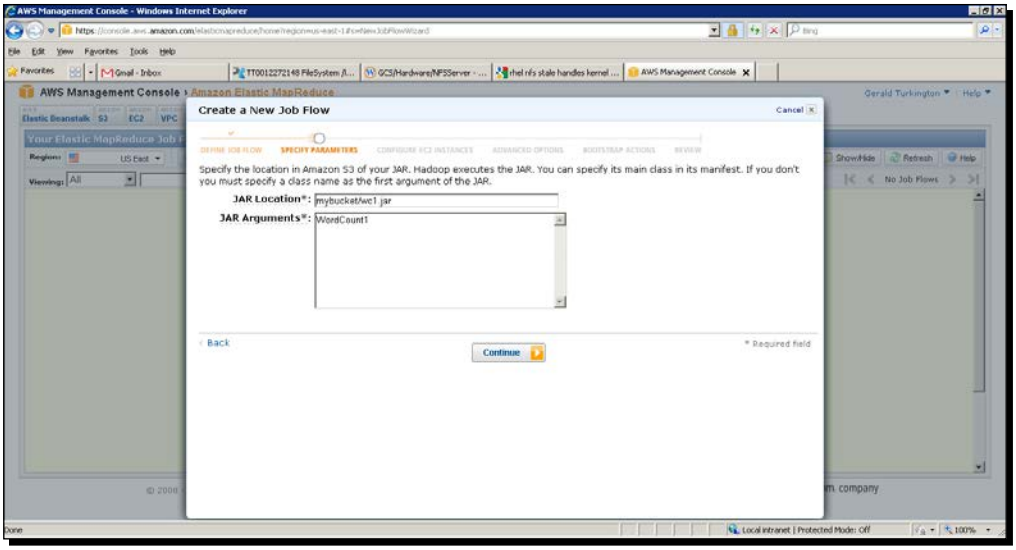
We will now show you how to run this same JAR file on EMR. Remember, as always, that this costs money!

1. Go to the AWS console at `http://aws.amazon.com/console`, sign in, and select **S3**.
2. You'll need two buckets: one to hold the JAR file and another for the job output. You can use existing buckets or create new ones.
3. Open the bucket where you will store the job file, click on **Upload**, and add the `wc1.jar` file created earlier.
4. Return to the main console home page, and then go to the EMR portion of the console by selecting **Elastic MapReduce**.

5. Click on the **Create a New Job Flow** button and you'll see a familiar screen as shown in the following screenshot:



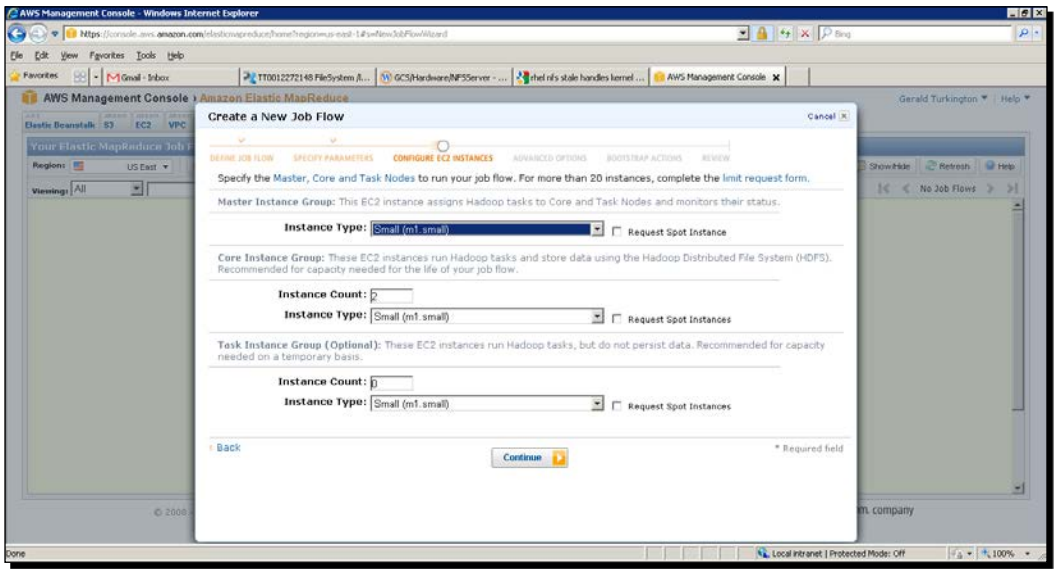
6. Previously, we used a sample application; to run our code, we need to perform different steps. Firstly, select the **Run your own application** radio button.
7. In the **Select a Job Type** combobox, select **Custom JAR**.
8. Click on the **Continue** button and you'll see a new form, as shown in the following screenshot:



We now specify the arguments to the job. Within our uploaded JAR file, our code—particularly the driver class—specifies aspects such as the `Mapper` and `Reducer` classes.

What we need to provide is the path to the JAR file and the input and output paths for the job. In the **JAR Location** field, put the location where you uploaded the JAR file. If the JAR file is called `wc1.jar` and you uploaded it into a bucket called `mybucket`, the path would be `mybucket/wc1.jar`.

In the **JAR Arguments** field, you need to enter the name of the main class and the input and output locations for the job. For files on **S3**, we can use URLs of the form `s3://bucketname/objectname`. Click on **Continue** and the familiar screen to specify the virtual machines for the job flow appears, as shown in the following screenshot:



Now continue through the job flow setup and execution as we did in *Chapter 2*, *Getting Hadoop Up and Running*.

What just happened?

The important lesson here is that we can reuse the code written on and for a local Hadoop cluster in EMR. Also, besides these first few steps, the majority of the EMR console is the same regardless of the source of the job code to be executed.

Through the remainder of this chapter, we will not explicitly show code being executed on EMR and will instead focus more on the local cluster, because running a JAR file on EMR is very easy.

The pre-0.20 Java MapReduce API

Our preference in this book is for the 0.20 and above versions of MapReduce Java API, but we'll need to take a quick look at the older APIs for two reasons:

1. Many online examples and other reference materials are written for the older APIs.
2. Several areas within the MapReduce framework are not yet ported to the new API, and we will need to use the older APIs to explore them.

The older API's classes are found primarily in the `org.apache.hadoop.mapred` package.

The new API classes use concrete `Mapper` and `Reducer` classes, while the older API had this responsibility split across abstract classes and interfaces.

An implementation of a `Mapper` class will subclass the abstract `MapReduceBase` class and implement the `Mapper` interface, while a custom `Reducer` class will subclass the same `MapReduceBase` abstract class but implement the `Reducer` interface.

We'll not explore `MapReduceBase` in much detail as its functionality deals with job setup and configuration, which aren't really core to understanding the `MapReduce` model. But the interfaces of pre-0.20 `Mapper` and `Reducer` are worth showing:

```
public interface Mapper<K1, V1, K2, V2>
{
void map( K1 key, V1 value, OutputCollector< K2, V2> output, Reporter
reporter) throws IOException ;
}

public interface Reducer<K2, V2, K3, V3>
{
void reduce( K2 key, Iterator<V2> values,
OutputCollector<K3, V3> output, Reporter reporter)
throws IOException ;
}
```

There are a few points to understand here:

- The generic parameters to the `OutputCollector` class show more explicitly how the result of the methods is presented as output.
- The old API used the `OutputCollector` class for this purpose, and the `Reporter` class to write status and metrics information to the Hadoop framework. The 0.20 API combines these responsibilities in the `Context` class.

- The `Reducer` interface uses an `Iterator` object instead of an `Iterable` object; this was changed as the latter works with the Java for each syntax and makes for cleaner code.
- Neither the `map` nor the `reduce` method could throw `InterruptedException` in the old API.

As you can see, the changes between the APIs alter how MapReduce programs are written but don't change the purpose or responsibilities of mappers or reducers. Don't feel obliged to become an expert in both APIs unless you need to; familiarity with either should allow you to follow the rest of this book.

Hadoop-provided mapper and reducer implementations

We don't always have to write our own `Mapper` and `Reducer` classes from scratch. Hadoop provides several common `Mapper` and `Reducer` implementations that can be used in our jobs. If we don't override any of the methods in the `Mapper` and `Reducer` classes in the new API, the default implementations are the identity `Mapper` and `Reducer` classes, which simply output the input unchanged.

Note that more such prewritten `Mapper` and `Reducer` implementations may be added over time, and currently the new API does not have as many as the older one.

The mappers are found at `org.apache.hadoop.mapreduce.lib.mapper`, and include the following:

- InverseMapper: This outputs (value, key)
- TokenCounterMapper: This counts the number of discrete tokens in each line of input

The reducers are found at `org.apache.hadoop.mapreduce.lib.reduce`, and currently include the following:

- IntSumReducer: This outputs the sum of the list of integer values per key
- LongSumReducer: This outputs the sum of the list of long values per key

Time for action – WordCount the easy way

Let's revisit WordCount, but this time use some of these predefined `map` and `reduce` implementations:

1. Create a new `WordCountPredefined.java` file containing the following code:

```
import org.apache.hadoop.conf.Configuration ;
import org.apache.hadoop.fs.Path;
```

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.map.TokenCounterMapper ;
import org.apache.hadoop.mapreduce.lib.reduce.IntSumReducer ;

public class WordCountPredefined
{
    public static void main(String[] args) throws Exception
    {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "word count1");
        job.setJarByClass(WordCountPredefined.class);
        job.setMapperClass(TokenCounterMapper.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

2. Now compile, create the JAR file, and run it as before.
3. Don't forget to delete the output directory before running the job, if you want to use the same location. Use the `hadoop fs -rmr` output, for example.

What just happened?

Given the ubiquity of WordCount as an example in the MapReduce world, it's perhaps not entirely surprising that there are predefined `Mapper` and `Reducer` implementations that together realize the entire WordCount solution. The `TokenCounterMapper` class simply breaks each input line into a series of `(token, 1)` pairs and the `IntSumReducer` class provides a final count by summing the number of values for each key.

There are two important things to appreciate here:

- Though WordCount was doubtless an inspiration for these implementations, they are in no way specific to it and can be widely applicable
- This model of having reusable mapper and reducer implementations is one thing to remember, especially in combination with the fact that often the best starting point for a new MapReduce job implementation is an existing one

Walking through a run of WordCount

To explore the relationship between mapper and reducer in more detail, and to expose some of Hadoop's inner working, we'll now go through just how WordCount (or indeed any MapReduce job) is executed.

Startup

The call to `Job.waitForCompletion()` in the driver is where all the action starts. The driver is the only piece of code that runs on our local machine, and this call starts the communication with the JobTracker. Remember that the JobTracker is responsible for all aspects of job scheduling and execution, so it becomes our primary interface when performing any task related to job management. The JobTracker communicates with the NameNode on our behalf and manages all interactions relating to the data stored on HDFS.

Splitting the input

The first of these interactions happens when the JobTracker looks at the input data and determines how to assign it to map tasks. Recall that HDFS files are usually split into blocks of at least 64 MB and the JobTracker will assign each block to one map task.

Our WordCount example, of course, used a trivial amount of data that was well within a single block. Picture a much larger input file measured in terabytes, and the split model makes more sense. Each segment of the file—or **split**, in MapReduce terminology—is processed uniquely by one map task.

Once it has computed the splits, the JobTracker places them and the JAR file containing the `Mapper` and `Reducer` classes into a job-specific directory on HDFS, whose path will be passed to each task as it starts.

Task assignment

Once the JobTracker has determined how many map tasks will be needed, it looks at the number of hosts in the cluster, how many TaskTrackers are working, and how many map tasks each can concurrently execute (a user-definable configuration variable). The JobTracker also looks to see where the various input data blocks are located across the cluster and attempts to define an execution plan that maximizes the cases when a TaskTracker processes a split/block located on the same physical host, or, failing that, it processes at least one in the same hardware rack.

This data locality optimization is a huge reason behind Hadoop's ability to efficiently process such large datasets. Recall also that, by default, each block is replicated across three different hosts, so the likelihood of producing a task/host plan that sees most blocks processed locally is higher than it may seem at first.

Task startup

Each TaskTracker then starts up a separate Java virtual machine to execute the tasks. This does add a startup time penalty, but it isolates the TaskTracker from problems caused by misbehaving map or reduce tasks, and it can be configured to be shared between subsequently executed tasks.

If the cluster has enough capacity to execute all the map tasks at once, they will all be started and given a reference to the split they are to process and the job JAR file. Each TaskTracker then copies the split to the local filesystem.

If there are more tasks than the cluster capacity, the JobTracker will keep a queue of pending tasks and assign them to nodes as they complete their initially assigned map tasks.

We are now ready to see the executed data of map tasks. If this all sounds like a lot of work, it is; and it explains why when running any MapReduce job, there is always a non-trivial amount of time taken as the system gets started and performs all these steps.

Ongoing JobTracker monitoring

The JobTracker doesn't just stop work now and wait for the TaskTrackers to execute all the mappers and reducers. It is constantly exchanging heartbeat and status messages with the TaskTrackers, looking for evidence of progress or problems. It also collects metrics from the tasks throughout the job execution, some provided by Hadoop and others specified by the developer of the map and reduce tasks, though we don't use any in this example.

Mapper input

In *Chapter 2*, *Getting Hadoop Up and Running*, our WordCount input was a simple one-line text file. For the rest of this walkthrough, let's assume it was a not-much-less trivial two-line text file:

```
This is a test
Yes this is
```

The driver class specifies the format and structure of the input file by using TextInputFormat, and from this Hadoop knows to treat this as text with the line number as the key and line contents as the value. The two invocations of the mapper will therefore be given the following input:

```
1 This is a test
2 Yes it is.
```

Mapper execution

The key/value pairs received by the mapper are the offset in the file of the line and the line contents respectively because of how the job is configured. Our implementation of the `map` method in `WordCountMapper` discards the key as we do not care where each line occurred in the file and splits the provided value into words using the `split` method on the standard Java String class. Note that better tokenization could be provided by use of regular expressions or the `StringTokenizer` class, but for our purposes this simple approach will suffice.

For each individual word, the mapper then emits a key comprised of the actual word itself, and a value of 1.



We add a few optimizations that we'll mention here, but don't worry too much about them at this point. You will see that we don't create the `IntWritable` object containing the value 1 each time, instead we create it as a static variable and re-use it in each invocation. Similarly, we use a single `Text` object and reset its contents for each execution of the method. The reason for this is that though it doesn't help much for our tiny input file, the processing of a huge data set would see the mapper potentially called thousands or millions of times. If each invocation potentially created a new object for both the key and value output, this would become a resource issue and likely cause much more frequent pauses due to garbage collection. We use this single value and know the `Context.write` method will not alter it.

Mapper output and reduce input

The output of the mapper is a series of pairs of the form `(word, 1)`; in our example these will be:

```
(This,1), (is, 1), (a, 1), (test., 1), (Yes, 1), (it, 1), (is, 1)
```

These output pairs from the mapper are not passed directly to the reducer. Between mapping and reducing is the shuffle stage where much of the magic of MapReduce occurs.

Partitioning

One of the implicit guarantees of the `Reduce` interface is that a single reducer will be given all the values associated with a given key. With multiple reduce tasks running across a cluster, each mapper output must therefore be partitioned into the separate outputs destined for each reducer. These partitioned files are stored on the local node filesystem.

The number of reduce tasks across the cluster is not as dynamic as that of mappers, and indeed we can specify the value as part of our job submission. Each TaskTracker therefore knows how many reducers are in the cluster and from this how many partitions the mapper output should be split into.



We'll address failure tolerance in a later chapter, but at this point an obvious question is what happens to this calculation if a reducer fails. The answer is that the JobTracker will ensure that any failed reduce tasks are reexecuted, potentially on a different node so a transient failure will not be an issue. A more serious issue, such as that caused by a data-sensitive bug or very corrupt data in a split will, unless certain steps are taken, cause the whole job to fail.

The optional partition function

Within the `org.apache.hadoop.mapreduce` package is the `Partitioner` class, an abstract class with the following signature:

```
public abstract class Partitioner<Key, Value>
{
public abstract int getPartition( Key key, Value value,
int numPartitions) ;
}
```

By default, Hadoop will use a strategy that hashes the output key to perform the partitioning. This functionality is provided by the `HashPartitioner` class within the `org.apache.hadoop.mapreduce.lib.partition` package, but it is necessary in some cases to provide a custom subclass of `Partitioner` with application-specific partitioning logic. This would be particularly true if, for example, the data provided a very uneven distribution when the standard hash function was applied.

Reducer input

The reducer TaskTracker receives updates from the JobTracker that tell it which nodes in the cluster hold map output partitions which need to be processed by its local reduce task. It then retrieves these from the various nodes and merges them into a single file that will be fed to the reduce task.

Reducer execution

Our `WordCountReducer` class is very simple; for each word it simply counts the number of elements in the array and emits the final `(Word, count)` output for each word.



We don't worry about any sort of optimization to avoid excess object creation here. The number of reduce invocations is typically smaller than the number of mappers, and consequently the overhead is less of a concern. However, feel free to do so if you find yourself with very tight performance requirements.

For our invocation of WordCount on our sample input, all but one word have only one value in the list of values; `is` has two.



Note that the word `this` and `This` had discrete counts because we did not attempt to ignore case sensitivity. Similarly, ending each sentence with a period would have stopped `is` having a count of two as `is` would be different from `is.`. Always be careful when working with textual data such as capitalization, punctuation, hyphenation, pagination, and other aspects, as they can skew how the data is perceived. In such cases, it's common to have a precursor MapReduce job that applies a normalization or clean-up strategy to the data set.

Reducer output

The final set of reducer output for our example is therefore:

```
(This, 1), (is, 2), (a, 1), (test, 1), (Yes, 1), (this, 1)
```

This data will be output to partition files within the output directory specified in the driver that will be formatted using the specified OutputFormat implementation. Each reduce task writes to a single file with the filename `part-r-nnnnn`, where `nnnnn` starts at `00000` and is incremented. This is, of course, what we saw in *Chapter 2*, *Getting Hadoop Up and Running*; hopefully the `part` prefix now makes a little more sense.

Shutdown

Once all tasks have completed successfully, the JobTracker outputs the final state of the job to the client, along with the final aggregates of some of the more important counters that it has been aggregating along the way. The full job and task history is available in the log directory on each node or, more accessibly, via the JobTracker web UI; point your browser to port 50030 on the JobTracker node.

That's all there is to it!

As you've seen, each MapReduce program sits atop a significant amount of machinery provided by Hadoop and the sketch provided is in many ways a simplification. As before, much of this isn't hugely valuable for such a small example, but never forget that we can use the same software and mapper/reducer implementations to do a WordCount on a much larger data set across a huge cluster, be it local or on EMR. The work that Hadoop does for you at that point is enormous and is what makes it possible to perform data analysis on such datasets; otherwise, the effort to manually implement the distribution, synchronization, and parallelization of code will be immense.

Apart from the combiner...maybe

There is one additional, and optional, step that we omitted previously. Hadoop allows the use of a combiner class to perform some early sorting of the output from the `map` method before it is retrieved by the reducer.

Why have a combiner?

Much of Hadoop's design is predicated on reducing the expensive parts of a job that usually equate to disk and network I/O. The output of the mapper is often large; it's not infrequent to see it many times the size of the original input. Hadoop does allow configuration options to help reduce the impact of the reducers transferring such large chunks of data across the network. The combiner takes a different approach, where it is possible to perform early aggregation to require less data to be transferred in the first place.

The combiner does not have its own interface; a combiner must have the same signature as the reducer and hence also subclasses the `Reduce` class from the `org.apache.hadoop.mapreduce` package. The effect of this is to basically perform a mini-reduce on the mapper for the output destined for each reducer.

Hadoop does not guarantee whether the combiner will be executed. At times, it may not be executed at all, while at times it may be used once, twice, or more times depending on the size and number of output files generated by the mapper for each reducer.

Time for action – WordCount with a combiner

Let's add a combiner to our first WordCount example. In fact, let's use our reducer as the combiner. Since the combiner must have the same interface as the reducer, this is something you'll often see, though note that the type of processing involved in the reducer will determine if it is a true candidate for a combiner; we'll discuss this later. Since we are looking to count word occurrences, we can do a partial count on the map node and pass these subtotals to the reducer.

1. Copy `WordCount1.java` to `WordCount2.java` and change the driver class to add the following line between the definition of the `Mapper` and `Reducer` classes:

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

2. Also change the class name to `WordCount2` and then compile it.

```
$ javac WordCount2.java
```

3. Create the JAR file.

```
$ jar cvf wc2.jar WordCount2*class
```

4. Run the job on Hadoop.

```
$ hadoop jar wc2.jar WordCount2 test.txt output
```

5. Examine the output.

```
$ hadoop fs -cat output/part-r-00000
```

What just happened?

This output may not be what you expected, as the value for the word `is` is now incorrectly specified as 1 instead of 2.

The problem lies in how the combiner and reducer will interact. The value provided to the reducer, which was previously `(is, 1, 1)`, is now `(is, 2)` because our combiner did its own summation of the number of elements for each word. However, our reducer does not look at the actual values in the `Iterable` object, it simply counts how many are there.

When you can use the reducer as the combiner

You need to be careful when writing a combiner. Remember that Hadoop makes no guarantees on how many times it may be applied to map output, it may be 0, 1, or more. It is therefore critical that the operation performed by the combiner can effectively be applied in such a way. Distributive operations such as summation, addition, and similar are usually safe, but, as shown previously, ensure the reduce logic isn't making implicit assumptions that might break this property.

Time for action – fixing WordCount to work with a combiner

Let's make the necessary modifications to WordCount to correctly use a combiner.

Copy `WordCount2.java` to a new file called `WordCount3.java` and change the `reduce` method as follows:

```
public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException
```

```
{
int total = 0 ;
for (IntWritable val : values))
{
total+= val.get() ;
}
            context.write(key, new IntWritable(total));
}
```

Remember to also change the class name to `WordCount3` and then compile, create the JAR file, and run the job as before.

What just happened?

The output is now as expected. Any map-side invocations of the combiner performs successfully and the reducer correctly produces the overall output value.



Would this have worked if the original reducer was used as the combiner and the new reduce implementation as the reducer? The answer is no, though our test example would not have demonstrated it. Because the combiner may be invoked multiple times on the map output data, the same errors would arise in the map output if the dataset was large enough, but didn't occur here due to the small input size. Fundamentally, the original reducer was incorrect, but this wasn't immediately obvious; watch out for such subtle logic flaws. This sort of issue can be really hard to debug as the code will reliably work on a development box with a subset of the data set and fail on the much larger operational cluster. Carefully craft your combiner classes and never rely on testing that only processes a small sample of the data.

Reuse is your friend

In the previous section we took the existing job class file and made changes to it. This is a small example of a very common Hadoop development workflow; use an existing job file as the starting point for a new one. Even if the actual mapper and reducer logic is very different, it's often a timesaver to take an existing working job as this helps you remember all the required elements of the mapper, reducer, and driver implementations.

Pop quiz – MapReduce mechanics

Q1. What do you always have to specify for a MapReduce job?

1. The classes for the mapper and reducer.
2. The classes for the mapper, reducer, and combiner.

3. The classes for the mapper, reducer, partitioner, and combiner.
4. None; all classes have default implementations.

Q2. How many times will a combiner be executed?

1. At least once.
2. Zero or one times.
3. Zero, one, or many times.
4. It's configurable.

Q3. You have a mapper that for each key produces an integer value and the following set of reduce operations:

- Reducer A: outputs the sum of the set of integer values.
- Reducer B: outputs the maximum of the set of values.
- Reducer C: outputs the mean of the set of values.
- Reducer D: outputs the difference between the largest and smallest values in the set.

Which of these reduce operations could safely be used as a combiner?

1. All of them.
2. A and B.
3. A, B, and D.
4. C and D.
5. None of them.

Hadoop-specific data types

Up to this point we've glossed over the actual data types used as the input and output of the map and reduce classes. Let's take a look at them now.

The Writable and WritableComparable interfaces

If you browse the Hadoop API for the `org.apache.hadoop.io` package, you'll see some familiar classes such as `Text` and `IntWritable` along with others with the `Writable` suffix.

This package also contains the `Writable` interface specified as follows:

```
import java.io.DataInput ;
import java.io.DataOutput ;
import java.io.IOException ;

public interface Writable
{
void write(DataOutput out) throws IOException ;
void readFields(DataInput in) throws IOException ;
}
```

The main purpose of this interface is to provide mechanisms for the serialization and deserialization of data as it is passed across the network or read and written from the disk. Every data type to be used as a value input or output from a mapper or reducer (that is, `V1`, `V2`, or `V3`) must implement this interface.

Data to be used as keys (`K1`, `K2`, `K3`) has a stricter requirement: in addition to `Writable`, it must also provide an implementation of the standard Java `Comparable` interface. This has the following specifications:

```
public interface Comparable
{
public int compareTO( Object obj) ;
}
```

The compare method returns `-1`, `0`, or `1` depending on whether the compared object is less than, equal to, or greater than the current object.

As a convenience interface, Hadoop provides the `WritableComparable` interface in the `org.apache.hadoop.io` package.

```
public interface WritableComparable extends Writable, Comparable
{}
```

Introducing the wrapper classes

Fortunately, you don't have to start from scratch; as you've already seen, Hadoop provides classes that wrap the Java primitive types and implement `WritableComparable`. They are provided in the `org.apache.hadoop.io` package.

Primitive wrapper classes

These classes are conceptually similar to the primitive wrapper classes, such as `Integer` and `Long` found in `java.lang`. They hold a single primitive value that can be set either at construction or via a setter method.

- BooleanWritable
- ByteWritable
- DoubleWritable
- FloatWritable
- IntWritable
- LongWritable
- VIntWritable – a variable length integer type
- VLongWritable – a variable length long type

Array wrapper classes

These classes provide writable wrappers for arrays of other `Writable` objects. For example, an instance of either could hold an array of `IntWritable` or `DoubleWritable` but not arrays of the raw int or float types. A specific subclass for the required `Writable` class will be required. They are as follows:

- ArrayWritable
- TwoDArrayWritable

Map wrapper classes

These classes allow implementations of the `java.util.Map` interface to be used as keys or values. Note that they are defined as `Map<Writable, Writable>` and effectively manage a degree of internal-runtime-type checking. This does mean that compile type checking is weakened, so be careful.

- `AbstractMapWritable`: This is a base class for other concrete `Writable` map implementations
- `MapWritable`: This is a general purpose map mapping `Writable` keys to `Writable` values
- `SortedMapWritable`: This is a specialization of the `MapWritable` class that also implements the `SortedMap` interface

Time for action – using the Writable wrapper classes

Let's write a class to show some of these wrapper classes in action:

1. Create the following as `WritablesTest.java`:

```
import org.apache.hadoop.io.* ;
import java.util.* ;

public class WritablesTest
{
    public static class IntArrayWritable extends ArrayWritable
    {
        public IntArrayWritable()
        {
            super(IntWritable.class) ;
        }
    }

    public static void main(String[] args)
    {
System.out.println("*** Primitive Writables ***") ;
        BooleanWritable bool1 = new BooleanWritable(true) ;
        ByteWritable byte1 = new ByteWritable( (byte)3) ;
        System.out.printf("Boolean:%s Byte:%d\n", bool1, byte1.
get()) ;

        IntWritable i1 = new IntWritable(5) ;
        IntWritable i2 = new IntWritable( 17) ;         System.
out.printf("I1:%d I2:%d\n", i1.get(), i2.get()) ;
        i1.set(i2.get()) ;
        System.out.printf("I1:%d I2:%d\n", i1.get(), i2.get()) ;
        Integer i3 = new Integer( 23) ;
        i1.set( i3) ;
        System.out.printf("I1:%d I2:%d\n", i1.get(), i2.get()) ;

System.out.println("*** Array Writables ***") ;
        ArrayWritable a = new ArrayWritable( IntWritable.class) ;
        a.set( new IntWritable[]{ new IntWritable(1), new
IntWritable(3), new IntWritable(5)}) ;

        IntWritable[] values = (IntWritable[])a.get() ;

        for (IntWritable i: values)
```

```
        System.out.println(i) ;

        IntArrayWritable ia = new IntArrayWritable() ;
        ia.set( new IntWritable[]{ new IntWritable(1), new
IntWritable(3), new IntWritable(5)}) ;

        IntWritable[] ivalues = (IntWritable[])ia.get() ;

        ia.set(new LongWritable[]{new LongWritable(1000l)}) ;

System.out.println("*** Map Writables ***") ;
        MapWritable m = new MapWritable() ;
        IntWritable key1 = new IntWritable(5) ;
        NullWritable value1 = NullWritable.get() ;
        m.put(key1, value1) ;
        System.out.println(m.containsKey(key1)) ;
        System.out.println(m.get(key1)) ;
        m.put(new LongWritable(1000000000), key1) ;
        Set<Writable> keys = m.keySet() ;

        for(Writable w: keys)
        System.out.println(w.getClass()) ;
    }
}
```

2. Compile and run the class, and you should get the following output:

```
*** Primitive Writables ***
Boolean:true Byte:3
I1:5 I2:17
I1:17 I2:17
I1:23 I2:17
*** Array Writables ***
1
3
5
*** Map Writables ***
true
(null)
class org.apache.hadoop.io.LongWritable
class org.apache.hadoop.io.IntWritable
```

What just happened?

This output should be largely self-explanatory. We create various `Writable` wrapper objects and show their general usage. There are several key points:

- As mentioned, there is no type-safety beyond `Writable` itself. So it is possible to have an array or map that holds multiple types, as shown previously.
- We can use autounboxing, for example, by supplying an `Integer` object to methods on `IntWritable` that expect an `int` variable.
- The inner class demonstrates what is needed if an `ArrayWritable` class is to be used as an input to a `reduce` function; a subclass with such a default constructor must be defined.

Other wrapper classes

- `CompressedWritable`: This is a base class to allow for large objects that should remain compressed until their attributes are explicitly accessed
- `ObjectWritable`: This is a general-purpose generic object wrapper
- `NullWritable`: This is a singleton object representation of a null value
- `VersionedWritable`: This is a base implementation to allow writable classes to track versions over time

Have a go hero – playing with Writables

Write a class that exercises the `NullWritable` and `ObjectWritable` classes in the same way as it does in the previous examples.

Making your own

As you have seen from the `Writable` and `Comparable` interfaces, the required methods are pretty straightforward; don't be afraid of adding this functionality if you want to use your own custom classes as keys or values within a MapReduce job.

Input/output

There is one aspect of our driver classes that we have mentioned several times without getting into a detailed explanation: the format and structure of the data input into and output from MapReduce jobs.

Files, splits, and records

We have talked about files being broken into splits as part of the job startup and the data in a split being sent to the mapper implementation. However, this overlooks two aspects: how the data is stored in the file and how the individual keys and values are passed to the mapper structure.

InputFormat and RecordReader

Hadoop has the concept of an **InputFormat** for the first of these responsibilities. The `InputFormat` abstract class in the `org.apache.hadoop.mapreduce` package provides two methods as shown in the following code:

```
public abstract class InputFormat<K, V>
{
public abstract List<InputSplit> getSplits( JobContext context) ;
RecordReader<K, V> createRecordReader(InputSplit split,
TaskAttemptContext context) ;
}
```

These methods display the two responsibilities of the `InputFormat` class:

- To provide the details on how to split an input file into the splits required for map processing
- To create a `RecordReader` class that will generate the series of key/value pairs from a split

The `RecordReader` class is also an abstract class within the `org.apache.hadoop.mapreduce` package:

```
public abstract class RecordReader<Key, Value> implements Closeable
{
public abstract void initialize(InputSplit split, TaskAttemptContext
context) ;
  public abstract boolean nextKeyValue()
throws IOException, InterruptedException ;
public abstract Key getCurrentKey()
throws IOException, InterruptedException ;
public abstract Value getCurrentValue()
throws IOException, InterruptedException ;
public abstract float getProgress()
throws IOException, InterruptedException ;
public abstract close() throws IOException ;
}
```

A `RecordReader` instance is created for each split and calls `getNextKeyValue` to return a Boolean indicating if another key/value pair is available and if so, the `getKey` and `getValue` methods are used to access the key and value respectively.

The combination of the `InputFormat` and `RecordReader` classes therefore are all that is required to bridge between any kind of input data and the key/value pairs required by MapReduce.

Hadoop-provided InputFormat

There are some Hadoop-provided `InputFormat` implementations within the `org.apache.hadoop.mapreduce.lib.input` package:

- `FileInputFormat`: This is an abstract base class that can be the parent of any file-based input
- `SequenceFileInputFormat`: This is an efficient binary file format that will be discussed in an upcoming section
- `TextInputFormat`: This is used for plain text files



The pre-0.20 API has additional InputFormats defined in the `org.apache.hadoop.mapred` package.

Note that `InputFormats` are not restricted to reading from files; `FileInputFormat` is itself a subclass of `InputFormat`. It is possible to have Hadoop use data that is not based on the files as the input to MapReduce jobs; common sources are relational databases or HBase.

Hadoop-provided RecordReader

Similarly, Hadoop provides a few common `RecordReader` implementations, which are also present within the `org.apache.hadoop.mapreduce.lib.input` package:

- `LineRecordReader`: This implementation is the default `RecordReader` class for text files that present the line number as the key and the line contents as the value
- `SequenceFileRecordReader`: This implementation reads the key/value from the binary `SequenceFile` container

Again, the pre-0.20 API has additional `RecordReader` classes in the `org.apache.hadoop.mapred` package, such as `KeyValueRecordReader`, that have not yet been ported to the new API.

OutputFormat and RecordWriter

There is a similar pattern for writing the output of a job coordinated by subclasses of `OutputFormat` and `RecordWriter` from the `org.apache.hadoop.mapreduce` package. We'll not explore these in any detail here, but the general approach is similar, though `OutputFormat` does have a more involved API as it has methods for tasks such as validation of the output specification.



It is this step that causes a job to fail if a specified output directory already exists. If you wanted different behavior, it would require a subclass of `OutputFormat` that overrides this method.

Hadoop-provided OutputFormat

The following OutputFormats are provided in the `org.apache.hadoop.mapreduce.output` package:

- `FileOutputFormat`: This is the base class for all file-based OutputFormats
- `NullOutputFormat`: This is a dummy implementation that discards the output and writes nothing to the file
- `SequenceFileOutputFormat`: This writes to the binary `SequenceFile` format
- `TextOutputFormat`: This writes a plain text file

Note that these classes define their required `RecordWriter` implementations as inner classes so there are no separately provided `RecordWriter` implementations.

Don't forget Sequence files

The `SequenceFile` class within the `org.apache.hadoop.io` package provides an efficient binary file format that is often useful as an output from a MapReduce job. This is especially true if the output from the job is processed as the input of another job. The `Sequence` files have several advantages, as follows:

- As binary files, they are intrinsically more compact than text files
- They additionally support optional compression, which can also be applied at different levels, that is, compress each record or an entire split
- The file can be split and processed in parallel

This last characteristic is important, as most binary formats—particularly those that are compressed or encrypted—cannot be split and must be read as a single linear stream of data. Using such files as input to a MapReduce job means that a single mapper will be used to process the entire file, causing a potentially large performance hit. In such a situation, it is preferable to either use a splitable format such as `SequenceFile`, or, if you cannot avoid receiving the file in the other format, do a preprocessing step that converts it into a splitable format. This will be a trade-off, as the conversion will take time; but in many cases—especially with complex map tasks—this will be outweighed by the time saved.

Summary

We have covered a lot of ground in this chapter and we now have the foundation to explore MapReduce in more detail. Specifically, we learned how key/value pairs is a broadly applicable data model that is well suited to MapReduce processing. We also learned how to write mapper and reducer implementations using the 0.20 and above versions of the Java API.

We then moved on and saw how a MapReduce job is processed and how the `map` and `reduce` methods are tied together by significant coordination and task-scheduling machinery. We also saw how certain MapReduce jobs require specialization in the form of a custom partitioner or combiner.

We also learned how Hadoop reads data to and from the filesystem. It uses the concept of `InputFormat` and `OutputFormat` to handle the file as a whole and `RecordReader` and `RecordWriter` to translate the format to and from key/value pairs.

With this knowledge, we will now move on to a case study in the next chapter, which demonstrates the ongoing development and enhancement of a MapReduce application that processes a large data set.

4
Developing MapReduce Programs

Now that we have explored the technology of MapReduce, we will spend this chapter looking at how to put it to use. In particular, we will take a more substantial dataset and look at ways to approach its analysis by using the tools provided by MapReduce.

In this chapter we will cover the following topics:

- Hadoop Streaming and its uses
- The UFO sighting dataset
- Using Streaming as a development/debugging tool
- Using multiple mappers in a single job
- Efficiently sharing utility files and data across the cluster
- Reporting job and task status and log information useful for debugging

Throughout this chapter, the goal is to introduce both concrete tools and ideas about how to approach the analysis of a new data set. We shall start by looking at how to use scripting programming languages to aid MapReduce prototyping and initial analysis. Though it may seem strange to learn the Java API in the previous chapter and immediately move to different languages, our goal here is to provide you with an awareness of different ways to approach the problems you face. Just as many jobs make little sense being implemented in anything but the Java API, there are other situations where using another approach is best suited. Consider these techniques as new additions to your tool belt and with that experience you will know more easily which is the best fit for a given scenario.

Using languages other than Java with Hadoop

We have mentioned previously that MapReduce programs don't have to be written in Java. Most programs are written in Java, but there are several reasons why you may want or need to write your map and reduce tasks in another language. Perhaps you have existing code to leverage or need to use third-party binaries—the reasons are varied and valid.

Hadoop provides a number of mechanisms to aid non-Java development, primary amongst these are **Hadoop Pipes** that provides a native C++ interface to Hadoop and **Hadoop Streaming** that allows any program that uses standard input and output to be used for map and reduce tasks. We will use Hadoop Streaming heavily in this chapter.

How Hadoop Streaming works

With the MapReduce Java API, both map and reduce tasks provide implementations for methods that contain the task functionality. These methods receive the input to the task as method arguments and then output results via the `Context` object. This is a clear and type-safe interface but is by definition Java specific.

Hadoop Streaming takes a different approach. With Streaming, you write a map task that reads its input from standard input, one line at a time, and gives the output of its results to standard output. The reduce task then does the same, again using only standard input and output for its data flow.

Any program that reads and writes from standard input and output can be used in Streaming, such as compiled binaries, Unixshell scripts, or programs written in a dynamic language such as Ruby or Python.

Why to use Hadoop Streaming

The biggest advantage to Streaming is that it can allow you to try ideas and iterate on them more quickly than using Java. Instead of a compile/jar/submit cycle, you just write the scripts and pass them as arguments to the Streaming jar file. Especially when doing initial analysis on a new dataset or trying out new ideas, this can significantly speed up development.

The classic debate regarding dynamic versus static languages balances the benefits of swift development against runtime performance and type checking. These dynamic downsides also apply when using Streaming. Consequently, we favor use of Streaming for up-front analysis and Java for the implementation of jobs that will be executed on the production cluster.

We will use Ruby for Streaming examples in this chapter, but that is a personal preference. If you prefer shell scripting or another language, such as Python, then take the opportunity to convert the scripts used here into the language of your choice.

Time for action – implementing WordCount using Streaming

Let's flog the dead horse of WordCount one more time and implement it using Streaming by performing the following steps:

1. Save the following file to `wcmapper.rb`:

```
#/bin/env ruby

while line = gets
    words = line.split("\t")
    words.each{ |word| puts word.strip+"\t1"}}
end
```

2. Make the file executable by executing the following command:

```
$ chmod +x wcmapper.rb
```

3. Save the following file to `wcreducer.rb`:

```
#!/usr/bin/env ruby

current = nil
count = 0

while line = gets
    word, counter = line.split("\t")

    if word == current
        count = count+1
    else
        puts current+"\t"+count.to_s if current
        current = word
        count = 1
    end
end
puts current+"\t"+count.to_s
```

4. Make the file executable by executing the following command:

```
$ chmod +x wcreducer.rb
```

5. Execute the scripts as a Streaming job using the datafile from the previous chapter:

```
$ hadoop jar hadoop/contrib/streaming/hadoop-streaming-1.0.3.jar
-file wcmapper.rb -mapper wcmapper.rb -file wcreducer.rb
-reducer wcreducer.rb -input test.txt -output output
packageJobJar: [wcmapper.rb, wcreducer.rb, /tmp/hadoop-
hadoop/hadoop-unjar1531650352198893161/] [] /tmp/
streamjob937274081293220534.jar tmpDir=null
12/02/05 12:43:53 INFO mapred.FileInputFormat: Total input paths
to process : 1
12/02/05 12:43:53 INFO streaming.StreamJob: getLocalDirs(): [/var/
hadoop/mapred/local]
12/02/05 12:43:53 INFO streaming.StreamJob: Running job:
job_201202051234_0005
…
12/02/05 12:44:01 INFO streaming.StreamJob:  map 100%  reduce 0%
12/02/05 12:44:13 INFO streaming.StreamJob:  map 100%  reduce 100%
12/02/05 12:44:16 INFO streaming.StreamJob: Job complete:
job_201202051234_0005
12/02/05 12:44:16 INFO streaming.StreamJob: Output: wcoutput
```

6. Check the result file:

```
$ hadoop fs -cat output/part-00000
```

What just happened?

Ignore the specifics of Ruby. If you don't know the language, it isn't important here.

Firstly, we created the script that will be our mapper. It uses the `gets` function to read a line from standard input, splits this into words, and uses the `puts` function to write the word and the value `1` to the standard output. We then made the file executable.

Our reducer is a little more complex for reasons we will describe in the next section. However, it performs the job we would expect, it counts the number of occurrences for each word, reads from standard input, and gives the output as the final value to standard output. Again we made sure to make the file executable.

Note that in both cases we are implicitly using Hadoop input and output formats discussed in the earlier chapters. It is the `TextInputFormat` property that processes the source file and provides each line one at a time to the map script. Conversely, the `TextOutputFormat` property will ensure that the output of reduce tasks is also
correctly written as textual data. We can of course modify these if required.

Next, we submitted the Streaming job to Hadoop via the rather cumbersome command line shown in the previous section. The reason for each file to be specified twice is that any file not available on each node must be packaged up by Hadoop and shipped across the cluster, which requires it to be specified by the `-file` option. Then, we also need to tell Hadoop which script performs the mapper and reducer roles.

Finally, we looked at the output of the job, which should be identical to the previous Java-based WordCount implementations

Differences in jobs when using Streaming

The Streaming WordCount mapper looks a lot simpler than the Java version, but the reducer appears to have more logic. Why? The reason is that the implied contract between Hadoop and our tasks changes when we use Streaming.

In Java we knew that our `map()` method would be invoked once for each input key/value pair and our `reduce()` method would be invoked for each key and its set of values.

With Streaming we don't have the concept of the `map` or `reduce` methods anymore, instead we have written scripts that process streams of received data. This changes how we need to write our reducer. In Java the grouping of values to each key was performed by Hadoop; each invocation of the `reduce` method would receive a single key and all its values. In Streaming, each instance of the `reduce` task is given the individual ungathered values one at a time.

Hadoop Streaming does sort the keys, for example, if a mapper emitted the following data:

```
First     1
Word      1
Word      1
A         1
First     1
```

The Streaming reducer would receive this data in the following order:

```
A         1
First     1
First     1
Word      1
Word      1
```

Hadoop still collects the values for each key and ensures that each key is passed only to a single reducer. In other words, a reducer gets all the values for a number of keys and they are grouped together; however, they are not packaged into individual executions of the reducer, that is, one per key, as with the Java API.

This should explain the mechanism used in the Ruby reducer; it first sets empty default values for the current word; then after reading each line it determines if this is another value for the current key, and if so, increments the count. If not, then there will be no more values for the previous key and its final output is sent to standard output and the counting begins again for the new word.

After reading so much in the earlier chapters about how great it is for Hadoop to do so much for us, this may seem a lot more complex, but after you write a few Streaming reducers it's actually not as bad as it may first appear. Also remember that Hadoop does still manage the assignment of splits to individual map tasks and the necessary coordination that sends the values for a given key to the same reducer. This behavior can be modified through configuration settings to change the number of mappers and reducers just as with the Java API.

Analyzing a large dataset

Armed with our abilities to write MapReduce jobs in both Java and Streaming, we'll now explore a more significant dataset than any we've looked at before. In the following section, we will attempt to show how to approach such analysis and the sorts of questions Hadoop allows you to ask of a large dataset.

Getting the UFO sighting dataset

We will use a public domain dataset of over 60,000 UFO sightings. This is hosted by InfoChimps at `http://www.infochimps.com/datasets/60000-documented-ufo-sightings-with-text-descriptions-and-metada`.

You will need to register for a free InfoChimps account to download a copy of the data.

The data comprises a series of UFO sighting records with the following fields:

1. **Sighting date**: This field gives the date when the UFO sighting occurred.
2. **Recorded date**: This field gives the date when the sighting was reported, often different to the sighting date.
3. **Location**: This field gives the location where the sighting occurred.
4. **Shape**: This field gives a brief summary of the shape of the UFO, for example, diamond, lights, cylinder.
5. **Duration**: This field gives the duration of how long the sighting lasted.
6. **Description**: This field gives free text details of the sighting.

Once downloaded, you will find the data in a few formats. We will be using the `.tsv` (tab-separated value) version.

Getting a feel for the dataset

When faced with a new dataset it is often difficult to get a feel for the nature, breadth, and quality of the data involved. There are several questions, the answers to which will affect how you approach the follow-on analysis, in particular:

- How big is the dataset?
- How complete are the records?
- How well do the records match the expected format?

The first is a simple question of scale; are we talking hundreds, thousands, millions, or more records? The second question asks how complete the records are. If you expect each record to have 10 fields (if this is structured or semi-structured data), how many have key fields populated with data? The last question expands on this point, how well do the records match your expectations of format and representation?

Time for action – summarizing the UFO data

Now we have the data, let's get an initial summarization of its size and how many records may be incomplete:

1. With the UFO **tab-separated value** (**TSV**) file on HDFS saved as `ufo.tsv`, save the following file to `summarymapper.rb`:

```
#!/usr/bin/env ruby

while line = gets
    puts "total\t1"
    parts = line.split("\t")
    puts "badline\t1" if parts.size != 6
    puts "sighted\t1" if !parts[0].empty?
    puts "recorded\t1" if !parts[1].empty?
    puts "location\t1" if !parts[2].empty?
    puts "shape\t1" if !parts[3].empty?
    puts "duration\t1" if !parts[4].empty?
    puts "description\t1" if !parts[5].empty?
end
```

2. Make the file executable by executing the following command:

```
$ chmod +x summarymapper.rb
```

3. Execute the job as follows by using Streaming:

```
$ hadoop jar hadoop/contrib/streaming/hadoop-streaming-1.0.3.jar
-file summarymapper.rb -mapper summarymapper.rb -file wcreducer.rb
-reducer wcreducer.rb -input ufo.tsv -output ufosummary
```

4. Retrieve the summary data:

```
$ hadoop fs -cat ufosummary/part-0000
```

What just happened?

Remember that our UFO sightings should have six fields as described previously. They are listed as follows:

- The date of the sighting
- The date the sighting was reported
- The location of the sighting
- The shape of the object
- The duration of the sighting
- A free text description of the event

The mapper examines the file and counts the total number of records in addition to identifying potentially incomplete records.

We produce the overall count by simply recording how many distinct records are encountered while processing the file. We identify potentially incomplete records by flagging those that either do not contain exactly six fields or have at least one field that has a null value.

Therefore, the implementation of the mapper reads each line and does three things as it proceeds through the file:

- It gives the output of a token to be incremented in the total number of records processed
- It splits the record on tab boundaries and records any occurrence of lines which do not result in six fields' values

- For each of the six expected fields it reports when the values present are other than an empty string, that is, there is data in the field, though this doesn't actually say anything about the quality of that data

We wrote this mapper intentionally to produce the output of the form `(token, count)`. Doing this allowed us to use our existing WordCount reducer from our earlier implementations as the reducer for this job. There are certainly more efficient implementations, but as this job is unlikely to be frequently executed, the convenience is worth it.

At the time of writing, the result of this job was as follows:

```
badline324
description61372
duration58961
location61377
recorded61377
shape58855
sighted61377
total61377
```

We see from these figures that we have 61,300records. All of these provide values for the sighted date, reported date, and location fields. Around 58,000-59,000 records have values for shape and duration, and almost all have a description.

When split on tab characters there were 372 lines found to not have exactly six fields. However, since only five records had no value for description, this suggests that the bad records typically have too many tabs as opposed to too few. We could of course alter our mapper to gather detailed information on this fact. This is likely due to tabs being used in the free text description, so for now we will do our analysis expecting most records to have correctly placed values for all the six fields, but not make any assumptions regarding further tabs in each record.

Examining UFO shapes

Out of all the fields in these reports, it was shape that immediately interested us most, as it could offer some interesting ways of grouping the data depending on what sort of information we have in that field.

Time for action – summarizing the shape data

Just as we provided a summarization for the overall UFO data set earlier, let's now do a more focused summarization on the data provided for UFO shapes:

1. Save the following to `shapemapper.rb`:

```
#!/usr/bin/env ruby

while line = gets
    parts = line.split("\t")
    if parts.size == 6
        shape = parts[3].strip
        puts shape+"\t1" if !shape.empty?
    end
end
```

2. Make the file executable:

```
$ chmod +x shapemapper.rb
```

3. Execute the job once again using the WordCount reducer:

```
$ hadoop jar hadoop/contrib/streaming/hadoop-streaming-1.0.3.jarr
--file shapemapper.rb -mapper shapemapper.rb -file wcreducer.rb
-reducer wcreducer.rb -input ufo.tsv -output shapes
```

4. Retrieve the shape info:

```
$ hadoop fs -cat shapes/part-00000
```

What just happened?

Our mapper here is pretty simple. It breaks each record into its constituent fields, discards any without exactly six fields, and gives a counter as the output for any non-empty shape value.

For our purposes here, we are happy to ignore any records that don't precisely match the specification we expect. Perhaps one record is the single UFO sighting that will prove it once and for all, but even so it wouldn't likely make much difference to our analysis. Think about the potential value of individual records before deciding to so easily discard some. If you are working primarily on large aggregations where you care mostly about trends, individual records likely don't matter. But in cases where single individual values could materially affect the analysis or must be accounted for, an approach of trying to parse and recover more conservatively rather than discard may be best. We'll talk more about this trade-off in *Chapter 6*, *When Things Break*.

After the usual routine of making the mapper executable and running the job we produced, data showing 29 different UFO shapes were reported. Here's some sample output tabulated in compact form for space reasons:

```
changed1 changing1533 chevron758 cigar1774
circle5250 cone265 crescent2 cross177
cylinder981 delta8 diamond909 disk4798
dome1 egg661 fireball3437 flare1
flash988 formation1775 hexagon1 light12140
other4574 oval2859 pyramid1 rectangle957
round2 sphere3614 teardrop592 triangle6036
unknown4459
```

As we can see, there is a wide variance in sighting frequency. Some such as pyramid occur only once, while light comprises more than a fifth of all reported shapes. Considering many UFO sightings are at night, it could be argued that a description of light is not terribly useful or specific and when combined with the values for other and unknown we see that around 21000 of our 58000 reported shapes may not actually be of any use. Since we are not about to run out and do additional research, this doesn't matter very much, but what's important is to start thinking of your data in these terms. Even these types of summary analysis can start giving an insight into the nature of the data and indicate what quality of analysis may be possible. In the case of reported shapes, for example, we have already discovered that out of our 61000 sightings only 58000 reported the shape and of these 21000 are of dubious value. We have already determined that our 61000 sample set only provides 37000 shape reports that we may be able to work with. If your analysis is predicated on a minimum number of samples, always be sure to do this sort of summarization up-front to determine if the data set will actually meet your needs.

Time for action – correlating of sighting duration to UFO shape

Let's do a little more detailed analysis in regards to this shape data. We wondered if there was any correlation between the duration of a sighting to the reported shape. Perhaps cigar-shaped UFOs hang around longer than the rest or formations always appear for the exact amount of time.

1. Save the following to `shapetimemapper.rb`:

```
#!/usr/bin/env ruby

pattern = Regexp.new /\d* ?((min)|(sec))/

while line = gets
parts = line.split("\t")
if parts.size == 6
```

```
shape = parts[3].strip
duration = parts[4].strip.downcase
if !shape.empty? && !duration.empty?
match = pattern.match(duration)
time = /\d*/.match(match[0])[0]
unit = match[1]
time = Integer(time)
time = time * 60 if unit == "min"
puts shape+"\t"+time.to_s
end
end
end
```

2. Make the file executable by executing the following command:

```
$ chmod +x shapetimemapper.rb
```

3. Save the following to `shapetimereducer.rb`:

```
#!/usr/bin/env ruby

current = nil
min = 0
max = 0
mean = 0
total = 0
count = 0

while line = gets
word, time = line.split("\t")
time = Integer(time)

if word == current
count = count+1
total = total+time
min = time if time < min
max = time if time > max
else
puts current+"\t"+min.to_s+" "+max.to_s+" "+(total/count).to_s if
current
current = word
count = 1
total = time
min = time
max = time
end
end
puts current+"\t"+min.to_s+" "+max.to_s+" "+(total/count).to_s
```

4. Make the file executable by executing the following command:

```
$ chmod +x shapetimereducer.rb
```

5. Run the job:

```
$ hadoop jar hadoop/contrib/streaminghHadoop-streaming-1.0.3.jar
-file shapetimemapper.rb -mapper shapetimemapper.rb -file
shapetimereducer.rb -reducer shapetimereducer.rb -input ufo.tsv
-output shapetime
```

6. Retrieve the results:

```
$ hadoop fs -cat shapetime/part-00000
```

What just happened?

Our mapper here is a little more involved than previous examples due to the nature of the duration field. Taking a quick look at some sample records, we found values as follows:

```
15 seconds
2 minutes
2 min
2minutes
5-10 seconds
```

In other words, there was a mixture of range and absolute values, different formatting and inconsistent terms for time units. Again for simplicity we decided on a limited interpretation of the data; we will take the absolute value if present, and the upper part of a range if not. We would assume that the strings `min` or `sec` would be present for the time units and would convert all timings into seconds. With some regular expression magic, we unpack the duration field into these parts and do the conversion. Note again that we simply discard any record that does not work as we expect, which may not always be appropriate.

The reducer follows the same pattern as our earlier example, starting with a default key and reading values until a new one is encountered. In this case, we want to capture the minimum, maximum, and mean for each shape, so use numerous variables to track the needed data.

Remember that Streaming reducers need to handle a series of values grouped into their associated keys and must identify when a new line has a changed key, and hence indicates the last value for the previous key that has been processed. In contrast, a Java reducer would be simpler as it only deals with the values for a single key in each execution.

After making both files executable we run the job and get the following results, where we removed any shape with less than 10 sightings and again made the output more compact for space reasons. The numbers for each shape are the minimum value, the maximum value, and mean respectively:

```
changing0 5400 670 chevron0 3600 333
cigar0 5400 370 circle0 7200 423
cone0 4500 498 cross2 3600 460
cylinder0 5760 380 diamond0 7800 519
disk0 5400 449 egg0 5400 383
fireball0 5400 236 flash0 7200 303
formation0 5400 434 light0 9000 462
other0 5400 418 oval0 5400 405
rectangle0 4200 352 sphere0 14400 396
teardrop0 2700 335 triangle0 18000 375
unknown0 6000 470
```

It is surprising to see the relatively narrow variance in the mean sighting duration across all shape types; most have the mean value between 350 and 430 seconds. Interestingly, we also see that the shortest mean duration is for fireballs and the maximum for changeable objects, both of which make some degree of intuitive sense. A fireball by definition wouldn't be a long-lasting phenomena and a changeable object would need a lengthy duration for its changes to be noticed.

Using Streaming scripts outside Hadoop

This last example with its more involved mapper and reducer is a good illustration of how Streaming can help MapReduce development in another way; you can execute the scripts outside of Hadoop.

It's generally good practice during MapReduce development to have a sample of the production data against which to test your code. But when this is on HDFS and you are writing Java map and reduce tasks, it can be difficult to debug problems or refine complex logic. With map and reduce tasks that read input from the command line, you can directly run them against some data to get quick feedback on the result. If you have a development environment that provides Hadoop integration or are using Hadoop in standalone mode, the problems are minimized; just remember that Streaming does give you this ability to try the scripts outside of Hadoop; it may be useful some day.

While developing these scripts the author noticed that the last set of records in his UFO datafile had data in a better structured manner than those at the start of the file. Therefore, to do a quick test on the mapper all that was required was:

```
$ tail ufo.tsv | shapetimemapper.rb
```

This principle can be applied to the full workflow to exercise both the map and reduce script.

Time for action – performing the shape/time analysis from the command line

It may not be immediately obvious how to do this sort of local command-line analysis, so let's look at an example.

With the UFO datafile on the local filesystem, execute the following command:

```
$ cat ufo.tsv | shapetimemapper.rb | sort| shapetimereducer.rb
```

What just happened?

With a single Unixcommand line, we produced output identical to our previous full MapReduce job. If you look at what the command line does, this makes sense.

Firstly, the input file is sent—a line at a time—to the mapper. The output of this is passed through the Unix sort utility and this sorted output is passed a line at a time to the reducer. This is of course a very simplified representation of our general MapReduce job workflow.

Then the obvious question is why should we bother with Hadoop if we can do equivalent analysis at the command line. The answer of course is our old friend, scale. This simple approach works fine for a file such as the UFO sightings, which though non-trivial, is only 71MB in size. To put this into context we could hold thousands of copies of this dataset on a single modern disk drive.

So what if the dataset was 71GB in size instead, or even 71TB? In the latter case, at least we would have to spread the data across multiple hosts, and then decide how to split the data, combine partial answers, and deal with the inevitable failures along the way. In other words,we would need something like Hadoop.

However, don't discount the use of command-line tools like this, such approaches should be well used during MapReduce development.

Java shape and location analysis

Let's return to the Java MapReduce API and consider some analysis of the shape and location data within the reports.

However, before we start writing code, let's think about how we've been approaching the per-field analysis of this dataset. The previous mappers have had a common pattern:

- Discard records determined to be corrupt
- Process valid records to extract the field of interest
- Output a representation of the data we care about for the record

Now if we were to write Java mappers to analyze location and then perhaps the sighting and reported time columns, we would follow a similar pattern. So can we avoid any of the consequent code duplication?

The answer is yes, through the use of `org.apache.hadoop.mapred.lib.ChainMapper`. This class provides a means by which multiple mappers are executed in sequence and it is the output of the final mapper that is passed to the reducer. `ChainMapper` is applicable not just for this type of data clean-up; when analyzing particular jobs, it is not an uncommon pattern that is useful to perform multiple map-type tasks before applying a reducer.

An example of this approach would be to write a validation mapper that could be used by all future field analysis jobs. This mapper would discard lines deemed corrupt, passing only valid lines to the actual business logic mapper that can now be focused on analyzing data instead of worrying about coarse-level validation.

An alternative approach here would be to do the validation within a custom `InputFormat` class that discards non-valid records; which approach makes the most sense will depend on your particular situation.

Each mapper in the chain is executed within a single JVM so there is no need to worry about the use of multiple mappers increasing our filesystem I/O load.

Time for action – using ChainMapper for field validation/analysis

Let's use this principle and employ the `ChainMapper` class to help us provide some record validation within our job:

1. Create the following class as `UFORecordValidationMapper.java`:

```
import java.io.IOException;

import org.apache.hadoop.io.* ;
import org.apache.hadoop.mapred.* ;
import org.apache.hadoop.mapred.lib.* ;

public class UFORecordValidationMapper extends MapReduceBase
implements Mapper<LongWritable, Text, LongWritable, Text>
{

    public void map(LongWritable key, Text value,
        OutputCollector<LongWritable, Text> output,
        Reporter reporter) throws IOException
{
String line = value.toString();
```

```
            if (validate(line))
                output.collect(key, value);
        }

            private boolean validate(String str)
            {
                String[] parts = str.split("\t") ;

                if (parts.length != 6)
                return false ;

                return true ;
            }
        }
```

2. Create the following as `UFOLocation.java`:

```
import java.io.IOException;
import java.util.Iterator ;
import java.util.regex.* ;

import org.apache.hadoop.conf.* ;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.* ;
import org.apache.hadoop.mapred.* ;
import org.apache.hadoop.mapred.lib.* ;

public class UFOLocation
{

    public static class MapClass extends MapReduceBase
implements Mapper<LongWritable, Text, Text, LongWritable>
{

private final static LongWritable one = new LongWritable(1);
private static Pattern locationPattern = Pattern.compile(
"[a-zA-Z]{2}[^a-zA-Z]*$") ;

public void map(LongWritable key, Text value,
OutputCollector<Text, LongWritable> output,
Reporter reporter) throws IOException
{
String line = value.toString();
        String[] fields = line.split("\t") ;
        String location = fields[2].trim() ;
```

```
            if (location.length() >= 2)
            {

                Matcher matcher = locationPattern.matcher(location) ;
                if (matcher.find() )
                {
                    int start = matcher.start() ;
                    String state = location.substring(start,start+2);

                    output.collect(new Text(state.toUpperCase()),
                           One);
                }
            }
        }
    }

    public static void main(String[] args) throws Exception
    {
        Configuration config = new Configuration() ;
    JobConf conf = new JobConf(config, UFOLocation.class);
    conf.setJobName("UFOLocation");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(LongWritable.class);

    JobConf mapconf1 = new JobConf(false) ;
    ChainMapper.addMapper( conf, UFORecordValidationMapper.class,
    LongWritable.class, Text.class, LongWritable.class,
    Text.class, true, mapconf1) ;

    JobConf mapconf2 = new JobConf(false) ;
    ChainMapper.addMapper( conf, MapClass.class,
    LongWritable.class, Text.class,
    Text.class, LongWritable.class, true, mapconf2) ;
    conf.setMapperClass(ChainMapper.class);
    conf.setCombinerClass(LongSumReducer.class);
    conf.setReducerClass(LongSumReducer.class);

    FileInputFormat.setInputPaths(conf,args[0]) ;
    FileOutputFormat.setOutputPath(conf, new Path(args[1])) ;

    JobClient.runJob(conf);
    }
    }
```

3. Compile both files:

```
$ javac UFORecordValidationMapper.java UFOLocation.java
```

4. Jar up the class files and submit the job to Hadoop:

```
$ Hadoop jar ufo.jar UFOLocation ufo.tsv output
```

5. Copy the output file to the local filesystem and examine it:

```
$ Hadoop fs -get output/part-00000 locations.txt
$ more locations.txt
```

What just happened?

There's quite a bit happening here, so let's look at it one piece at a time.

The first mapper is our simple validation mapper. The class follows the same interface as the standard MapReduce API and the `map` method simply returns the result of a utility validation method. We split this out into a separate method to highlight the functionality of the mapper, but the checks could easily have been within the main `map` method itself. For simplicity, we keep to our previous validation strategy of looking for the number of fields and discarding lines that don't break into exactly six tab-delimited fields.

Note that the `ChainMapper` class has unfortunately been one of the last components to be migrated to the context object API and as of Hadoop 1.0, it can only be used with the older API. It remains a valid concept and useful tool but until Hadoop 2.0, where it will finally be migrated into the `org.apache.hadoop.mapreduce.lib.chain` package, its current use requires the older approach.

The other file contains another mapper implementation and an updated driver in the main method. The mapper looks for a two-letter sequence at the end of the location field in a UFO sighting report. From some manual examination of data, it is obvious that most location fields are of the form `city, state`, where the standard two-character abbreviation is used for the state.

Some records, however, add trailing parenthesis, periods, or other punctuation. Some others are simply not in this format. For our purposes, we are happy to discard those records and focus on those that have the trailing two-character state abbreviation we are looking for.

The map method extracts this from the location field using another regular expression and gives the output as the capitalized form of the abbreviation along with a simple count.

The driver for the job has the most changes as the previous configuration involving a single `map` class is replaced with multiple calls on the `ChainMapper` class.

The general model is to create a new configuration object for each mapper, then add the mapper to the `ChainMapper` class along with a specification of its input and output, and a reference to the overall job configuration object.

Notice that the two mappers have different signatures. Both input a key of type `LongWritable` and value of type `Text` which are also the output types of `UFORecordValidationMapper`. `UFOLocationMapper` however outputs the reverse with a key of type Text and a value of type `LongWritable`.

The important thing here is to match the input from the final mapper in the chain (`UFOLocationMapper`) with the inputs expected by the `reduce` class (`LongSumReducer`). When using the`ChainMapper` class the mappers in the chain can have different input and output as long as the following are true:

- For all but the final mapper each map output matches the input of the subsequent mapper in the chain
- For the final mapper, its output matches the input of the reducer

We compile these classes and put them in the same jar file. This is the first time we have bundled the output from more than one Java source file together. As may be expected, there is no magic here; the usual rules on jar files, path, and class names apply. Because in this case we have both our classes in the same package, we don't have to worry about an additional import in the driver class file.

We then run the MapReduce job and examine the output, which is not quite as expected.

Have a go hero

Use the Java API and the previousChainMapper example to reimplement the mappers previously written in Ruby that produce the shape frequency and duration reports.

Too many abbreviations

The following are the first few entries from our result file of the previous job:

```
AB      286
AD      6
AE      7
AI      6
AK      234
AL      548
AM      22
AN      161
…
```

The file had 186 different two-character entries. Plainly, our approach of extracting the final character digraph from the location field was not sufficiently robust.

We have a number of issues with the data which becomes apparent after a manual analysis of the source file:

- There is inconsistency in the capitalization of the state abbreviations
- A non-trivial number of sightings are from outside the U.S. and though they may follow a similar `(city, area)` pattern, the abbreviation is not one of the 50 we'd expect
- Some fields simply don't follow the pattern at all, yet would still be captured by our regular expression

We need to filter these results, ideally by normalizing the U.S. records into correct state output and by gathering everything else into a broader category.

To perform this task we need to add to the mapper some notion of what the valid U.S. state abbreviations are. We could of course hardcode this into the mapper but that does not seem right. Although we are for now going to treat all non-U.S. sightings as a single category, we may wish to extend that over time and perhaps do a breakdown by country. If we hardcode the abbreviations, we would need to recompile our mapper each time.

Using the Distributed Cache

Hadoop gives us an alternative mechanism to achieve the goal of sharing reference data across all tasks in the job, the Distributed Cache. This can be used to efficiently make available common read-only files that are used by the map or reduce tasks to all nodes. The files can be text data as in this case but could also be additional jars, binary data, or archives; anything is possible.

The files to be distributed are placed on HDFS and added to the DistributedCache within the job driver. Hadoop copies the files onto the local filesystem of each node prior to job execution, meaning every task has local access to the files.

An alternative is to bundle needed files into the job jar submitted to Hadoop. This does tie the data to the job jar making it more difficult to share across jobs and requires the jar to be rebuilt if the data changes.

Time for action – using the Distributed Cache to improve location output

Let's now use the Distributed Cache to share a list of U.S. state names and abbreviations across the cluster:

1. Create a datafile called `states.txt` on the local filesystem. It should have the state abbreviation and full name tab separated, one per line. Or retrieve the file from this book's homepage. The file should start like the following:

```
AL      Alabama
AK      Alaska
AZ      Arizona
AR      Arkansas
CA      California

…
```

2. Place the file on HDFS:

```
$ hadoop fs -put states.txt states.txt
```

3. Copy the previous `UFOLocation.java` file to UFOLocation2.java file and make the changes by adding the following import statements:

```
import java.io.* ;
import java.net.* ;
import java.util.* ;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.filecache.DistributedCache ;
```

4. Add the following line to the driver main method after the job name is set:

```
DistributedCache.addCacheFile(new URI ("/user/hadoop/states.txt"),
conf) ;
```

5. Replace the `map` class as follows:

```
    public static class MapClass extends MapReduceBase
implements Mapper<LongWritable, Text, Text, LongWritable>
    {

        private final static LongWritable one = new
LongWritable(1);
        private static Pattern locationPattern = Pattern.compile(
"[a-zA-Z]{2}[^a-zA-Z]*$") ;
        private Map<String, String> stateNames ;

        @Override
```

```
        public void configure( JobConf job)
        {
            try
            {
                Path[] cacheFiles = DistributedCache.
getLocalCacheFiles(job) ;
                setupStateMap( cacheFiles[0].toString()) ;
            } catch (IOException e)
{
System.err.println("Error reading state file.") ;
                    System.exit(1) ;
}
        }

        private void setupStateMap(String filename)
throws IOException
        {
            Map<String, String> states = new HashMap<String,
String>() ;
            BufferedReader reader = new BufferedReader( new
FileReader(filename)) ;
            String line = reader.readLine() ;
            while (line != null)
            {
                String[] split = line.split("\t") ;
                states.put(split[0], split[1]) ;
                line = reader.readLine() ;
            }

            stateNames = states ;
        }

        public void map(LongWritable key, Text value,
            OutputCollector<Text, LongWritable> output,
            Reporter reporter) throws IOException
        {
            String line = value.toString();
        String[] fields = line.split("\t") ;
        String location = fields[2].trim() ;
        if (location.length() >= 2)
        {

            Matcher matcher = locationPattern.matcher(location) ;
            if (matcher.find() )
```

```
                {
                    int start = matcher.start() ;
                    String state = location.substring(start, start+2)
;

                    output.collect(newText(lookupState(state.
toUpperCase())), one);
                }
            }
        }

        private String lookupState( String state)
        {
            String fullName = stateNames.get(state) ;

            return fullName == null? "Other": fullName ;
            }
}
```

6. Compile these classes and submit the job to Hadoop. Then retrieve the result file.

What just happened?

We first created the lookup file we will use in our job and placed it on HDFS. Files to be added to the Distributed Cache must initially be copied onto the HDFS filesystem.

After creating our new job file, we added the required class imports. Then we modified the driver class to add the file we want on each node to be added to the DistributedCache. The filename can be specified in multiple ways, but the easiest way is with an absolute path to the file location on HDFS.

There were a number of changes to our mapper class. We added an overridden `configure` method, which we use to populate a map that will be used to associate state abbreviations with their full name.

The `configure` method is called on task startup and the default implementation does nothing. In our overridden version, we retrieve the array of files that have been added to the Distributed Cache. As we know there is only one file in the cache we feel safe in using the first index in this array, and pass that to a `utility` method that parses the file and uses the contents to populate the state abbreviation lookup map. Notice that once the file reference is retrieved, we can access the file with standard Java I/O classes; it is after all just a file on the local filesystem.

We add another method to perform the lookup that takes the string extracted from the location field and returns either the full name of the state if there is a match or the string `Other` otherwise. This is called prior to the map result being written via the `OutputCollector` class.

The result of this job should be similar to the following data:

```
Alabama   548
Alaska    234
Arizona   2097
Arkansas  534
California       7679
…
Other     4531…
…
```

This works fine but we have been losing some information along the way. In our validation mapper, we simply drop any lines which don't meet our six field criteria. Though we don't care about individual lost records, we may care if the number of dropped records is very large. Currently, our only way of determining that is to sum the number of records for each recognized state and subtract from the total number of records in the file. We could also try to have this data flow through the rest of the job to be gathered in a special reduced key but that also seems wrong. Fortunately, there is a better way.

Counters, status, and other output

At the end of every MapReducejob, we see output related to counters such as the following output:

```
12/02/12 06:28:51 INFO mapred.JobClient: Counters: 22
12/02/12 06:28:51 INFO mapred.JobClient:   Job Counters
12/02/12 06:28:51 INFO mapred.JobClient:     Launched reduce tasks=1
12/02/12 06:28:51 INFO mapred.JobClient:     Launched map tasks=18
12/02/12 06:28:51 INFO mapred.JobClient:     Data-local map tasks=18
12/02/12 06:28:51 INFO mapred.JobClient:   SkippingTaskCounters
12/02/12 06:28:51 INFO mapred.JobClient:     MapProcessedRecords=61393
…
```

It is possible to add user-defined counters that will likewise be aggregated from all tasks and reported in this final output as well as in the MapReduce web UI.

Time for action – creating counters, task states, and writing log output

We'll modify our `UFORecordValidationMapper` to report statistics about skipped records and also highlight some other facilities for recording information about a job:

1. Create the following as the `UFOCountingRecordValidationMapper.java` file:

```
import java.io.IOException;

import org.apache.hadoop.io.* ;
import org.apache.hadoop.mapred.* ;
import org.apache.hadoop.mapred.lib.* ;

public class UFOCountingRecordValidationMapper extends
MapReduceBase
implements Mapper<LongWritable, Text, LongWritable, Text>
{

    public enum LineCounters
    {
        BAD_LINES,
        TOO_MANY_TABS,
        TOO_FEW_TABS
    } ;

    public void map(LongWritable key, Text value,
        OutputCollector<LongWritable, Text> output,
        Reporter reporter) throws IOException
    {
        String line = value.toString();

        if (validate(line, reporter))
Output.collect(key, value);
    }

    private boolean validate(String str, Reporter reporter)
    {
        String[] parts = str.split("\t") ;

        if (parts.length != 6)
        {
            if (parts.length < 6)
            {
```

```
reporter.incrCounter(LineCounters.TOO_FEW_TABS, 1) ;
            }
            else
            {
                reporter.incrCounter(LineCounters.TOO_MANY_TABS,
1) ;
            }

            reporter.incrCounter(LineCounters.BAD_LINES, 1) ;

if((reporter.getCounter(
LineCounters.BAD_LINES).getCounter()%10)
== 0)
            {
                reporter.setStatus("Got 10 bad lines.") ;
                System.err.println("Read another 10 bad lines.") ;
            }

            return false ;
        }
        return true ;
    }
        }
```

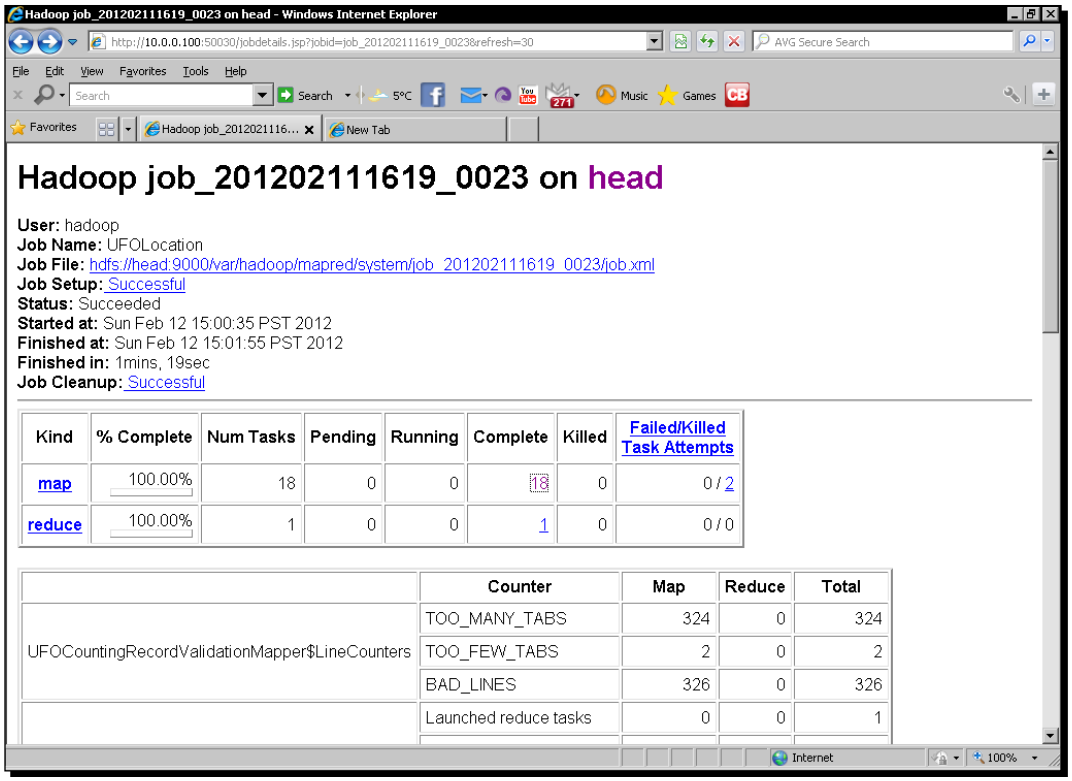
2. Make a copy of the `UFOLocation2.java` file as the `UFOLocation3.java` file to use this new mapper instead of `UFORecordValidationMapper`:

```
…
        JobConf mapconf1 = new JobConf(false) ;
        ChainMapper.addMapper( conf,
UFOCountingRecordValidationMapper.class,
            LongWritable.class, Text.class, LongWritable.class,
Text.class,
            true, mapconf1) ;
```

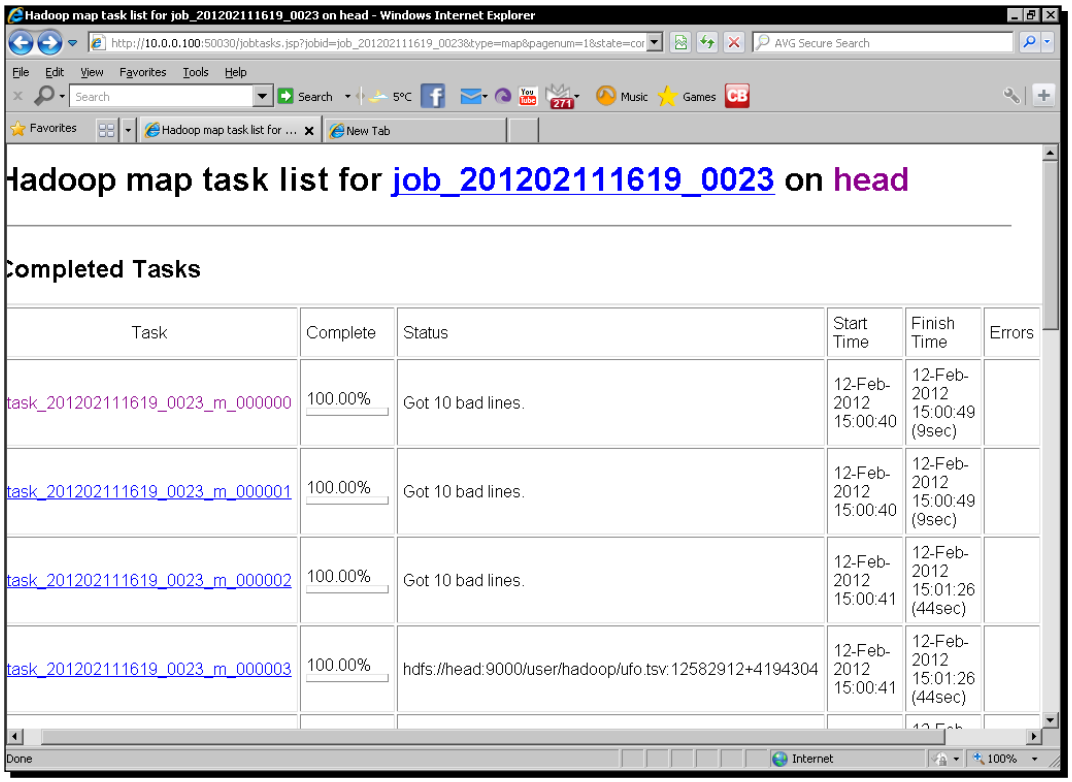
3. Compile the files, jar them up, and submit the job to Hadoop:

```
…
12/02/12 06:28:51 INFO mapred.JobClient: Counters: 22
12/02/12 06:28:51 INFO mapred.JobClient:   UFOCountingRecordValida
tionMapper$LineCounters
12/02/12 06:28:51 INFO mapred.JobClient:     TOO_MANY_TABS=324
12/02/12 06:28:51 INFO mapred.JobClient:     BAD_LINES=326
12/02/12 06:28:51 INFO mapred.JobClient:     TOO_FEW_TABS=2
12/02/12 06:28:51 INFO mapred.JobClient:   Job Counters
```

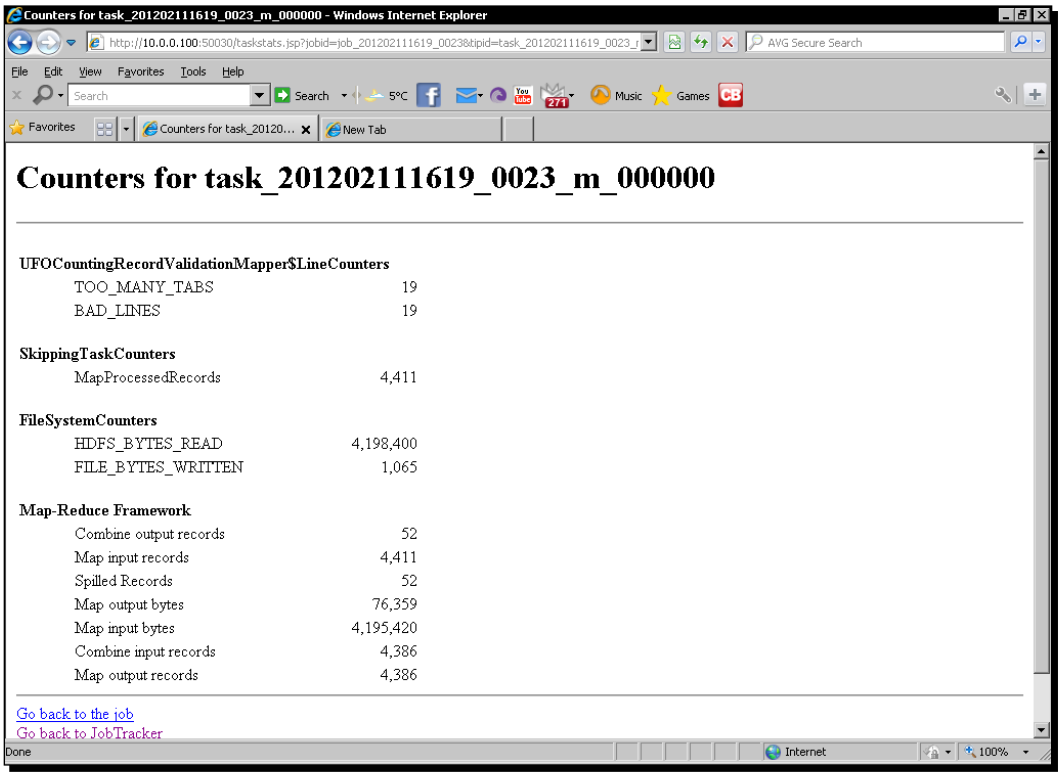
4. Use a web browser to go to the MapReduce web UI (remember by default it is on port 50030 on the JobTracker host). Select the job at the bottom of the **Completed Jobs** list and you should see a screen similar to the following screenshot:



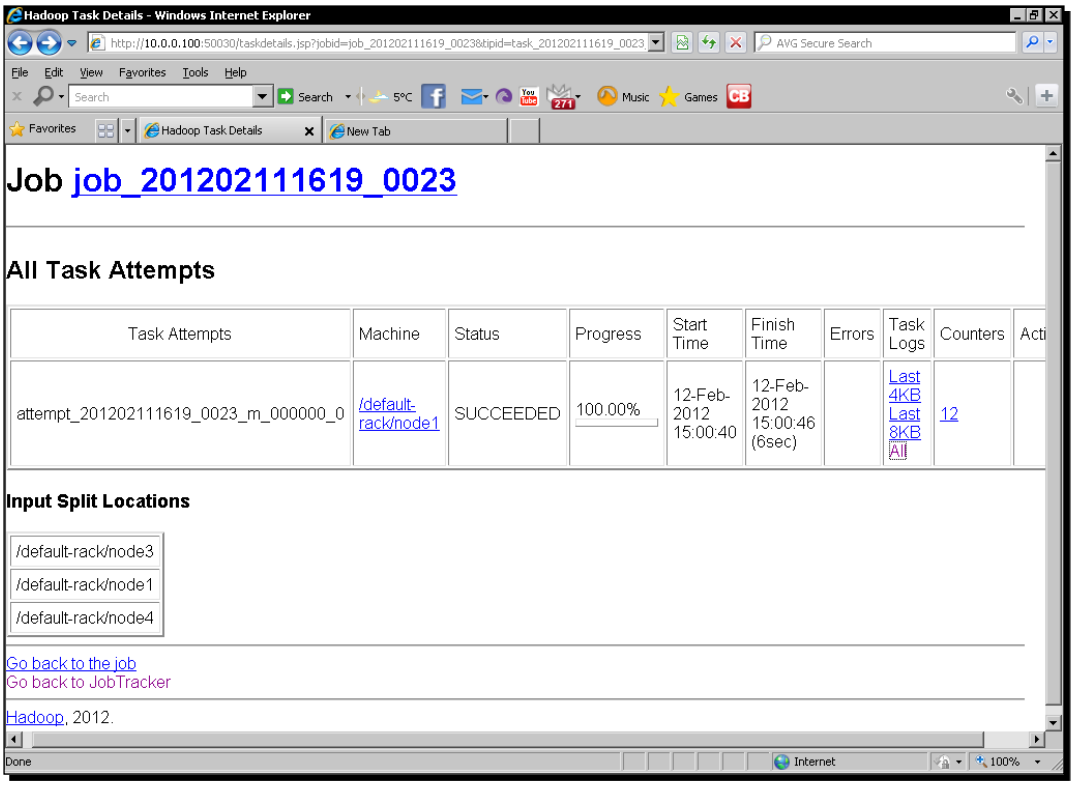
5. Click on the link to the map tasks and you should see an overview screen like the following screenshot:



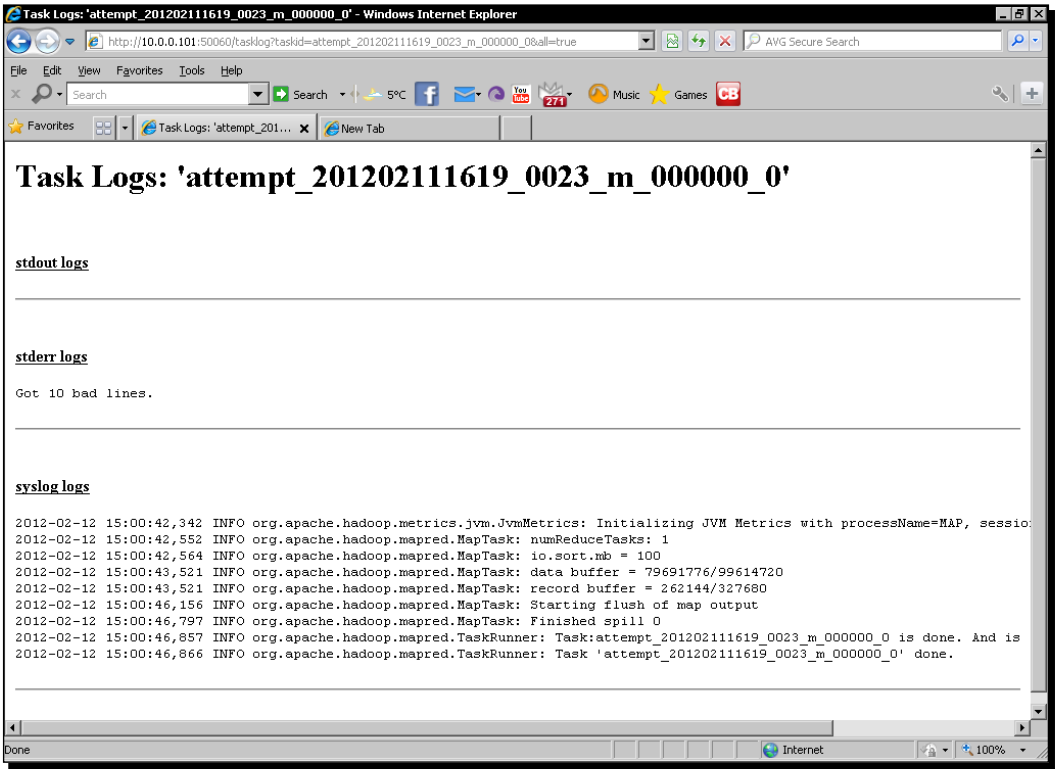
6. For one of the tasks with our custom status message, click on the link to its counters. This should give a screen similar to the one shown as follows:



7. Go back to the task list and click on the task ID to get the task overview similar to the following screenshot:



8. Under the **Task Logs** column are options for the amount of data to be displayed. Click on **All** and the following screenshot should be displayed:



9. Now log into one of the task nodes and look through the files stored under `hadoop/logs/userlogs`. There is a directory for each task attempt and several files within each; the one to look for is `stderr`.

What just happened?

The first thing we need to do in order to add new counters is to create a standard Java enumeration that will hold them. In this case we created what Hadoop would consider a counter group called **LineCounters** and within that there are three counters for the total number of bad lines, and finer grained counters for the number of lines with either too few or too many fields. This is all you need to do to create a new set of counters; define the enumeration and once you start setting the counter values, they will be automatically understood by the framework.

To add to a counter we simply increment it via the `Reporter` object, in each case here we add one each time we encounter a bad line, one with fewer than six fields, and one with more than six fields.

We also retrieve the `BAD_LINE` counter for a task and if it is a multiple of 10, do the following:

- Set the task status to reflect this fact
- Write a similar message to `stderr` with the standard Java `System.err.println` mechanism

We then go to the MapReduce UI and validate whether we can see both the counter totals in the job overview as well as tasks with the custom state message in the task list.

We then explored the web UI, looking at the counters for an individual job, then under the detail page for a task we saw, we can click on through the log files for the task.

We then looked at one of the nodes to see that Hadoop also captures the logs from each task in a directory on the filesystem under the `{HADOOP_HOME}/logs/userlogs` directory. Under subdirectories for each task attempt, there are files for the standard streams as well as the general task logs. As you will see, a busy node can end up with a large number of task log directories and it is not always easy to identify the task directories of interest. The web interface proved itself to be a more efficient view on this data.



If you are using the Hadoop `context` object API, then counters are accessed through the `Context.getCounter().increment()` method.

Too much information!

After not worrying much about how to get status and other information out of our jobs, it may suddenly seem like we've got too many confusing options. The fact of the matter is that when running a fully distributed cluster in particular, there really is no way around the fact that the data may be spread across every node. With Java code we can't as easily mock its usage on the command line as we did with our Ruby Streaming tasks; so care needs to be taken to think about what information will be needed at runtime. This should include details concerning both the general job operation (additional statistics) as well as indicators of problems that may need further investigation.

Counters, task status messages, and good old-fashioned Java logging can work together. If there is a situation you care about, set it as a counter that will record each time it occurs and consider setting the status message of the task that encountered it. If there is some specific data, write that to `stderr`. Since counters are so easily visible, you can know pretty quickly post job completion if the situation of interest occurred. From this, you can go to the web UI and see all the tasks in which the situation was encountered at a glance. From here, you can click through to examine the more detailed logs for the task.

In fact, you don't need to wait until the job completes; counters and task status messages are updated in the web UI as the job proceeds, so you can start the investigation as soon as either counters or task status messages alert you to the situation. This is particularly useful in very long running jobs where the errors may cause you to abort the job.

Summary

This chapter covered development of a MapReduce job, highlighting some of the issues and approaches you are likely to face frequently. In particular, we learned how Hadoop Streaming provides a means to use scripting languages to write map and reduce tasks, and how using Streaming can be an effective tool for early stages of job prototyping and initial data analysis.

We also learned that writing tasks in a scripting language can provide the additional benefit of using command-line tools to directly test and debug the code. Within the Java API, we looked at the `ChainMapper` class that provides an efficient way of decomposing a complex map task into a series of smaller, more focused ones.

We then saw how the Distributed Cache provides a mechanism for efficient sharing of data across all nodes. It copies files from HDFS onto the local filesystem on each node, providing local access to the data. We also learned how to add job counters by defining a Java enumeration for the counter group and using framework methods to increment their values, and how to use a combination of counters, task status messages, and debug logs to develop an efficient job analysis workflow.

We expect most of these techniques and ideas to be the ones that you will encounter frequently as you develop MapReduce jobs. In the next chapter, we will explore a series of more advanced techniques that are less often encountered but are invaluable when they are.

5
Advanced MapReduce Techniques

Now that we have looked at a few details of the fundamentals of MapReduce and its usage, it's time to examine some more techniques and concepts involved in MapReduce. This chapter will cover the following topics:

- Performing joins on data
- Implementing graph algorithms in MapReduce
- How to represent complex datatypes in a language-independent fashion

Along the way, we'll use the case studies as examples in order to highlight other aspects such as tips and tricks and identifying some areas of best practice.

Simple, advanced, and in-between

Including the word "advanced" in a chapter title is a little dangerous, as complexity is a subjective concept. So let's be very clear about the material covered here. We don't, for even a moment, suggest that this is the pinnacle of distilled wisdom that would otherwise take years to acquire. Conversely, we also don't claim that some of the techniques and problems covered in this chapter will have occurred to someone new to the world of Hadoop.

For the purposes of this chapter, therefore, we use the term "advanced" to cover things that you don't see in the first days or weeks, or wouldn't necessarily appreciate if you did. These are some techniques that provide both specific solutions to particular problems but also highlight ways in which the standard Hadoop and related APIs can be employed to address problems that are not obviously suited to the MapReduce processing model. Along the way, we'll also point out some alternative approaches that we don't implement here but which may be useful sources for further research.

Our first case study is a very common example of this latter case; performing join-type operations within MapReduce.

Joins

Few problems use a single set of data. In many cases, there are easy ways to obviate the need to try and process numerous discrete yet related data sets within the MapReduce framework.

The analogy here is, of course, to the concept of **join** in a relational database. It is very natural to segment data into numerous tables and then use SQL statements that join tables together to retrieve data from multiple sources. The canonical example is where a main table has only ID numbers for particular facts, and joins against other tables are used to extract data about the information referred to by the unique ID.

When this is a bad idea

It is possible to implement joins in MapReduce. Indeed, as we'll see, the problem is less about the ability to do it and more the choice of which of many potential strategies to employ.

However, MapReduce joins are often difficult to write and easy to make inefficient. Work with Hadoop for any length of time, and you will come across a situation where you need to do it. However, if you very frequently need to perform MapReduce joins, you may want to ask yourself if your data is well structured and more relational in nature than you first assumed. If so, you may want to consider **Apache Hive** (the main topic of *Chapter 8*, *A Relational View on Data with Hive*) or **Apache Pig** (briefly mentioned in the same chapter). Both provide additional layers atop Hadoop that allow data processing operations to be expressed in high-level languages; in the case of Hive, through a variant of SQL.

Map-side versus reduce-side joins

That caveat out of the way, there are two basic approaches to join data in Hadoop and these are given their names depending on where in the job execution the join occurs. In either case, we need to bring multiple data streams together and perform the join through some logic. The basic difference between these two approaches is whether the multiple data streams are combined within the mapper or reducer functions.

Map-side joins, as the name implies, read the data streams into the mapper and uses logic within the mapper function to perform the join. The great advantage of a map-side join is that by performing all joining—and more critically data volume reduction—within the mapper, the amount of data transferred to the reduce stage is greatly minimized. The drawback of map-side joins is that you either need to find a way of ensuring one of the data sources is very small or you need to define the job input to follow very specific criteria. Often, the only way to do that is to preprocess the data with another MapReduce job whose sole purpose is to make the data ready for a map-side join.

In contrast, a **reduce-side join** has the multiple data streams processed through the map stage without performing any join logic and does the joining in the reduce stage. The potential drawback of this approach is that all the data from each source is pulled through the shuffle stage and passed into the reducers, where much of it may then be discarded by the join operation. For large data sets, this can become a very significant overhead.

The main advantage of the reduce-side join is its simplicity; you are largely responsible for how the jobs are structured and it is often quite straightforward to define a reduce-side join approach for related data sets. Let's look at an example.

Matching account and sales information

A common situation in many companies is that sales records are kept separate from the client data. There is, of course, a relationship between the two; usually a sales record contains the unique ID of the user account through which the sale was performed.

In the Hadoop world, these would be represented by two types of data files: one containing records of the user IDs and information for sales, and the other would contain the full data for each user account.

Frequent tasks require reporting that uses data from both these sources; say, for example, we wanted to see the total number of sales and total value for each user but do not want to associate it with an anonymous ID number, but rather with a name. This may be valuable when customer service representatives wish to call the most frequent customers—data from the sales records—but want to be able to refer to the person by name and not just a number.

Time for action – reduce-side join using MultipleInputs

We can perform the report explained in the previous section using a reduce-side join by performing the following steps:

1. Create the following tab-separated file and name it `sales.txt`:

```
00135.992012-03-15
00212.492004-07-02
00413.422005-12-20
003499.992010-12-20
00178.952012-04-02
00221.992006-11-30
00293.452008-09-10
0019.992012-05-17
```

2. Create the following tab-separated file and name it `accounts.txt`:

```
001John AllenStandard2012-03-15
002Abigail SmithPremium2004-07-13
003April StevensStandard2010-12-20
004Nasser HafezPremium2001-04-23
```

3. Copy the datafiles onto HDFS.

```
$ hadoop fs -mkdir sales
$ hadoop fs -put sales.txt sales/sales.txt
$ hadoop fs -mkdir accounts
$ hadoop fs -put accounts/accounts.txt
```

4. Create the following file and name it `ReduceJoin.java`:

```
import java.io.* ;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.input.*;

public class ReduceJoin
{

    public static class SalesRecordMapper
extends Mapper<Object, Text, Text, Text>
{

        public void map(Object key, Text value, Context context)
throws IOException, InterruptedException
        {
            String record = value.toString() ;
            String[] parts = record.split("\t") ;

            context.write(new Text(parts[0]), new
Text("sales\t"+parts[1])) ;
        }
    }

    public static class AccountRecordMapper
extends Mapper<Object, Text, Text, Text>
{
```

```
        public void map(Object key, Text value, Context context)
throws IOException, InterruptedException
        {
            String record = value.toString() ;
            String[] parts = record.split("\t") ;

            context.write(new Text(parts[0]), new
Text("accounts\t"+parts[1])) ;
        }
    }

    public static class ReduceJoinReducer
    extends Reducer<Text, Text, Text, Text>
    {

        public void reduce(Text key, Iterable<Text> values,
            Context context)
            throws IOException, InterruptedException
        {
            String name = "" ;
double total = 0.0 ;
            int count = 0 ;

            for(Text t: values)
            {
                String parts[] = t.toString().split("\t") ;

                if (parts[0].equals("sales"))
                {
                    count++ ;
                    total+= Float.parseFloat(parts[1]) ;
                }
                else if (parts[0].equals("accounts"))
                {
                    name = parts[1] ;
                }
            }

            String str = String.format("%d\t%f", count, total) ;
            context.write(new Text(name), new Text(str)) ;
        }
    }
```

```
    public static void main(String[] args) throws Exception
{
        Configuration conf = new Configuration();
        Job job = new Job(conf, "Reduce-side join");
        job.setJarByClass(ReduceJoin.class);
        job.setReducerClass(ReduceJoinReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
MultipleInputs.addInputPath(job, new Path(args[0]),
TextInputFormat.class, SalesRecordMapper.class) ;
MultipleInputs.addInputPath(job, new Path(args[1]),
TextInputFormat.class, AccountRecordMapper.class) ;
        Path outputPath = new Path(args[2]);
        FileOutputFormat.setOutputPath(job, outputPath);
outputPath.getFileSystem(conf).delete(outputPath);

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

5. Compile the file and add it to a JAR file.

```
$ javac ReduceJoin.java
$ jar -cvf join.jar *.class
```

6. Run the job by executing the following command:

```
$ hadoop jar join.jarReduceJoin sales accounts outputs
```

7. Examine the result file.

```
$ hadoop fs -cat /user/garry/outputs/part-r-00000
John Allen    3      124.929998
Abigail Smith 3      127.929996
April Stevens 1      499.989990
Nasser Hafez  1      13.420000
```

What just happened?

Firstly, we created the datafiles to be used in this example. We created two small data sets as this makes it easier to track the result output. The first data set we defined was the account details with four columns, as follows:

- The account ID
- The client name

- The type of account
- The date the account was opened

We then created a sales record with three columns:

- The account ID of the purchaser
- The value of the sale
- The date of the sale

Naturally, real account and sales records would have many more fields than the ones mentioned here. After creating the files, we placed them onto HDFS.

We then created the `ReduceJoin.java` file, which looks very much like the previous MapReduce jobs we have used. There are a few aspects to this job that make it special and allow us to implement a join.

Firstly, the class has two defined mappers. As we have seen before, jobs can have multiple mappers executed in a chain; but in this case, we wish to apply different mappers to each of the input locations. Accordingly, we have the sales and account data defined into the `SalesRecordMapper` and `AccountRecordMapper` classes. We used the `MultipleInputs` class from the `org.apache.hadoop.mapreduce.lib.io` package as follows:

```
MultipleInputs.addInputPath(job, new Path(args[0]),
TextInputFormat.class, SalesRecordMapper.class) ;
MultipleInputs.addInputPath(job, new Path(args[1]),
TextInputFormat.class, AccountRecordMapper.class) ;
```

As you can see, unlike in previous examples where we add a single input location, the `MultipleInputs` class allows us to add multiple sources and associate each with a distinct input format and mapper.

The mappers are pretty straightforward; the `SalesRecordMapper` class emits an output of the form `<account number>, <sales value>` while the `AccountRecordMapper` class emits an output of the form `<account number>, <client name>`. We therefore have the order value and client name for each sale being passed into the reducer where the actual join will happen.

Notice that both mappers actually emit more than the required values.
The `SalesRecordMapper` class prefixes its value output with `sales` while the `AccountRecordMapper` class uses the tag `account`.

If we look at the reducer, we can see why this is so. The reducer retrieves each record for a given key, but without these explicit tags we would not know if a given value came from the sales or account mapper and hence would not understand how to treat the data value.

The `ReduceJoinReducer` class therefore treats the values in the `Iterator` object differently, depending on which mapper they came from. Values from the `AccountRecordMapper` class—and there should be only one—are used to populate the client name in the final output. For each sales record—likely to be multiple, as most clients buy more than a single item—the total number of orders is counted as is the overall combined value. The output from the reducer is therefore a key of the account holder name and a value string containing the number of orders and the total order value.

We compile and execute the class; notice how we provide three arguments representing the two input directories as well as the single output source. Because of how the `MultipleInputs` class is configured, we must also ensure we specify the directories in the right order; there is no dynamic mechanism to determine which type of file is in which location.

After execution, we examine the output file and confirm that it does indeed contain the overall totals for named clients as expected.

DataJoinMapper and TaggedMapperOutput

There is a way of implementing a reduce-side join in a more sophisticated and object-oriented fashion. Within the `org.apache.hadoop.contrib.join` package are classes such as `DataJoinMapperBase` and `TaggedMapOutput` that provide an encapsulated means of deriving the tags for map output and having them processed at the reducer. This mechanism means you don't have to define explicit tag strings as we did previously and then carefully parse out the data received at the reducer to determine from which mapper the data came; there are methods in the provided classes that encapsulate this functionality.

This capability is particularly valuable when using numeric or other non-textual data. For creating our own explicit tags as in the previous example, we would have to convert types such as integers into strings to allow us to add the required prefix tag. This will be more inefficient than using the numeric types in their normal form and relying on the additional classes to implement the tag.

The framework allows for quite sophisticated tag generation as well as concepts such as tag grouping that we didn't implement previously. There is additional work required to use this mechanism that includes overriding additional methods and using a different map base class. For straightforward joins such as in the previous example, this framework may be overkill, but if you find yourself implementing very complex tagging logic, it may be worth a look.

Implementing map-side joins

For a join to occur at a given point, we must have access to the appropriate records from each data set at that point. This is where the simplicity of the reduce-side join comes into its own; though it incurs the expense of additional network traffic, processing it by definition ensures that the reducer has all records associated with the join key.

If we wish to perform our join in the mapper, it isn't as easy to make this condition hold true. We can't assume that our input data is sufficiently well structured to allow associated records to be read simultaneously. We generally have two classes of approach here: obviate the need to read from multiple external sources or preprocess the data so that it is amenable for map-side joining.

Using the Distributed Cache

The simplest way of realizing the first approach is to take all but one data set and make it available in the Distributed Cache that we used in the previous chapter. The approach can be used for multiple data sources, but for simplicity let's discuss just two.

If we have one large data set and one smaller one, such as with the sales and account info earlier, one option would be to package up the account info and push it into the Distributed Cache. Each mapper would then read this data into an efficient data structure, such as a hash table that uses the join key as the hash key. The sales records are then processed, and during the processing of record each the needed account information can be retrieved from the hash table.

This mechanism is very effective and when one of the smaller data sets can easily fit into memory, it is a great approach. However, we are not always that lucky, and sometimes the smallest data set is still too large to be copied to every worker machine and held in memory.

Have a go hero - Implementing map-side joins

Take the previous sales/account record example and implement a map-side join using the Distributed Cache. If you load the account records into a hash table that maps account ID numbers to client names, you can use the account ID to retrieve the client name. Do this within the mapper while processing the sales records.

Pruning data to fit in the cache

If the smallest data set is still too big to be used in the Distributed Cache, all is not necessarily lost. Our earlier example, for instance, extracted only two fields from each record and discarded the other fields not required by the job. In reality, an account will be described by many attributes, and this sort of reduction will limit the data size dramatically. Often the data available to Hadoop is this full data set, but what we need is only a subset of the fields.

In such a case, therefore, it may be possible to extract from the full data set only the fields that are needed during the MapReduce job, and in doing so create a pruned data set that is small enough to be used in the cache.



This is a very similar concept to the underlying **column-oriented databases**. Traditional relational databases store data a row at a time, meaning that the full row needs to be read to extract a single column. A column-based database instead stores each column separately, allowing a query to read only the columns in which it is interested.

If you take this approach, you need to consider what mechanism will be used to generate the data subset and how often this will be done. The obvious approach is to write another MapReduce job that does the necessary filtering and this output is then used in the Distributed Cache for the follow-on job. If the smaller data set changes only rarely, you may be able to get away with generating the pruned data set on a scheduled basis; for example, refresh it every night. Otherwise, you will need to make a chain of two MapReduce jobs: one to produce the pruned data set and the other to perform the join operation using the large set and the data in the Distributed Cache.

Using a data representation instead of raw data

Sometimes, one of the data sources is not used to retrieve additional data but is instead used to derive some fact that is then used in a decision process. We may, for example, be looking to filter sales records to extract only those for which the shipping address was in a specific locale.

In such a case, we can reduce the required data size down to a list of the applicable sales records that may more easily fit into the cache. We can again store it as a hash table, where we are just recording the fact that the record is valid, or even use something like a sorted list or a tree. In cases where we can accept some false positives while still guaranteeing no false negatives, a **Bloom filter** provides an extremely compact way of representing such information.

As can be seen, applying this approach to enable a map-side join requires creativity and not a little luck in regards to the nature of the data set and the problem at hand. But remember that the best relational database administrators spend significant time optimizing queries to remove unnecessary data processing; so it's never a bad idea to ask if you truly need to process all that data.

Using multiple mappers

Fundamentally, the previous techniques are trying to remove the need for a full cross data set join. But sometimes this is what you have to do; you may simply have very large data sets that cannot be combined in any of these clever ways.

There are classes within the `org.apache.hadoop.mapreduce.lib.join` package that support this situation. The main class of interest is `CompositeInputFormat`, which applies a user-defined function to combine records from multiple data sources.

The main limitation of this approach is that the data sources must already be indexed based on the common key, in addition to being both sorted and partitioned in the same way. The reason for this is simple: when reading from each source, the framework needs to know if a given key is present at each location. If we know that each partition is sorted and contains the same key range, simple iteration logic can do the required matching.

This situation is obviously not going to happen by accident, so again you may find yourself writing preprocess jobs to transform all the input data sources into the correct sort and partition structure.



This discussion starts to touch on distributed and parallel join algorithms; both topics are of extensive academic and commercial research. If you are interested in the ideas and want to learn more of the underlying theory, go searching on `http://scholar.google.com`.

To join or not to join...

After our tour of joins in the MapReduce world, let's come back to the original question: are you really sure you want to be doing this? The choice is often between a relatively easily implemented yet inefficient reduce-side join, and more efficient but more complex map-side alternatives. We have seen that joins can indeed be implemented in MapReduce, but they aren't always pretty. This is why we advise the use of something like Hive or Pig if these types of problems comprise a large portion of your workload. Obviously, we can use tools such as those that do their own translation into MapReduce code under the hood and directly implement both map-side and reduce-side joins, but it's often better to use a well-engineered and well-optimized library for such workloads instead of building your own. That is after all why you are using Hadoop and not writing your own distributed processing framework!

Graph algorithms

Any good computer scientist will tell you that the graph data structure is one of the most powerful tools around. Many complex systems are best represented by graphs and a body of knowledge going back at least decades (centuries if you get more mathematical about it) provides very powerful algorithms to solve a vast variety of graph problems. But by their very nature, graphs and their algorithms are often very difficult to imagine in a MapReduce paradigm.

Graph 101

Let's take a step back and define some terminology. A graph is a structure comprising of nodes (also called vertices) that are connected by links called **edges**. Depending on the type of graph, the edges may be bidirectional or unidirectional and may have weights associated with them. For example, a city road network can be seen as a graph where the roads are the edges, and intersections and points of interest are nodes. Some streets are one-way and some are not, some have tolls, some are closed at certain times of day, and so forth.

For transportation companies, there is much money to be made by optimizing the routes taken from one point to another. Different graph algorithms can derive such routes by taking into account attributes such as one-way streets and other costs expressed as weights that make a given road more attractive or less so.

For a more current example, think of the social graph popularized by sites such as Facebook where the nodes are people and the edges are the relationships between them.

Graphs and MapReduce – a match made somewhere

The main reason graphs don't look like many other MapReduce problems is due to the stateful nature of graph processing, which can be seen in the path-based relationship between elements and often between the large number of nodes processed together for a single algorithm. Graph algorithms tend to use notions of the global state to make determinations about which elements to process next and modify such global knowledge at each step.

In particular, most of the well-known algorithms often execute in an incremental or reentrant fashion, building up structures representing processed and pending nodes, and working through the latter while reducing the former.

MapReduce problems, on the other hand, are conceptually stateless and typically based upon a divide-and-conquer approach where each Hadoop worker host processes a small subset of the data, writing out a portion of the final result where the total job output is viewed as the simple collection of these smaller outputs. Therefore, when implementing graph algorithms in Hadoop, we need to express algorithms that are fundamentally stateful and conceptually single-threaded in a stateless parallel and distributed framework. That's the challenge!

Most of the well-known graph algorithms are based upon search or traversal of the graph, often to find routes—frequently ranked by some notion of cost—between nodes. The most fundamental graph traversal algorithms are **depth-first search** (**DFS**) and **breadth-first search** (**BFS**).The difference between the algorithms is the ordering in which a node is processed in relationship to its neighbors.

We will look at representing an algorithm that implements a specialized form of such a traversal; for a given starting node in the graph, determine the distance between it and every other node in the graph.

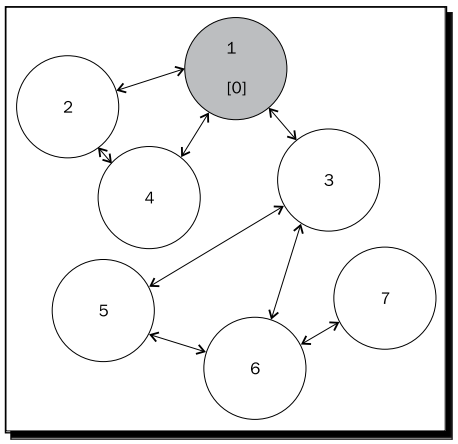


As can be seen, the field of graph algorithms and theory is a huge one that we barely scratch the surface of here. If you want to find out more, the Wikipedia entry on graphs is a good starting point; it can be found at `http://en.wikipedia.org/wiki/Graph_(abstract_data_type)`.

Representing a graph

The first problem we face is how to represent the graph in a way we can efficiently process using MapReduce. There are several well-known graph representations known as pointer-based, adjacency matrix, and adjacency list. In most implementations, these representations often assume a single process space with a global view of the whole graph; we need to modify the representation to allow individual nodes to be processed in discrete map and reduce tasks.

We'll use the graph shown here in the following examples. The graph does have some extra information that will be explained later.



Our graph is quite simple; it has only seven nodes, and all but one of the edges is bidirectional. We are also using a common coloring technique that is used in standard graph algorithms, as follows:

- White nodes are yet to be processed
- Gray nodes are currently being processed
- Black nodes have been processed

As we process our graph in the following steps, we will expect to see the nodes move through these stages.

Time for action – representing the graph

Let's define a textual representation of the graph that we'll use in the following examples.

Create the following as `graph.txt`:

```
12,3,40C
21,4
31,5,6
41,2
53,6
63,5
76
```

What just happened?

We defined a file structure that will represent our graph, based somewhat on the adjacency list approach. We assumed that each node has a unique ID and the file structure has four fields, as follows:

- The node ID
- A comma-separated list of neighbors
- The distance from the start node
- The node status

In the initial representation, only the starting node has values for the third and fourth columns: its distance from itself is 0 and its status is "C", which we'll explain later.

Our graph is directional—more formally referred to as a directed graph—that is to say, if node 1 lists node 2 as a neighbor, there is only a return path if node 2 also lists node 1 as its neighbor. We see this in the graphical representation where all but one edge has an arrow on both ends.

Overview of the algorithm

Because this algorithm and corresponding MapReduce job is quite involved, we'll explain it before showing the code, and then demonstrate it in use later.

Given the previous representation, we will define a MapReduce job that will be executed multiple times to get the final output; the input to a given execution of the job will be the output from the previous execution.

Based on the color code described in the previous section, we will define three states for a node:

- **Pending**: The node is yet to be processed; it is in the default state (white)
- **Currently processing**: The node is being processed (gray)
- **Done**: The final distance for the node has been determined (black)

The mapper

The mapper will read in the current representation of the graph and treat each node as follows:

- If the node is marked as Done, it gives output with no changes.
- If the node is marked as Currently processing, its state is changed to Done and gives output with no other changes. Each of its neighbors gives output as per the current record with its distance incremented by one, but with no neighbors; node 1 doesn't know node 2's neighbors, for example.
- If the node is marked Pending, its state is changed to Currently processing and it gives output with no further changes.

The reducer

The reducer will receive one or more records for each node ID, and it will combine their values into the final output node record for that stage.

The general algorithm for the reducer is as follows:

- A Done record is the final output and no further processing of the values is performed
- For other nodes, the final output is built up by taking the list of neighbors, where it is to be found, and the highest distance and state

Iterative application

If we apply this algorithm once, we will get node 1 marked as Done, several more (its immediate neighbors) as Current, and a few others as Pending. Successive applications of the algorithm will see all nodes move to their final state; as each node is encountered, its neighbors are brought into the processing pipeline. We will show this later.

Time for action – creating the source code

We'll now see the source code to implement our graph traversal. Because the code is lengthy, we'll break it into multiple steps; obviously they should all be together in a single source file.

1. Create the following as `GraphPath.java` with these imports:

```
import java.io.* ;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.output.*;

public class GraphPath
{
```

2. Create an inner class to hold an object-oriented representation of a node:

```
// Inner class to represent a node
    public static class Node
    {
// The integer node id
        private String id ;
// The ids of all nodes this node has a path to
        private String neighbours ;
// The distance of this node to the starting node
        private int distance ;
// The current node state
        private String state ;

// Parse the text file representation into a Node object
        Node( Text t)
        {
            String[] parts = t.toString().split("\t") ;
this.id = parts[0] ;
this.neighbours = parts[1] ;
            if (parts.length<3 || parts[2].equals(""))
this.distance = -1 ;
            else
this.distance = Integer.parseInt(parts[2]) ;
```

```
            if (parts.length< 4 || parts[3].equals(""))
this.state = "P" ;
            else
this.state = parts[3] ;
        }

// Create a node from a key and value object pair
        Node(Text key, Text value)
        {
            this(new Text(key.toString()+"\t"+value.toString())) ;
        }

        Public String getId()
        {return this.id ;
        }

        public String getNeighbours()
        {
            return this.neighbours ;
        }

        public int getDistance()
        {
            return this.distance ;
        }

        public String getState()
        {
            return this.state ;
        }
    }
```

3. Create the mapper for the job. The mapper will create a new `Node` object for its input and then examine it, and based on its state do the appropriate processing.

```
    public static class GraphPathMapper
extends Mapper<Object, Text, Text, Text>
{

       public void map(Object key, Text value, Context context)
throws IOException, InterruptedException
{
         Node n = new Node(value) ;

         if (n.getState().equals("C"))
```

```
          {
//  Output the node with its state changed to Done
            context.write(new Text(n.getId()), new
Text(n.getNeighbours()+"\t"+n.getDistance()+"\t"+"D")) ;

                for (String neighbour:n.getNeighbours().
split(","))
                {
// Output each neighbour as a Currently processing node
// Increment the distance by 1; it is one link further away
                    context.write(new Text(neighbour), new
Text("\t"+(n.getDistance()+1)+"\tC")) ;
                }
            }
            else
            {
// Output a pending node unchanged
                context.write(new Text(n.getId()), new
Text(n.getNeighbours()+"\t"+n.getDistance()
+"\t"+n.getState())) ;
            }

        }
    }
```

4. Create the reducer for the job. As with the mapper, this reads in a representation of a node and gives as output a different value depending on the state of the node. The basic approach is to collect from the input the largest value for the state and distance columns, and through this converge to the final solution.

```
    public static class GraphPathReducer
extends Reducer<Text, Text, Text, Text>
{

        public void reduce(Text key, Iterable<Text> values,
            Context context)
            throws IOException, InterruptedException
{
// Set some default values for the final output
            String neighbours = null ;
int distance = -1 ;
String state = "P" ;

            for(Text t: values)
{
                Node n = new Node(key, t) ;
```

```
                if (n.getState().equals("D"))
                {
// A done node should be the final output; ignore the remaining
// values
neighbours = n.getNeighbours() ;
                    distance = n.getDistance() ;
                    state = n.getState() ;
                    break ;
                }

// Select the list of neighbours when found
                if (n.getNeighbours() != null)
neighbours = n.getNeighbours() ;

// Select the largest distance
                if (n.getDistance() > distance)
distance = n.getDistance() ;

// Select the highest remaining state
                if (n.getState().equals("D") ||
(n.getState().equals("C") &&state.equals("P")))
state=n.getState() ;
            }

// Output a new node representation from the collected parts
            context.write(key, new
Text(neighbours+"\t"+distance+"\t"+state)) ;
        }
    }
```

5. Create the job driver:

```
    public static void main(String[] args) throws Exception
{
        Configuration conf = new Configuration();
        Job job = new Job(conf, "graph path");
        job.setJarByClass(GraphPath.class);
        job.setMapperClass(GraphPathMapper.class);
        job.setReducerClass(GraphPathReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

What just happened?

The job here implements the previously described algorithm that we'll execute in the following sections. The job setup is pretty standard, and apart from the algorithm definition the only new thing here is the use of an inner class to represent nodes.

The input to a mapper or reducer is often a flattened representation of a more complex structure or object. We could just use that representation, but in this case this would result in the mapper and reducer bodies being full of text and string manipulation code that would obscure the actual algorithm.

The use of the `Node` inner class allows the mapping from the flat file to object representation that is to be encapsulated in an object that makes sense in terms of the business domain. This also makes the mapper and reducer logic clearer as comparisons between object attributes are more semantically meaningful than comparisons with slices of a string identified only by absolute index positions.

Time for action – the first run

Let's now perform the initial execution of this algorithm on our starting representation of the graph:

1. Put the previously created `graph.txt` file onto HDFS:

```
$ hadoop fs -mkdirgraphin
$ hadoop fs -put graph.txtgraphin/graph.txt
```

2. Compile the job and create the JAR file:

```
$ javac GraphPath.java
$ jar -cvf graph.jar *.class
```

3. Execute the MapReduce job:

```
$ hadoop jar graph.jarGraphPathgraphingraphout1
```

4. Examine the output file:

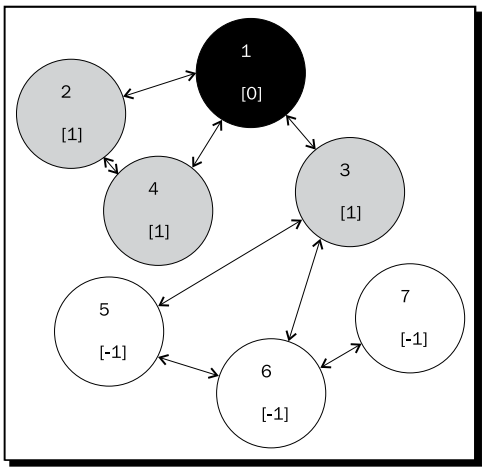
```
$ hadoop fs -cat /home/user/hadoop/graphout1/part-r00000
12,3,40D
21,41C
31,5,61C
41,21C
53,6-1P
63,5-1P
76-1P
```

What just happened?

After putting the source file onto HDFS and creating the job JAR file, we executed the job in Hadoop. The output representation of the graph shows a few changes, as follows:

- Node 1 is now marked as Done; its distance from itself is obviously 0
- Nodes 2, 3, and 4 – the neighbors of node 1 — are marked as Currently processing
- All other nodes are Pending

Our graph now looks like the following figure:



Given the algorithm, this is to be expected; the first node is complete and its neighboring nodes, extracted through the mapper, are in progress. All other nodes are yet to begin processing.

Time for action – the second run

If we take this representation as the input to another run of the job, we would expect nodes 2, 3, and 4 to now be complete, and for their neighbors to now be in the Current state. Let's see; execute the following steps:

1. Execute the MapReduce job by executing the following command:

```
$ hadoop jar graph.jarGraphPathgraphout1graphout2
```

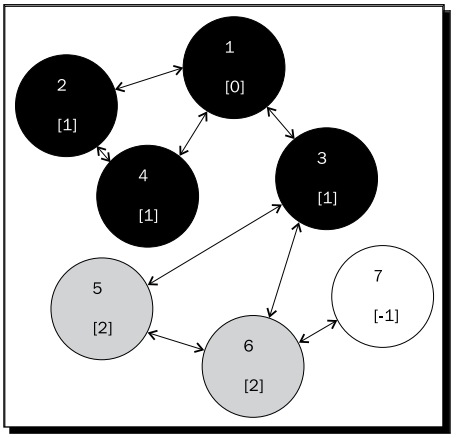
2. Examine the output file:

```
$ hadoop fs -cat /home/user/hadoop/graphout2/part-r000000
12,3,40D
21,41D
```

```
31,5,61D
41,21D
53,62C
63,52C
76-1P
```

What just happened?

As expected, nodes 1 through 4 are complete, nodes 5 and 6 are in progress, and node 7 is still pending, as seen in the following figure:



If we run the job again, we should expect nodes 5 and 6 to be Done and any unprocessed neighbors to become Current.

Time for action – the third run

Let's validate that assumption by running the algorithm for the third time.

1. Execute the MapReduce job:

```
$ hadoop jar graph.jarGraphPathgraphout2graphout3
```

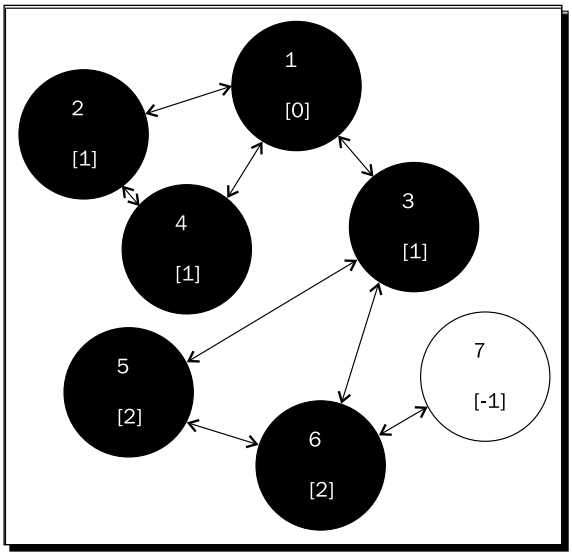
2. Examine the output file:

```
$ hadoop fs -cat /user/hadoop/graphout3/part-r-00000
12,3,40D
21,41D
31,5,61D
```

```
41,21D
53,62D
63,52D
76-1P
```

What just happened?

We now see nodes 1 through 6 are complete. But node 7 is still pending and no nodes are currently being processed, as shown in the following figure:



The reason for this state is that though node 7 has a link to node 6, there is no edge in the reverse direction. Node 7 is therefore effectively unreachable from node 1. If we run the algorithm one final time, we should expect to see the graph unchanged.

Time for action – the fourth and last run

Let's perform the fourth execution to validate that the output has now reached its final stable state.

1. Execute the MapReduce job:

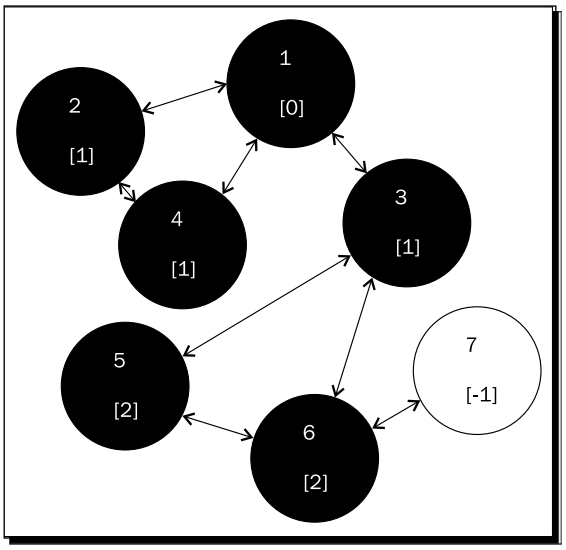
```
$ hadoop jar graph.jarGraphPathgraphout3graphout4
```

2. Examine the output file:

```
$ hadoop fs -cat /user/hadoop/graphout4/part-r-00000
12,3,40D
21,41D
31,5,61D
41,21D
53,62D
63,52D
76-1P
```

What just happened?

The output is as expected; since node 7 is not reachable by node 1 or any of its neighbors, it will remain Pending and never be processed further. Consequently, our graph is unchanged as shown in the following figure:



The one thing we did not build into our algorithm was an understanding of a terminating condition; the process is complete if a run does not create any new D or C nodes.

The mechanism we use here is manual, that is, we knew by examination that the graph representation had reached its final stable state. There are ways of doing this programmatically, however. In a later chapter, we will discuss custom job counters; we can, for example, increment a counter every time a new D or C node is created and only reexecute the job if that counter is greater than zero after the run.

Running multiple jobs

The previous algorithm is the first time we have explicitly used the output of one MapReduce job as the input to another. In most cases, the jobs are different; but, as we have seen, there is value in repeatedly applying an algorithm until the output reaches a stable state.

Final thoughts on graphs

For anyone familiar with graph algorithms, the previous process will seem very alien. This is simply a consequence of the fact that we are implementing a stateful and potentially recursive global and reentrant algorithm as a series of serial stateless MapReduce jobs. The important fact is not in the particular algorithm used; the lesson is in how we can take flat text structures and a series of MapReduce jobs, and from this implement something like graph traversal. You may have problems that at first don't appear to have any way of being implemented in the MapReduce paradigm; consider some of the techniques used here and remember that many algorithms can be modeled in MapReduce. They may look very different from the traditional approach, but the goal is the correct output and not an implementation of a known algorithm.

Using language-independent data structures

A criticism often leveled at Hadoop, and which the community has been working hard to address, is that it is very Java-centric. It may appear strange to accuse a project fully implemented in Java of being Java-centric, but the consideration is from a client's perspective.

We have shown how Hadoop Streaming allows the use of scripting languages to implement map and reduce tasks and how Pipes provides similar mechanisms for C++. However, one area that does remain Java-only is the nature of the input formats supported by Hadoop MapReduce. The most efficient format is SequenceFile, a binary splittable container that supports compression. However, SequenceFiles have only a Java API; they cannot be written or read in any other language.

We could have an external process creating data to be ingested into Hadoop for MapReduce processing, and the best way we could do this is either have it simply as an output of text type or do some preprocessing to translate the output format into SequenceFiles to be pushed onto HDFS. We also struggle here to easily represent complex data types; we either have to flatten them to a text format or write a converter across two binary formats, neither of which is an attractive option.

Candidate technologies

Fortunately, there have been several technologies released in recent years that address the question of cross-language data representations. They are **Protocol Buffers** (created by Google and hosted at `http://code.google.com/p/protobuf`), **Thrift** (originally created by Facebook and now an Apache project at `http://thrift.apache.org`), and **Avro** (created by Doug Cutting, the original creator of Hadoop). Given its heritage and tight Hadoop integration, we will use Avro to explore this topic. We won't cover Thrift or Protocol Buffers in this book, but both are solid technologies; if the topic of data serialization interests you, check out their home pages for more information.

Introducing Avro

Avro, with its home page at `http://avro.apache.org`, is a data-persistence framework with bindings for many programming languages. It creates a binary structured format that is both compressible and splittable, meaning it can be efficiently used as the input to MapReduce jobs.

Avro allows the definition of hierarchical data structures; so, for example, we can create a record that contains an array, an enumerated type, and a subrecord. We can create these files in any programming language, process them in Hadoop, and have the result read by a third language.

We'll talk about these aspects of language independence over the next sections, but this ability to express complex structured types is also very valuable. Even if we are using only Java, we could employ Avro to allow us to pass complex data structures in and out of mappers and reducers. Even things like graph nodes!

Time for action – getting and installing Avro

Let's download Avro and get it installed on our system.

1. Download the latest stable version of Avro from `http://avro.apache.org/releases.html`.
2. Download the latest version of the ParaNamer library from `http://paranamer.codehaus.org`.
3. Add the classes to the build classpath used by the Java compiler.

```
$ export CLASSPATH=avro-1.7.2.jar:${CLASSPATH}
$ export CLASSPATH=avro-mapred-1.7.2.jar:${CLASSPATH}
$ export CLASSPATH=paranamer-2.5.jar:${CLASSPATH
```

4. Add existing JAR files from the Hadoop distribution to the `build` classpath.

```
Export CLASSPATH=${HADOOP_HOME}/lib/Jackson-core-asl-
1.8.jar:${CLASSPATH}

Export CLASSPATH=${HADOOP_HOME}/lib/Jackson-mapred-asl-
1.8.jar:${CLASSPATH}

Export CLASSPATH=${HADOOP_HOME}/lib/commons-cli-
1.2.jar:${CLASSPATH}
```

5. Add the new JAR files to the Hadoop `lib` directory.

```
$cpavro-1.7.2.jar ${HADOOP_HOME}/lib

$cpavro-1.7.2.jar ${HADOOP_HOME}/lib

$cpavro-mapred-1.7.2.jar ${HADOOP_HOME}/lib
```

What just happened?

Setting up Avro is a little involved; it is a much newer project than the other Apache tools we'll be using, so it requires more than a single download of a tarball.

We download the Avro and Avro-mapred JAR files from the Apache website. There is also a dependency on ParaNamer that we download from its home page at `codehaus.org`.



The ParaNamer home page has a broken download link at the time of writing; as an alternative, try the following link:

`http://search.maven.org/remotecontent?filepath=com/thoughtworks/paranamer/paranamer/2.5/paranamer-2.5.jar`

After downloading these JAR files, we need to add them to the classpath used by our environment; primarily for the Java compiler. We add these files, but we also need to add to the `build` classpath several packages that ship with Hadoop because they are required to compile and run Avro code.

Finally, we copy the three new JAR files into the Hadoop `lib` directory on each host in the cluster to enable the classes to be available for the map and reduce tasks at runtime. We could distribute these JAR files through other mechanisms, but this is the most straightforward means.

Avro and schemas

One advantage Avro has over tools such as Thrift and Protocol Buffers, is the way it approaches the schema describing an Avro datafile. While the other tools always require the schema to be available as a distinct resource, Avro datafiles encode the schema in their header, which allows for the code to parse the files without ever seeing a separate schema file.

Avro supports but does not require code generation that produces code tailored to a specific data schema. This is an optimization that is valuable when possible but not a necessity.

We can therefore write a series of Avro examples that never actually use the datafile schema, but we'll only do that for parts of the process. In the following examples, we will define a schema that represents a cut-down version of the UFO sighting records we used previously.

Time for action – defining the schema

Let's now create this simplified UFO schema in a single Avro schema file.

Create the following as `ufo.avsc`:

```
{ "type": "record",
  "name": "UFO_Sighting_Record",
  "fields" : [
    {"name": "sighting_date", "type": "string"},
    {"name": "city", "type": "string"},
    {"name": "shape", "type": ["null", "string"]},
    {"name": "duration", "type": "float"}
]
}
```

What just happened?

As can be seen, Avro uses JSON in its schemas, which are usually saved with the `.avsc` extension. We create here a schema for a format that has four fields, as follows:

- The **Sighting_date** field of type string to hold a date of the form `yyyy-mm-dd`
- The **City** field of type string that will contain the city's name where the sighting occurred
- The **Shape** field, an optional field of type string, that represents the UFO's shape
- The **Duration** field gives a representation of the sighting duration in fractional minutes

With the schema defined, we will now create some sample data.

Time for action – creating the source Avro data with Ruby

Let's create the sample data using Ruby to demonstrate the cross-language capabilities of Avro.

1. Add the `rubygems` package:

```
$ sudo apt-get install rubygems
```

2. Install the Avro gem:

```
$ gem install avro
```

3. Create the following as `generate.rb`:

```
require 'rubygems'
require 'avro'

file = File.open('sightings.avro', 'wb')
schema = Avro::Schema.parse(
File.open("ufo.avsc", "rb").read)

writer = Avro::IO::DatumWriter.new(schema)
dw = Avro::DataFile::Writer.new(file, writer, schema)
dw<< {"sighting_date" => "2012-01-12", "city" => "Boston", "shape"
=> "diamond", "duration" => 3.5}
dw<< {"sighting_date" => "2011-06-13", "city" => "London", "shape"
=> "light", "duration" => 13}
dw<< {"sighting_date" => "1999-12-31", "city" => "New York",
"shape" => "light", "duration" => 0.25}
dw<< {"sighting_date" => "2001-08-23", "city" => "Las Vegas",
"shape" => "cylinder", "duration" => 1.2}
dw<< {"sighting_date" => "1975-11-09", "city" => "Miami",
"duration" => 5}
dw<< {"sighting_date" => "2003-02-27", "city" => "Paris", "shape"
=> "light", "duration" => 0.5}
dw<< {"sighting_date" => "2007-04-12", "city" => "Dallas", "shape"
=> "diamond", "duration" => 3.5}
dw<< {"sighting_date" => "2009-10-10", "city" => "Milan", "shape"
=> "formation", "duration" => 0}
dw<< {"sighting_date" => "2012-04-10", "city" => "Amsterdam",
"shape" => "blur", "duration" => 6}
dw<< {"sighting_date" => "2006-06-15", "city" => "Minneapolis",
"shape" => "saucer", "duration" => 0.25}
dw.close
```

4. Run the program and create the datafile:

```
$ ruby generate.rb
```

What just happened?

Before we use Ruby, we ensure the `rubygems` package is installed on our Ubuntu host. We then install the preexisting Avro gem for Ruby. This provides the libraries we need to read and write Avro files from, within the Ruby language.

The Ruby script itself simply reads the previously created schema and creates a datafile with 10 test records. We then run the program to create the data.

This is not a Ruby tutorial, so I will leave analysis of the Ruby API as an exercise for the reader; its documentation can be found at `http://rubygems.org/gems/avro`.

Time for action – consuming the Avro data with Java

Now that we have some Avro data, let's write some Java code to consume it:

1. Create the following as `InputRead.java`:

```
import java.io.File;
import java.io.IOException;

import org.apache.avro.file.DataFileReader;
import org.apache.avro.generic.GenericData;
import org.apache.avro. generic.GenericDatumReader;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.DatumReader;

public class InputRead
{
    public static void main(String[] args) throws IOException
    {
        String filename = args[0] ;

        File file=new File(filename) ;
DatumReader<GenericRecord> reader= new
GenericDatumReader<GenericRecord>();
DataFileReader<GenericRecord>dataFileReader=new
DataFileReader<GenericRecord>(file,reader);

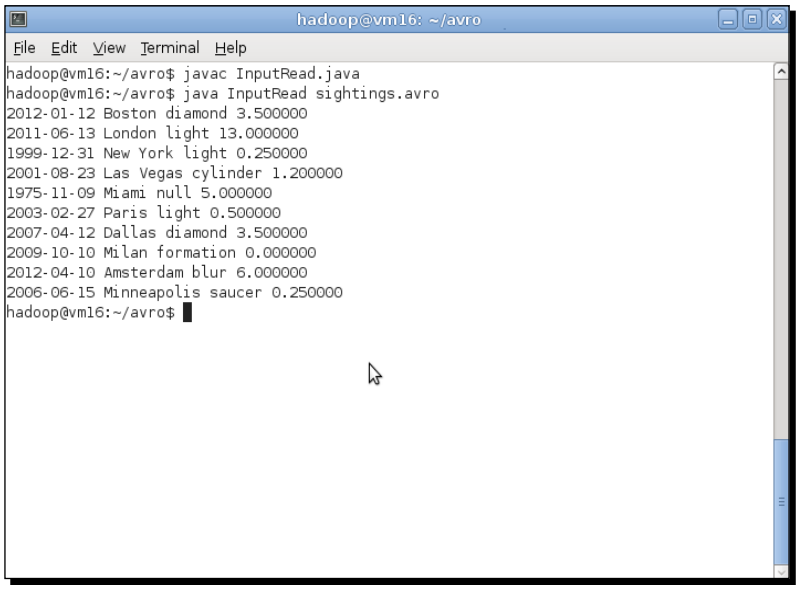
        while (dataFileReader.hasNext())
        {
GenericRecord result=dataFileReader.next();
            String output = String.format("%s %s %s %f",
result.get("sighting_date"), result.get("city"),
result.get("shape"), result.get("duration")) ;
```

```
System.out.println(output) ;
        }
    }
}
```

2. Compile and run the program:

```
$ javacInputRead.java
$ java InputReadsightings.avro
```

The output will be as shown in the following screenshot:



What just happened?

We created the Java class `InputRead`, which takes the filename passed as a command-line argument and parses this as an Avro datafile. When Avro reads from a datafile, each individual element is called a **datum** and each datum will follow the structure defined in the schema.

In this case, we don't use an explicit schema; instead, we read each datum into the `GenericRecord` class, and from this extract each field by explicitly retrieving it by name.

The `GenericRecord` class is a very flexible class in Avro; it can be used to wrap any record structure, such as our UFO-sighting type. Avro also supports primitive types such as integers, floats, and booleans as well as other structured types such as arrays and enums. In these examples, we'll use records as the most common structure, but this is only a convenience.

Using Avro within MapReduce

Avro's support for MapReduce revolves around several Avro-specific variants of other familiar classes, whereas we'd normally expect a new datafile format to be supported in Hadoop through new `InputFormat` and `OutputFormat` classes, we'll use **AvroJob**, **AvroMapper**, and **AvroReducer** instead of the non-Avro versions. AvroJob expects Avro datafiles as its input and output, so instead of specifying input and output format types, we configure it with details of the input and output Avro schemas.

The main difference for our mapper and reducer implementations are the types used. Avro, by default, has a single input and output, whereas we're used to our `Mapper` and `Reducer` classes having a key/value input and a key/value output. Avro also introduces the `Pair` class, which is often used to emit intermediate key/value data.

Avro does also support **AvroKey** and **AvroValue**, which can wrap other types, but we'll not use those in the following examples.

Time for action – generating shape summaries in MapReduce

In this section we will write a mapper that takes as input the UFO sighting record we defined earlier. It will output the shape and a count of `1`, and the reducer will take this shape and count records and produce a new structured Avro datafile type containing the final counts for each UFO shape. Perform the following steps:

1. Copy the `sightings.avro` file to HDFS.

```
$ hadoopfs -mkdiravroin
$ hadoopfs -put sightings.avroavroin/sightings.avro
```

2. Create the following as `AvroMR.java`:

```
import java.io.IOException;
import org.apache.avro.Schema;
import org.apache.avro.generic.*;
import org.apache.avro.Schema.Type;
import org.apache.avro.mapred.*;
import org.apache.avro.reflect.ReflectData;
import org.apache.avro.util.Utf8;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.io.* ;
import org.apache.hadoop.util.*;

// Output record definition
```

```
class UFORecord
{
UFORecord()
    {
    }

    public String shape ;
    public long count ;
}

public class AvroMR extends Configured  implements Tool
{
// Create schema for map output
    public static final Schema PAIR_SCHEMA =
Pair.getPairSchema(Schema.create(Schema.Type.STRING),
Schema.create(Schema.Type.LONG));
// Create schema for reduce output
    public final static Schema OUTPUT_SCHEMA =
ReflectData.get().getSchema(UFORecord.class);

    @Override
    public int run(String[] args) throws Exception
    {
JobConfconf = new JobConf(getConf(), getClass());
conf.setJobName("UFO count");

        String[] otherArgs = new GenericOptionsParser(conf, args).
getRemainingArgs();
        if (otherArgs.length != 2)
        {
System.err.println("Usage: avro UFO counter <in><out>");
System.exit(2);

        }

FileInputFormat.addInputPath(conf, new Path(otherArgs[0]));
        Path outputPath = new Path(otherArgs[1]);
FileOutputFormat.setOutputPath(conf, outputPath);
outputPath.getFileSystem(conf).delete(outputPath);
        Schema input_schema =
Schema.parse(getClass().getResourceAsStream("ufo.avsc"));
AvroJob.setInputSchema(conf, input_schema);
AvroJob.setMapOutputSchema(conf,
Pair.getPairSchema(Schema.create(Schema.Type.STRING),
```

```
Schema.create(Schema.Type.LONG)));

AvroJob.setOutputSchema(conf, OUTPUT_SCHEMA);
AvroJob.setMapperClass(conf, AvroRecordMapper.class);
AvroJob.setReducerClass(conf, AvroRecordReducer.class);
conf.setInputFormat(AvroInputFormat.class) ;
JobClient.runJob(conf);

        return 0 ;
    }

    public static class AvroRecordMapper extends
AvroMapper<GenericRecord, Pair<Utf8, Long>>
    {
        @Override
        public void map(GenericRecord in, AvroCollector<Pair<Utf8,
Long>> collector, Reporter reporter) throws IOException
        {
            Pair<Utf8,Long> p = new Pair<Utf8,Long>(PAIR_SCHEMA) ;
Utf8 shape = (Utf8)in.get("shape") ;
            if (shape != null)
            {
p.set(shape, 1L) ;
collector.collect(p);
            }
        }
    }

    public static class AvroRecordReducer extends
AvroReducer<Utf8,
Long, GenericRecord>
    {
        public void reduce(Utf8 key, Iterable<Long> values,
AvroCollector<GenericRecord> collector,
            Reporter reporter) throws IOException
        {
            long sum = 0;
            for (Long val : values)
            {
                sum += val;
            }

GenericRecord value = new
GenericData.Record(OUTPUT_SCHEMA);
```

```
value.put("shape", key);
value.put("count", sum);

collector.collect(value);
        }
    }

    public static void main(String[] args) throws Exception
    {
int res = ToolRunner.run(new Configuration(), new AvroMR(),
args);
System.exit(res);
    }
}
```

3. Compile and run the job:

```
$ javacAvroMR.java
$ jar -cvfavroufo.jar *.class ufo.avsc
   $ hadoop jar ~/classes/avroufo.jarAvroMRavroinavroout
```

4. Examine the output directory:

```
$ hadoopfs -lsavroout
Found 3 items
-rw-r--r--   1 … /user/hadoop/avroout/_SUCCESS
drwxr-xr-x   - hadoopsupergroup          0 … /user/hadoop/
avroout/_logs
-rw-r--r--   1 …  /user/hadoop/avroout/part-00000.avro
```

5. Copy the output file to the local filesystem:

```
$ hadoopfs -get /user/hadoop/avroout/part-00000.avroresult.avro
```

What just happened?

We created the `Job` class and examined its various components. The actual logic within the `Mapper` and `Reducer` classes is relatively straightforward: the `Mapper` class just extracts the shape column and emits it with a count of `1`; the reducer then counts the total number of entries for each shape. The interesting aspects are around the defined input and output types to the `Mapper` and `Reducer` classes and how the job is configured.

The `Mapper` class has an input type of `GenericRecord` and an output type of `Pair`. The `Reducer` class has a corresponding input type of `Pair` and output type of `GenericRecord`.

The `GenericRecord` class passed to the `Mapper` class wraps a datum that is the UFO sighting record represented in the input file. This is how the `Mapper` class is able to retrieve the **Shape** field by name.

Recall that `GenericRecords` may or may not be explicitly created with a schema, and in either case the structure can be determined from the datafile. For the `GenericRecord` output by the `Reducer` class, we do pass a schema but use a new mechanism for its creation.

Within the previously mentioned code, we created the additional `UFORecord` class and used Avro reflection to generate its schema dynamically at runtime. We were then able to use this schema to create a `GenericRecord` class specialized to wrap that particular record type.

Between the `Mapper` and `Reducer` classes we use the Avro `Pair` type to hold a key and value pair. This allows us to express the same logic for the `Mapper` and `Reducer` classes that we used in the original WordCount example back in *Chapter 2*, *Getting Hadoop Up and Running*; the Mapper class emits singleton counts for each value and the reducer sums these into an overall total for each shape.

In addition to the `Mapper` and `Reducer` classes' input and output, there is some configuration unique to a job processing Avro data:

```
Schema input_schema = Schema.parse(getClass().
getResourceAsStream("ufo.avsc")) ;
AvroJob.setInputSchema(conf, input_schema);
AvroJob.setMapOutputSchema(conf,           Pair.getPairSchema(Schema.
create(Schema.Type.STRING), Schema.create(Schema.Type.LONG)));

AvroJob.setOutputSchema(conf, OUTPUT_SCHEMA);
AvroJob.setMapperClass(conf, AvroRecordMapper.class);
AvroJob.setReducerClass(conf, AvroRecordReducer.class);
```

These configuration elements demonstrate the criticality of schema definition to Avro; though we can do without it, we must set the expected input and output schema types. Avro will validate the input and output against the specified schemas, so there is a degree of data type safety. For the other elements, such as setting up the `Mapper` and `Reducer` classes, we simply set those on AvroJob instead of the more generic classes, and once done, the MapReduce framework will perform appropriately.

This example is also the first time we've explicitly implemented the `Tool` interface. When running the Hadoop command-line program, there are a series of arguments (such as `-D`) that are common across all the multiple subcommands. If a job class implements the `Tool` interface as mentioned in the previous section, it automatically gets access to any of these standard options passed on the command line. It's a useful mechanism that prevents lots of code duplication.

Time for action – examining the output data with Ruby

Now that we have the output data from the job, let's examine it again using Ruby.

1. Create the following as `read.rb`:

```
require 'rubygems'
require 'avro'

file = File.open('res.avro', 'rb')
reader = Avro::IO::DatumReader.new()
dr = Avro::DataFile::Reader.new(file, reader)

dr.each {|record|
print record["shape"]," ",record["count"],"\n"
}
dr.close
```

2. Examine the created result file.

```
$ ruby read.rb
blur 1
cylinder 1
diamond 2
formation 1
light 3
saucer 1
```

What just happened?

As before, we'll not analyze the Ruby Avro API. The example created a Ruby script that opens an Avro datafile, iterates through each datum, and displays it based on explicitly named fields. Note that the script does not have access to the schema for the datafile; the information in the header provides enough data to allow each field to be retrieved.

Time for action – examining the output data with Java

To show that the data is accessible from multiple languages, let's also display the job output using Java.

1. Create the following as `OutputRead.java`:

```
import java.io.File;
import java.io.IOException;
```

```
import org.apache.avro.file.DataFileReader;
import org.apache.avro.generic.GenericData;
import org.apache.avro. generic.GenericDatumReader;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.DatumReader;

public class OutputRead
{
    public static void main(String[] args) throws IOException
    {
        String filename = args[0] ;

        File file=new File(filename) ;
DatumReader<GenericRecord> reader= new
GenericDatumReader<GenericRecord>();
DataFileReader<GenericRecord>dataFileReader=new
DataFileReader<GenericRecord>(file,reader);

        while (dataFileReader.hasNext())
        {
GenericRecord result=dataFileReader.next();
            String output = String.format("%s %d",
result.get("shape"), result.get("count")) ;
System.out.println(output) ;
        }
    }
}
```

2. Compile and run the program:

```
$ javacOutputResult.java
$ java OutputResultresult.avro
blur 1
cylinder 1
diamond 2
formation 1
light 3
saucer 1
```

What just happened?

We added this example to show the Avro data being read by more than one language. The code is very similar to the earlier `InputRead` class; the only difference is that the named fields are used to display each datum as it is read from the datafile.

Have a go hero – graphs in Avro

As previously mentioned, we worked hard to reduce representation-related complexity in our `GraphPath` class. But with mappings to and from flat lines of text and objects, there was an overhead in managing these transformations.

With its support for nested complex types, Avro can natively support a representation of a node that is much closer to the runtime object. Modify the `GraphPath` class job to read and write the graph representation to an Avro datafile comprising of datums for each node. The following example schema may be a good starting point, but feel free to enhance it:

```
{ "type": "record",
  "name": "Graph_representation",
  "fields" : [
{"name": "node_id", "type": "int"},
    {"name": "neighbors", "type": "array", "items:"int" },
    {"name": "distance", "type": "int"},
  {"name": "status", "type": "enum",
"symbols": ["PENDING", "CURRENT", "DONE"
},]
]
}
```

Going forward with Avro

There are many features of Avro we did not cover in this case study. We focused only on its value as an at-rest data representation. It can also be used within a **remote procedure call** (**RPC**) framework and can optionally be used as the default RPC format in Hadoop 2.0. We didn't use Avro's code generation facilities that produce a much more domain-focused API. Nor did we cover issues such as Avro's ability to support schema evolution that, for example, allows new fields to be added to recent records without invalidating old datums or breaking existing clients. It's a technology you are very likely to see more of in the future.

Summary

This chapter has used three case studies to highlight some more advanced aspects of Hadoop and its broader ecosystem. In particular, we covered the nature of join-type problems and where they are seen, how reduce-side joins can be implemented with relative ease but with an efficiency penalty, and how to use optimizations to avoid full joins in the map-side by pushing data into the Distributed Cache.

We then learned how full map-side joins can be implemented, but require significant input data processing; how other tools such as Hive and Pig should be investigated if joins are a frequently encountered use case; and how to think about complex types like graphs and how they can be represented in a way that can be used in MapReduce.

We also saw techniques for breaking graph algorithms into multistage MapReduce jobs, the importance of language-independent data types, how Avro can be used for both language independence as well as complex Java-consumed types, and the Avro extensions to the MapReduce APIs that allow structured types to be used as the input and output to MapReduce jobs.

This now concludes our coverage of the programmatic aspects of the Hadoop MapReduce framework. We will now move on in the next two chapters to explore how to manage and scale a Hadoop environment.

6
When Things Break

One of the main promises of Hadoop is resilience to failure and an ability to survive failures when they do happen. Tolerance to failure will be the focus of this chapter.

In particular, we will cover the following topics:

- How Hadoop handles failures of DataNodes and TaskTrackers
- How Hadoop handles failures of the NameNode and JobTracker
- The impact of hardware failure on Hadoop
- How to deal with task failures caused by software bugs
- How dirty data can cause tasks to fail and what to do about it

Along the way, we will deepen our understanding of how the various components of Hadoop fit together and identify some areas of best practice.

Failure

With many technologies, the steps to be taken when things go wrong are rarely covered in much of the documentation and are often treated as topics only of interest to the experts. With Hadoop, it is much more front and center; much of the architecture and design of Hadoop is predicated on executing in an environment where failures are both frequent and expected.

Embrace failure

In recent years, a different mindset than the traditional one has been described by the term **embrace failure**. Instead of hoping that failure does not happen, accept the fact that it will and know how your systems and processes will respond when it does.

Or at least don't fear it

That's possibly a stretch, so instead, our goal in this chapter is to make you feel more comfortable about failures in the system. We'll be killing the processes of a running cluster, intentionally causing the software to fail, pushing bad data into our jobs, and generally causing as much disruption as we can.

Don't try this at home

Often when trying to break a system, a test instance is abused, leaving the operational system protected from the disruption. We will not advocate doing the things given in this chapter to an operational Hadoop cluster, but the fact is that apart from one or two very specific cases, you could. The goal is to understand the impact of the various types of failures so that when they do happen on the business-critical system, you will know whether it is a problem or not. Fortunately, the majority of cases are handled for you by Hadoop.

Types of failure

We will generally categorize failures into the following five types:

- Failure of a node, that is, DataNode or TaskTracker process
- Failure of a cluster's masters, that is, NameNode or JobTracker process
- Failure of hardware, that is, host crash, hard drive failure, and so on
- Failure of individual tasks within a MapReduce job due to software errors
- Failure of individual tasks within a MapReduce job due to data problems

We will explore each of these in turn in the following sections.

Hadoop node failure

The first class of failure that we will explore is the unexpected termination of the individual DataNode and TaskTracker processes. Given Hadoop's claims of managing system availability through survival of failures on its commodity hardware, we can expect this area to be very solid. Indeed, as clusters grow to hundreds or thousands of hosts, failures of individual nodes are likely to become quite commonplace.

Before we start killing things, let's introduce a new tool and set up the cluster properly.

The dfsadmin command

As an alternative tool to constantly viewing the HDFS web UI to determine the cluster status, we will use the `dfsadmin` command-line tool:

```
$ Hadoop dfsadmin
```

This will give a list of the various options the command can take; for our purposes we'll be using the `-report` option. This gives an overview of the overall cluster state, including configured capacity, nodes, and files as well as specific details about each configured node.

Cluster setup, test files, and block sizes

We will need a fully distributed cluster for the following activities; refer to the setup instructions given earlier in the book. The screenshots and examples that follow use a cluster of one host for the JobTracker and NameNode and four slave nodes for running the DataNode and TaskTracker processes.



Remember that you don't need physical hardware for each node, we use virtual machines for our cluster.

In normal usage, 64 MB is the usual configured block size for a Hadoop cluster. For our testing purposes, that is terribly inconvenient as we'll need pretty large files to get meaningful block counts across our multinode cluster.

What we can do is reduce the configured block size; in this case, we will use 4 MB. Make the following modifications to the `hdfs-site.xml` file within the Hadoop `conf` directory:

```
<property>
<name>dfs.block.size</name>
<value>4194304</value>
;</property>
<property>
<name>dfs.namenode.logging.level</name>
<value>all</value>
</property>
```

The first property makes the required changes to the block size and the second one increases the NameNode logging level to make some of the block operations more visible.



Both these settings are appropriate for this test setup but would rarely be seen on a production cluster. Though the higher NameNode logging may be required if a particularly difficult problem is being investigated, it is highly unlikely you would ever want a block size as small as 4 MB. Though the smaller block size will work fine, it will impact Hadoop's efficiency.

We also need a reasonably-sized test file that will comprise of multiple 4 MB blocks. We won't actually be using the content of the file, so the type of file is irrelevant. But you should copy the largest file you can onto HDFS for the following sections. We used a CD ISO image:

```
$ Hadoop fs –put cd.iso file1.data
```

Fault tolerance and Elastic MapReduce

The examples in this book are for a local Hadoop cluster because this allows some of the failure mode details to be more explicit. EMR provides exactly the same failure tolerance as the local cluster, so the failure scenarios described here apply equally to a local Hadoop cluster and the one hosted by EMR.

Time for action – killing a DataNode process

Firstly, we'll kill a DataNode. Recall that the DataNode process runs on each host in the HDFS cluster and is responsible for the management of blocks within the HDFS filesystem. Because Hadoop, by default, uses a replication factor of 3 for blocks, we should expect a single DataNode failure to have no direct impact on availability, rather it will result in some blocks temporarily falling below the replication threshold. Execute the following steps to kill a DataNode process:

1. Firstly, check on the original status of the cluster and check whether everything is healthy. We'll use the `dfsadmin` command for this:

```
$ Hadoop dfsadmin -report
Configured Capacity: 81376493568 (75.79 GB)
Present Capacity: 61117323920 (56.92 GB)
DFS Remaining: 59576766464 (55.49 GB)
DFS Used: 1540557456 (1.43 GB)
DFS Used%: 2.52%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0

-------------------------------------------------
```

```
Datanodes available: 4 (4 total, 0 dead)

Name: 10.0.0.102:50010
Decommission Status : Normal
Configured Capacity: 20344123392 (18.95 GB)
DFS Used: 403606906 (384.91 MB)
Non DFS Used: 5063119494 (4.72 GB)
DFS Remaining: 14877396992(13.86 GB)
DFS Used%: 1.98%
DFS Remaining%: 73.13%
Last contact: Sun Dec 04 15:16:27 PST 2011
…
```

Now log onto one of the nodes and use the `jps` command to determine the process ID of the DataNode process:

```
$ jps
2085 TaskTracker
2109 Jps
1928 DataNode
```

2. Use the **process ID** (**PID**) of the DataNode process and kill it:

```
$ kill -9  1928
```

3. Check that the DataNode process is no longer running on the host:

```
$ jps
2085 TaskTracker
```

4. Check the status of the cluster again by using the `dfsadmin` command:

```
$ Hadoop dfsadmin -report
Configured Capacity: 81376493568 (75.79 GB)
Present Capacity: 61117323920 (56.92 GB)
DFS Remaining: 59576766464 (55.49 GB)
DFS Used: 1540557456 (1.43 GB)
DFS Used%: 2.52%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
```

```
-------------------------------------------------
Datanodes available: 4 (4 total, 0 dead)
…
```

5. The key lines to watch are the lines reporting on blocks, live nodes, and the last contact time for each node. Once the last contact time for the dead node is around 10 minutes, use the command more frequently until the block and live node values change:

```
$ Hadoop dfsadmin -report
Configured Capacity: 61032370176 (56.84 GB)
Present Capacity: 46030327050 (42.87 GB)
DFS Remaining: 44520288256 (41.46 GB)
DFS Used: 1510038794 (1.41 GB)
DFS Used%: 3.28%
Under replicated blocks: 12
Blocks with corrupt replicas: 0
Missing blocks: 0

-------------------------------------------------
Datanodes available: 3 (4 total, 1 dead)
…
```

6. Repeat the process until the count of under-replicated blocks is once again `0`:

```
$ Hadoop dfsadmin -report
…
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0

-------------------------------------------------
Datanodes available: 3 (4 total, 1 dead)
…
```

What just happened?

The high-level story is pretty straightforward; Hadoop recognized the loss of a node and worked around the problem. However, quite a lot is going on to make that happen.

When we killed the DataNode process, the process on that host was no longer available to serve or receive data blocks as part of the read/write operations. However, we were not actually accessing the filesystem at the time, so how did the NameNode process know this particular DataNode was dead?

NameNode and DataNode communication

The answer lies in the constant communication between the NameNode and DataNode processes that we have alluded to once or twice but never really explained. This occurs through a constant series of heartbeat messages from the DataNode reporting on its current state and the blocks it holds. In return, the NameNode gives instructions to the DataNode, such as notification of the creation of a new file or an instruction to retrieve a block from another node.

It all begins when the NameNode process starts up and begins receiving status messages from the DataNode. Recall that each DataNode knows the location of its NameNode and will continuously send status reports. These messages list the blocks held by each DataNode and from this, the NameNode is able to construct a complete mapping that allows it to relate files and directories to the blocks from where they are comprised and the nodes on which they are stored.

The NameNode process monitors the last time it received a heartbeat from each DataNode and after a threshold is reached, it assumes the DataNode is no longer functional and marks it as dead.



The exact threshold after which a DataNode is assumed to be dead is not configurable as a single HDFS property. Instead, it is calculated from several other properties such as defining the heartbeat interval. As we'll see later, things are a little easier in the MapReduce world as the timeout for TaskTrackers is controlled by a single configuration property.

Once a DataNode is marked as dead, the NameNode process determines the blocks which were held on that node and have now fallen below their replication target. In the default case, each block held on the killed node would have been one of the three replicas, so each block for which the node held a replica will now have only two replicas across the cluster.

In the preceding example, we captured the state when 12 blocks were still under-replicated, that is they did not have enough replicas across the cluster to meet the replication target. When the NameNode process determines the under-replicated blocks, it assigns other DataNodes to copy these blocks from the hosts where the existing replicas reside. In this case we only had to re-replicate a very small number of blocks; in a live cluster, the failure of a node can result in a period of high network traffic as the affected blocks are brought up to their replication factor.

Note that if a failed node returns to the cluster, we have the situation of blocks having more than the required number of replicas; in such a case the NameNode process will send instructions to remove the surplus replicas. The specific replica to be deleted is chosen randomly, so the result will be that the returned node will end up retaining some of its blocks and deleting the others.

Have a go hero – NameNode log delving

We configured the NameNode process to log all its activities. Have a look through these very verbose logs and attempt to identify the replication requests being sent.

The final output shows the status after the under-replicated blocks have been copied to the live nodes. The cluster is down to only three live nodes but there are no under-replicated blocks.



A quick way to restart the dead nodes across all hosts is to use the `start-all.sh` script. It will attempt to start everything but is smart enough to detect the running services, which means you get the dead nodes restarted without the risk of duplicates.

Time for action – the replication factor in action

Let's repeat the preceding process, but this time, kill two DataNodes out of our cluster of four. We will give an abbreviated walk-through of the activity as it is very similar to the previous *Time for action* section:

1. Restart the dead DataNode and monitor the cluster until all nodes are marked as live.
2. Pick two DataNodes, use the process ID, and kill the DataNode processes.
3. As done previously, wait for around 10 minutes then actively monitor the cluster state via `dfsadmin`, paying particular attention to the reported number of under-replicated blocks.
4. Wait until the cluster has stabilized with an output similar to the following:

```
Configured Capacity: 61032370176 (56.84 GB)
Present Capacity: 45842373555 (42.69 GB)
DFS Remaining: 44294680576 (41.25 GB)
DFS Used: 1547692979 (1.44 GB)
DFS Used%: 3.38%
Under replicated blocks: 125
Blocks with corrupt replicas: 0
```

```
Missing blocks: 0

-------------------------------------------------
Datanodes available: 2 (4 total, 2 dead)
…
```

What just happened?

This is the same process as before; the difference is that due to two DataNode failures there were significantly more blocks that fell below the replication factor, many going down to a single remaining replica. Consequently, you should see more activity in the reported number of under-replicated blocks as it first increase because nodes fail and then drop as re-replication occurs. These events can also be seen in the NameNode logs.

Note that though Hadoop can use re-replication to bring those blocks with only a single remaining replica up to two replicas, this still leaves the blocks in an under-replicated state. With only two live nodes in the cluster, it is now impossible for any block to meet the default replication target of three.

We have been truncating the `dfsadmin` output for space reasons; in particular, we have been omitting the reported information for each node. However, let's take a look at the first node in our cluster through the previous stages. Before we started killing any DataNode, it reported the following:

```
Name: 10.0.0.101:50010
Decommission Status : Normal
Configured Capacity: 20344123392 (18.95 GB)
DFS Used: 399379827 (380.88 MB)
Non DFS Used: 5064258189 (4.72 GB)
DFS Remaining: 14880485376(13.86 GB)
DFS Used%: 1.96%
DFS Remaining%: 73.14%
Last contact: Sun Dec 04 15:16:27 PST 2011
```

After a single DataNode was killed and all blocks had been re-replicated as necessary, it reported the following:

```
Name: 10.0.0.101:50010
Decommission Status : Normal
Configured Capacity: 20344123392 (18.95 GB)
DFS Used: 515236022 (491.37 MB)
Non DFS Used: 5016289098 (4.67 GB)
DFS Remaining: 14812598272(13.8 GB)
```

```
DFS Used%: 2.53%
DFS Remaining%: 72.81%
Last contact: Sun Dec 04 15:31:22 PST 2011
```

The thing to note is the increase in the local DFS storage on the node. This shouldn't be a surprise. With a dead node, the others in the cluster need to add some additional block replicas and that will translate to a higher storage utilization on each.

Finally, the following is the node's report after two other DataNodes were killed:

```
Name: 10.0.0.101:50010
Decommission Status : Normal
Configured Capacity: 20344123392 (18.95 GB)
DFS Used: 514289664 (490.46 MB)
Non DFS Used: 5063868416 (4.72 GB)
DFS Remaining: 14765965312(13.75 GB)
DFS Used%: 2.53%
DFS Remaining%: 72.58%
Last contact: Sun Dec 04 15:43:47 PST 2011
```

With two dead nodes it may seem as if the remaining live nodes should consume even more local storage space, but this isn't the case and it's yet again a natural consequence of the replication factor.

If we have four nodes and a replication factor of 3, each block will have a replica on three of the live nodes in the cluster. If a node dies, the blocks living on the other nodes are unaffected, but any blocks with a replica on the dead node will need a new replica created. However, with only three live nodes, each node will hold a replica of every block. If a second node fails, the situation will result into under-replicated blocks and Hadoop does not have anywhere to put the additional replicas. Since both remaining nodes already hold a replica of each block, their storage utilization does not increase.

Time for action – intentionally causing missing blocks

The next step should be obvious; let's kill three DataNodes in quick succession.



This is the first of the activities we mentioned that you really should not do on a production cluster. Although there will be no data loss if the steps are followed properly, there is a period when the existing data is unavailable.

The following are the steps to kill three DataNodes in quick succession:

1. Restart all the nodes by using the following command:

```
$ start-all.sh
```

2. Wait until Hadoop `dfsadmin -report` shows four live nodes.
3. Put a new copy of the test file onto HDFS:

```
$ Hadoop fs -put file1.data file1.new
```

4. Log onto three of the cluster hosts and kill the DataNode process on each.
5. Wait for the usual 10 minutes then start monitoring the cluster via `dfsadmin` until you get output similar to the following that reports the missing blocks:

```
…
Under replicated blocks: 123
Blocks with corrupt replicas: 0
Missing blocks: 33

-------------------------------------------------
Datanodes available: 1 (4 total, 3 dead)
…
```

6. Try and retrieve the test file from HDFS:

```
$ hadoop fs -get file1.new  file1.new
11/12/04 16:18:05 INFO hdfs.DFSClient: No node available for
block: blk_1691554429626293399_1003 file=/user/hadoop/file1.new
11/12/04 16:18:05 INFO hdfs.DFSClient: Could not obtain block
blk_1691554429626293399_1003 from any node:  java.io.IOException:
No live nodes contain current block
…
get: Could not obtain block: blk_1691554429626293399_1003 file=/
user/hadoop/file1.new
```

7. Restart the dead nodes using the `start-all.sh` script:

```
$ start-all.sh
```

8. Repeatedly monitor the status of the blocks:

```
$ Hadoop dfsadmin -report | grep -i blocks
Under replicated blockss: 69
Blocks with corrupt replicas: 0
Missing blocks: 35
```

```
$ Hadoop dfsadmin -report | grep -i blocks
Under replicated blockss: 0
Blocks with corrupt replicas: 0
Missing blocks: 30
```

9. Wait until there are no reported missing blocks then copy the test file onto the local filesystem:

```
$ Hadoop fs -get file1.new file1.new
```

10. Perform an MD5 check on this and the original file:

```
$ md5sum file1.*
f1f30b26b40f8302150bc2a494c1961d  file1.data
f1f30b26b40f8302150bc2a494c1961d  file1.new
```

What just happened?

After restarting the killed nodes, we copied the test file onto HDFS again. This isn't strictly necessary as we could have used the existing file but due to the shuffling of the replicas, a clean copy gives the most representative results.

We then killed three DataNodes as before and waited for HDFS to respond. Unlike the previous examples, killing these many nodes meant it was certain that some blocks would have all of their replicas on the killed nodes. As we can see, this is exactly the result; the remaining single node cluster shows over a hundred blocks that are under-replicated (obviously only one replica remains) but there are also 33 missing blocks.

Talking of blocks is a little abstract, so we then try to retrieve our test file which, as we know, effectively has 33 holes in it. The attempt to access the file fails as Hadoop could not find the missing blocks required to deliver the file.

We then restarted all the nodes and tried to retrieve the file again. This time it was successful, but we took an added precaution of performing an MD5 cryptographic check on the file to confirm that it was bitwise identical to the original one — which it is.

This is an important point: though node failure may result in data becoming unavailable, there may not be a permanent data loss if the node recovers.

When data may be lost

Do not assume from this example that it's impossible to lose data in a Hadoop cluster. For general use it is very hard, but disaster often has a habit of striking in just the wrong way.

As seen in the previous example, a parallel failure of a number of nodes equal to or greater than the replication factor has a chance of resulting in missing blocks. In our example of three dead nodes in a cluster of four, the chances were high; in a cluster of 1000, it would be much lower but still non-zero. As the cluster size increases, so does the failure rate and having three node failures in a narrow window of time becomes less and less unlikely. Conversely, the impact also decreases but rapid multiple failures will always carry a risk of data loss.

Another more insidious problem is recurring or partial failures, for example, when power issues across the cluster cause nodes to crash and restart. It is possible for Hadoop to end up chasing replication targets, constantly asking the recovering hosts to replicate under-replicated blocks, and also seeing them fail mid-way through the task. Such a sequence of events can also raise the potential of data loss.

Finally, never forget the human factor. Having a replication factor equal to the size of the cluster—ensuring every block is on every node—won't help you when a user accidentally deletes a file or directory.

The summary is that data loss through system failure is pretty unlikely but is possible through almost inevitable human action. Replication is not a full alternative to backups; ensure that you understand the importance of the data you process and the impact of the types of loss discussed here.



The most catastrophic losses in a Hadoop cluster are actually caused by NameNode failure and filesystem corruption; we'll discuss this topic in some detail in the next chapter.

Block corruption

The reports from each DataNode also included a count of the corrupt blocks, which we have not referred to. When a block is first stored, there is also a hidden file written to the same HDFS directory containing cryptographic checksums for the block. By default, there is a checksum for each 512-byte chunk within the block.

Whenever any client reads a block, it will also retrieve the list of checksums and compare these to the checksums it generates on the block data it has read. If there is a checksum mismatch, the block on that particular DataNode will be marked as corrupt and the client will retrieve a different replica. On learning of the corrupt block, the NameNode will schedule a new replica to be made from one of the existing uncorrupted replicas.

If the scenario seems unlikely, consider that faulty memory, disk drive, storage controller, or numerous issues on an individual host could cause some corruption to a block as it is initially being written while being stored or when being read. These are rare events and the chances of the same corruption occurring on all DataNodes holding replicas of the same block become exceptionally remote. However, remember as previously mentioned that replication is not a full alternative to backup and if you need 100 percent data availability, you likely need to think about off-cluster backup.

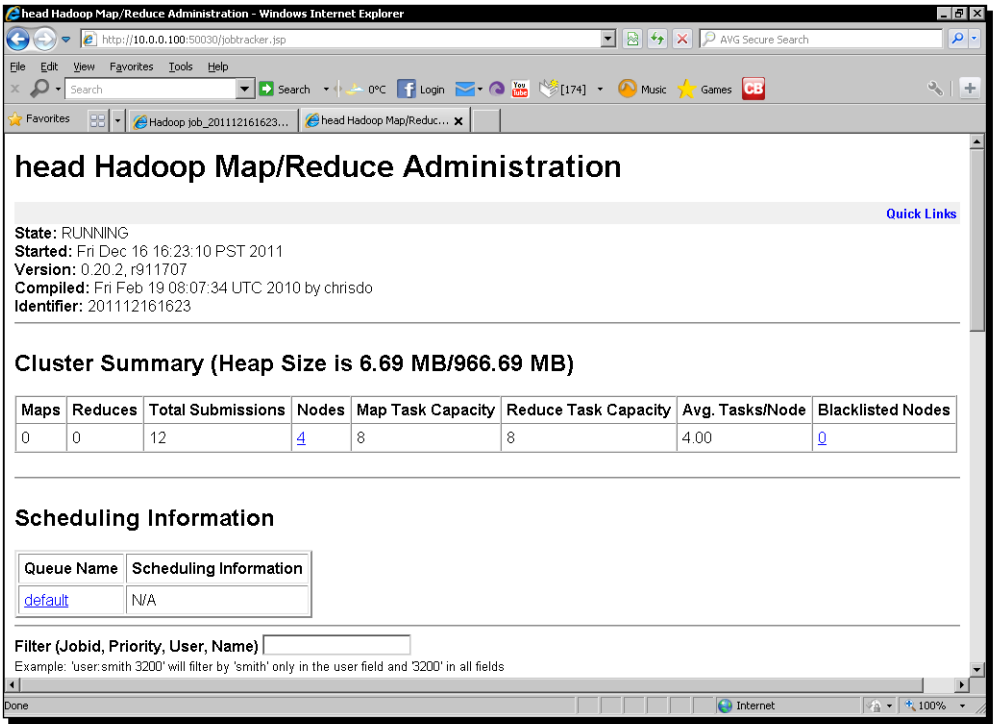
Time for action – killing a TaskTracker process

We've abused HDFS and its DataNode enough; now let's see what damage we can do to MapReduce by killing some TaskTracker processes.

Though there is an `mradmin` command, it does not give the sort of status reports we are used to with HDFS. So we'll use the MapReduce web UI (located by default on port 50070 on the JobTracker host) to monitor the MapReduce cluster health.

Perform the following steps:

1. Ensure everything is running via the `start-all.sh` script then point your browser at the MapReduce web UI. The page should look like the following screenshot:



2. Start a long-running MapReduce job; the example pi estimator with large values is great for this:

```
$ Hadoop jar Hadoop/Hadoop-examples-1.0.4.jar pi 2500 2500
```

3. Now log onto a cluster node and use `jps` to identify the TaskTracker process:

```
$ jps
21822 TaskTracker
3918 Jps
3891 DataNode
```

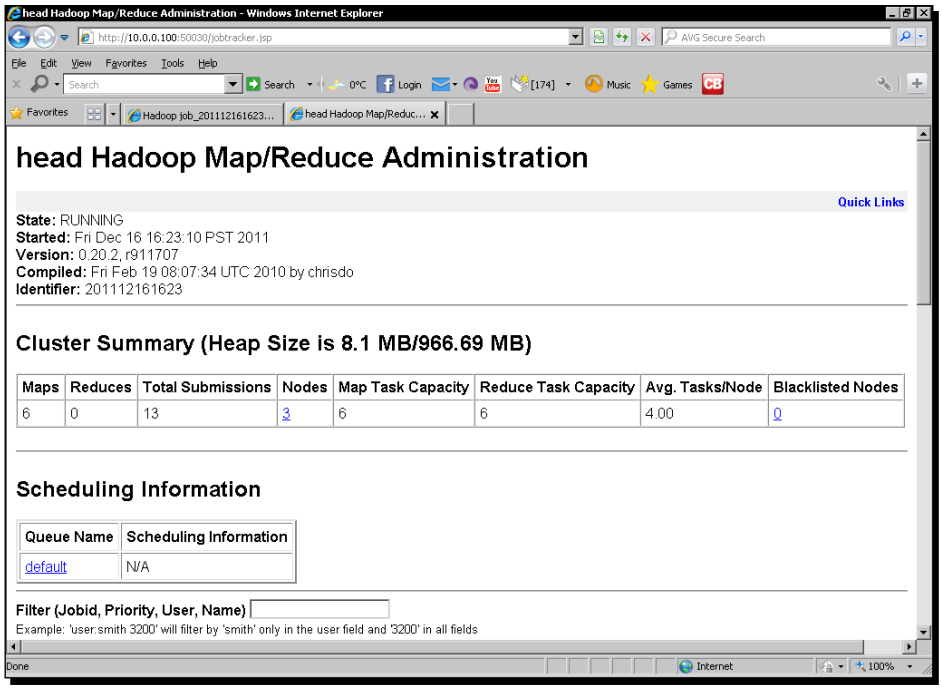
4. Kill the TaskTracker process:

```
$ kill -9 21822
```

5. Verify that the TaskTracker is no longer running:

```
$jps
3918 Jps
3891 DataNode
```

6. Go back to the MapReduce web UI and after 10 minutes you should see that the number of nodes and available map/reduce slots change as shown in the following screenshot:

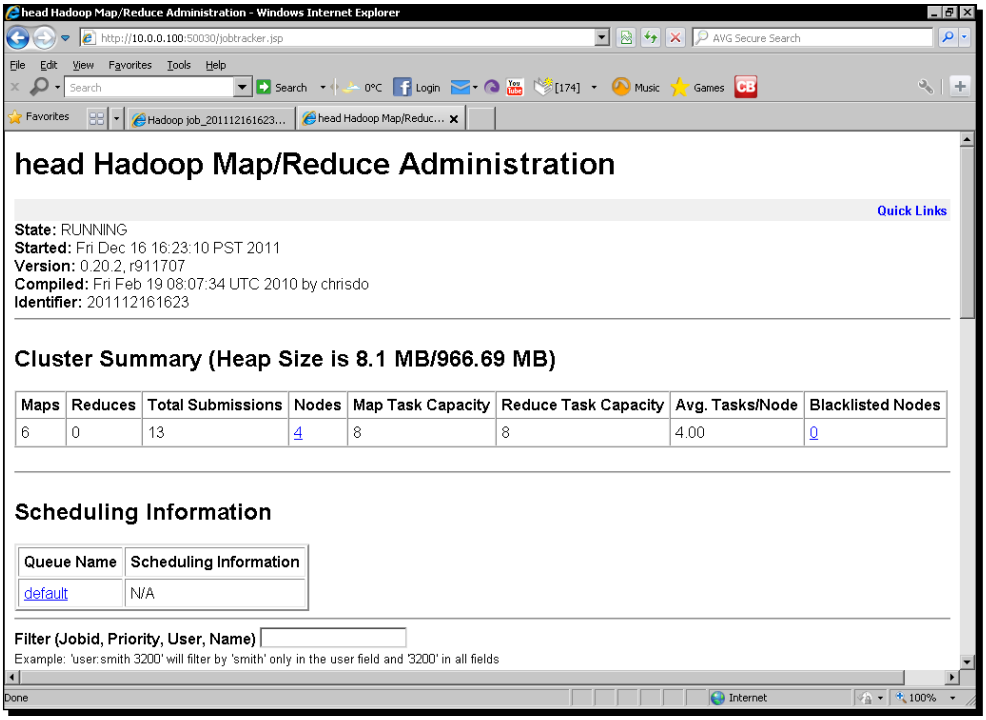


7. Monitor the job progress in the original window; it should be proceeding, even if it is slow.

8. Restart the dead TaskTracker process:

```
$ start-all.sh
```

9. Monitor the MapReduce web UI. After a little time the number of nodes should be back to its original number as shown in the following screenshot:



What just happened?

The MapReduce web interface provides a lot of information on both the cluster as well as the jobs it executes. For our interests here, the important data is the cluster summary that shows the currently executing number of map and reduce tasks, the total number of submitted jobs, the number of nodes and their map and reduce capacity, and finally, any blacklisted nodes.

The relationship of the JobTracker process to the TaskTracker process is quite different than that between NameNode and DataNode but a similar heartbeat/monitoring mechanism is used.

The TaskTracker process frequently sends heartbeats to the JobTracker, but instead of status reports of block health, they contain progress reports of the assigned task and available capacity. Each node has a configurable number of map and reduce task slots (the default for each is two), which is why we see four nodes and eight map and reduce slots in the first web UI screenshot.

When we kill the TaskTracker process, its lack of heartbeats is measured by the JobTracker process and after a configurable amount of time, the node is assumed to be dead and we see the reduced cluster capacity reflected in the web UI.



The timeout for a TaskTracker process to be considered dead is modified by the `mapred.tasktracker.expiry.interval` property, configured in `mapred-site.xml`.

When a TaskTracker process is marked as dead, the JobTracker process also considers its in-progress tasks as failed and re-assigns them to the other nodes in the cluster. We see this implicitly by watching the job proceed successfully despite a node being killed.

After the TaskTracker process is restarted it sends a heartbeat to the JobTracker, which marks it as alive and reintegrates it into the MapReduce cluster. This we see through the cluster node and task slot capacity returning to their original values as can be seen in the final screenshot.

Comparing the DataNode and TaskTracker failures

We'll not perform similar two or three node killing activities with TaskTrackers as the task execution architecture renders individual TaskTracker failures relatively unimportant. Because the TaskTracker processes are under the control and coordination of JobTracker, their individual failures have no direct effect other than to reduce the cluster execution capacity. If a TaskTracker instance fails, the JobTracker will simply schedule the failed tasks on a healthy TaskTracker process in the cluster. The JobTracker is free to reschedule tasks around the cluster because TaskTracker is conceptually stateless; a single failure does not affect other parts of the job.

In contrast, loss of a DataNode—which is intrinsically stateful—can affect the persistent data held on HDFS, potentially making it unavailable.

This highlights the nature of the various nodes and their relationship to the overall Hadoop framework. The DataNode manages data, and the TaskTracker reads and writes that data. Catastrophic failure of every TaskTracker would still leave us with a completely functional HDFS; a similar failure of the NameNode process would leave a live MapReduce cluster that is effectively useless (unless it was configured to use a different storage system).

Permanent failure

Our recovery scenarios so far have assumed that the dead node can be restarted on the same physical host. But what if it can't due to the host having a critical failure? The answer is simple; you can remove the host from the slave's file and Hadoop will no longer try to start a DataNode or TaskTracker on that host. Conversely, if you get a replacement machine with a different hostname, add this new host to the same file and run `start-all.sh`.



Note that the slave's file is only used by tools such as the `start/stop` and `slaves.sh` scripts. You don't need to keep it updated on every node, but only on the hosts where you generally run such commands. In practice, this is likely to be either a dedicated head node or the host where the NameNode or JobTracker processes run. We'll explore these setups in *Chapter 7*, *Keeping Things Running*.

Killing the cluster masters

Though the failure impact of DataNode and TaskTracker processes is different, each individual node is relatively unimportant. Failure of any single TaskTracker or DataNode is not a cause for concern and issues only occur if multiple others fail, particularly in quick succession. But we only have one JobTracker and NameNode; let's explore what happens when they fail.

Time for action – killing the JobTracker

We'll first kill the JobTracker process which we should expect to impact our ability to execute MapReduce jobs but not affect the underlying HDFS filesystem.

1. Log on to the JobTracker host and kill its process.
2. Attempt to start a test MapReduce job such as Pi or WordCount:

```
$ Hadoop jar wc.jar WordCount3 test.txt output
Starting Job
11/12/11 16:03:29 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9001. Already tried 0 time(s).
11/12/11 16:03:30 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9001. Already tried 1 time(s).
…
11/12/11 16:03:38 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9001. Already tried 9 time(s).
```

```
java.net.ConnectException: Call to /10.0.0.100:9001 failed on
connection exception: java.net.ConnectException: Connection
refused
  at org.apache.hadoop.ipc.Client.wrapException(Client.java:767)
  at org.apache.hadoop.ipc.Client.call(Client.java:743)
  at org.apache.hadoop.ipc.RPC$Invoker.invoke(RPC.java:220)
…
```

3. Perform some HDFS operations:

```
$ hadoop fs -ls /
Found 2 items
drwxr-xr-x   - hadoop supergroup          0 2011-12-11 19:19 /user
drwxr-xr-x   - hadoop supergroup          0 2011-12-04 20:38 /var
$ hadoop fs -cat test.txt
This is a test file
```

What just happened?

After killing the JobTracker process we attempted to launch a MapReduce job. From the walk-through in *Chapter 2*, *Getting Hadoop Up and Running*, we know that the client on the machine where we are starting the job attempts to communicate with the JobTracker process to initiate the job scheduling activities. But in this case there was no running JobTracker, this communication did not happen and the job failed.

We then performed a few HDFS operations to highlight the point in the previous section; a non-functional MapReduce cluster will not directly impact HDFS, which will still be available to all clients and operations.

Starting a replacement JobTracker

The recovery of the MapReduce cluster is also pretty straightforward. Once the JobTracker process is restarted, all the subsequent MapReduce jobs are successfully processed.

Note that when the JobTracker was killed, any jobs that were in flight are lost and need to be restarted. Watch out for temporary files and directories on HDFS; many MapReduce jobs write temporary data to HDFS that is usually cleaned up on job completion. Failed jobs—especially the ones failed due to a JobTracker failure—are likely to leave such data behind and this may require a manual clean-up.

Have a go hero – moving the JobTracker to a new host

But what happens if the host on which the JobTracker process was running has a fatal hardware failure and cannot be recovered? In such situations you will need to start a new JobTracker process on a different host. This requires all nodes to have their `mapred-site.xml` file updated with the new location and the cluster restarted. Try this! We'll talk about it more in the next chapter.

Time for action – killing the NameNode process

Let's now kill the NameNode process, which we should expect to directly stop us from accessing HDFS and by extension, prevent the MapReduce jobs from executing:



Don't try this on an operationally important cluster. Though the impact will be short-lived, it effectively kills the entire cluster for a period of time.

1. Log onto the NameNode host and list the running processes:

```
$ jps
2372 SecondaryNameNode
2118 NameNode
2434 JobTracker
5153 Jps
```

2. Kill the NameNode process. Don't worry about SecondaryNameNode, it can keep running.

3. Try to access the HDFS filesystem:

```
$ hadoop fs -ls /
11/12/13 16:00:05 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 0 time(s).
11/12/13 16:00:06 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 1 time(s).
11/12/13 16:00:07 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 2 time(s).
11/12/13 16:00:08 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 3 time(s).
11/12/13 16:00:09 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 4
time(s).
…
Bad connection to FS. command aborted.
```

4. Submit the MapReduce job:

```
$ hadoop jar hadoop/hadoop-examples-1.0.4.jar  pi 10 100
Number of Maps  = 10
Samples per Map = 100
11/12/13 16:00:35 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 0 time(s).
11/12/13 16:00:36 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 1 time(s).
11/12/13 16:00:37 INFO ipc.Client: Retrying connect to server:
/10.0.0.100:9000. Already tried 2 time(s).
…
java.lang.RuntimeException: java.net.ConnectException: Call
to /10.0.0.100:9000 failed on connection exception: java.net.
ConnectException: Connection refused
  at org.apache.hadoop.mapred.JobConf.getWorkingDirectory(JobConf.
java:371)
  at org.apache.hadoop.mapred.FileInputFormat.
setInputPaths(FileInputFormat.java:309)
…
Caused by: java.net.ConnectException: Call to /10.0.0.100:9000
failed on connection exception: java.net.ConnectException:
Connection refused
…
```

5. Check the running processes:

```
$ jps
2372 SecondaryNameNode
5253 Jps
2434 JobTracker
Restart the NameNode
$ start-all.sh
```

6. Access HDFS:

```
$ Hadoop fs -ls /
Found 2 items

drwxr-xr-x   - hadoop supergroup          0 2011-12-16 16:18 /user

drwxr-xr-x   - hadoop supergroup          0 2011-12-16 16:23 /var
```

What just happened?

We killed the NameNode process and tried to access the HDFS filesystem. This of course failed; without the NameNode there is no server to receive our filesystem commands.

We then tried to submit a MapReduce job and this also failed. From the abbreviated exception stack trace you can see that while trying to set up the input paths for the job data, the JobTracker also tried and failed to connect to NameNode.

We then confirmed that the JobTracker process is healthy and it was the NameNode's unavailability that caused the MapReduce task to fail.

Finally, we restarted the NameNode and confirmed that we could once again access the HDFS filesystem.

Starting a replacement NameNode

With the differences identified so far between the MapReduce and HDFS clusters, it shouldn't be a surprise to learn that restarting a new NameNode on a different host is not as simple as moving the JobTracker. To put it more starkly, having to move NameNode due to a hardware failure is probably the worst crisis you can have with a Hadoop cluster. Unless you have prepared carefully, the chance of losing all your data is very high.

That's quite a statement and we need to explore the nature of the NameNode process to understand why this is the case.

The role of the NameNode in more detail

So far we've spoken of the NameNode process as the coordinator between the DataNode processes and the service responsible for ensuring the configuration parameters, such as block replication values, are honored. This is an important set of tasks but it's also very operationally focused. The NameNode process also has the responsibility of managing the HDFS filesystem metadata; a good analogy is to think of it holding the equivalent of the file allocation table in a traditional filesystem.

File systems, files, blocks, and nodes

When accessing HDFS you rarely care about blocks. You want to access a given file at a certain location in the filesystem. To facilitate this, the NameNode process is required to maintain numerous pieces of information:

- The actual filesystem contents, the names of all the files, and their containing directories
- Additional metadata about each of these elements, such as size, ownership, and replication factor

- The mapping of which blocks hold the data for each file
- The mapping of which nodes in the cluster hold which blocks and from this, the current replication state of each

All but the last of the preceding points is persistent data that must be maintained across restarts of the NameNode process.

The single most important piece of data in the cluster – fsimage

The NameNode process stores two data structures to disk, the `fsimage` file and the edits log of changes to it. The `fsimage` file holds the key filesystem attributes mentioned in the previous section; the name and details of each file and directory on the filesystem and the mapping of the blocks that correspond to each.

If the `fsimage` file is lost, you have a series of nodes holding blocks of data without any knowledge of which blocks correspond to which part of which file. In fact, you don't even know which files are supposed to be constructed in the first place. Loss of the `fsimage` file leaves you with all the filesystem data but renders it effectively useless.

The `fsimage` file is read by the NameNode process at startup and is held and manipulated in memory for performance reasons. To avoid changes to the filesystem being lost, any modifications made are written to the edits log throughout the NameNode's uptime. The next time it restarts, it looks for this log at startup and uses it to update the `fsimage` file which it then reads into memory.



This process can be optimized by the use of the SecondaryNameNode which we'll mention later.

DataNode startup

When a DataNode process starts up, it commences its heartbeat process by reporting to the NameNode process on the blocks it holds. As explained earlier in this chapter, this is how the NameNode process knows which node should be used to service a request for a given block. If the NameNode process itself restarts, it uses the re-establishment of the heartbeats with all the DataNode processes to construct its mapping of blocks to nodes.

With the DataNode processes potentially coming in and out of the cluster, there is little use in this mapping being stored persistently as the on-disk state would often be out-of-date with the current reality. This is why the NameNode process does not persist the location of which blocks are held on which nodes.

Safe mode

If you look at the HDFS web UI or the output of `dfsadmin` shortly after starting an HDFS cluster, you will see a reference to the cluster being in **safe mode** and the required threshold of the reported blocks before it will leave safe mode. This is the DataNode block reporting mechanism at work.

As an additional safeguard, the NameNode process will hold the HDFS filesystem in a read-only mode until it has confirmed that a given percentage of blocks meet their replication threshold. In the usual case this will simply require all the DataNode processes to report in, but if some have failed, the NameNode process will need to schedule some re-replication before safe mode can be left.

SecondaryNameNode

The most unfortunately named entity in Hadoop is the **SecondaryNameNode**. When one learns of the critical `fsimage` file for the first time, this thing called SecondaryNameNode starts to sound like a helpful mitigation. Is it perhaps, as the name suggests, a second copy of the NameNode process running on another host that can take over when the primary fails? No, it isn't. SecondaryNameNode has a very specific role; it periodically reads in the state of the `fsimage` file and the edits log and writes out an updated `fsimage` file with the changes in the log applied. This is a major time saver in terms of NameNode startup. If the NameNode process has been running for a significant period of time, the edits log will be huge and it will take a very long time (easily several hours) to apply all the changes to the old `fsimage` file's state stored on the disk. The SecondaryNameNode facilitates a faster startup.

So what to do when the NameNode process has a critical failure?

Would it help to say don't panic? There are approaches to NameNode failure and this is such an important topic that we have an entire section on it in the next chapter. But for now, the main point is that you can configure the NameNode process to write its `fsimage` file and edits log to multiple locations. Typically, a network filesystem is added as a second location to ensure a copy of the `fsimage` file outside the NameNode host.

But the process of moving to a new NameNode process on a new host requires manual effort and your Hadoop cluster is dead in the water until you do. This is something you want to have a process for and that you have tried (successfully!) in a test scenario. You really don't want to be learning how to do this when your operational cluster is down, your CEO is shouting at you, and the company is losing money.

BackupNode/CheckpointNode and NameNode HA

Hadoop 0.22 replaced SecondaryNameNode with two new components, **BackupNode** and **CheckpointNode**. The latter of these is effectively a renamed SecondaryNameNode; it is responsible for updating the `fsimage` file at regular checkpoints to decrease the NameNode startup time.

The BackupNode, however, is a step closer to the goal of a fully functional hot-backup for the NameNode. It receives a constant stream of filesystem updates from the NameNode and its in-memory state is up-to-date at any point in time, with the current state held in the master NameNode. If the NameNode dies, the BackupNode is much more capable of being brought into service as a new NameNode. The process isn't automatic and requires manual intervention and a cluster restart, but it takes some of the pain out of a NameNode failure.

Remember that Hadoop 1.0 is a continuation of the Version 0.20 branch, so it does not contain the features mentioned previously.

Hadoop 2.0 will take these extensions to the next logical step: a fully automatic NameNode failover from the current master NameNode to an up-to-date backup NameNode. This NameNode **High Availability** (**HA**) is one of the most long-requested changes to the Hadoop architecture and will be a welcome addition when complete.

Hardware failure

When we killed the various Hadoop components earlier, we were—in most cases—using termination of the Hadoop processes as a proxy for the failure of the hosting physical hardware. From experience, it is quite rare to see the Hadoop processes fail without some underlying host issue causing the problem.

Host failure

Actual failure of the host is the simplest case to consider. A machine could fail due to a critical hardware issue (failed CPU, blown power supply, stuck fans, and so on), causing sudden failure of the Hadoop processes running on the host. Critical bugs in system-level software (kernel panics, I/O locks, and so on) can also have the same effect.

Generally speaking, if the failure causes a host to crash, reboot, or otherwise become unreachable for a period of time, we can expect Hadoop to act just as demonstrated throughout this chapter.

Host corruption

A more insidious problem is when a host appears to be functioning but is in reality producing corrupt results. Examples of this could be faulty memory resulting in corruption of data or disk sector errors, resulting in data on the disk being damaged.

For HDFS, this is where the status reports of corrupted blocks that we discussed earlier come into play.

For MapReduce there is no equivalent mechanism. Just as with most other software, the TaskTracker relies on data being written and read correctly by the host and has no means to detect corruption in either task execution or during the shuffle stage.

The risk of correlated failures

There is a phenomenon that most people don't consider until it bites them; sometimes the cause of a failure will also result in subsequent failures and greatly increase the chance of encountering a data loss scenario.

As an example, I once worked on a system that used four networking devices. One of these failed and no one cared about it; there were three remaining devices, after all. Until they all failed in an 18-hour period. Turned out they all contained hard drives from a faulty batch.

It doesn't have to be quite this exotic; more frequent causes will be due to faults in the shared services or facilities. Network switches can fail, power distribution can spike, air conditioning can fail, and equipment racks can short-circuit. As we'll see in the next chapter Hadoop doesn't assign blocks to random locations, it actively seeks to adopt a placement strategy that provides some protection from such failures in shared services.

We are again talking about unlikely scenarios, most often a failed host is just that and not the tip of a failure-crisis iceberg. However, remember to never discount the unlikely scenarios, especially when taking clusters to progressively larger scale.

Task failure due to software

As mentioned earlier, it is actually relatively rare to see the Hadoop processes themselves crash or otherwise spontaneously fail. What you are likely to see more of in practice are failures caused by the tasks, that is faults in the map or reduce tasks that you are executing on the cluster.

Failure of slow running tasks

We will first look at what happens if tasks hang or otherwise appear to Hadoop to have stopped making progress.

Time for action – causing task failure

Let's cause a task to fail; before we do, we will need to modify the default timeouts:

1. Add this configuration property to `mapred-site.xml`:

```
<property>
<name>mapred.task.timeout</name>
<value>30000</value>
</property>
```

2. We will now modify our old friend WordCount from *Chapter 3*, *Understanding MapReduce*. Copy `WordCount3.java` to a new file called `WordCountTimeout.java` and add the following imports:

```
import java.util.concurrent.TimeUnit ;
import org.apache.hadoop.fs.FileSystem ;
import org.apache.hadoop.fs.FSDataOutputStream ;
```

3. Replace the `map` method with the following one:

```
    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
String lockfile = "/user/hadoop/hdfs.lock" ;
  Configuration config = new Configuration() ;
FileSystem hdfs = FileSystem.get(config) ;
Path path = new Path(lockfile) ;
if (!hdfs.exists(path))
{
byte[] bytes = "A lockfile".getBytes() ;
  FSDataOutputStream out = hdfs.create(path) ;
out.write(bytes, 0, bytes.length);
out.close() ;
TimeUnit.SECONDS.sleep(100) ;
}

String[] words = value.toString().split(" ") ;

for (String str: words)
{
        word.set(str);
        context.write(word, one);

    }
    }
  }
```

4. Compile the file after changing the class name, jar it up, and execute it on the cluster:

```
$ Hadoop jar wc.jar WordCountTimeout test.txt output

…

11/12/11 19:19:51 INFO mapred.JobClient:  map 50% reduce 0%

11/12/11 19:20:25 INFO mapred.JobClient:  map 0% reduce 0%

11/12/11 19:20:27 INFO mapred.JobClient: Task Id : attempt_2011121
11821_0004_m_000000_0, Status : FAILED

Task attempt_201112111821_0004_m_000000_0 failed to report status
for 32 seconds. Killing!

11/12/11 19:20:31 INFO mapred.JobClient:  map 100% reduce 0%

11/12/11 19:20:43 INFO mapred.JobClient:  map 100% reduce 100%

11/12/11 19:20:45 INFO mapred.JobClient: Job complete:
job_201112111821_0004

11/12/11 19:20:45 INFO mapred.JobClient: Counters: 18

11/12/11 19:20:45 INFO mapred.JobClient:   Job Counters

…
```

What just happened?

We first modified a default Hadoop property that manages how long a task can seemingly make no progress before the Hadoop framework considers it for termination.

Then we modified WordCount3 to add some logic that causes the task to sleep for 100 seconds. We used a lock file on HDFS to ensure that only a single task instance sleeps. If we just had the sleep statement in the map operation without any checks, every mapper would timeout and the job would fail.

Have a go hero – HDFS programmatic access

We said we would not really deal with programmatic access to HDFS in this book. However, take a look at what we have done here and browse through the Javadoc for these classes. You will find that the interface largely follows the patterns for access to a standard Java filesystem.

Then we compile, jar up the classes, and execute the job on the cluster. The first task goes to sleep and after exceeding the threshold we set (the value was specified in milliseconds), Hadoop kills the task and reschedules another mapper to process the split assigned to the failed task.

Hadoop's handling of slow-running tasks

Hadoop has a balancing act to perform here. It wants to terminate tasks that have got stuck or, for other reasons, are running abnormally slowly; but sometimes complex tasks simply take a long time. This is especially true if the task relies on any external resources to complete its execution.

Hadoop looks for evidence of progress from a task when deciding how long it has been idle/quiet/stuck. Generally this could be:

- Emitting results
- Writing values to counters
- Explicitly reporting progress

For the latter, Hadoop provides the `Progressable` interface which contains one method of interest:

```
Public void progress() ;
```

The `Context` class implements this interface, so any mapper or reducer can call `context.progress()` to show it is alive and continuing to process.

Speculative execution

Typically, a MapReduce job will comprise of many discrete maps and reduce task executions. When run across a cluster, there is a real risk that a misconfigured or ill host will cause its tasks to run significantly slower than the others.

To address this, Hadoop will assign duplicate maps or reduce tasks across the cluster towards the end of the map or reduce phase. This speculative task execution is aimed at preventing one or two slow running tasks from causing a significant impact on the overall job execution time.

Hadoop's handling of failing tasks

Tasks won't just hang; sometimes they'll explicitly throw exceptions, abort, or otherwise stop executing in a less silent way than the ones mentioned previously.

Hadoop has three configuration properties that control how it responds to task failures, all set in `mapred-site.xml`:

- `mapred.map.max.attempts`: A given map task will be retried this many times before causing the job to fail

- `mapred.reduce.max.attempts`: A given reduce task will be retried these many times before causing the job to fail
- `mapred.max.tracker.failures`: The job will fail if this many individual task failures are recorded

The default value for all of these is 4.



Note that it does not make sense for `mapred.tracker.max.failures` to be set to a value smaller than either of the other two properties.

Which of these you consider setting will depend on the nature of your data and jobs. If your jobs access external resources that may occasionally cause transient errors, increasing the number of repeat failures of a task may be useful. But if the task is very data-specific, these properties may be less applicable as a task that fails once will do so again. However, note that a default value higher than 1 does make sense as in a large complex system various transient failures are always possible.

Have a go hero – causing tasks to fail

Modify the WordCount example; instead of sleeping, have it throw a RuntimeException based on a random number. Modify the cluster configuration and explore the relationship between the configuration properties that manage how many failed tasks will cause the whole job to fail.

Task failure due to data

The final types of failure that we will explore are those related to data. By this, we mean tasks that crash because a given record had corrupt data, used the wrong data types or formats, or a wide variety of related problems. We mean those cases where the data received diverges from expectations.

Handling dirty data through code

One approach to dirty data is to write mappers and reducers that deal with data defensively. So, for example, if the value received by the mapper should be a comma-separated list of values, first validate the number of items before processing the data. If the first value should be a string representation of an integer, ensure that the conversion into a numerical type has solid error handling and default behavior.

The problem with this approach is that there will always be some type of weird data input that was not considered, no matter how careful you were. Did you consider receiving values in a different unicode character set? What about multiple character sets, null values, badly terminated strings, wrongly encoded escape characters, and so on?

If the data input to your jobs is something you generate and/or control, these possibilities are less of a concern. However, if you are processing data received from external sources, there will always be grounds for surprise.

Using Hadoop's skip mode

The alternative is to configure Hadoop to approach task failures differently. Instead of looking upon a failed task as an atomic event, Hadoop can instead attempt to identify which records may have caused the problem and exclude them from future task executions. This mechanism is known as **skip mode**. This can be useful if you are experiencing a wide variety of data issues where coding around them is not desirable or practical. Alternatively, you may have little choice if, within your job, you are using third-party libraries for which you may not have the source code.

Skip mode is currently available only for jobs written to the pre 0.20 version of API, which is another consideration.

Time for action – handling dirty data by using skip mode

Let's see skip mode in action by writing a MapReduce job that receives the data that causes it to fail:

1. Save the following Ruby script as `gendata.rb`:

```
File.open("skipdata.txt", "w") do |file|
  3.times do
    500000.times{file.write("A valid record\n")}
    5.times{file.write("skiptext\n")}
  end
  500000.times{file.write("A valid record\n")}
End
```

2. Run the script:

```
$ ruby gendata.rb
```

3. Check the size of the generated file and its number of lines:

```
$ ls -lh skipdata.txt
-rw-rw-r-- 1 hadoop hadoop 29M 2011-12-17 01:53 skipdata.txt
~$ cat skipdata.txt | wc -l
2000015
```

4. Copy the file onto HDFS:

```
$ hadoop fs -put skipdata.txt skipdata.txt
```

5. Add the following property definition to `mapred-site.xml`:

```
<property>
<name>mapred.skip.map.max.skip.records</name>
<value5</value>
</property>
```

6. Check the value set for `mapred.max.map.task.failures` and set it to `20` if it is lower.

7. Save the following Java file as `SkipData.java`:

```
import java.io.IOException;

import org.apache.hadoop.conf.* ;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.* ;
import org.apache.hadoop.mapred.* ;
import org.apache.hadoop.mapred.lib.* ;

public class SkipData
{

    public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, LongWritable>
    {

        private final static LongWritable one = new
LongWritable(1);
        private Text word = new Text("totalcount");

        public void map(LongWritable key, Text value,
            OutputCollector<Text, LongWritable> output,
                Reporter reporter) throws IOException
                {
                    String line = value.toString();

                if (line.equals("skiptext"))
                throw new RuntimeException("Found skiptext") ;
                output.collect(word, one);
            }
        }

        public static void main(String[] args) throws Exception
        {
            Configuration config = new Configuration() ;
```

```
            JobConf conf = new JobConf(config, SkipData.class);
            conf.setJobName("SkipData");

            conf.setOutputKeyClass(Text.class);
            conf.setOutputValueClass(LongWritable.class);

            conf.setMapperClass(MapClass.class);
            conf.setCombinerClass(LongSumReducer.class);
            conf.setReducerClass(LongSumReducer.class);

            FileInputFormat.setInputPaths(conf,args[0]) ;
            FileOutputFormat.setOutputPath(conf, new
Path(args[1])) ;

            JobClient.runJob(conf);
        }
    }
```

8. Compile this file and jar it into `skipdata.jar`.

9. Run the job:

```
$ hadoop jar skip.jar SkipData skipdata.txt output
…
11/12/16 17:59:07 INFO mapred.JobClient:  map 45% reduce 8%
11/12/16 17:59:08 INFO mapred.JobClient: Task Id : attempt_2011121
61623_0014_m_000003_0, Status : FAILED
java.lang.RuntimeException: Found skiptext
  at SkipData$MapClass.map(SkipData.java:26)
  at SkipData$MapClass.map(SkipData.java:12)
  at org.apache.hadoop.mapred.MapRunner.run(MapRunner.java:50)
  at org.apache.hadoop.mapred.MapTask.runOldMapper(MapTask.
java:358)
  at org.apache.hadoop.mapred.MapTask.run(MapTask.java:307)
  at org.apache.hadoop.mapred.Child.main(Child.java:170)

11/12/16 17:59:11 INFO mapred.JobClient:  map 42% reduce 8%
...
11/12/16 18:01:26 INFO mapred.JobClient:  map 70% reduce 16%
11/12/16 18:01:35 INFO mapred.JobClient:  map 71% reduce 16%
11/12/16 18:01:43 INFO mapred.JobClient: Task Id : attempt_2011111
61623_0014_m_000003_2, Status : FAILED
java.lang.RuntimeException: Found skiptext
...
11/12/16 18:12:44 INFO mapred.JobClient:  map 99% reduce 29%
11/12/16 18:12:50 INFO mapred.JobClient:  map 100% reduce 29%
```

```
11/12/16 18:13:00 INFO mapred.JobClient:  map 100% reduce 100%
11/12/16 18:13:02 INFO mapred.JobClient: Job complete:
job_201112161623_0014
...
```

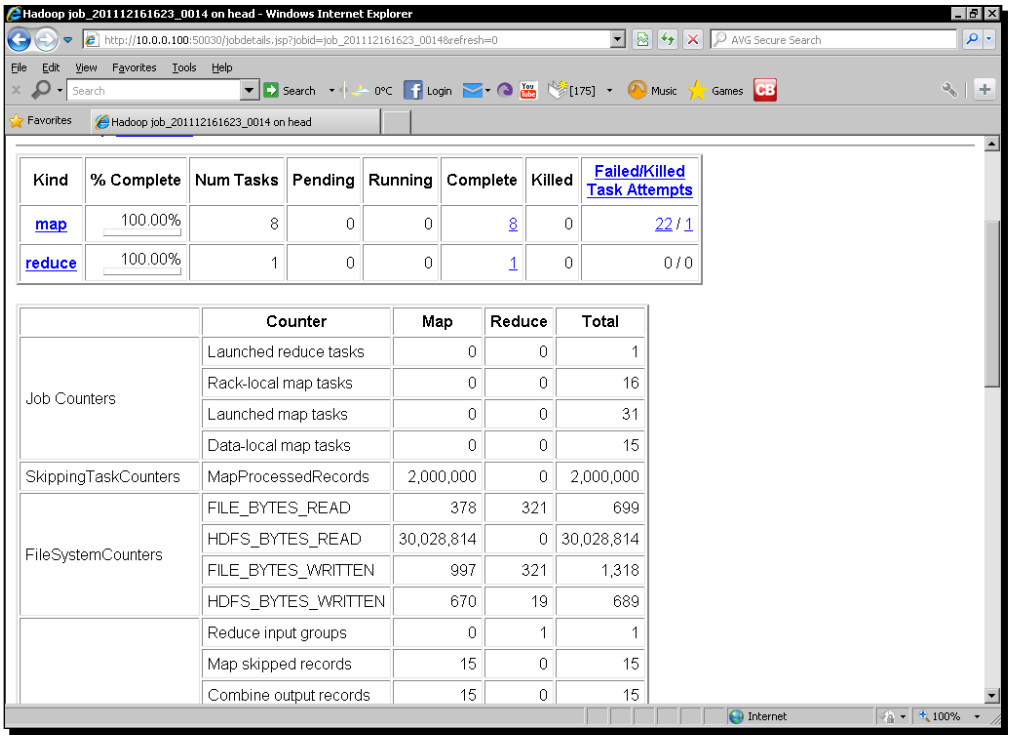
10. Examine the contents of the job output file:

```
$ hadoop fs -cat output/part-00000
totalcount  2000000
```

11. Look in the output directory for skipped records:

```
$ hadoop fs -ls output/_logs/skip
Found 15 items
-rw-r--r--   3 hadoop supergroup        203 2011-12-16 18:05 /
user/hadoop/output/_logs/skip/attempt_201112161623_0014_m_000001_3
-rw-r--r--   3 hadoop supergroup        211 2011-12-16 18:06 /
user/hadoop/output/_logs/skip/attempt_201112161623_0014_m_000001_4
…
```

12. Check the job details from the MapReduce UI to observe the recorded statistics as shown in the following screenshot:



What just happened?

We had to do a lot of setup here so let's walk through it a step at a time.

Firstly, we needed to configure Hadoop to use skip mode; it is disabled by default. The key configuration property was set to `5`, meaning that we didn't want the framework to skip any set of records greater than this number. Note that this includes the invalid records, and by setting this property to `0` (the default) Hadoop will not enter skip mode.

We also check to ensure that Hadoop is configured with a sufficiently high threshold for repeated task attempt failures, which we will explain shortly.

Next we needed a test file that we could use to simulate dirty data. We wrote a simple Ruby script that generated a file with 2 million lines that we would treat as valid with three sets of five bad records interspersed through the file. We ran this script and confirmed that the generated file did indeed have 2,000,015 lines. This file was then put on HDFS where it would be the job input.

We then wrote a simple MapReduce job that effectively counts the number of valid records. Every time the line reads from the input as the valid text we emit an additional count of 1 to what will be aggregated as a final total. When the invalid lines are encountered, the mapper fails by throwing an exception.

We then compile this file, jar it up, and run the job. The job takes a while to run and as seen from the extracts of the job status, it follows a pattern that we have not seen before. The map progress counter will increase but when a task fails, the progress will drop back then start increasing again. This is skip mode in action.

Every time a key/value pair is passed to the mapper, Hadoop by default increments a counter that allows it to keep track of which record caused a failure.



If your map or reduce tasks process their input through mechanisms other than directly receiving all data via the arguments to the map or reduce method (for example, from asynchronous processes or caches) you will need to ensure you explicitly update this counter manually.

When a task fails, Hadoop retries it on the same block but attempts to work around the invalid records. Through a binary search approach, the framework performs retries across the data until the number of skipped records is no greater than the maximum value we configured earlier, that is 5. This process does require multiple task retries and failures as the framework seeks the optimal batch to skip, which is why we had to ensure the framework was configured to be tolerant of a higher-than-usual number of repeated task failures.

We watched the job continue following this back and forth process and on completion checked the contents of the output file. This showed 2,000,000 processed records, that is the correct number of valid records in our input file. Hadoop successfully managed to skip only the three sets of five invalid records.

We then looked within the `_logs` directory in the job output directory and saw that there is a skip directory containing the sequence files of the skipped records.

Finally, we looked at the MapReduce web UI to see the overall job status, which included both the number of records processed while in skip mode as well as the number of records skipped. Note that the total number of failed tasks was 22, which is greater than our threshold for failed map attempts, but this number is aggregate failures across multiple tasks.

To skip or not to skip...

Skip mode can be very effective but as we have seen previously, there is a performance penalty caused by Hadoop having to determine which record range to skip. Our test file was actually quite helpful to Hadoop; the bad records were nicely grouped in three groups and only accounted for a tiny fraction of the overall data set. If there were many more invalid records in the input data and they were spread much more widely across the file, a more effective approach may have been to use a precursor MapReduce job to filter out all the invalid records.

This is why we have presented the topics of writing code to handle bad data and using skip mode consecutively. Both are valid techniques that you should have in your tool belt. There is no single answer to when one or the other is the best approach, you need to consider the input data, performance requirements, and opportunities for hardcoding before making a decision.

Summary

We have caused a lot of destruction in this chapter and I hope you never have to deal with this much failure in a single day with an operational Hadoop cluster. There are some key learning points from the experience.

In general, component failures are not something to fear in Hadoop. Particularly with large clusters, failure of some component or host will be pretty commonplace and Hadoop is engineered to handle this situation. HDFS, with its responsibility to store data, actively manages the replication of each block and schedules new copies to be made when the DataNode processes die.

MapReduce has a stateless approach to TaskTracker failure and in general simply schedules duplicate jobs if one fails. It may also do this to prevent the misbehaving hosts from slowing down the whole job.

Failure of the HDFS and MapReduce master nodes is a more significant failure. In particular, the NameNode process holds critical filesystem data and you must actively ensure you have it set up to allow a new NameNode process to take over.

In general, hardware failures will look much like the previous process failures, but always be aware of the possibility of correlated failures. If tasks fail due to software errors, Hadoop will retry them within configurable thresholds. Data-related errors can be worked around by employing skip mode, though it will come with a performance penalty.

Now that we know how to handle failures in our cluster, we will spend the next chapter working through the broader issues of cluster setup, health, and maintenance.

7
Keeping Things Running

Having a Hadoop cluster is not all about writing interesting programs to do clever data analysis. You also need to maintain the cluster, and keep it tuned and ready to do the data crunching you want.

In this chapter we will cover:

- More about Hadoop configuration properties
- How to select hardware for your cluster
- How Hadoop security works
- Managing the NameNode
- Managing HDFS
- Managing MapReduce
- Scaling the cluster

Although these topics are operationally focused, they do give us an opportunity to explore some aspects of Hadoop we have not looked at before. Therefore, even if you won't be personally managing the cluster, there should be useful information here for you too.

A note on EMR

One of the main benefits of using cloud services such as those offered by Amazon Web Services is that much of the maintenance overhead is borne by the service provider. Elastic MapReduce can create Hadoop clusters tied to the execution of a single task (non-persistent job flows) or allow long-running clusters that can be used for multiple jobs (persistent job flows). When non-persistent job flows are used, the actual mechanics of how the underlying Hadoop cluster is configured and run are largely invisible to the user. Consequently, users employing non-persistent job flows will not need to consider many of the topics in this chapter. If you are using EMR with persistent job flows, many topics (but not all) do become relevant.

We will generally talk about local Hadoop clusters in this chapter. If you need to reconfigure a persistent job flow, use the same Hadoop properties but set them as described in *Chapter 3*, *Writing MapReduce Jobs*.

Hadoop configuration properties

Before we look at running the cluster, let's talk a little about Hadoop's configuration properties. We have been introducing many of these along the way, and there are a few additional points worth considering.

Default values

One of the most mystifying things to a new Hadoop user is the large number of configuration properties. Where do they come from, what do they mean, and what are their default values?

If you have the full Hadoop distribution—that is, not just the binary distribution—the following XML files will answer your questions:

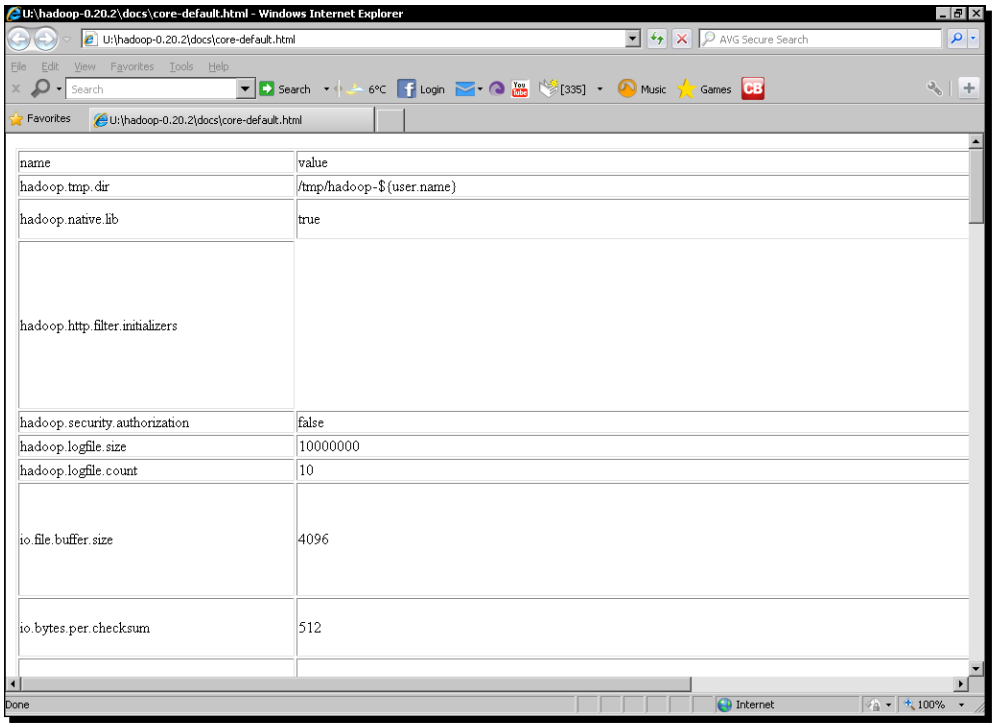
- `Hadoop/src/core/core-default.xml`
- `Hadoop/src/hdfs/hdfs-default.xml`
- `Hadoop/src/mapred/mapred-default.xml`

Time for action – browsing default properties

Fortunately, the XML documents are not the only way of looking at the default values; there are also more readable HTML versions, which we'll now take a quick look at.

These files are not included in the Hadoop binary-only distribution; if you are using that, you can also find these files on the Hadoop website.

1. Point your browser at the `docs/core-default.html` file within your Hadoop distribution directory and browse its contents. It should look like the next screenshot:



2. Now, similarly, browse these other files:

   - `Hadoop/docs/hdfs-default.html`
   - `Hadoop/docs/mapred-default.html`

What just happened?

As you can see, each property has a name, default value, and a brief description. You will also see there are indeed a very large number of properties. Do not expect to understand all of these now, but do spend a little time browsing to get a flavor for the type of customization allowed by Hadoop.

Additional property elements

When we have previously set properties in the configuration files, we have used an XML element of the following form:

```
<property>
<name>the.property.name</name>
<value>The property value</value>
</property>
```

There are an additional two optional XML elements we can add, `description` and `final`. A fully described property using these additional elements now looks as follows:

```
<property>
<name>the.property.name</name>
<value>The default property value</value>
<description>A textual description of the property</description>
<final>Boolean</final>
</property>
```

The description element is self-explanatory and provides the location for the descriptive text we saw for each property in the preceding HTML files.

The `final` property has a similar meaning as in Java: any property marked `final` cannot be overridden by values in any other files or by other means; we will see this shortly. Use this for those properties where for performance, integrity, security, or other reasons, you wish to enforce cluster-wide values.

Default storage location

You will see properties that modify where Hadoop stores its data on both the local disk and HDFS. There's one property used as the basis for many others `hadoop.tmp.dir`, which is the root location for all Hadoop files, and its default value is `/tmp`.

Unfortunately, many Linux distributions—including Ubuntu—are configured to remove the contents of this directory on each reboot. This means that if you do not override this property, you will lose all your HDFS data on the next host reboot. Therefore, it is worthwhile to set something like the following in `core-site.xml`:

```
<property>
<name>hadoop.tmp.dir</name>
<value>/var/lib/hadoop</value>
</property>
```

Remember to ensure the location is writable by the user who will start Hadoop, and that the disk the directory is located on has enough space. As you will see later, there are a number of other properties that allow more granular control of where particular types of data are stored.

Where to set properties

We have previously used the configuration files to specify new values for Hadoop properties. This is fine, but does have an overhead if we are trying to find the best value for a property or are executing a job that requires special handling.

It is possible to use the `JobConf` class to programmatically set configuration properties on the executing job. There are two types of methods supported, the first being those that are dedicated to setting a specific property, such as the ones we've seen for setting the job name, input, and output formats, among others. There are also methods to set properties such as the preferred number of map and reduce tasks for the job.

In addition, there are a set of generic methods, such as the following:

- `Void set(String key, String value);`
- `Void setIfUnset(String key, String value);`
- `Void setBoolean( String key, Boolean value);`
- `Void setInt(String key, int value);`

These are more flexible and do not require specific methods to be created for each property we wish to modify. However, they also lose compile time checking meaning you can use an invalid property name or assign the wrong type to a property and will only find out at runtime.



This ability to set property values both programmatically and in the configuration files is an important reason for the ability to mark a property as `final`. For properties for which you do not want any submitted job to have the ability to override them, set them as final within the master configuration files.

Setting up a cluster

Before we look at how to keep a cluster running, let's explore some aspects of setting it up in the first place.

How many hosts?

When considering a new Hadoop cluster, one of the first questions is how much capacity to start with. We know that we can add additional nodes as our needs grow, but we also want to start off in a way that eases that growth.

There really is no clear-cut answer here, as it will depend largely on the size of the data sets you will be processing and the complexity of the jobs to be executed. The only near-absolute is to say that if you want a replication factor of *n*, you should have at least that many nodes. Remember though that nodes will fail, and if you have the same number of nodes as the default replication factor then any single failure will push blocks into an under-replicated state. In most clusters with tens or hundreds of nodes, this is not a concern; but for very small clusters with a replication factor of 3, the safest approach would be a five-node cluster.

Calculating usable space on a node

An obvious starting point for the required number of nodes is to look at the size of the data set to be processed on the cluster. If you have hosts with 2 TB of disk space and a 10 TB data set, the temptation would be to assume that five nodes is the minimum number needed.

This is incorrect, as it omits consideration of the replication factor and the need for temporary space. Recall that the output of mappers is written to the local disk to be retrieved by the reducers. We need to account for this non-trivial disk usage.

A good rule of thumb would be to assume a replication factor of 3, and that 25 percent of what remains should be accounted for as temporary space. Using these assumptions, the calculation of the needed cluster for our 10 TB data set on 2 TB nodes would be as follows:

- Divide the total storage space on a node by the replication factor:

  2 TB/3 = 666 GB

- Reduce this figure by 25 percent to account for temp space:

  666 GB * 0.75 = 500 GB

- Each 2 TB node therefore has approximately 500 GB (0.5 TB) of usable space
- Divide the data set size by this figure:

  10 TB / 500 GB = 20

So our 10 TB data set will likely need a 20 node cluster as a minimum, four times our naïve estimate.

This pattern of needing more nodes than expected is not unusual and should be remembered when considering how high-spec you want the hosts to be; see the *Sizing hardware* section later in this chapter.

Location of the master nodes

The next question is where the NameNode, JobTracker, and SecondaryNameNode will live. We have seen that a DataNode can run on the same host as the NameNode and the TaskTracker can co-exist with the JobTracker, but this is unlikely to be a great setup for a production cluster.

As we will see, the NameNode and SecondaryNameNode have some specific resource requirements, and anything that affects their performance is likely to slow down the entire cluster operation.

The ideal situation would be to have the NameNode, JobTracker, and SecondaryNameNode on their own dedicated hosts. However, for very small clusters, this would result in a significant increase in the hardware footprint without necessarily reaping the full benefit.

If at all possible, the first step should be to separate the NameNode, JobTracker, and SecondaryNameNode onto a single dedicated host that does not have any DataNode or TaskTracker processes running. As the cluster continues to grow, you can add an additional server host and then move the NameNode onto its own host, keeping the JobTracker and SecondaryNameNode co-located. Finally, as the cluster grows yet further, it will make sense to move to full separation.



As discussed in *Chapter 6*, *Keeping Things Running*, Hadoop 2.0 will split the Secondary NameNode into Backup NameNodes and Checkpoint NameNodes. Best practice is still evolving, but aiming towards having a dedicated host each for the NameNode and at least one Backup NameNode looks sensible.

Sizing hardware

The amount of data to be stored is not the only consideration regarding the specification of the hardware to be used for the nodes. Instead, you have to consider the amount of processing power, memory, storage types, and networking available.

Much has been written about selecting hardware for a Hadoop cluster, and once again there is no single answer that will work for all cases. The big variable is the types of MapReduce tasks that will be executed on the data and, in particular, if they are bounded by CPU, memory, I/O, or something else.

Processor / memory / storage ratio

A good way of thinking of this is to look at potential hardware in terms of the CPU / memory / storage ratio. So, for example, a quad-core host with 8 GB memory and 2 TB storage could be thought of as having two cores and 4 GB memory per 1 TB of storage.

Then look at the types of MapReduce jobs you will be running, does that ratio seem appropriate? In other words, does your workload require proportionally more of one of these resources or will a more balanced configuration be sufficient?

This is, of course, best assessed by prototyping and gathering metrics, but that isn't always possible. If not, consider what part of the job is the most expensive. For example, some of the jobs we have seen are I/O bound and read data from the disk, perform simple transformations, and then write results back to the disk. If this was typical of our workload, we could likely use hardware with more storage—especially if it was delivered by multiple disks to increase I/O—and use less CPU and memory.

Conversely, jobs that perform very heavy number crunching would need more CPU, and those that create or use large data structures would benefit from memory.

Think of it in terms of limiting factors. If your job was running, would it be CPU-bound (processors at full capacity; memory and I/O to spare), memory-bound (physical memory full and swapping to disk; CPU and I/O to spare), or I/O-bound (CPU and memory to spare, but data being read/written to/from disk at maximum possible speed)? Can you get hardware that eases that bound?

This is of course a limitless process, as once you ease one bound another will manifest itself. So always remember that the idea is to get a performance profile that makes sense in the context of your likely usage scenario.

What if you really don't know the performance characteristics of your jobs? Ideally, try to find out, do some prototyping on any hardware you have and use that to inform your decision. However, if even that is not possible, you will have to go for a configuration and try it out. Remember that Hadoop supports heterogeneous hardware—though having uniform specifications makes your life easier in the end—so build the cluster to the minimum possible size and assess the hardware. Use this knowledge to inform future decisions regarding additional host purchases or upgrades of the existing fleet.

EMR as a prototyping platform

Recall that when we configured a job on Elastic MapReduce we chose the type of hardware for both the master and data/task nodes. If you plan to run your jobs on EMR, you have a built-in capability to tweak this configuration to find the best combination of hardware specifications to price and execution speed.

However, even if you do not plan to use EMR full-time, it can be a valuable prototyping platform. If you are sizing a cluster but do not know the performance characteristics of your jobs, consider some prototyping on EMR to gain better insight. Though you may end up spending money on the EMR service that you had not planned, this will likely be a lot less than the cost of finding out you have bought completely unsuitable hardware for your cluster.

Special node requirements

Not all hosts have the same hardware requirements. In particular, the host for the NameNode may look radically different to those hosting the DataNodes and TaskTrackers.

Recall that the NameNode holds an in-memory representation of the HDFS filesystem and the relationship between files, directories, blocks, nodes, and various metadata concerning all of this. This means that the NameNode will tend to be memory bound and may require larger memory than any other host, particularly for very large clusters or those with a huge number of files. Though 16 GB may be a common memory size for DataNodes/TaskTrackers, it's not unusual for the NameNode host to have 64 GB or more of memory. If the NameNode ever ran out of physical memory and started to use swap space, the impact on cluster performance would likely be severe.

However, though 64 GB is large for physical memory, it's tiny for modern storage, and given that the filesystem image is the only data stored by the NameNode, we don't need the massive storage common on the DataNode hosts. We care much more about NameNode reliability so are likely to have several disks in a redundant configuration. Consequently, the NameNode host will benefit from multiple small drives (for redundancy) rather than large drives.

Overall, therefore, the NameNode host is likely to look quite different from the other hosts in the cluster; this is why we made the earlier recommendations regarding moving the NameNode to its own host as soon as budget/space allows, as its unique hardware requirements are more easily satisfied this way.



The SecondaryNameNode (or CheckpointNameNode and BackupNameNode in Hadoop 2.0) share the same hardware requirements as the NameNode. You can run it on a more generic host while in its secondary capacity, but if you do ever need to switch and make it the NameNode due to failure of the primary hardware, you may be in trouble.

Storage types

Though you will find strong opinions on some of the previous points regarding the relative importance of processor, memory, and storage capacity, or I/O, such arguments are usually based around application requirements and hardware characteristics and metrics. Once we start discussing the type of storage to be used, however, it is very easy to get into flame war situations, where you will find extremely entrenched opinions.

Commodity versus enterprise class storage

The first argument will be over whether it makes most sense to use hard drives aimed at the commodity/consumer segments or those aimed at enterprise customers. The former (primarily SATA disks) are larger, cheaper, and slower, and have lower quoted figures for **mean time between failures** (**MTBF**). Enterprise disks will use technologies such as SAS or Fiber Channel, and will on the whole be smaller, more expensive, faster, and have higher quoted MTBF figures.

Single disk versus RAID

The next question will be on how the disks are configured. The enterprise-class approach would be to use **Redundant Arrays of Inexpensive Disks** (**RAID**) to group multiple disks into a single logical storage device that can quietly survive one or more disk failures. This comes with the cost of a loss in overall capacity and an impact on the read/write rates achieved.

The other position is to treat each disk independently to maximize total storage and aggregate I/O, at the cost of a single disk failure causing host downtime.

Finding the balance

The Hadoop architecture is, in many ways, predicated on the assumption that hardware will fail. From this perspective, it is possible to argue that there is no need to use any traditional enterprise-focused storage features. Instead, use many large, cheap disks to maximize the total storage and read and write from them in parallel to do likewise for I/O throughput. A single disk failure may cause the host to fail, but the cluster will, as we have seen, work around this failure.

This is a completely valid argument and in many cases makes perfect sense. What the argument ignores, however, is the cost of bringing a host back into service. If your cluster is in the next room and you have a shelf of spare disks, host recovery will likely be a quick, painless, and inexpensive task. However, if you have your cluster hosted by a commercial collocation facility, any hands-on maintenance may cost a lot more. This is even more the case if you are using fully-managed servers where you have to pay the provider for maintenance tasks. In such a situation, the extra cost and reduced capacity and I/O from using RAID may make sense.

Network storage

One thing that will almost never make sense is to use networked storage for your primary cluster storage. Be it block storage via a **Storage Area Network** (**SAN**) or file-based via **Network File System** (**NFS**) or similar protocols, these approaches constrain Hadoop by introducing unnecessary bottlenecks and additional shared devices that would have a critical impact on failure.

Sometimes, however, you may be forced for non-technical reasons to use something like this. It's not that it won't work, just that it changes how Hadoop will perform in regards to speed and tolerance to failures, so be sure you understand the consequences if this happens.

Hadoop networking configuration

Hadoop's support of networking devices is not as sophisticated as it is for storage, and consequently you have fewer hardware choices to make compared to CPU, memory, and storage setup. The bottom line is that Hadoop can currently support only one network device and cannot, for example, use all 4-gigabit Ethernet connections on a host for an aggregate of 4-gigabit throughput. If you need network throughput greater than that provided by a single-gigabit port then, unless your hardware or operating system can present multiple ports as a single device to Hadoop, the only option is to use a 10-gigabit Ethernet device.

How blocks are placed

We have talked a lot about HDFS using replication for redundancy, but have not explored how Hadoop chooses where to place the replicas for a block.

In most traditional server farms, the various hosts (as well as networking and other devices) are housed in standard-sized racks that stack the equipment vertically. Each rack will usually have a common power distribution unit that feeds it and will often have a network switch that acts as the interface between the broader network and all the hosts in the rack.

Given this setup, we can identify three broad types of failure:

- Those that affect a single host (for example, CPU/memory/disk/motherboard failure)
- Those that affect a single rack (for example, power unit or switch failure)
- Those that affect the entire cluster (for example, larger power/network failures, cooling/environmental outages)



Remember that Hadoop currently does not support a cluster that is spread across multiple data centers, so instances of the third type of failure will quite likely bring down your cluster.

By default, Hadoop will treat each node as if it is in the same physical rack. This implies that the bandwidth and latency between any pair of hosts is approximately equal and that each node is equally likely to suffer a related failure as any other.

Rack awareness

If, however, you do have a multi-rack setup, or another configuration that otherwise invalidates the previous assumptions, you can add the ability for each node to report its rack ID to Hadoop, which will then take this into account when placing replicas.

In such a setup, Hadoop tries to place the first replica of a node on a given host, the second on another within the same rack, and the third on a host in a different rack.

This strategy provides a good balance between performance and availability. When racks contain their own network switches, communication between hosts inside the rack often has lower latency than that with external hosts. This strategy places two replicas within a rack to ensure maximum speed of writing for these replicas, but keeps one outside the rack to provide redundancy in the event of a rack failure.

The rack-awareness script

If the `topology.script.file.name` property is set and points to an executable script on the filesystem, it will be used by the NameNode to determine the rack for each host.

Note that the property needs to be set and the script needs to exist only on the NameNode host.

The NameNode will pass to the script the IP address of each node it discovers, so the script is responsible for a mapping from node IP address to rack name.

If no script is specified, each node will be reported as a member of a single default rack.

Time for action – examining the default rack configuration

Let's take a look at how the default rack configuration is set up in our cluster.

1. Execute the following command:

```
$ Hadoop fsck -rack
```

2. The result should include output similar to the following:

```
Default replication factor:    3
Average block replication:     3.3045976
Corrupt blocks:                0
Missing replicas:              18 (0.5217391 %)
Number of data-nodes:          4
Number of racks:               1


The filesystem under path '/' is HEALTHY
```

What just happened?

Both the tool used and its output are of interest here. The tool is **hadoop fsck**, which can be used to examine and fix filesystem problems. As can be seen, this includes some information not dissimilar to our old friend `hadoop dfsadmin`, though that tool is focused more on the state of each node in detail while `hadoop fsck` reports on the internals of the filesystem as a whole.

One of the things it reports is the total number of racks in the cluster, which, as seen in the preceding output, has the value `1`, as expected.



This command was executed on a cluster that had recently been used for some HDFS resilience testing. This explains the figures for average block replication and under-replicated blocks.

If a block ends up with more than the required number of replicas due to a host temporarily failing, the host coming back into service will put the block above the minimum replication factor. Along with ensuring that blocks have replicas added to meet the replication factor, Hadoop will also delete excess replicas to return blocks to the replication factor.

Time for action – adding a rack awareness script

We can enhance the default flat rack configuration by creating a script that derives the rack location for each host.

1. Create a script in the Hadoop user's home directory on the NameNode host called `rack-script.sh`, containing the following text. Remember to change the IP address to one of your HDFS nodes.

```
#!/bin/bash

if [ $1 = "10.0.0.101" ]; then
    echo -n "/rack1 "
else
    echo -n "/default-rack "
fi
```

2. Make this script executable.

```
$ chmod +x rack-script.sh
```

3. Add the following property to `core-site.xml` on the NameNode host:

```
<property>
<name>topology.script.file.name</name>
<value>/home/Hadoop/rack-script.sh</value>
</property>
```

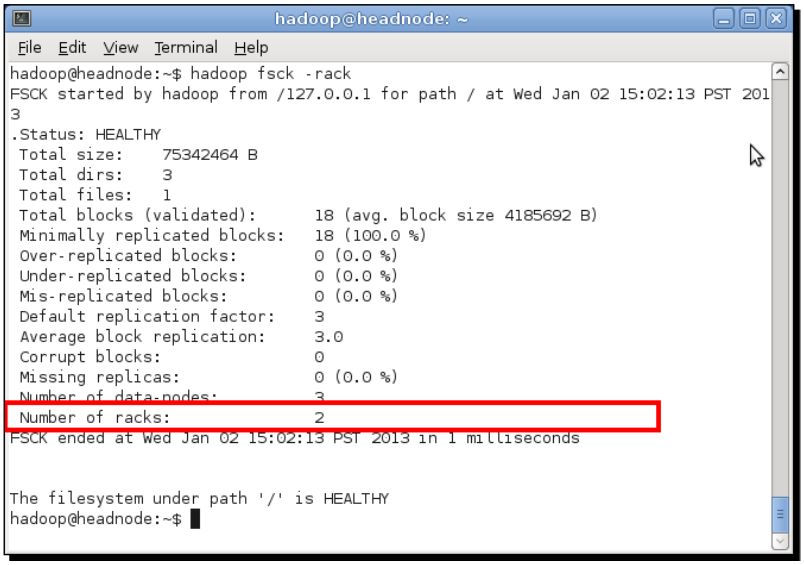
4. Restart HDFS.

```
$ start-dfs.sh
```

5. Check the filesystem via `fsck`.

```
$ Hadoop fsck –rack
```

The output of the preceding command can be shown in the following screenshot:



What just happened?

We first created a simple script that returns one value for a named node and a default value for all others. We placed this on the NameNode host and added the needed configuration property to the NameNode `core-site.xml` file.

After starting HDFS, we used `hadoop fsck` to report on the filesystem and saw that we now have a two-rack cluster. With this knowledge, Hadoop will now employ more sophisticated block placement strategies, as described previously.



Using an external host file

A common approach is to keep a separate data file akin to the `/etc/hosts` file on Unix and use this to specify the IP/rack mapping, one per line. This file can then be updated independently and read by the rack-awareness script.

What is commodity hardware anyway?

Let's revisit the question of the general characteristics of the hosts used for your cluster, and whether they should look more like a commodity white box server or something built for a high-end enterprise environment.

Part of the problem is that "commodity" is an ambiguous term. What looks cheap and cheerful for one business may seem luxuriously high-end for another. We suggest considering the following points to keep in mind when selecting hardware and then remaining happy with your decision:

- With your hardware, are you paying a premium for reliability features that duplicate some of Hadoop's fault-tolerance capabilities?
- Are the higher-end hardware features you are paying for addressing the need or risk that you have confirmed is realistic in your environment?
- Have you validated the cost of the higher-end hardware to be higher than dealing with cheaper / less reliable hardware?

Pop quiz – setting up a cluster

Q1. Which of the following is most important when selecting hardware for your new Hadoop cluster?

1. The number of CPU cores and their speed.
2. The amount of physical memory.
3. The amount of storage.
4. The speed of the storage.
5. It depends on the most likely workload.

Q2. Why would you likely not want to use network storage in your cluster?

1. Because it may introduce a new single point of failure.
2. Because it most likely has approaches to redundancy and fault-tolerance that may be unnecessary given Hadoop's fault tolerance.
3. Because such a single device may have inferior performance to Hadoop's use of multiple local disks simultaneously.
4. All of the above.

Q3. You will be processing 10 TB of data on your cluster. Your main MapReduce job processes financial transactions, using them to produce statistical models of behavior and future forecasts. Which of the following hardware choices would be your first choice for the cluster?

1. 20 hosts each with fast dual-core processors, 4 GB memory, and one 500 GB disk drive.
2. 30 hosts each with fast dual-core processors, 8 GB memory, and two 500 GB disk drives.
3. 30 hosts each with fast quad-core processors, 8 GB memory, and one 1 TB disk drive.
4. 40 hosts each with 16 GB memory, fast quad-core processors, and four 1 TB disk drives.

Cluster access control

Once you have the shiny new cluster up and running, you need to consider questions of access and security. Who can access the data on the cluster—is there sensitive data that you really don't want the whole user base to see?

The Hadoop security model

Until very recently, Hadoop had a security model that could, at best, be described as "marking only". It associated an owner and group with each file but, as we'll see, did very little validation of a given client connection. Strong security would manage not only the markings given to a file but also the identities of all connecting users.

Time for action – demonstrating the default security

When we have previously shown listings of files, we have seen user and group names for them. However, we have not really explored what that means. Let's do so.

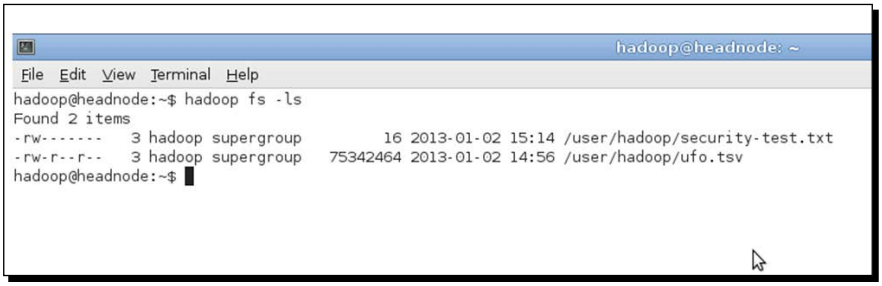
1. Create a test text file in the Hadoop user's home directory.

```
$ echo "I can read this!" >  security-test.txt
$ hadoop fs -put security-test.txt  security-test.txt
```

2. Change the permissions on the file to be accessible only by the owner.

```
$ hadoop fs -chmod 700 security-test.txt
$ hadoop fs -ls
```

The output of the preceding command can be shown in the following screenshot:



3. Confirm you can still read the file.

```
$ hadoop fs -cat security-test.txt
```

You'll see the following line on the screen:

```
I can read this!
```

4. Connect to another node in the cluster and try to read the file from there.

```
$ ssh node2
$ hadoop fs -cat security-test.txt
```

You'll see the following line on the screen:

```
I can read this!
```

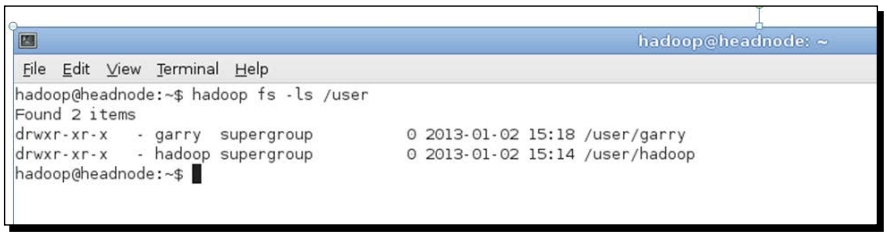
5. Log out from the other node.

```
$ exit
```

6. Create a home directory for another user and give them ownership.

```
$ hadoop m[Kfs -mkdir /user/garry
$ hadoop fs -chown garry /user/garry
$ hadoop fs -ls /user
```

The output of the preceding command can be shown in the following screenshot:



7. Switch to that user.

```
$ su garry
```

8. Try to read the test file in the Hadoop user's home directory.

```
$ hadoop/bin/hadoop fs -cat /user/hadoop/security-test.txt

cat: org.apache.hadoop.security.AccessControlException: Permission
denied: user=garry, access=READ, inode="security-test.txt":hadoop:
supergroup:rw-------
```

9. Place a copy of the file in this user's home directory and again make it accessible only by the owner.

```
$ Hadoop/bin/Hadoop fs -put security-test.txt security-test.txt

$ Hadoop/bin/Hadoop fs -chmod 700 security-test.txt

$ hadoop/bin/hadoop fs -ls
```

The output of the preceding command can be shown in following screenshot:



10. Confirm this user can access the file.

```
$ hadoop/bin/hadoop fs -cat security-test.txt
```

You'll see the following line on the screen:

```
I can read this!
```

11. Return to the Hadoop user.

```
$ exit
```

12. Try and read the file in the other user's home directory.

```
$ hadoop fs -cat /user/garry/security-test.txt
```

You'll see the following line on the screen:

```
I can read this!
```

What just happened?

We firstly used our Hadoop user to create a test file in its home directory on HDFS. We used the `-chmod` option to `hadoop fs`, which we have not seen before. This is very similar to the standard Unix **chmod** tool that gives various levels of read/write/execute access to the file owner, group members, and all users.

We then went to another host and tried to access the file, again as the Hadoop user. Not surprisingly, this worked. But why? What did Hadoop know about the Hadoop user that allowed it to give access to the file?

To explore this, we then created another home directory on HDFS (you can use any other account on the host you have access to), and gave it ownership by using the `-chown` option to `hadoop fs`. This should once again look similar to standard Unix `-chown`. Then we switched to this user and attempted to read the file stored in the Hadoop user's home directory. This failed with the security exception shown before, which is again what we expected. Once again, we copied a test file into this user's home directory and made it only accessible by the owner.

But we then muddied the waters by switching back to the Hadoop user and tried to access the file in the other account's home directory, which, surprisingly, worked.

User identity

The answer to the first part of the puzzle is that Hadoop uses the Unix ID of the user executing the HDFS command as the user identity on HDFS. So any commands executed by a user called `alice` will create files with an owner named `alice` and will only be able to read or write files to which this user has the correct access.

The security-minded will realize that to access a Hadoop cluster all one needs to do is create a user with the same name as an already existing HDFS user on any host that can connect to the cluster. So, for instance, in the previous example, any user named `hadoop` created on any host that can access the NameNode can read all files accessible by the user `hadoop`, which is actually even worse than it seems.

The super user

The previous step saw the Hadoop user access another user's files. Hadoop treats the user ID that started the cluster as the super user, and gives it various privileges, such as the ability to read, write, and modify any file on HDFS. The security-minded will realize even more the risk of having users called `hadoop` randomly created on hosts outside the Hadoop administrator's control.

More granular access control

The preceding situation has caused security to be a major weakness in Hadoop since its inception. The community has, however, not been standing still, and after much work the very latest versions of Hadoop support a more granular and stronger security model.

To avoid reliance on simple user IDs, the developers need to learn the user identity from somewhere, and the Kerberos system was chosen with which to integrate. This does require the establishment and maintenance of services outside the scope of this book, but if such security is important to you, consult the Hadoop documentation. Note that this support does allow integration with third-party identity systems such as Microsoft Active Directory, so it is quite powerful.

Working around the security model via physical access control

If the burden of Kerberos is too great, or security is a nice-to-have rather than an absolute, there are ways of mitigating the risk. One favored by me is to place the entire cluster behind a firewall with tight access control. In particular, only allow access to the NameNode and JobTracker services from a single host that will be treated as the cluster head node and to which all users connect.



Accessing Hadoop from non-cluster hosts

Hadoop does not need to be running on a host for it to use the command-line tools to access HDFS and run MapReduce jobs. As long as Hadoop is installed on the host and its configuration files have the correct locations of the NameNode and JobTracker, these will be found when invoking commands such as `Hadoop fs` and `Hadoop jar`.

This model works because only one host is used to interact with Hadoop; and since this host is controlled by the cluster administrator, normal users should be unable to create or access other user accounts.

Remember that this approach is not providing security. It is putting a hard shell around a soft system that reduces the ways in which the Hadoop security model can be subverted.

Managing the NameNode

Let's do some more risk reduction. In *Chapter 6, When Things Break*, I probably scared you when talking about the potential consequences of a failure of the host running the NameNode. If that section did not scare you, go back and re-read it—it should have. The summary is that the loss of the NameNode could see you losing every single piece of data on the cluster. This is because the NameNode writes a file called `fsimage` that contains all the metadata for the filesystem and records which blocks comprise which files. If the loss of the NameNode host makes the `fsimage` unrecoverable, all the HDFS data is likewise lost.

Configuring multiple locations for the fsimage class

The NameNode can be configured to simultaneously write `fsimage` to multiple locations. This is purely a redundancy mechanism, the same data is written to each location and there is no attempt to use multiple storage devices for increased performance. Instead, the policy is that multiple copies of `fsimage` will be harder to lose.

Time for action – adding an additional fsimage location

Let's now configure our NameNode to simultaneously write multiple copies of `fsimage` to give us our desired data resilience. To do this, we require an NFS-exported directory.

1. Ensure the cluster is stopped.

```
$ stopall.sh
```

2. Add the following property to `Hadoop/conf/core-site.xml`, modifying the second path to point to an NFS-mounted location to which the additional copy of NameNode data can be written.

```
<property>
<name>dfs.name.dir</name>
<value>${hadoop.tmp.dir}/dfs/name,/share/backup/namenode</value>
</property>
```

3. Delete any existing contents of the newly added directory.

```
$ rm -f /share/backup/namenode
```

4. Start the cluster.

```
$ start-all.sh
```

5. Verify that `fsimage` is being written to both the specified locations by running the `md5sum` command against the two files specified before (change the following code depending on your configured locations):

```
$ md5sum /var/hadoop/dfs/name/image/fsimage
a25432981b0ecd6b70da647e9b94304a  /var/hadoop/dfs/name/image/
fsimage
$ md5sum /share/backup/namenode/image/fsimage
a25432981b0ecd6b70da647e9b94304a  /share/backup/namenode/image/
fsimage
```

What just happened?

Firstly, we ensured the cluster was stopped; though changes to the core configuration files are not reread by a running cluster, it's a good habit to get into in case that capability is ever added to Hadoop.

We then added a new property to our cluster configuration, specifying a value for the `data.name.dir` property. This property takes a list of comma-separated values and writes `fsimage` to each of these locations. Note how the `hadoop.tmp.dir` property discussed earlier is de-referenced, as would be seen when using Unix variables. This syntax allows us to base property values on others and inherit changes when the parent properties are updated.



Do not forget all required locations

The default value for this property is `${Hadoop.tmp.dir}/dfs/name`. When adding an additional value, remember to explicitly add the default one also, as shown before. Otherwise, only the single new value will be used for the property.

Before starting the cluster, we ensure the new directory exists and is empty. If the directory doesn't exist, the NameNode will fail to start as should be expected. If, however, the directory was previously used to store NameNode data, Hadoop will also fail to start as it will identify that both directories contain different NameNode data and it does not know which one is correct.

Be careful here! Especially if you are experimenting with various NameNode data locations or swapping back and forth between nodes; you really do not want to accidentally delete the contents from the wrong directory.

After starting the HDFS cluster, we wait for a moment and then use MD5 cryptographic checksums to verify that both locations contain the identical `fsimage`.

Where to write the fsimage copies

The recommendation is to write `fsimage` to at least two locations, one of which should be the remote (such as a NFS) filesystem, as in the previous example. `fsimage` is only updated periodically, so the filesystem does not need high performance.

In our earlier discussion regarding the choice of hardware, we alluded to other considerations for the NameNode host. Because of `fsimage` criticality, it may be useful to ensure it is written to more than one disk and to perhaps invest in disks with higher reliability, or even to write `fsimage` to a RAID array. If the host fails, using the copy written to the remote filesystem will be the easiest option; but just in case that has also experienced problems, it's good to have the choice of pulling another disk from the dead host and using it on another to recover the data.

Swapping to another NameNode host

We have ensured that `fsimage` is written to multiple locations and this is the single most important prerequisite for managing a swap to a different NameNode host. Now we need to actually do it.

This is something you really should not do on a production cluster. Absolutely not when trying for the first time, but even beyond that it's not a risk-free process. But do practice on other clusters and get an idea of what you'll do when disaster strikes.

Having things ready before disaster strikes

You don't want to be exploring this topic for the first time when you need to recover the production cluster. There are several things to do in advance that will make disaster recovery much less painful, not to mention possible:

- Ensure the NameNode is writing the `fsimage` to multiple locations, as done before.
- Decide which host will be the new NameNode location. If this is a host currently being used for a DataNode and TaskTracker, ensure it has the right hardware needed to host the NameNode and that the reduction in cluster performance due to the loss of these workers won't be too great.
- Make a copy of the `core-site.xml` and `hdfs-site.xml` files, place them (ideally) on an NFS location, and update them to point to the new host. Any time you modify the current configuration files, remember to make the same changes to these copies.
- Copy the `slaves` file from the NameNode onto either the new host or the NFS share. Also, make sure you keep it updated.
- Know how you will handle a subsequent failure in the new host. How quickly can you likely repair or replace the original failed host? Which host will be the location of the NameNode (and SecondaryNameNode) in the interim?

Ready? Let's do it!

Time for action – swapping to a new NameNode host

In the following steps we keep the new configuration files on an NFS share mounted to `/share/backup` and change the paths to match where you have the new files. Also use a different string to grep; we use a portion of the IP address we know isn't shared with any other host in the cluster.

1. Log on to the current NameNode host and shut down the cluster.

```
$ stop-all.sh
```

2. Halt the host that runs the NameNode.

```
$ sudo poweroff
```

3. Log on to the new NameNode host and confirm the new configuration files have the correct NameNode location.

```
$ grep 110 /share/backup/*.xml
```

4. On the new host, first copy across the `slaves` file.

```
$ cp /share/backup/slaves Hadoop/conf
```

5. Now copy across the updated configuration files.

```
$ cp /share/backup/*site.xml Hadoop/conf
```

6. Remove any old NameNode data from the local filesystem.

```
$ rm -f /var/Hadoop/dfs/name/*
```

7. Copy the updated configuration files to every node in the cluster.

```
$ slaves.sh cp /share/backup/*site.xml Hadoop/conf
```

8. Ensure each node now has the configuration files pointing to the new NameNode.

```
$ slaves.sh grep 110 hadoop/conf/*site.xml
```

9. Start the cluster.

```
$ start-all.sh
```

10. Check HDFS is healthy, from the command line.

```
$ Hadoop fs ls /
```

11. Verify whether HDFS is accessible from the web UI.

What just happened?

First, we shut down the cluster. This is a little un-representative as most failures see the NameNode die in a much less friendly way, but we do not want to talk about issues of filesystem corruption until later in the chapter.

We then shut down the old NameNode host. Though not strictly necessary, it is a good way of ensuring that nothing accesses the old host and gives you an incorrect view on how well the migration has occurred.

Before copying across files, we take a quick look at `core-site.xml` and `hdfs-site.xml` to ensure the correct values are specified for the `fs.default.dir` property in `core-site.xml`.

We then prepare the new host by firstly copying across the `slaves` configuration file and the cluster configuration files and then removing any old NameNode data from the local directory. Refer to the preceding steps about being very careful in this step.

Next, we use the `slaves.sh` script to get each host in the cluster to copy across the new configuration files. We know our new NameNode host is the only one with 110 in its IP address, so we grep for that in the files to ensure all are up-to-date (obviously, you will need to use a different pattern for your system).

At this stage, all should be well; we start the cluster and access via both the command-line tools and UI to confirm it is running as expected.

Don't celebrate quite yet!

Remember that even with a successful migration to a new NameNode, you aren't done quite yet. You decided in advance how to handle the SecondaryNameNode and which host would be the new designated NameNode host should the newly migrated one fail. To be ready for that, you will need to run through the "Be prepared" checklist mentioned before once more and act appropriately.



Do not forget to consider the chance of correlated failures. Investigate the cause of the NameNode host failure in case it is the start of a bigger problem.

What about MapReduce?

We did not mention moving the JobTracker as that is a much less painful process as shown in *Chapter 6*, *When Things Break*. If your NameNode and JobTracker are running on the same host, you will need to modify the preceding approach by also keeping a new copy of `mapred-site.xml`, which has the location of the new host in the `mapred.job.tracker` property.

Have a go hero – swapping to a new NameNode host

Perform a migration of both the NameNode and JobTracker from one host to another.

Managing HDFS

As we saw when killing and restarting nodes in *Chapter 6*, *When Things Break*, Hadoop automatically manages many of the availability concerns that would consume a lot of effort on a more traditional filesystem. There are some things, however, that we still need to be aware of.

Where to write data

Just as the NameNode can have multiple locations for storage of `fsimage` specified via the `dfs.name.dir` property, we explored earlier that there is a similar-appearing property called `dfs.data.dir` that allows HDFS to use multiple data locations on a host, which we will look at now.

This is a useful mechanism that works very differently from the NameNode property. If multiple directories are specified in `dfs.data.dir`, Hadoop will view these as a series of independent locations that it can use in parallel. This is useful if you have multiple physical disks or other storage devices mounted at distinct points on the filesystem. Hadoop will use these multiple devices intelligently, maximizing not only the total storage capacity but also by balancing reads and writes across the locations to gain maximum throughput. As mentioned in the *Storage types* section, this is the approach that maximizes these factors at the cost of a single disk failure causing the whole host to fail.

Using balancer

Hadoop works hard to place data blocks on HDFS in a way that maximizes both performance and redundancy. However, in certain situations, the cluster can become unbalanced, with a large discrepancy between the data held on the various nodes. The classic situation that causes this is when a new node is added to the cluster. By default, Hadoop will consider the new node as a candidate for block placement alongside all other nodes, meaning that it will remain lightly utilized for a significant period of time. Nodes that have been out of service or have otherwise suffered issues may also have collected a smaller number of blocks than their peers.

Hadoop includes a tool called the balancer, started and stopped by the `start-balancer.sh` and `stop-balancer.sh` scripts respectively, to handle this situation.

When to rebalance

Hadoop does not have any automatic alarms that will alert you to an unbalanced filesystem. Instead, you need to keep an eye on the data reported by both `hadoop fsck` and `hadoop fsadmin` and watch for imbalances across the nodes.

In reality, this is not something you usually need to worry about, as Hadoop is very good at managing block placement and you likely only need to consider running the balancer to remove major imbalances when adding new hardware or when returning faulty nodes to service. To maintain maximum cluster health, however, it is not uncommon to have the balancer run on a scheduled basis (for example, nightly) to keep the block balancing within a specified threshold.

MapReduce management

As we saw in the previous chapter, the MapReduce framework is generally more tolerant of problems and failures than HDFS. The JobTracker and TaskTrackers have no persistent data to manage and, consequently, the management of MapReduce is more about the handling of running jobs and tasks than servicing the framework itself.

Command line job management

The `hadoop job` command-line tool is the primary interface for this job management. As usual, type the following to get a usage summary:

```
$ hadoop job --help
```

The options to the command are generally self-explanatory; it allows you to start, stop, list, and modify running jobs in addition to retrieving some elements of job history. Instead of examining each individually, we will explore the use of several of these subcommands together in the next section.

Have a go hero – command line job management

The MapReduce UI also provides access to a subset of these capabilities. Explore the UI and see what you can and cannot do from the web interface.

Job priorities and scheduling

So far, we have generally run a single job against our cluster and waited for it to complete. This has hidden the fact that, by default, Hadoop places subsequent job submissions into a **First In, First Out** (**FIFO**) queue. When a job finishes, Hadoop simply starts executing the next job in the queue. Unless we use one of the alternative schedulers that we will discuss in later sections, the FIFO scheduler dedicates the full cluster to the sole currently running job.

For small clusters with a pattern of job submission that rarely sees jobs waiting in the queue, this is completely fine. However, if jobs are often waiting in the queue, issues can arise. In particular, the FIFO model takes no account of job priority or resources needed. A long-running but low-priority job will execute before faster high-priority jobs that were submitted later.

To address this situation, Hadoop defines five levels of job priority: `VERY_HIGH`, `HIGH`, `NORMAL`, `LOW`, and `VERY_LOW`. A job defaults to `NORMAL` priority, but this can be changed with the `hadoop job -set-priority` command.

Time for action – changing job priorities and killing a job

Let's explore job priorities by changing them dynamically and watching the result of killing a job.

1. Start a relatively long-running job on the cluster.

   ```
   $ hadoop jar hadoop-examples-1.0.4.jar pi 100 1000
   ```

2. Open another window and submit a second job.

   ```
   $ hadoop jar hadoop-examples-1.0.4.jar wordcount test.txt out1
   ```

3. Open another window and submit a third.

   ```
   $ hadoop jar hadoop-examples-1.0.4.jar wordcount test.txt out2
   ```

4. List the running jobs.

   ```
   $ Hadoop job -list
   ```

   You'll see the following lines on the screen:

   ```
   3 jobs currently running
   JobId  State  StartTime  UserName  Priority  SchedulingInfo
   job_201201111540_0005  1  1326325810671  hadoop  NORMAL  NA
   job_201201111540_0006  1  1326325938781  hadoop  NORMAL  NA
   job_201201111540_0007  1  1326325961700  hadoop  NORMAL  NA
   ```

5. Check the status of the running job.

   ```
   $ Hadoop job -status job_201201111540_0005
   ```

   You'll see the following lines on the screen:

   ```
   Job: job_201201111540_0005
   file: hdfs://head:9000/var/hadoop/mapred/system/
   job_201201111540_0005/job.xml
   tracking URL: http://head:50030/jobdetails.
   jsp?jobid=job_201201111540_000
   map() completion: 1.0
   reduce() completion: 0.32666665
   Counters: 18
   ```

6. Raise the priority of the last submitted job to `VERY_HIGH`.
   ```
   $ Hadoop job -set-priority job_201201111540_0007 VERY_HIGH
   ```
7. Kill the currently running job.
   ```
   $ Hadoop job -kill job_201201111540_0005
   ```
8. Watch the other jobs to see which begins processing.

What just happened?

We started a job on the cluster and then queued up another two jobs, confirming that the queued jobs were in the expected order by using `hadoop job -list`. The `hadoop job -list all` command would have listed both completed as well as the current jobs and `hadoop job -history` would have allowed us to examine the jobs and their tasks in much more detail. To confirm the submitted job was running, we used `hadoop job -status` to get the current map and reduce task completion status for the job, in addition to the job counters.

We then used `hadoop job -set-priority` to increase the priority of the job currently last in the queue.

After using `hadoop job -kill` to abort the currently running job, we confirmed the job with the increased priority that executed next, even though the job remaining in the queue was submitted beforehand.

Alternative schedulers

Manually modifying job priorities in the FIFO queue certainly does work, but it requires active monitoring and management of the job queue. If we think about the problem, the reason we are having this difficulty is the fact that Hadoop dedicates the entire cluster to each job being executed.

Hadoop offers two additional job schedulers that take a different approach and share the cluster among multiple concurrently executing jobs. There is also a plugin mechanism by which additional schedulers can be added. Note that this type of resource sharing is one of those problems that is conceptually simple but is in reality very complex and is an area of much academic research. The goal is to maximize resource allocation not only at a point in time, but also over an extended period while honoring notions of relative priority.

Capacity Scheduler

The **Capacity Scheduler** uses multiple job queues (to which access control can be applied) to which jobs are submitted, each of which is allocated a portion of the cluster resources. You could, for example, have a queue for large long-running jobs that is allocated 90 percent of the cluster and one for smaller high-priority jobs allocated the remaining 10 percent. If both queues have jobs submitted, the cluster resources will be allocated in this proportion.

If, however, one queue is empty and the other has jobs to execute, the Capacity Scheduler will temporarily allocate the capacity of the empty queue to the busy one. Once a job is submitted to the empty queue, it will regain its capacity as the currently running tasks complete execution. This approach gives a reasonable balance between the desired resource allocation and preventing long periods of unused capacity.

Though disabled by default, the Capacity Scheduler supports job priorities within each queue. If a high priority job is submitted after a low priority one, its tasks will be scheduled in preference to the other jobs as capacity becomes available.

Fair Scheduler

The **Fair Scheduler** segments the cluster into pools into which jobs are submitted; there is often a correlation between the user and the pool. Though by default each pool gets an equal share of the cluster, this can be modified.

Within each pool, the default model is to share the pool across all jobs submitted to that pool. Therefore, if the cluster is split into pools for Alice and Bob, each of whom submit three jobs, the cluster will execute all six jobs in parallel. It is possible to place total limits on the number of concurrent jobs running in a pool, as too many running at once will potentially produce a large amount of temporary data and provide overall inefficient processing.

As with the Capacity Scheduler, the Fair Scheduler will over-allocate cluster capacity to other pools if one is empty, and then reclaim it as the pool receives jobs. It also supports job priorities within a pool to preferentially schedule tasks of high priority jobs over those with a lower priority.

Enabling alternative schedulers

Each of the alternative schedulers is provided as a JAR file in `capacityScheduler` and `fairScheduler` directories within the `contrib` directory in the Hadoop installation. To enable a scheduler, either add its JAR to the `hadoop/lib` directory or explicitly place it on the classpath. Note that each scheduler requires its own set of properties to configure its usage. Refer to the documentation for each for more details.

When to use alternative schedulers

The alternative schedulers are very effective, but are not really needed on small clusters or those with no need to ensure multiple job concurrency or execution of late-arriving but high-priority jobs. Each has multiple configuration parameters and requires tuning to get optimal cluster utilization. But for any large cluster with multiple users and varying job priorities, they can be essential.

Scaling

You have data and you have a running Hadoop cluster; now you get more of the former and need more of the latter. We have said repeatedly that Hadoop is an easily scalable system. So let us add some new capacity.

Adding capacity to a local Hadoop cluster

Hopefully, at this point, you should feel pretty underwhelmed at the idea of adding another node to a running cluster. All through *Chapter 6*, *When Things Break*, we constantly killed and restarted nodes. Adding a new node is really no different; all you need to do is perform the following steps:

1. Install Hadoop on the host.
2. Set the environment variables shown in *Chapter 2*, *Getting Up and Running*.
3. Copy the configuration files into the `conf` directory on the installation.
4. Add the host's DNS name or IP address to the `slaves` file on the node from which you usually run commands such as `slaves.sh` or cluster start/stop scripts.

And that's it!

Have a go hero – adding a node and running balancer

Try out the process of adding a new node and afterwards examine the state of HDFS. If it is unbalanced, run the balancer to fix things. To help maximize the effect, ensure there is a reasonable amount of data on HDFS before adding the new node.

Adding capacity to an EMR job flow

If you are using Elastic MapReduce, for non-persistent clusters, the concept of scaling does not always apply. Since you specify the number and type of hosts required when setting up the job flow each time, you need only ensure that the cluster size is appropriate for the job to be executed.

Expanding a running job flow

However, sometimes you may have a long-running job that you want to complete more quickly. In such a case, you can add more nodes to the running job flow. Recall that EMR has three different types of node: master nodes for NameNode and JobTracker, core nodes for HDFS, and task nodes for MapReduce workers. In this case, you could add additional task nodes to help crunch the MapReduce job.

Another scenario is where you have defined a job flow comprising a series of MapReduce jobs instead of just one. EMR now allows the job flow to be modified between steps in such a series. This has the advantage of each job being given a tailored hardware configuration that gives better control of balancing performance against cost.

The canonical model for EMR is for the job flow to pull its source data from S3, process that data on a temporary EMR Hadoop cluster, and then write results back to S3. If, however, you have a very large data set that requires frequent processing, the copying back and forth of data could become too time-consuming. Another model that can be employed in such a situation is to use a persistent Hadoop cluster within a job flow that has been sized with enough core nodes to store the needed data on HDFS. When processing is performed, increase capacity as shown before by assigning more task nodes to the job flow.



These tasks to resize running job flows are not currently available from the AWS Console and need to be performed through the API or command line tools.

Summary

This chapter covered how to build, maintain, and expand a Hadoop cluster. In particular, we learned where to find the default values for Hadoop configuration properties and how to set them programmatically on a per-job level. We learned how to choose hardware for a cluster and the value in understanding your likely workload before committing to purchases, and how Hadoop can use awareness of the physical location of hosts to optimize its block placement strategy through the use of rack awareness.

We then saw how the default Hadoop security model works, its weaknesses and how to mitigate them, how to mitigate the risks of NameNode failure we introduced in *Chapter 6*, *When Things Break*, and how to swap to a new NameNode host if disaster strikes. We learned more about block replica placement, how the cluster can become unbalanced, and what to do if it does.

We also saw the Hadoop model for MapReduce job scheduling and learned how job priorities can modify the behavior, how the Capacity Scheduler and Fair Scheduler give a more sophisticated way of managing cluster resources across multiple concurrent job submissions, and how to expand a cluster with a new capacity.

This completes our exploration of core Hadoop in this book. In the remaining chapters, we will look at other systems and tools that build atop Hadoop to provide more sophisticated views on data and integration with other systems. We will start with a relational view on the data in HDFS through the use of Hive.

8
A Relational View on Data with Hive

MapReduce is a powerful paradigm which enables complex data processing that can reveal valuable insights. However, it does require a different mindset and some training and experience on the model of breaking processing analytics into a series of map and reduce steps. There are several products that are built atop Hadoop to provide higher-level or more familiar views on the data held within HDFS. This chapter will introduce one of the most popular of these tools, **Hive***.*

In this chapter, we will cover:

- What Hive is and why you may want to use it
- How to install and configure Hive
- Using Hive to perform SQL-like analysis of the UFO data set
- How Hive can approximate common features of a relational database such as joins and views
- How to efficiently use Hive across very large data sets
- How Hive allows the incorporation of user-defined functions into its queries
- How Hive complements another common tool, Pig

Overview of Hive

Hive is a data warehouse that uses MapReduce to analyze data stored on HDFS. In particular, it provides a query language called **HiveQL** that closely resembles the common **Structured Query Language** (**SQL**) standard.

Why use Hive?

In *Chapter 4*, *Developing MapReduce Programs*, we introduced Hadoop Streaming and explained that one large benefit of Streaming is how it allows faster turn-around in the development of MapReduce jobs. Hive takes this a step further. Instead of providing a way of more quickly developing map and reduce tasks, it offers a query language based on the industry standard SQL. Hive takes these HiveQL statements and immediately and automatically translates the queries into one or more MapReduce jobs. It then executes the overall MapReduce program and returns the results to the user. Whereas Hadoop Streaming reduces the required code/compile/submit cycle, Hive removes it entirely and instead only requires the composition of HiveQL statements.

This interface to Hadoop not only accelerates the time required to produce results from data analysis, it significantly broadens who can use Hadoop and MapReduce. Instead of requiring software development skills, anyone with a familiarity with SQL can use Hive.

The combination of these attributes is that Hive is often used as a tool for business and data analysts to perform ad hoc queries on the data stored on HDFS. Direct use of MapReduce requires map and reduce tasks to be written before the job can be executed which means a necessary delay from the idea of a possible query to its execution. With Hive, the data analyst can work on refining HiveQL queries without the ongoing involvement of a software developer. There are of course operational and practical limitations (a badly written query will be inefficient regardless of technology) but the broad principle is compelling.

Thanks, Facebook!

Just as we earlier thanked Google, Yahoo!, and Doug Cutting for their contributions to Hadoop and the technologies that inspired it, it is to Facebook that we must now direct thanks.

Hive was developed by the Facebook Data team and, after being used internally, it was contributed to the Apache Software Foundation and made freely available as open source software. Its homepage is `http://hive.apache.org`.

Setting up Hive

In this section, we will walk through the act of downloading, installing, and configuring Hive.

Prerequisites

Unlike Hadoop, there are no Hive masters, slaves, or nodes. Hive runs as a client application that processes HiveQL queries, converts them into MapReduce jobs, and submits these to a Hadoop cluster.

Although there is a mode suitable for small jobs and development usage, the usual situation is that Hive will require an existing functioning Hadoop cluster.

Just as other Hadoop clients don't need to be executed on the actual cluster nodes, Hive can be executed on any host where the following are true:

- Hadoop is installed on the host (even if no processes are running)
- The `HADOOP_HOME` environment variable is set and points to the location of the Hadoop installation
- The `${HADOOP_HOME}/bin` directory is added to the system or user path

Getting Hive

You should download the latest stable Hive version from `http://hive.apache.org/releases.html`.

The Hive getting started guide at `http://cwiki.apache.org/confluence/display/Hive/GettingStarted` will give recommendations on version compatibility, but as a general principle, you should expect the most recent stable versions of Hive, Hadoop, and Java to work together.

Time for action – installing Hive

Let's now set up Hive so we can start using it in action.

1. Download the latest stable version of Hive and move it to the location to which you wish to have it installed:

```
$ mv hive-0.8.1.tar.gz /usr/local
```

2. Uncompress the package:

```
$ tar –xzf hive-0.8.1.tar.gz
```

3. Set the `HIVE_HOME` variable to the installation directory:

```
$ export HIVE_HOME=/usr/local/hive
```

4. Add the Hive home directory to the path variable:

```
$ export PATH=${HIVE_HOME}/bin:${PATH}
```

5. Create directories required by Hive on HDFS:

```
$ hadoop fs -mkdir /tmp
$ hadoop fs -mkdir /user/hive/warehouse
```

6. Make both of these directories group writeable:

```
$ hadoop fs -chmod g+w /tmp
$ hadoop fs -chmod g+w /user/hive/warehouse
```

7. Try to start Hive:

```
$ hive
```

You will receive the following response:

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties


Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203031500_480385673.txt
hive>
```

8. Exit the Hive interactive shell:

```
$ hive> quit;
```

What just happened?

After downloading the latest stable Hive release, we copied it to the desired location and uncompressed the archive file. This created a directory, `hive-<version>`.

Similarly, as we previously defined `HADOOP_HOME` and added the `bin` directory within the installation to the path variable, we then did something similar with HIVE_HOME and its bin directory.



Remember that to avoid having to set these variables every time you log in, add them to your shell login script or to a separate configuration script that you source when you want to use Hive.

We then created two directories on HDFS that Hive requires and changed their attributes to make them group writeable. The `/tmp` directory is where Hive will, by default, write transient data created during query execution and will also place output data in this location. The `/user/hive/warehouse` directory is where Hive will store the data that is written into its tables.

After all this setup, we run the `hive` command and a successful installation will give output similar to the one mentioned above. Running the `hive` command with no arguments enters an interactive shell; the `hive>` prompt is analogous to the `sql>` or `mysql>` prompts familiar from relational database interactive tools.

We then exit the interactive shell by typing `quit;`. Note the trailing semicolon `;`. HiveQL is, as mentioned, very similar to SQL and follows the convention that all commands must be terminated by a semicolon. Pressing *Enter* without a semicolon will allow commands to be continued on subsequent lines.

Using Hive

With our Hive installation, we will now import and analyze the UFO data set introduced in *Chapter 4*, *Developing MapReduce Programs*.

When importing any new data into Hive, there is generally a three-stage process:

1. Create the specification of the table into which the data is to be imported.
2. Import the data into the created table.
3. Execute HiveQL queries against the table.

This process should look very familiar to those with experience with relational databases. Hive gives a structured query view of our data and to enable that, we must first define the specification of the table's columns and import the data into the table before we can execute any queries.



We assume a general level of familiarity with SQL and will be focusing more on how to get things done with Hive than in explaining particular SQL constructs in detail. A SQL reference may be handy for those with little familiarity with the language, though we will make sure you know what each statement does, even if the details require deeper SQL knowledge.

Time for action – creating a table for the UFO data

Perform the following steps to create a table for the UFO data:

1. Start the Hive interactive shell:

```
$ hive
```

2. Create a table for the UFO data set, splitting the statement across multiple lines for easy readability:

```
hive> CREATE TABLE ufodata(sighted STRING, reported STRING,
sighting_location STRING,    > shape STRING, duration STRING,
description STRING COMMENT 'Free text description')
COMMENT 'The UFO data set.' ;
```

You should see the following lines once you are done:

```
OK
Time taken: 0.238 seconds
```

3. List all existing tables:

```
hive> show tables;
```

You will receive the following output:

```
OK
ufodata
Time taken: 0.156 seconds
```

4. Show tables matching a regular expression:

```
hive> show tables '.*data';
```

You will receive the following output:

```
OK
ufodata
Time taken: 0.065 seconds
```

5. Validate the table specification:

```
hive> describe ufodata;
```

You will receive the following output:

```
OK
sighted  string
reported  string
sighting_location  string
shape  string
duration  string
description  string  Free text description
Time taken: 0.086 seconds
```

6. Display a more detailed description of the table:

```
hive> describe extended ufodata;
```

You will receive the following output:

```
OK
```

```
sighted  string
reported  string
…
Detailed Table Information  Table(tableName:ufodata,
dbName:default, owner:hadoop, createTime:1330818664,
lastAccessTime:0, retention:0,
…
…location:hdfs://head:9000/user/hive/warehouse/
ufodata, inputFormat:org.apache.hadoop.mapred.
TextInputFormat, outputFormat:org.apache.hadoop.hive.ql.io.
HiveIgnoreKeyTextOutputFormat, compressed:false, numBuckets:-1,
```

What just happened?

After starting the interactive Hive interpreter, we used the `CREATE TABLE` command to define the structure of the UFO data table. As with standard SQL, this requires that each column in the table has a name and datatype. HiveQL also offers optional comments on each column and on the overall table, as shown previously where we add one column and one table comment.

For the UFO data, we use `STRING` as the data type; HiveQL, as with SQL, supports a variety of datatypes:

- **Boolean types**: `BOOLEAN`
- **Integer types**: `TINYINT`, `INT`, `BIGINT`
- **Floating point types**: `FLOAT`, `DOUBLE`
- **Textual types**: `STRING`

After creating the table, we use the `SHOW TABLES` statement to verify that the table has been created. This command lists all tables and in this case, our new UFO table is the only one in the system.

We then use a variant on `SHOW TABLES` that takes an optional Java regular expression to match against the table name. In this case, the output is identical to the previous command, but in systems with a large number of tables—especially when you do not know the exact name—this variant can be very useful.



We have seen the table exists but we have not validated whether it was created properly. We next do this by using the `DESCRIBE TABLE` command to display the specification of the named table. We see that all is as specified (though note the table comment is not shown by this command) and then use the `DESCRIBE TABLE EXTENDED` variant to get much more information about the table.

We have omitted much of this final output though a few points of interest are present. Note the input format is specified as `TextInputFormat`; by default, Hive will assume any HDFS files inserted into a table are stored as text files.

We also see that the table data will be stored in a directory under the `/user/hive/warehouse` HDFS directory we created earlier.



A note on case sensitivity:

HiveQL, as with SQL, is not case sensitive in terms of keywords, columns, or table names. By convention, SQL statements use uppercase for SQL language keywords and we will generally follow this when using HiveQL within files, as shown later. However, when typing interactive commands, we will frequently take the line of least resistance and use lowercase.

Time for action – inserting the UFO data

Now that we have created a table, let us load the UFO data into it.

1. Copy the UFO data file onto HDFS:

```
$ hadoop fs -put ufo.tsv /tmp/ufo.tsv
```

2. Confirm that the file was copied:

```
$ hadoop fs -ls /tmp
```

   You will receive the following response:

```
Found 2 items
drwxrwxr-x   - hadoop supergroup          0 … 14:52 /tmp/hive-hadoop
-rw-r--r--   3 hadoop supergroup   75342464 2012-03-03 16:01 /tmp/ufo.tsv
```

3. Enter the Hive interactive shell:

```
$ hive
```

4. Load the data from the previously copied file into the `ufodata` table:

```
hive> LOAD DATA INPATH '/tmp/ufo.tsv' OVERWRITE INTO TABLE ufodata;
```

   You will receive the following response:

```
Loading data to table default.ufodata
Deleted hdfs://head:9000/user/hive/warehouse/ufodata
```

```
OK
Time taken: 5.494 seconds
```

5. Exit the Hive shell:

```
hive> quit;
```

6. Check the location from which we copied the data file:

```
$ hadoop fs -ls /tmp
```

You will receive the following response:

```
Found 1 items
drwxrwxr-x   - hadoop supergroup          0 … 16:10 /tmp/hive-
hadoop
```

What just happened?

We first copied onto HDFS the tab-separated file of UFO sightings used previously in *Chapter 4*, *Developing MapReduce Programs*. After validating the file's presence on HDFS, we started the Hive interactive shell and used the `LOAD DATA` command to load the file into the `ufodata` table.

Because we are using a file already on HDFS, the path was specified by `INPATH` alone. We could have loaded directly from a file on the local filesystem (obviating the need for the prior explicit HDFS copy) by using `LOCAL INPATH`.

We specified the `OVERWRITE` statement which will delete any existing data in the table before loading the new data. This obviously should be used with care, as can be seen from the output of the command, the directory holding the table data is removed by use of `OVERWRITE`.

Note the command took only a little over five seconds to execute, significantly longer than it would have taken to copy the UFO data file onto HDFS.



Though we specified an explicit file in this example, it is possible to load multiple files with a single statement by specifying a directory as the `INPATH` location; in such a case, all files within the directory will be loaded into the table.

After exiting the Hive shell, we look again at the directory into which we copied the data file and find it is no longer there. If a `LOAD` statement is given a path to data on HDFS, it will not simply copy this into `/user/hive/datawarehouse`, but will move it there instead. If you want to analyze data on HDFS that is used by other applications, then either create a copy or use the `EXTERNAL` mechanism that will be described later.

Validating the data

Now that we have loaded the data into our table, it is good practice to do some quick validating queries to confirm all is as expected. Sometimes our initial table definition turns out to be incorrect.

Time for action – validating the table

The easiest way to do some initial validation is to perform some summary queries to validate the import. This is similar to the types of activities for which we used Hadoop Streaming in *Chapter 4*, *Developing MapReduce Programs*.

1. Instead of using the Hive shell, pass the following HiveQL to the `hive` command-line tool to count the number of entries in the table:

```
$ hive -e "select count(*) from ufodata;"
```

   You will receive the following response:

```
Total MapReduce jobs = 1
Launching Job 1 out of 1
…
Hadoop job information for Stage-1: number of mappers: 1; number
of reducers: 1
2012-03-03 16:15:15,510 Stage-1 map = 0%,  reduce = 0%
2012-03-03 16:15:21,552 Stage-1 map = 100%,  reduce = 0%
2012-03-03 16:15:30,622 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201202281524_0006
MapReduce Jobs Launched:
Job 0: Map: 1  Reduce: 1   HDFS Read: 75416209 HDFS Write: 6
SUCESS
Total MapReduce CPU Time Spent: 0 msec
OK
61393
Time taken: 28.218 seconds
```

2. Display a sample of five values for the sighted column:

```
$ hive -e  "select sighted from ufodata limit 5;"
```

   You will receive the following response:

```
Total MapReduce jobs = 1
Launching Job 1 out of 1
```

```
…
OK
19951009  19951009   Iowa City, IA      Man repts. witnessing
&quot;flash, followed by a classic UFO, w/ a tailfin at
back.&quot; Red color on top half of tailfin. Became triangular.
19951010  19951011   Milwaukee, WI    2 min.  Man  on Hwy 43 SW
of Milwaukee sees large, bright blue light streak by his car,
descend, turn, cross road ahead, strobe. Bizarre!
19950101  19950103   Shelton, WA      Telephoned Report:CA
woman visiting daughter witness discs and triangular ships over
Squaxin Island in Puget Sound. Dramatic.  Written report, with
illustrations, submitted to NUFORC.
19950510  19950510   Columbia, MO    2 min.  Man repts. son&apos;s
bizarre sighting of small humanoid creature in back yard.  Reptd.
in Acteon Journal, St. Louis UFO newsletter.
19950611  19950614   Seattle, WA      Anonymous caller repts.
sighting 4 ufo&apos;s in NNE sky, 45 deg. above horizon.  (No
other facts reptd.  No return tel. #.)
Time taken: 11.693 seconds
```

What just happened?

In this example, we use the `hive -e` command to directly pass HiveQL to the Hive tool instead of using the interactive shell. The interactive shell is useful when performing a series of Hive operations. For simple statements, it is often more convenient to use this approach and pass the query string directly to the command-line tool. This also shows that Hive can be called from scripts like any other Unix tool.



When using `hive –e`, it is not necessary to terminate the HiveQL string with a semicolon, but if you are like me, the habit is hard to break. If you want multiple commands in a single string, they must obviously be separated by semicolons.

The result of the first query is 61393, the same number of records we saw when analyzing the UFO data set previously with direct MapReduce. This tells us the entire data set was indeed loaded into the table.

We then execute a second query to select five values of the first column in the table, which should return a list of five dates. However, the output instead includes the entire record which has been loaded into the first column.

The issue is that though we relied on Hive loading our data file as a text file, we didn't take into account the separator between columns. Our file is tab separated, but Hive, by default, expects its input files to have fields separated by the ASCII code 00 (control-A).

Time for action – redefining the table with the correct column separator

Let's fix our table specification as follows:

1. Create the following file as `commands.hql`:

```
DROP TABLE ufodata ;
CREATE TABLE ufodata(sighted string, reported string, sighting_
location string,
shape string, duration string, description string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' ;
LOAD DATA INPATH '/tmp/ufo.tsv' OVERWRITE INTO TABLE ufodata ;
```

2. Copy the data file onto HDFS:

```
$ hadoop fs -put ufo.tsv /tmp/ufo.tsv
```

3. Execute the HiveQL script:

```
$ hive -f commands.hql
```

   You will receive the following response:

```
OK
Time taken: 5.821 seconds
OK
Time taken: 0.248 seconds
Loading data to table default.ufodata
Deleted hdfs://head:9000/user/hive/warehouse/ufodata
OK
Time taken: 0.285 seconds
```

4. Validate the number of rows in the table:

```
$ hive -e "select count(*) from ufodata;"
```

   You will receive the following response:

```
OK
61393
Time taken: 28.077 seconds
```

5. Validate the contents of the reported column:

```
$ hive -e "select reported from ufodata limit 5"
```

You will receive the following response:

```
OK
19951009
19951011
19950103
19950510
19950614
Time taken: 14.852 seconds
```

What just happened?

We introduced a third way to invoke HiveQL commands in this example. In addition to using the interactive shell or passing query strings to the Hive tool, we can have Hive read and execute the contents of a file containing a series of HiveQL statements.

We first created such a file that deletes the old table, creates a new one, and loads the data file into it.

The main differences with the table specification are the `ROW FORMAT` and `FIELDS TERMINATED BY` statements. We need both these commands as the first tells Hive that the row contains multiple delimited fields, while the second specifies the actual separator. As can be seen here, we can use both explicit ASCII codes as well as common tokens such as `\t` for tab.



Be careful with the separator specification as it must be precise and is case sensitive. Do not waste a few hours by accidentally writing `\T` instead of `\t` as I did recently.

Before running the script, we copy the data file onto HDFS again—the previous copy was removed by the `DELETE` statement—and then use `hive -f` to execute the HiveQL file.

As before, we then execute two simple `SELECT` statements to first count the rows in the table and then extract the specific values from a named column for a small number of rows.

The overall row count is, as should be expected, the same as before, but the second statement now produces what looks like correct data, showing that the rows are now correctly being split into their constituent fields.

Hive tables – real or not?

If you look closely at the time taken by the various commands in the preceding example, you'll see a pattern which may at first seem strange. Loading data into a table takes about as long as creating the table specification, but even the simple count of all row statements takes significantly longer. The output also shows that table creation and the loading of data do not actually cause MapReduce jobs to be executed, which explains the very short execution times.

When loading data into a Hive table, the process is different from what may be expected with a traditional relational database. Although Hive copies the data file into its working directory, it does not actually process the input data into rows at that point. What it does instead is create metadata around the data which is then used by subsequent HiveQL queries.

Both the `CREATE TABLE` and `LOAD DATA` statements, therefore, do not truly create concrete table data as such, instead they produce the metadata that will be used when Hive is generating MapReduce jobs to access the data conceptually stored in the table.

Time for action – creating a table from an existing file

So far we have loaded data into Hive directly from files over which Hive effectively takes control. It is also possible, however, to create tables that model data held in files external to Hive. This can be useful when we want the ability to perform Hive processing over data written and managed by external applications or otherwise required to be held in directories outside the Hive warehouse directory. Such files are not moved into the Hive warehouse directory or deleted when the table is dropped.

1. Save the following to a file called `states.hql`:

```
CREATE EXTERNAL TABLE states(abbreviation string, full_name
string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LOCATION '/tmp/states' ;
```

2. Copy the data file onto HDFS and confirm its presence afterwards:

```
$ hadoop fs -put states.txt /tmp/states/states.txt
$ hadoop fs -ls /tmp/states
```

You will receive the following response:

```
Found 1 items
-rw-r--r--   3 hadoop supergroup        654 2012-03-03 16:54 /tmp/
states/states.txt
```

3. Execute the HiveQL script:

```
$ hive -f states.hql
```

You will receive the following response:

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203031655_1132553792.txt
OK
Time taken: 3.954 seconds
OK
Time taken: 0.594 seconds
```

4. Check the source data file:

```
$ hadoop fs -ls /tmp/states
```

You will receive the following response:

```
Found 1 items
-rw-r--r--   3 hadoop supergroup        654 … /tmp/states/states.
txt
```

5. Execute a sample query against the table:

```
$ hive -e "select full_name from states where abbreviation like
'CA'"
```

You will receive the following response:

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203031655_410945775.txt
Total MapReduce jobs = 1
...
OK
California
Time taken: 15.75 seconds
```

What just happened?

The HiveQL statement to create an external table only differs slightly from the forms of `CREATE TABLE` we used previously. The `EXTERNAL` keyword specifies that the table exists in resources that Hive does not control and the `LOCATION` clause specifies where the source file or directory are to be found.

After creating the HiveQL script, we copied the source file onto HDFS. For this table, we used the data file from *Chapter 4*, *Developing MapReduce Programs*, which maps U.S. states to their common two-letter abbreviation.

After confirming the file was in the expected location on HDFS, we executed the query to create the table and checked the source file again. Unlike previous table creations that moved the source file into the `/user/hive/warehouse` directory, the `states.txt` file is still in the HDFS location into which it was copied.

Finally, we executed a query against the table to confirm it was populated with the source data and the expected result confirms this. This highlights an additional difference with this form of `CREATE TABLE`; for our previous non-external tables, the table creation statement does not ingest any data into the table, a subsequent `LOAD DATA` or (as we'll see later) `INSERT` statement performs the actual table population. With table definitions that include the `LOCATION` specification, we can create the table and ingest data in a single statement.

We now have two tables in Hive; the larger table with UFO sighting data and a smaller one mapping U.S. state abbreviations to their full names. Wouldn't it be a useful combination to use data from the second table to enrich the location column in the former?

Time for action – performing a join

Joins are a very frequently used tool in SQL, though sometimes appear a little intimidating to those new to the language. Essentially a **join** allows rows in multiple tables to be logically combined together based on a conditional statement. Hive has rich support for joins which we will now examine.

1. Create the following as `join.hql`:

```
SELECT t1.sighted, t2.full_name
FROM ufodata t1 JOIN states t2
ON (LOWER(t2.abbreviation) = LOWER(SUBSTR( t1.sighting_location,
(LENGTH(t1.sighting_location)-1))))
LIMIT 5 ;
```

2. Execute the query:

```
$ hive -f join.hql
```

You will receive the following response:

```
OK
20060930  Alaska
20051018  Alaska
20050707  Alaska
20100112  Alaska
20100625  Alaska
Time taken: 33.255 seconds
```

What just happened?

The actual `join` query is relatively straightforward; we want to extract the sighted date and location for a series of records but instead of the raw location field, we wish to map this into the full state name. The HiveQL file we created performs such a query. The join itself is specified by the standard `JOIN` keyword and the matching condition is contained in the `ON` clause.

Things are complicated by a restriction of Hive in that it only supports equijoins, that is, those where the ON clause contains an equality check. It is not possible to specify a join condition using operators such as `>`, `?`, `<`, or as we would have preferred to use here, the `LIKE` keyword.

Instead, therefore, we have an opportunity to introduce several of Hive's built-in functions, in particular, those to convert a string to lowercase (`LOWER`), to extract a substring from a string (`SUBSTR`) and to return the number of characters in a string (`LENGTH`).

We know that most location entries are of the form "city, state_abbreviation." So we use `SUBSTR` to extract the third and second from last characters in the string, using `length` to calculate the indices. We convert both the state abbreviation and extracted string to lower case via `LOWER` because we cannot assume that all entries in the sighting table will correctly use uniform capitalization.

After executing the script, we get the expected sample lines of output that indeed include the sighting date and full state name instead of the abbreviation.

Note the use of the `LIMIT` clause that simply constrains how many output rows will be returned from the query. This is also an indication that HiveQL is most similar to SQL dialects such as those found in open source databases such as MySQL.

This example shows an inner join; Hive also supports left and right outer joins as well as left semi joins. There are a number of subtleties around the use of joins in Hive (such as the aforementioned equijoin restriction) and you should really read through the documentation on the Hive homepage if you are likely to use joins, especially when using very large tables.



This is not a criticism of Hive alone; joins are incredibly powerful tools but it is probably fair to say that badly written joins or those created in ignorance of critical constraints have brought more relational databases to a grinding halt than any other type of SQL query.

Have a go hero – improve the join to use regular expressions

As well as the string functions we used previously, Hive also has functions such as `RLIKE` and `REGEXP_EXTRACT` that provide direct support for Java-like regular expression manipulation. Rewrite the preceding join specification using regular expressions to make a more accurate and elegant join statement.

Hive and SQL views

Another powerful SQL feature supported by Hive is views. These are useful when instead of a static table the contents of a logical table are specified by a `SELECT` statement and subsequent queries can then be executed against this dynamic view (hence the name) of the underlying data.

Time for action – using views

We can use views to hide the underlying query complexity such as the previous join example. Let us now create a view to do just that.

1. Create the following as `view.hql`:

```
CREATE VIEW IF NOT EXISTS usa_sightings (sighted, reported,
shape, state)
AS select t1.sighted, t1.reported, t1.shape, t2.full_name
FROM ufodata t1 JOIN states t2
ON (LOWER(t2.abbreviation) = LOWER(substr( t1.sighting_location,
(LENGTH(t1.sighting_location)-1)))) ;
```

2. Execute the script:

```
$ hive -f view.hql
```

You will receive the following response:

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203040557_1017700649.txt
OK
Time taken: 5.135 seconds
```

3. Execute the script again:

```
$ hive -f view.hql
```

You will receive the following response:

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203040557_851275946.txt
OK
Time taken: 4.828 seconds
```

4. Execute a test query against the view:

```
$ hive -e "select count(state) from usa_sightings where state =
'California'"
```

You will receive the following response:

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203040558_1729315866.txt
Total MapReduce jobs = 2
Launching Job 1 out of 2
…
2012-03-04 05:58:12,991 Stage-1 map = 0%,  reduce = 0%
2012-03-04 05:58:16,021 Stage-1 map = 50%,  reduce = 0%
2012-03-04 05:58:18,046 Stage-1 map = 100%,  reduce = 0%
2012-03-04 05:58:24,092 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201203040432_0027
Launching Job 2 out of 2
…
2012-03-04 05:58:33,650 Stage-2 map = 0%,  reduce = 0%
2012-03-04 05:58:36,673 Stage-2 map = 100%,  reduce = 0%
2012-03-04 05:58:45,730 Stage-2 map = 100%,  reduce = 100%
Ended Job = job_201203040432_0028
MapReduce Jobs Launched:
Job 0: Map: 2  Reduce: 1   HDFS Read: 75416863 HDFS Write: 116
SUCESS
Job 1: Map: 1  Reduce: 1   HDFS Read: 304 HDFS Write: 5 SUCESS
Total MapReduce CPU Time Spent: 0 msec.
```

```
OK
7599
Time taken: 47.03 seconds
```

5. Delete the view:

```
$ hive -e "drop view usa_sightings"
```

You will receive the following output on your screen:

```
OK
Time taken: 5.298 seconds
```

What just happened?

We firstly created the view using the `CREATE VIEW` statement. This is similar to `CREATE TABLE` but has two main differences:

- The column definitions include only the name as the type, which will be determined from the underlying query
- The `AS` clause specifies the `SELECT` statement that will be used to generate the view

We use the previous join statement to generate the view, so in effect we are creating a table that has the location field normalized to the full state name without directly requiring the user to deal with how that normalization is performed.

The optional `IF NOT EXISTS` clause (which can also be used with `CREATE TABLE`) means that Hive will ignore duplicate attempts to create the view. Without this clause, repeated attempts to create the view will generate errors, which isn't always the desired behavior.

We then execute this script twice to both create the view and to demonstrate that the inclusion of the `IF NOT EXISTS` clause is preventing errors as we intended.

With the view created, we then execute a query against it, in this case, to simply count how many of the sightings took place in California. All our previous Hive statements that generate MapReduce jobs have only produced a single one; this query against our view requires a pair of chained MapReduce jobs. Looking at the query and the view specification, this isn't necessarily surprising; it's not difficult to imagine how the view would be realized by the first MapReduce job and its output fed to the subsequent counting query performed as the second job. As a consequence, you will also see this two-stage job take much longer than any of our previous queries.

Hive is actually smarter than this. If the outer query can be folded into the view creation, then Hive will generate and execute only one MapReduce job. Given the time taken to hand-develop a series of co-operating MapReduce jobs this is a great example of the benefits Hive can offer. Though a hand-written MapReduce job (or series of jobs) is likely to be much more efficient, Hive is a great tool for determining which jobs are useful in the first place. It is better to run a slow Hive query to determine an idea isn't as useful as hoped instead of spending a day developing a MapReduce job to come to the same conclusion.

We have mentioned that views can hide underlying complexity; this does often mean that executing views is intrinsically slow. For large-scale production workloads, you will want to optimize the SQL and possibly remove the view entirely.

After running the query, we delete the view through the `DROP VIEW` statement, which demonstrates again the similarity between how HiveQL (and SQL) handle tables and views.

Handling dirty data in Hive

The observant among you may notice that the number of California sightings reported by this query is different from the number we generated in *Chapter 4*, *Developing MapReduce Programs*. Why?

Recall that before running our Hadoop Streaming or Java MapReduce jobs in *Chapter 4*, *Developing MapReduce Programs*, we had a mechanism to ignore input rows that were malformed. Then while processing the data, we used more precise regular expressions to extract the two-letter state abbreviation from the location field. However, in Hive, we did no such pre-processing and relied on quite crude mechanisms to extract the abbreviation.

On the latter, we could use some of Hive's previously mentioned functions that support regular expressions but for the former, we'd at best be forced to add complex validation `WHERE` clauses to many of our queries.

A frequent pattern is to instead preprocess data before it is imported into Hive, so for example, in this case, we could run a MapReduce job to remove all malformed records in the input file and another to do the normalization of the location field in advance.

Have a go hero – do it!

Write MapReduce jobs (it could be one or two) to do this pre-processing of the input data and generate a cleaned-up file more suited for direct importation into Hive. Then write a script to execute the jobs, create a Hive table, and import the new file into the table. This will also show how easily and powerfully scriptable Hadoop and Hive can be together.

Time for action – exporting query output

We have previously either loaded large quantities of data into Hive or extracted very small quantities as query results. We can also export large result sets; let us look at an example.

1. Recreate the previously used view:

```
$ hive -f view.hql
```

2. Create the following file as `export.hql`:

```
INSERT OVERWRITE DIRECTORY '/tmp/out'
SELECT reported, shape, state
FROM usa_sightings
WHERE state = 'California' ;
```

3. Execute the script:

```
$ hive -f export.hql
```

You will receive the following response:

```
2012-03-04 06:20:44,571 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201203040432_0029
Moving data to: /tmp/out
7599 Rows loaded to /tmp/out
MapReduce Jobs Launched:
Job 0: Map: 2  Reduce: 1   HDFS Read: 75416863 HDFS Write: 210901
SUCESS
Total MapReduce CPU Time Spent: 0 msec
OK
Time taken: 46.669 seconds
```

4. Look in the specified output directory:

```
$ hadoop fs -ls /tmp/out
```

You will receive the following response:

```
Found 1 items
-rw-r--r--   3 hadoop supergroup     210901 … /tmp/out/000000_1
```

5. Examine the output file:

```
$ hadoop fs -cat /tmp/out/000000_1 | head
```

You will receive the following output on your screen:

```
20021014_ light_California
20050224_ other_California
20021001_ egg_California
20030527_ sphere_California
20050813_ light_California
20040701_ other_California
20031007_ light_California
```

What just happened?

After reusing the previous view, we created our HiveQL script using the `INSERT OVERWRITE DIRECTORY` command. This, as the name suggests, places the results of the subsequent statement into the specified location. The `OVERWRITE` modifier is again optional and simply determines if any existing content in the location is to be firstly removed or not. The `INSERT` command is followed by a `SELECT` statement which produces the data to be written to the output location. In this example, we use a query on our previously created view which you will recall is built atop a join, demonstrating how the query here can be arbitrarily complex.

There is an additional optional `LOCAL` modifier for occasions when the output data is to be written to the local filesystem of the host running the Hive command instead of HDFS.

When we run the script, the MapReduce output is mostly as we have come to expect but with the addition of a line stating how many rows have been exported to the specified output location.

After running the script, we check the output directory and see if the result file is there and when we look at it, the contents are as we would expect.



Just as Hive's default separator for text files in inputs is ASCII code 0001 ('\a'), it also uses this as the default separator for output files, as shown in the preceding example.

The `INSERT` command can also be used to populate one table with the results of a query on others and we will look at that next. First, we need to explain a concept we will use at the same time.

Partitioning the table

We mentioned earlier that badly written joins have a long and disreputable history of causing relational databases to spend huge amounts of time grinding through unnecessary work. A similar sad tale can be told of queries that perform full table scans (visiting every row in the table) instead of using indices that allow direct access to rows of interest.

For data stored on HDFS and mapped into a Hive table, the default situation almost demands full table scans. With no way of segmenting data into a more organized structure that allows processing to apply only to the data subset of interest, Hive is forced to process the entire data set. For our UFO file of approximately 70 MB, this really is not a problem as we see the file processed in tens of seconds. However, what if it was a thousand times larger?

As with traditional relational databases, Hive allows tables to be partitioned based on the values of virtual columns and for these values to then be used in query predicates later.

In particular, when a table is created, it can have one or more partition columns and when loading data into the table, the specified values for these columns will determine the partition into which the data is written.

The most common partitioning strategy for tables that see lots of data ingested on a daily basis is for the partition column to be the date. Future queries can then be constrained to process only that data contained within a particular partition. Under the covers, Hive stores each partition in its own directory and files, which is how it can then apply MapReduce jobs only on the data of interest. Through the use of multiple partition columns, it is possible to create a rich hierarchical structure and for large tables with queries that require only small subsets of data it is worthwhile spending some time deciding on the optimal partitioning strategy.

For our UFO data set, we will use the year of the sighting as the partition value but we have to use a few less common features to make it happen. Hence, after this introduction, let us now make some partitions!

Time for action – making a partitioned UFO sighting table

We will create a new table for the UFO data to demonstrate the usefulness of partitioning.

1. Save the following query as `createpartition.hql`:

```
CREATE TABLE partufo(sighted string, reported string, sighting_
location string,shape string, duration string, description string)
PARTITIONED BY (year string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' ;
```

2. Save the following query as `insertpartition.hql`:

```
SET hive.exec.dynamic.partition=true ;
SET hive.exec.dynamic.partition.mode=nonstrict ;

INSERT OVERWRITE TABLE partufo partition (year)
SELECT sighted, reported, sighting_location, shape, duration,
description,
SUBSTR(TRIM(sighted), 1,4)  FROM ufodata ;
```

3. Create the partitioned table:

```
$ hive -f createpartition.hql
```

You will receive the following response:

```
Logging initialized using configuration in jar:file:/opt/hive-
0.8.1/lib/hive-common-0.8.1.jar!/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_
hadoop_201203101838_17331656.txt
OK
Time taken: 4.754 seconds
```

4. Examine the created table:

```
OK
sighted  string
reported  string
sighting_location  string
shape  string
duration  string
description  string
year  string
Time taken: 4.704 seconds
```

5. Populate the table:

```
$ hive -f insertpartition.hql
```

You will see the following lines on the screen:

```
Total MapReduce jobs = 2
…
…
```

```
Ended Job = job_201203040432_0041
Ended Job = 994255701, job is filtered out (removed at runtime).
Moving data to: hdfs://head:9000/tmp/hive-hadoop/hive_2012-03-
10_18-38-36_380_1188564613139061024/-ext-10000
Loading data to table default.partufo partition (year=null)
       Loading partition {year=1977}
       Loading partition {year=1880}
       Loading partition {year=1975}
       Loading partition {year=2007}
       Loading partition {year=1957}
…
Table default.partufo stats: [num_partitions: 100, num_files: 100,
num_rows: 0, total_size: 74751215, raw_data_size: 0]
61393 Rows loaded to partufo
…
OK
Time taken: 46.285 seconds
```

6. Execute a `count` command against a partition:

```
$ hive -e "select count(*)from partufo where year  = '1989'"
```

You will receive the following response:

```
OK
249
Time taken: 26.56 seconds
```

7. Execute a similar query on the non-partitioned table:

```
$ hive -e "select count(*) from ufodata where sighted like
'1989%'"
```

You will receive the following response:

```
OK
249
Time taken: 28.61 seconds
```

8. List the contents of the Hive directory housing the partitioned table:

```
$ Hadoop fs -ls /user/hive/warehouse/partufo
```

You will receive the following response:

```
Found 100 items
drwxr-xr-x   - hadoop supergroup          0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=0000
drwxr-xr-x   - hadoop supergroup          0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=1400
drwxr-xr-x   - hadoop supergroup          0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=1762
drwxr-xr-x   - hadoop supergroup          0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=1790
drwxr-xr-x   - hadoop supergroup          0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=1860
drwxr-xr-x   - hadoop supergroup          0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=1864
drwxr-xr-x   - hadoop supergroup          0 2012-03-10 18:38 /
user/hive/warehouse/partufo/year=1865
```

What just happened?

We created two HiveQL scripts for this example. The first of these creates the new partitioned table. As we can see, it looks very much like the previous `CREATE TABLE` statements; the difference is in the additional `PARTITIONED BY` clause.

After we execute this script, we describe the table and see that from a HiveQL perspective the table appears just like the previous `ufodata` table but with the addition of an extra column for the year. This allows the column to be treated as any other when it comes to specifying conditions in `WHERE` clauses, even though the column data does not actually exist in the on-disk data files.

We next execute the second script which performs the actual loading of data into the partitioned table. There are several things of note here.

Firstly, we see that the `INSERT` command can be used with tables just as we previously did for directories. The `INSERT` statement has a specification of where the data is to go and a subsequent `SELECT` statement gathers the required data from existing tables or views.

The partitioning mechanism used here is taking advantage of a relatively new feature in Hive, dynamic partitions. In most cases, the partition clause in this statement would include an explicit value for the year column. But though that would work if we were uploading a day's data into a daily partition, it isn't suitable for our type of data file where the various rows should be inserted into a variety of partitions. By simply specifying the column name with no value, the partition name will be automatically generated by the value of the year column returned from the `SELECT` statement.

This hopefully explains the strange final clause in the `SELECT` statement; after specifying all the standard columns from `ufodata`, we add a specification that extracts a string containing the first four characters of the sighting column. Remember that because the partitioned table sees the year partition column as the seventh column, this means we are assigning the year component of the sighted string to the year column in each row. Consequently, each row is inserted into the partition associated with its sighting year.

To prove this is working as expected, we then perform two queries; one counts all records in the partition for 1989 in the partitioned table, the other counts the records in `ufodata` that begin with the string "1989", that is, the component used to dynamically create the partitions previously.

As can be seen, both queries return the same result, verifying that our partitioning strategy is working as expected. We also note that the partitioned query is a little faster than the other, though not by very much. This is likely due to the MapReduce start up times dominating the processing of our relatively modest data set.

Finally, we take a look inside the directory where Hive stores the data for the partitioned table and see that there is indeed a directory for each of the 100 dynamically-generated partitions. Any time we now express HiveQL statements that refer to specific partitions, Hive can perform a significant optimization by processing only the data found in the appropriate partitions' directories.

Bucketing, clustering, and sorting... oh my!

We will not explore it in detail here, but hierarchical partition columns are not the full extent of how Hive can optimize data access patterns within subsets of data. Within a partition, Hive provides a mechanism to further gather rows into buckets using a hash function on specified `CLUSTER BY` columns. Within a bucket, the rows can be kept in sorted order using specified `SORT BY` columns. We could, for example, have bucketed our data based on the UFO shape and within each bucket sorted on the sighting date.

These aren't necessarily features you'll need to use on day 1 with Hive, but if you find yourself using larger and larger data sets, then considering this type of optimization may help query processing time significantly.

User-Defined Function

Hive provides mechanisms for you to hook custom code directly into the HiveQL execution. This can be in the form of adding new library functions or by specifying **Hive transforms**, which work quite similarly to Hadoop Streaming. We will look at user-defined functions in this section as they are where you are most likely to have an early need to add custom code. Hive transforms are a somewhat more involved mechanism by which you can add custom map and reduce classes that are invoked by the Hive runtime. If transforms are of interest, they are well documented on the Hive wiki.

Time for action – adding a new User Defined Function (UDF)

Let us show how to create and invoke some custom Java code via a new UDF.

1. Save the following code as `City.java`:

```
package com.kycorsystems ;

import java.util.regex.Matcher ;
import java.util.regex.Pattern ;
import org.apache.hadoop.hive.ql.exec.UDF ;
import org.apache.hadoop.io.Text ;

public class City extends UDF
{
    private static Pattern pattern = Pattern.compile(
        "[a-zA-z]+?[\\. ]*[a-zA-z]+?[\\, ][^a-zA-Z]") ;

    public Text evaluate( final Text str)
    {
        Text result ;
        String location = str.toString().trim() ;
        Matcher matcher = pattern.matcher(location) ;

        if (matcher.find())
        {
            result = new Text(                location.
substring(matcher.start(), matcher.end()-2)) ;
        }
        else
        {
            result = new Text("Unknown") ;
        }
        return result ;
    }
}
```

2. Compile this file:

```
$ javac -cp hive/lib/hive-exec-0.8.1.jar:hadoop/hadoop-1.0.4-core.
jar  -d . City.java
```

3. Package the generated class file into a JAR file:

```
$ jar cvf city.jar com
```

You will receive the following response:

```
added manifest
adding: com/(in = 0) (out= 0)(stored 0%)
adding: com/kycorsystems/(in = 0) (out= 0)(stored 0%)
adding: com/kycorsystems/City.class(in = 1101) (out= 647)(deflated
41%)
```

4. Start the interactive Hive shell:

```
$ hive
```

5. Add the new JAR file to the Hive classpath:

```
hive> add jar city.jar;
```

You will receive the following response:

```
Added city.jar to class path
Added resource: city.jar
```

6. Confirm that the JAR file was added:

```
hive> list jars;
```

You will receive the following response:

```
file:/opt/hive-0.8.1/lib/hive-builtins-0.8.1.jar
city.jar
```

7. Register the new code with a function name:

```
hive> create temporary function city as 'com.kycorsystems.City' ;
```

You will receive the following response:

```
OK
Time taken: 0.277 seconds
```

8. Execute a query using the new function:

```
hive> select city(sighting_location), count(*) as total
    > from partufo
    > where year = '1999'
    > group by city(sighting_location)
    > having  total > 15 ;
```

You will receive the following response:

```
Total MapReduce jobs = 1
Launching Job 1 out of 1
…
OK
Chicago  19
Las Vegas  19
Phoenix  19
Portland  17
San Diego  18
Seattle  26
Unknown  34
Time taken: 29.055 seconds
```

What just happened?

The Java class we wrote extends the base `org.apache.hadoop.hive.exec.ql.UDF` (User Defined Function) class. Into this class, we define a method for returning a city name given a location string that follows the general pattern we have seen previously.

UDF does not actually define a series of **evaluate** methods based on type; instead, you are free to add your own with arbitrary arguments and return types. Hive uses Java Reflection to select the correct evaluation method, and if you require a finer-grained selection, you can develop your own utility class that implements the `UDFMethodResolver` interface.

The regular expression used here is a little unwieldy; we wish to extract the name of the city, assuming it will be followed by a state abbreviation. However, inconsistency in how the names are delineated and handling of multi-word names gives us the regular expression seen before. Apart from this, the class is pretty straightforward.

We compile the `City.java` file, adding the necessary JARs from both Hive and Hadoop as we do so.



Remember, of course, that the specific JAR filenames may be different if you are not using the same versions of both Hadoop and Hive.

We then bundle the generated class file into a JAR and start the Hive interactive shell.

After creating the JAR, we need to configure Hive to use it. This is a two-step process. Firstly, we use the `add jar` command to add the new JAR file to the classpath used by Hive. After doing so, we use the `list jars` command to confirm that our new JAR has been registered in the system.

Adding the JAR only tells Hive that some code exists, it does not say how we wish to refer to the function within our HiveQL statements. The `CREATE FUNCTION` command does this—associating a function name (in this case, `city`) with the fully qualified Java class that provides the implementation (in this case, `com.kycorsystems.City`).

With both the JAR file added to the classpath and the function created, we can now refer to our `city()` function within our HiveQL statements.

We next ran an example query that demonstrates the new function in action. Going back to the partitioned UFO sightings table, we thought it would be interesting to see where the most UFO sightings were occurring as everyone prepared for the end-of-millennium apocalypse.

As can be seen from the HiveQL statement, we can use our new function just like any other and indeed the only way to know which functions are built-in and which are UDFs is through familiarity with the standard Hive function library.

The result shows a significant concentration of sightings in the north-west and south-west of the USA, Chicago being the only exception. We did get quite a few `Unknown` results however, and it would require further analysis to determine if that was due to locations outside of the U.S. or if we need to further refine our regular expression.

To preprocess or not to preprocess...

Let us re-visit an earlier topic; the potential need to pre-process data into a cleaner form before it is imported into Hive. As can be seen from the preceding example, we could perform similar processing on the fly through a series of UDFs. We could, for example, add functions called `state` and `country` that extract or infer the further region and nation components from the location sighting string. There are rarely concrete rules for which approach is best, but a few guidelines may help.

If, as is the case here, we are unlikely to actually process the full location string for reasons other than to extract the distinct components, then preprocessing likely makes more sense. Instead of performing expensive text processing every time the column is accessed, we could either normalize it into a more predictable format or even break it out into separate city/region/country columns.

If, however, a column is usually used in HiveQL in its original form and additional processing is the exceptional case, then there is likely little benefit to an expensive processing step across the entire data set.

Use the strategy that makes the most sense for your data and workloads. Remember that UDFs are for much more than this sort of text processing, they can be used to encapsulate any type of logic that you wish to apply to data in your tables.

Hive versus Pig

Search the Internet for articles about Hive and it won't be long before you find many comparing Hive to another Apache project called **Pig**. Some of the most common questions around this comparison are why both exist, when to use one over the other, which is better, and which makes you look cooler when wearing the project t-shirt in a bar.

The overlap between the projects is that whereas Hive looks to present a familiar SQL-like interface to data, Pig uses a language called **Pig Latin** that specifies dataflow pipelines. Just as Hive translates HiveQL into MapReduce which it then executes, Pig performs similar MapReduce code generation from the Pig Latin scripts.

The biggest difference between HiveQL and Pig Latin is the amount of control expressed over how the job will be executed. HiveQL, just like SQL, specifies what is to be done but says almost nothing about how to actually structure the implementation. The HiveQL query planner is responsible for determining in which order to perform particular parts of the HiveQL command, in which order to evaluate functions, and so on. These decisions are made by Hive at runtime, analogous to a traditional relational database query planner, and this is also the level at which Pig Latin operates.

Both approaches obviate the need to write raw MapReduce code; they differ in the abstractions they provide.

The choice of Hive versus Pig will depend on your needs. If having a familiar SQL interface to the data is important as a means of making the data in Hadoop available to a wider audience, then Hive is the obvious choice. If instead you have personnel who think in terms of data pipelines and need finer-grained control over how the jobs are executed, then Pig may be a better fit. The Hive and Pig projects are looking for closer integration so hopefully the false sense of competition will decrease and instead both will be seen as complementary ways of decreasing the Hadoop knowledge required to execute MapReduce jobs.

What we didn't cover

In this overview of Hive, we have covered its installation and setup, the creation and manipulation of tables, views, and joins. We have looked at how to move data into and out of Hive, how to optimize data processing, and explored several of Hive's built-in functions.

In reality, we have barely scratched the surface. In addition to more depth on the previous topics and a variety of related concepts, we didn't even touch on topics such as the **MetaStore** where Hive stores its configuration and metadata or **SerDe** (**serialize/deserialize**) objects, which can be used to read data from more complex file formats such as JSON.

Hive is an incredibly rich tool with many powerful and complex features. If Hive is something that you feel may be of value to you, then it is recommended that after running through the examples in this chapter that you spend some quality time with the documentation on the Hive website. There you will also find links to the user mailing list, which is a great source of information and help.

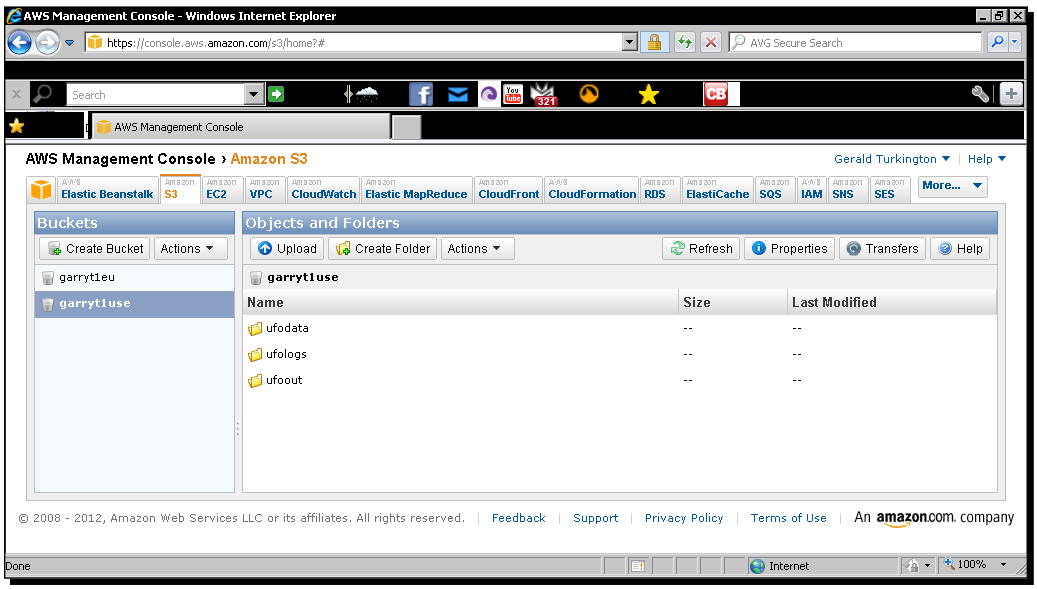
Hive on Amazon Web Services

Elastic MapReduce has significant support for Hive with some specific mechanisms to help its integration with other AWS services.

Time for action – running UFO analysis on EMR

Let us explore the use of EMR with Hive by doing some UFO analysis on the platform.

1. Log in to the AWS management console at `http://aws.amazon.com/console`.
2. Every Hive job flow on EMR runs from an S3 bucket and we need to select the bucket we wish to use for this purpose. Select **S3** to see the list of the buckets associated with your account and then choose the bucket from which to run the example, in the example below, we select the bucket called garryt1use.
3. Use the web interface to create three directories called `ufodata`, `ufoout`, and `ufologs` within that bucket. The resulting list of the bucket's contents should look like the following screenshot:



4. Double-click on the `ufodata` directory to open it and within it create two subdirectories called `ufo` and `states`.

5. Create the following as `s3test.hql`, click on the **Upload** link within the `ufodata` directory, and follow the prompts to upload the file:

```
CREATE EXTERNAL TABLE IF NOT EXISTS ufodata(sighted string,
reported string, sighting_location string,
shape string, duration string, description string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LOCATION '${INPUT}/ufo' ;

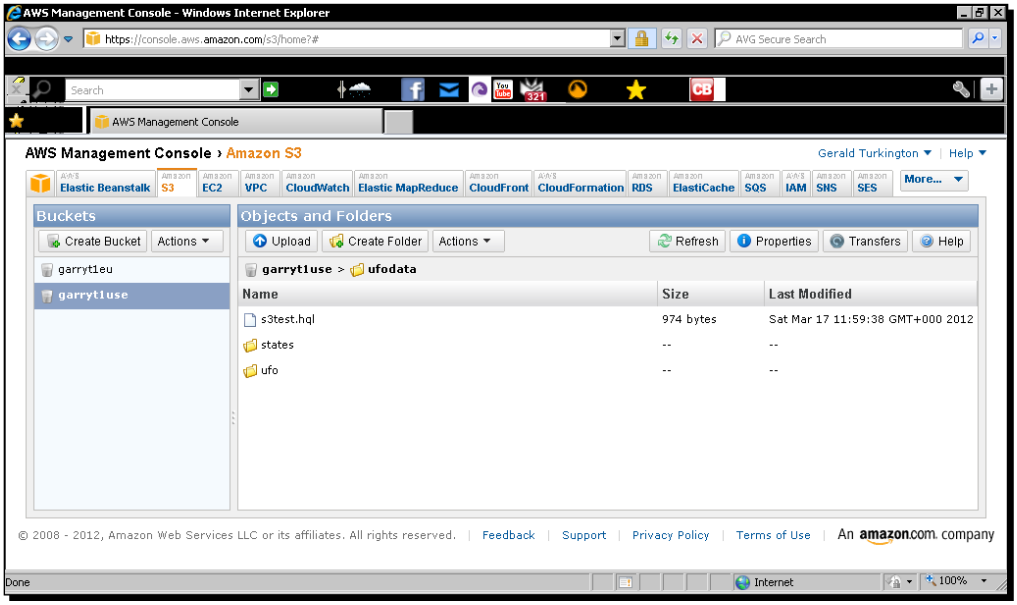
CREATE EXTERNAL TABLE IF NOT EXISTS states(abbreviation string,
full_name string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LOCATION '${INPUT}/states' ;

CREATE VIEW IF NOT EXISTS usa_sightings (sighted, reported, shape,
state)
AS SELECT t1.sighted, t1.reported, t1.shape, t2.full_name
FROM ufodata t1 JOIN states t2
ON (LOWER(t2.abbreviation) = LOWER(SUBSTR( t1.sighting_location,
(LENGTH(t1.sighting_location)-1)))) ;

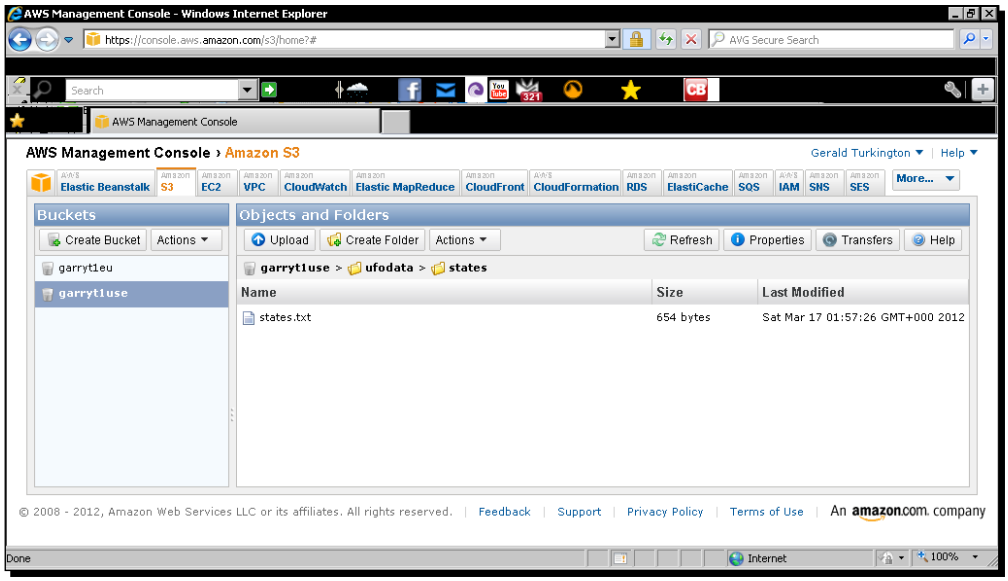
CREATE EXTERNAL TABLE IF NOT EXISTS state_results ( reported
string, shape string, state string)
ROW FORMAT DELIMITED
FFIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION '${OUTPUT}/states' ;

INSERT OVERWRITE TABLE state_results
SELECT reported, shape, state
FROM usa_sightings
WHERE state = 'California' ;
```

The contents of `ufodata` should now look like the following screenshot:

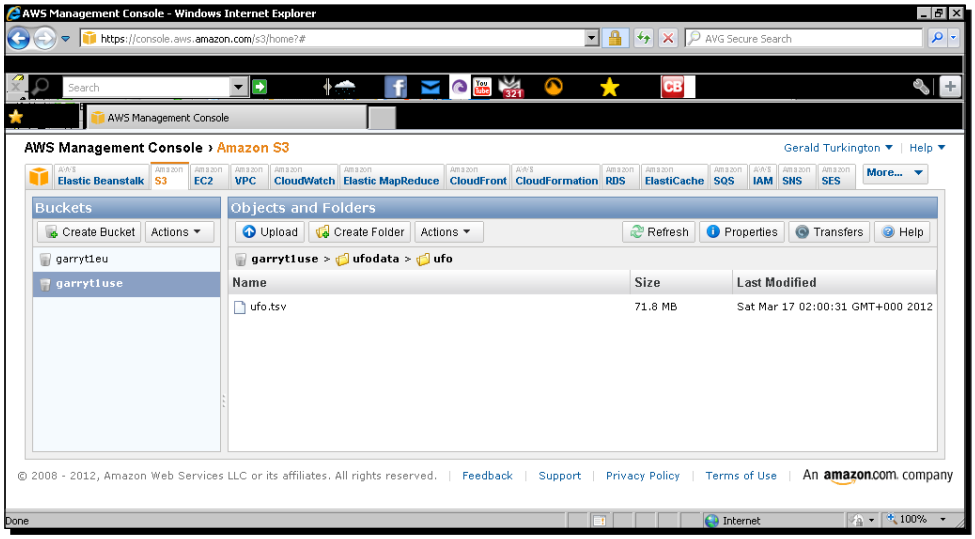


6. Double-click the `states` directory to open it and into this, upload the `states.txt` file used earlier. The directory should now look like the following screenshot:

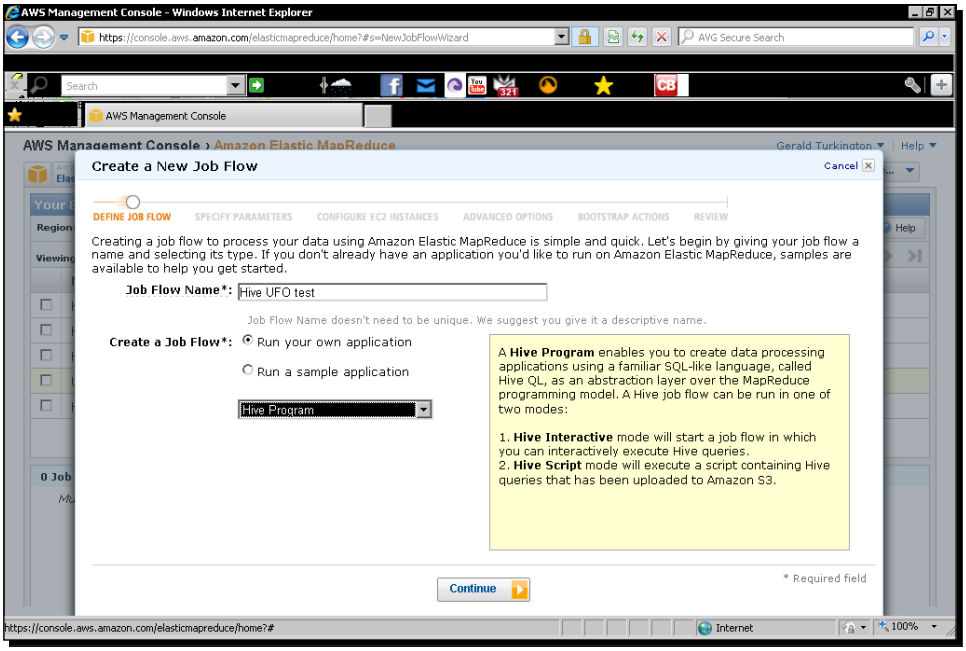


7. Click on the `ufodata` component at the top of the file list to return to this directory.

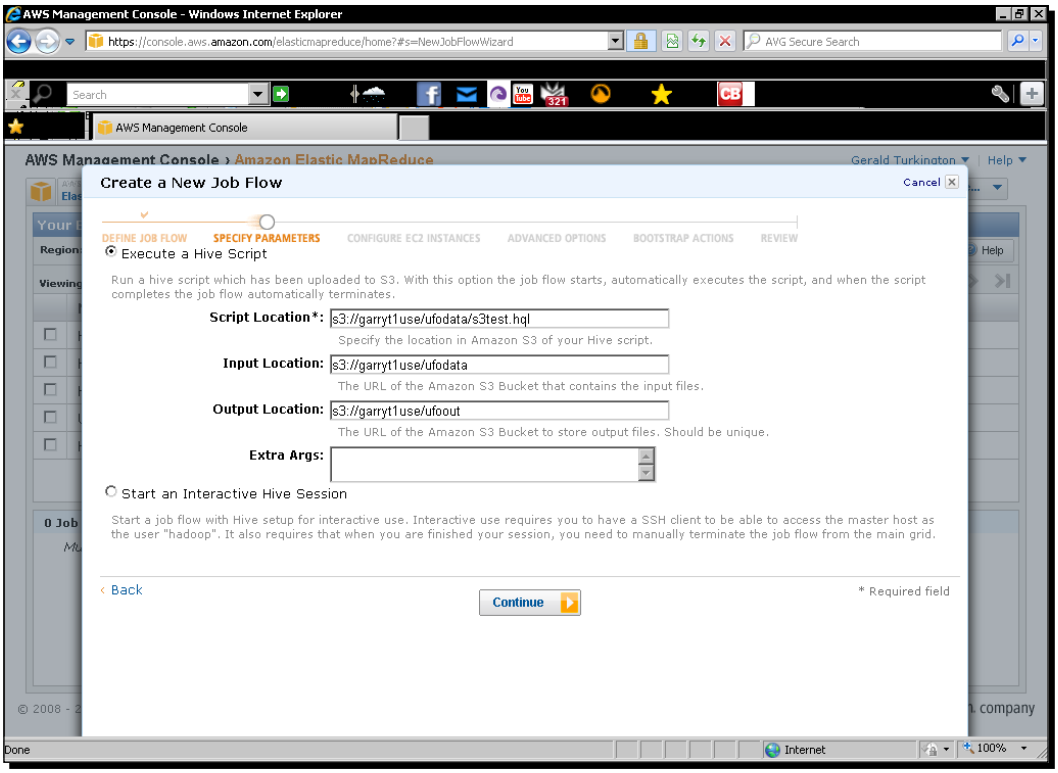
8. Double-click on the `ufo` directory to open it and into this, upload the `ufo.tsv` file used earlier. The directory should now look like the following screenshot:



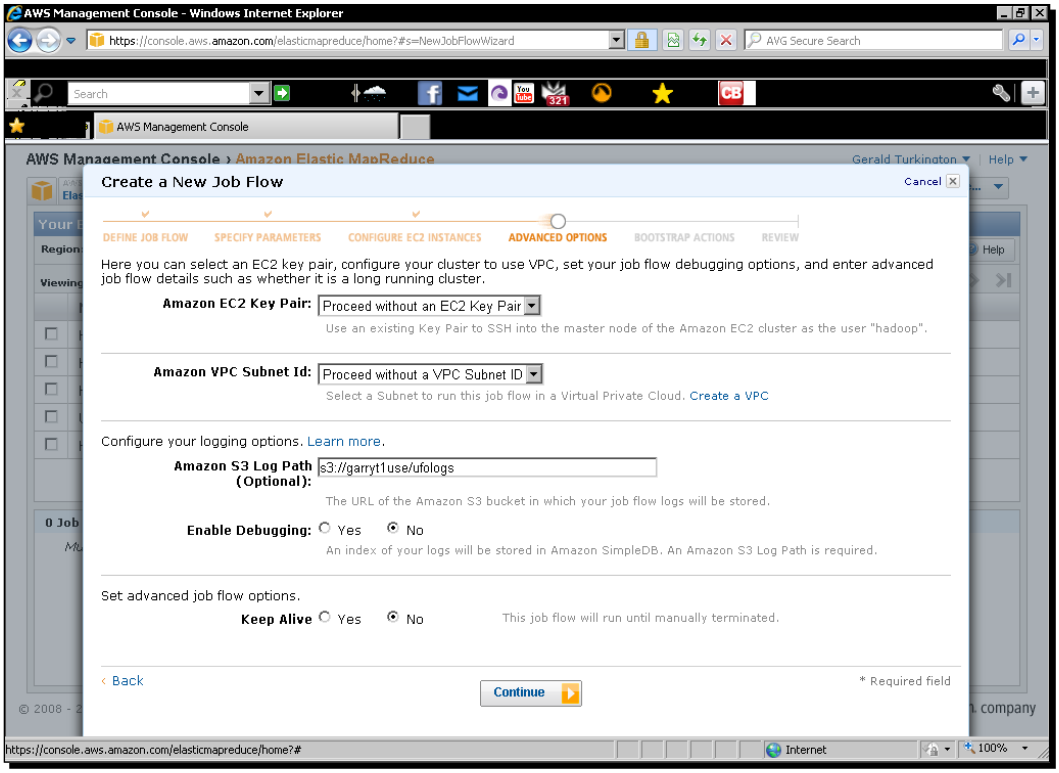
9. Now select **Elastic MapReduce** and click on **Create a New Job Flow**. Then select the option **Run your own application** and select a Hive application, as shown in the following screenshot:



10. Click on **Continue** and then fill in the required details for the Hive job flow. Use the following screenshot as a guide, but remember to change the bucket name (the first component in the `s3:// URLs`) to the bucket you set up before:



11. Click on **Continue**, review the number and the type of hosts to be used, and then click on **Continue** once again. Then fill in the name of the directory for the logs, as shown in the following screenshot:



12. Click on **Continue**. Then do the same through the rest of the job creation process as there are no other default options that need to be changed for this example. Finally start the job flow and monitor its progress from the management console.

13. Once the job has completed successfully, go back to **S3** and double-click on the `ufoout` directory. Within that should be a directory called `states` and within that, a file named something like `0000000`. Double-click to download the file and verify that its contents look something like the following:

```
20021014   light   California
20050224   other   California
20021001   egg     California
20030527   sphere  California
```

What just happened?

Before we actually execute our EMR job flow, we needed to do a bit of setup in the preceding example. Firstly, we used the S3 web interface to prepare the directory structure for our job. We created three main directories to hold the input data, into which to write results and one for EMR to place logs of the job flow execution.

The HiveQL script is a modification of several of the Hive commands used earlier in this chapter. It creates the tables for the UFO sighting data and state names as well as the view joining them. Then it creates a new table with no source data and uses an `INSERT OVERWRITE TABLE` to populate the table with the results of a query.

The unique feature in this script is the way we specify the `LOCATION` clauses for each of the tables. For the input tables, we use a path relative to a variable called `INPUT` and do likewise with the `OUTPUT` variable for the result table.

Note that Hive in EMR expects the location of table data to be a directory and not a file. This is the reason for us previously creating subdirectories for each table into which we uploaded the specific source file instead of specifying the table with the direct path to the data files themselves.

After setting up the required file and directory structure within our S3 bucket, we went to the EMR web console and started the job flow creation process.

After specifying that we wish to use our own program and that it would be a Hive application, we filled in a screen with the key data required for our job flow:

- The location of the HiveQL script itself
- The directory containing input data
- The directory to be used for output data

The path to the HiveQL script is an explicit path and does not require any explanation. However, it is important to realize how the other values are mapped into the variables used within our Hive script.

The value for the input path is available to the Hive script as the `INPUT` variable and this is how we then specify the directory containing the UFO sighting data as `${INPUT}/ufo`. Similarly, the output value specified in this form will be used as the `OUTPUT` variable within our Hive script.

We did not make any changes to the default host setup, which will be one small master and two small core nodes. On the next screen, we added the location into which we wanted EMR to write the logs produced by the job flow execution.

Though optional, it is useful to capture these logs, particularly in the early stages of running a new script, though obviously S3 storage does have a cost. EMR can also write indexed log data into **SimpleDB** (another AWS service), but we did not show that in action here.

After completing the job flow definition, we started it and on successful execution, went to the S3 interface to browse to the output location, which happily contained the data we were expecting.

Using interactive job flows for development

When developing a new Hive script to be executed on EMR, the previous batch job execution is not a good fit. There is usually a several minute latency between job flow creation and execution and if the job fails, then the cost of several hours of EC2 instance time will have been incurred (partial hours are rounded up).

Instead of selecting the option to create an EMR job flow to run a Hive script, as in the previous example, we can start a Hive job flow in interactive mode. This effectively spins up a Hadoop cluster without requiring a named script. You can then SSH into the master node as the Hadoop user where you will find Hive installed and configured. It is much more efficient to do the script development in this environment and then, if required, set up the batch script job flows to automatically execute the script in production.

Have a go hero – using an interactive EMR cluster

Start up an interactive Hive job flow in EMR. You will need to have SSH credentials already registered with EC2 so that you can connect to the master node. Run the previous script directly from the master node, remembering to pass the appropriate variables to the script.

Integration with other AWS products

With a local Hadoop/Hive installation, the question of where data lives usually comes down to HDFS or local filesystems. As we have seen previously, Hive within EMR gives another option with its support for external tables whose data resides in S3.

Another AWS service with similar support is **DynamoDB** (at `http://aws.amazon.com/dynamodb`), a hosted NoSQL database solution in the cloud. Hive job flows within EMR can declare external tables that either read data from DynamoDB or use it as the destination for query output.

This is a very powerful model as it allows Hive to be used to process and combine data from multiple sources while the mechanics of mapping data from one system into Hive tables happens transparently. It also allows Hive to be used as a mechanism for moving data from one system to another. The act of getting data frequently into such hosted services from existing stores is a major adoption hurdle.

Summary

We have looked at Hive in this chapter and learned how it provides many tools and features that will be familiar to anyone who uses relational databases. Instead of requiring development of MapReduce applications, Hive makes the power of Hadoop available to a much broader community.

In particular, we downloaded and installed Hive, learning that it is a client application that translates its HiveQL language into MapReduce code, which it submits to a Hadoop cluster. We explored Hive's mechanism for creating tables and running queries against these tables. We saw how Hive can support various underlying data file formats and structures and how to modify those options.

We also appreciated that Hive tables are largely a logical construct and that behind the scenes, all the SQL-like operations on tables are in fact executed by MapReduce jobs on HDFS files. We then saw how Hive supports powerful features such as joins and views and how to partition our tables to aid in efficient query execution.

We used Hive to output the results of a query to files on HDFS and saw how Hive is supported by Elastic MapReduce, where interactive job flows can be used to develop new Hive applications, and then ran automatically in batch mode.

As we have mentioned several times in this book, Hive looks like a relational database but is not really one. However, in many cases you will find existing relational databases are part of the broader infrastructure into which you need integrate. Performing that integration and how to move data across these different types of data sources will be the topic of the next chapter.

9
Working with Relational Databases

As we saw in the previous chapter, Hive is a great tool that provides a relational database-like view of the data stored in Hadoop. However, at the end of the day, it is not truly a relational database. It does not fully implement the SQL standard, and its performance and scale characteristics are vastly different (not better or worse, just different) from a traditional relational database.

In many cases, you will find a Hadoop cluster sitting alongside and used with (not instead of) relational databases. Often the business flows will require data to be moved from one store to the other; we will now explore such integration.

In this chapter, we will:

- Identify some common Hadoop/RDBMS use cases
- Explore how we can move data from RDBMS into HDFS and Hive
- Use Sqoop as a better solution for such problems
- Move data with exports from Hadoop into an RDBMS
- Wrap up with a discussion of how this can be applied to AWS

Common data paths

Back in *Chapter 1*, *What It's All About*, we touched on what we believe to be an artificial choice that causes a lot of controversy; to use Hadoop or a traditional relational database. As explained there, it is our contention that the thing to focus on is identifying the right tool for the task at hand and that this is likely to lead to a situation where more than one technology is employed. It is worth looking at a few concrete examples to illustrate this idea.

Hadoop as an archive store

When an RDBMS is used as the main data repository, there often arises issues of scale and data retention. As volumes of new data increase, what is to be done with the older and less valuable data?

Traditionally, there are two main approaches to this situation:

- Partition the RDBMS to allow higher performance of more recent data; sometimes the technology allows older data to be stored on slower and less expensive storage systems
- Archive the data onto tape or another offline store

Both approaches are valid, and the decision between the two often rests on just whether or not the older data is required for timely access. These are two extreme cases as the former maximizes for access at the cost of complexity and infrastructure expense, while the latter reduces costs but makes data less accessible.

The model being seen recently is for the most current data to be kept in the relational database and the older data to be pushed into Hadoop. This can either be onto HDFS as structured files or into Hive to retain the RDBMS interface. This gives the best of both worlds, allowing the lower-volume, more recent data to be accessible by high-speed, low-latency SQL queries, while the much larger volume of archived data will be accessed from Hadoop. The data therefore remains available for use cases requiring either types of access; this would be needed on a platform that does require additional integration for any queries that need to span both the recent and archive data.

Because of Hadoop's scalability, this model gives great future growth potential; we know we can continue to increase the amount of archive data being stored while retaining the ability to run analytics against it.

Hadoop as a preprocessing step

Several times in our Hive discussion, we highlighted opportunities where some preprocessing jobs to massage or otherwise clean up the data would be hugely useful. The unfortunate fact is that, in many (most?) big data situations, the large volumes of data coming from multiple sources mean that dirty data is simply a given. Although most MapReduce jobs only require a subset of the overall data to be processed, we should still expect to find incomplete or corrupt data across the data set. Just as Hive can benefit from preprocessing data, a traditional relational database can as well.

Hadoop can be a great tool here; it can pull data from multiple sources, combine them for necessary transformations, and clean up prior to the data being inserted into the relational database.

Hadoop as a data input tool

Hadoop is not just valuable in that it makes data better and is well suited to being ingested into a relational database. In addition to such tasks, Hadoop can also be used to generate additional data sets or data views that are then served from the relational database. Common patterns here are situations such as when we wish to display not only the primary data for an account but to also display alongside it secondary data generated from account history. Such views could be summaries of transactions against types of expenditure for the previous months. This data is held within Hadoop, from which can be generated the actual summaries that may be pushed back into the database for quicker display.

The serpent eats its own tail

Reality is often more complex than these well-defined situations, and it's not uncommon for the data flow between Hadoop and the relational database to be described by circles and arcs instead of a single straight line. The Hadoop cluster may, for example, do the preprocessing step on data that is then ingested into the RDBMS and then receive frequent transaction dumps that are used to build aggregates, which are sent back to the database. Then, once the data gets older than a certain threshold, it is deleted from the database but kept in Hadoop for archival purposes.

Regardless of the situation, the ability to get data from Hadoop to a relational database and back again is a critical aspect of integrating Hadoop into your IT infrastructure. So, let's see how to do it.

Setting up MySQL

Before reading and writing data from a relational database, we need a running relational database. We will use MySQL in this chapter because it is freely and widely available and many developers have used it at some point in their career. You can of course use any RDBMS for which a JDBC driver is available, but if you do so, you'll need to modify the aspects of this chapter that require direct interaction with the database server.

Time for action – installing and setting up MySQL

Let's get MySQL installed and configured with the basic databases and access rights.

1. On an Ubuntu host, install MySQL using `apt-get`:

```
$ apt-get update
$ apt-get install mysql-server
```

2. Follow the prompts, and when asked, choose a suitable root password.

3. Once installed, connect to the MySQL server:

```
$ mysql -h localhost -u root -p
```

4. Enter the root password when prompted:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 40
…
Mysql>
```

5. Create a new database to use for the examples in this chapter:

```
Mysql> create database hadooptest;
```

   You will receive the following response:

```
Query OK, 1 row affected (0.00 sec)
```

6. Create a user account with full access to the database:

```
Mysql>  grant all on hadooptest.* to 'hadoopuser'@'%' identified
by 'password';
```

   You will receive the following response:

```
Query OK, 0 rows affected (0.01 sec)
```

7. Reload the user privileges to have the user changes take effect:

```
Mysql> flush privileges;
```

   You will receive the following response:

```
Query OK, 0 rows affected (0.01 sec)
```

8. Log out as root:

```
mysql> quit;
```

   You will receive the following response:

```
Bye
```

9. Log in as the newly created user, entering the password when prompted:

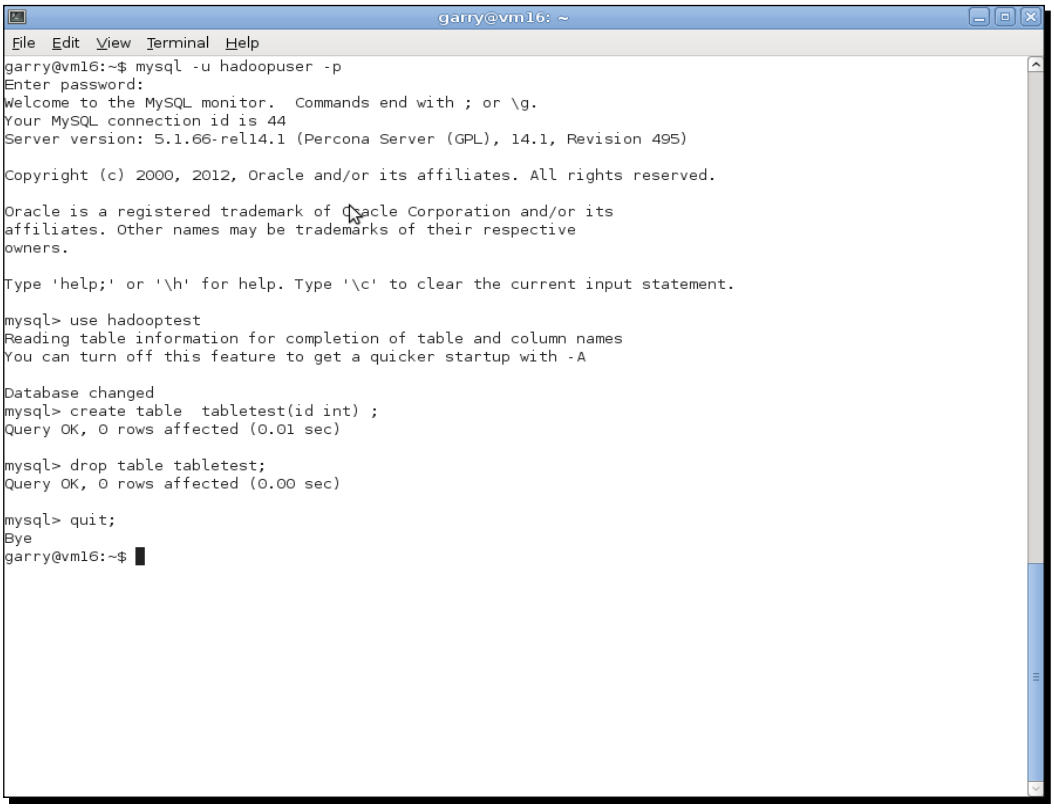
```
$ mysql -u hadoopuser -p
```

10. Change to the newly created database:

```
mysql> use hadooptest;
```

11. Create a test table, drop it to confirm the user has the privileges in this database, and then log out:

```
mysql> create table tabletest(id int);
mysql> drop table tabletest;
mysql> quit;
```



What just happened?

Due to the wonders of package managers such as `apt`, installing complex software such as MySQL is really very easy. We just use the standard process to install a package; under Ubuntu (and most other distributions in fact), requesting the main server package for MySQL will bring along all needed dependencies as well as the client packages.

During the install, you will be prompted for the root password on the database. Even if this is a test database instance that no one will use and that will have no valuable data, please give the root user a strong password. Having weak root passwords is a bad habit, and we do not want to encourage it.

After MySQL is installed, we connect to the database using the `mysql` command-line utility. This takes a range of options, but the ones we will use are as follows:

- `-h`: This option is used to specify the hostname of the database (the local machine is assumed if none is given)
- `-u`: This option is used for the username with which to connect (the default is the current Linux user)
- `-p`: This option is used to be prompted for the user password

MySQL has the concept of multiple databases, each of which is a collective grouping of tables. Every table needs be associated with a database. MySQL has several built-in databases, but we use the `CREATE DATABASE` statement to create a new one called `hadooptest` for our later work.

MySQL refuses connections/requests to perform actions unless the requesting user has explicitly been given the needed privileges to perform the action. We do not want to do everything as the root user (a bad practice and quite dangerous since the root can modify/delete everything), so we create a new user called `hadoopuser` by using the `GRANT` statement.

The `GRANT` statement we used actually does three distinct things:

- Creates the `hadoopuser` account
- Sets the `hadoopuser` password; we set it to `password`, which obviously you should never do; pick something easy to memorize
- Gives `hadoopuser` all privileges on the `hadooptest` database and all its tables

We issue the `FLUSH PRIVILEGES` command to have these changes take effect and then we log out as root and connect as the new user to check whether all is working.

The `USE` statement here is a little superfluous. In future, we can instead add the database name to the `mysql` command-line tool to automatically change to that database.

Connecting as the new user is a good sign, but to gain full confidence, we create a new table in the `hadooptest` database and then drop it. Success here shows that `hadoopuser` does indeed have the requested privileges to modify the database.

Did it have to be so hard?

We are perhaps being a little cautious here by checking every step of the process along the way. However, I have found in the past that subtle typos, in the `GRANT` statement in particular, can result in really hard-to-diagnose problems later on. And to continue our paranoia, let's make one change to the default MySQL configuration that we won't need quite yet, but which if we don't do, we'll be sorry later.

For any production database, you would of course not have security-sensitive statements, such as `GRANT`, present that were typed in from a book. Refer to the documentation of your database to understand user accounts and privileges.

Time for action – configuring MySQL to allow remote connections

We need to change the common default MySQL behavior, which will prevent us from accessing the database from other hosts.

1. Edit `/etc/mysql/my.cnf` in your favorite text editor and look for this line:

   ```
   bind-address = 127.0.0.1
   ```

2. Change it to this:

   ```
   # bind-address = 127.0.0.1
   ```

3. Restart MySQL:

   ```
   $ restart mysql
   ```

What just happened?

Most out-of-the-box MySQL configurations allow access only from the same host on which the server is running. This is absolutely the correct default from a security standpoint. However, it can also cause real confusion if, for example, you launch MapReduce jobs that try to access the database on that host. You may see the job fail with connection errors. If that happens, you fire up the `mysql` command-line client on the host; this will succeed. Then, perhaps, you will write a quick JDBC client to test connectivity. This will also work. Only when you try these steps from one of the Hadoop worker nodes will the problem be apparent. Yes, this has bit ten me several times in the past!

The previous change tells MySQL to bind to all available interfaces and thus be accessible from remote clients.

After making the change, we need to restart the server. In Ubuntu 11.10, many of the service scripts have been ported to the `Upstart` framework, and we can use the handy `restart` command directly.

If you are using a distribution other than Ubuntu—or potentially even a different version of Ubuntu—the global MySQL configuration file may be in a different location; `/etc/my.cnf`, for example, on CentOS and Red Hat Enterprise Linux.

Don't do this in production!

Or at least not without thinking about the consequences. In the earlier example, we gave a really bad password to the new user; do not do that. However, especially don't do something like that if you then make the database available across the network. Yes, it is a test database with no valuable data, but it is amazing how many test databases live for a very long time and start getting more and more critical. And will you remember to remove that user with the weak password after you are done?

Enough lecturing. Databases need data. Let's add a table to the `hadooptest` database that we'll use throughout this chapter.

Time for action – setting up the employee database

No discussion of databases is complete without the example of an employee table, so we will follow tradition and start there.

1. Create a tab-separated file named `employees.tsv` with the following entries:

```
Alice  Engineering  50000  2009-03-12
Bob    Sales  35000  2011-10-01
Camille  Marketing  40000  2003-04-20
David  Executive  75000  2001-03-20
Erica  Support  34000  2011-07-07
```

2. Connect to the MySQL server:

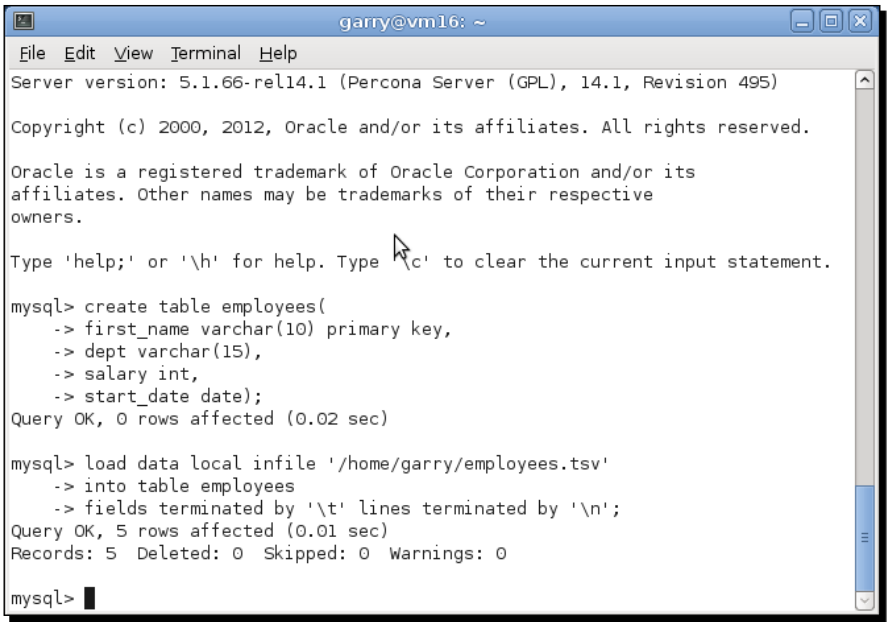
```
$ mysql -u hadoopuser -p hadooptest
```

3. Create the table:

```
Mysql> create table employees(
first_name varchar(10) primary key,
dept varchar(15),
salary int,
start_date date
) ;
```

4. Load the data from the file into the database:

```
mysql> load data local infile '/home/garry/employees.tsv'
    -> into table employees
    -> fields terminated by '\t' lines terminated by '\n' ;
```



What just happened?

This is pretty standard database stuff. We created a tab-separated data file, created the table in the database, and then used the `LOAD DATA LOCAL INFILE` statement to import the data into the table.

We are using a very small set of data here as it is really for illustration purposes only.

Be careful with data file access rights

Don't omit the `LOCAL` part from the `LOAD DATA` statement; doing so sees MySQL try and load the file as the MySQL user, and this usually results in access problems.

Getting data into Hadoop

Now that we have put in all that up-front effort, let us look at ways of bringing the data out of MySQL and into Hadoop.

Using MySQL tools and manual import

The simplest way to export data into Hadoop is to use existing command-line tools and statements. To export an entire table (or indeed an entire database), MySQL offers the `mysqldump` utility. To do a more precise export, we can use a `SELECT` statement of the following form:

```
SELECT col1, col2 from table
INTO OUTFILE '/tmp/out.csv'
FIELDS TERMINATED by ',', LINES TERMINATED BY '\n';
```

Once we have an export file, we can move it into HDFS using `hadoop fs -put` or into Hive through the methods discussed in the previous chapter.

Have a go hero – exporting the employee table into HDFS

We don't want this chapter to turn into a MySQL tutorial, so look up the syntax of the `mysqldump` utility, and use it or the `SELECT … INTO OUTFILE` statement to export the employee table into a tab-separated file you then copy onto HDFS.

Accessing the database from the mapper

For our trivial example, the preceding approaches are fine, but what if you need to export a much larger set of data, especially if it then is to be processed by a MapReduce job?

The obvious approach is that of direct JDBC access within a MapReduce input job that pulls the data from the database and writes it onto HDFS, ready for additional processing.

This is a valid technique, but there are a few not-so-obvious gotchas.

You need to be careful how much load you place on the database. Throwing this sort of job onto a very large cluster could very quickly melt the database as hundreds or thousands of mappers try to simultaneously open connections and read the same table. The simplest access pattern is also likely to see one query per row, which obviates the ability to use more efficient bulk access statements. Even if the database can take the load, it is quite possible for the database network connection to quickly become the bottleneck.

To effectively parallelize the query across all the mappers, you need a strategy to partition the table into segments each mapper will retrieve. You then need to determine how each mapper is to have its segment parameters passed in.

If the retrieved segments are large, there is a chance that you will end up with long-running tasks that get terminated by the Hadoop framework unless you explicitly report progress.

That is actually quite a lot of work for a conceptually simple task. Wouldn't it be much better to use an existing tool for the purpose? There is indeed such a tool that we will use throughout the rest of this chapter, Sqoop.

A better way – introducing Sqoop

Sqoop was created by Cloudera (`http://www.cloudera.com`), a company that provides numerous services related to Hadoop in addition to producing its own packaging of the Hadoop distribution, something we will discuss in *Chapter 11*, *Where to Go Next*.

As well as providing this packaged Hadoop product, the company has also created a number of tools that have been made available to the community, and one of these is Sqoop. Its job is to do exactly what we need, to copy data between Hadoop and relational databases. Though originally developed by Cloudera, it has been contributed to the Apache Software Foundation, and its homepage is `http://sqoop.apache.org`.

Time for action – downloading and configuring Sqoop

Let's download and get Sqoop installed and configured.

1. Go to the Sqoop homepage, select the link for the most stable version that is no earlier than 1.4.1, and match it with the version of Hadoop you are using. Download the file.
2. Copy the retrieved file where you want it installed on your system; then uncompress it:

```
$mv sqoop-1.4.1-incubating__hadoop-1.0.0.tar.gz_ /usr/local
$ cd /usr/local
$ tar –xzf sqoop-1.4.1-incubating__hadoop-1.0.0.tar.gz_
```

3. Make a symlink:

```
$ ln -s sqoop-1.4.1-incubating__hadoop-1.0.0 sqoop
```

4. Update your environment:

```
$ export SQOOP_HOME=/usr/local/sqoop
$ export PATH=${SQOOP_HOME}/bin:${PATH}
```

5. Download the JDBC driver for your database; for MySQL, we find it at `http://dev.mysql.com/downloads/connector/j/5.0.html`.
6. Copy the downloaded JAR file into the Sqoop `lib` directory:

```
$ cp mysql-connector-java-5.0.8-bin.jar /opt/sqoop/lib
```

7. Test Sqoop:

```
$ sqoop help
```

You will see the following output:

```
usage: sqoop COMMAND [ARGS]
Available commands:
  codegen            Generate code to interact with database
records
…
  version            Display version information

See 'sqoop help COMMAND' for information on a specific command.
```

What just happened?

Sqoop is a pretty straightforward tool to install. After downloading the required version from the Sqoop homepage—being careful to pick the one that matches our Hadoop version—we copied and unpacked the file.

Once again, we needed to set an environment variable and added the Sqoop `bin` directory to our path so we can either set these directly in our shell, or as before, add these steps to a configuration file we can source prior to a development session.

Sqoop needs access to the JDBC driver for your database; for us, we downloaded the MySQL Connector and copied it into the Sqoop `lib` directory. For the most popular databases, this is as much configuration as Sqoop requires; if you want to use something exotic, consult the Sqoop documentation.

After this minimal install, we executed the `sqoop` command-line utility to validate that it is working properly.



You may see warning messages from Sqoop telling you that additional variables such as `HBASE_HOME` have not been defined. As we are not talking about HBase in this book, we do not need this setting and will be omitting such warnings from our screenshots.

Sqoop and Hadoop versions

We were very specific in the version of Sqoop to be retrieved before; much more so than for previous software downloads. In Sqoop versions prior to 1.4.1, there is a dependency on an additional method on one of the core Hadoop classes that was only available in the Cloudera Hadoop distribution or versions of Hadoop after 0.21.

Unfortunately, the fact that Hadoop 1.0 is effectively a continuation of the 0.20 branch meant that Sqoop 1.3, for example, would work with Hadoop 0.21 but not 0.20 or 1.0. To avoid this version confusion, we recommend using version 1.4.1 or later, which removes the dependency.

There is no additional MySQL configuration required; we would discover if the server had not been configured to allow remote clients, as described earlier, through use of Sqoop.

Sqoop and HDFS

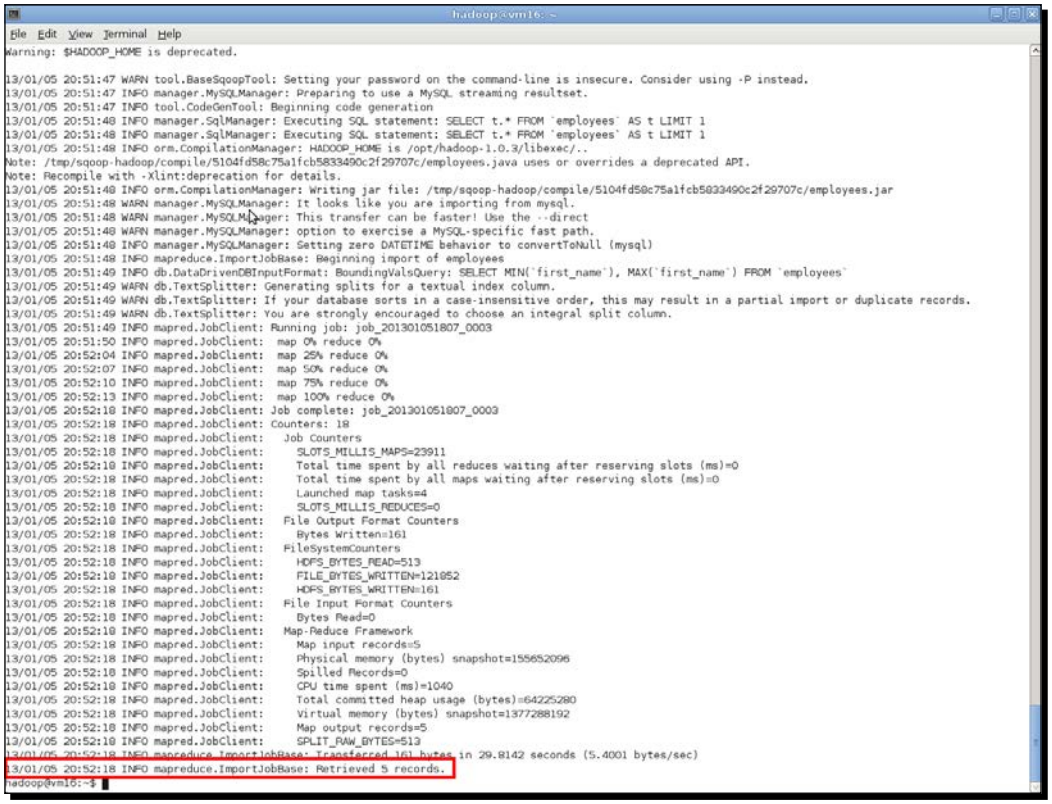
The simplest import we can perform is to dump data from a database table onto structured files on HDFS. Let's do that.

Time for action – exporting data from MySQL to HDFS

We'll use a straightforward example here, where we just pull all the data from a single MySQL table and write it to a single file on HDFS.

1. Run Sqoop to export data from MySQL onto HDFS:

```
$ sqoop import --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser \ > --password password --table employees
```



2. Examine the output directory:

```
$ hadoop fs -ls employees
```

You will receive the following response:

```
Found 6 items
-rw-r--r--   3 hadoop supergroup          0 2012-05-21 04:10 /
user/hadoop/employees/_SUCCESS
drwxr-xr-x   - hadoop supergroup          0 2012-05-21 04:10 /
user/hadoop/employees/_logs
-rw-r--r--   3 … /user/hadoop/employees/part-m-00000
-rw-r--r--   3 … /user/hadoop/employees/part-m-00001
-rw-r--r--   3 … /user/hadoop/employees/part-m-00002
-rw-r--r--   3 … /user/hadoop/employees/part-m-00003
```

3. Display one of the result files:

```
$ hadoop fs -cat /user/hadoop/employees/part-m-00001
```

You will see the following output:

```
Bob,Sales,35000,2011-10-01
Camille,Marketing,40000,2003-04-20
```

What just happened?

We did not need any preamble; a single Sqoop statement is all we require here. As can be seen, the Sqoop command line takes many options; let's unpack them one at a time.

The first option in Sqoop is the type of task to be performed; in this case, we wish to import data from a relational source into Hadoop. The `--connect` option specifies the JDBC URI for the database, of the standard form `jdbc:<driver>://<host>/<database>`. Obviously, you need to change the IP or hostname to the server where your database is running.

We use the `--username` and `--password` options to specify those attributes and finally use `--table` to indicate from which table we wish to retrieve the data. That is it! Sqoop does the rest.

The Sqoop output is relatively verbose, but do read it as it gives a good idea of exactly what is happening.



Repeated executions of Sqoop may however include a nested error about a generated file already existing. Ignore that for now.

Firstly, in the preceding steps, we see Sqoop telling us not to use the `--password` option as it is inherently insecure. Sqoop has an alternative `-P` command, which prompts for the password; we will use that in future examples.

We also get a warning about using a textual primary key column and that it's a very bad idea; more on that in a little while.

After all the setup and warnings, however, we see Sqoop execute a MapReduce job and complete it successfully.

By default, Sqoop places the output files into a directory in the home directory of the user who ran the job. The files will be in a directory of the same name as the source table. To verify this, we used `hadoop fs -ls` to check this directory and confirmed that it contained several files, likely more than we would have expected, given such a small table. Note that we slightly abbreviated the output here to allow it to fit on one line.

We then examined one of the output files and discovered the reason for the multiple files; even though the table is tiny, it was still split across multiple mappers, and hence, output files. Sqoop uses four map tasks by default. It may look a little strange in this case, but the usual situation will be a much larger data import. Given the desire to copy data onto HDFS, this data is likely to be the source of a future MapReduce job, so multiple files makes perfect sense.

Mappers and primary key columns

We intentionally set up this situation by somewhat artificially using a textual primary key column in our employee data set. In reality, the primary key would much more likely be an auto-incrementing, numeric employee ID. However, this choice highlighted the nature of how Sqoop processes tables and its use of primary keys.

Sqoop uses the primary key column to determine how to divide the source data across its mappers. But, as the warnings before state, this means we are reliant on string-based comparisons, and in an environment with imperfect case significance, the results may be incorrect. The ideal situation is to use a numeric column as suggested.

Alternatively, it is possible to control the number of mappers using the `-m` option. If we use `-m 1`, there will be a single mapper and no attempt will be made to partition the primary key column. For small data sets such as ours, we can also do this to ensure a single output file.

This is not just an option; if you try to import from a table with no primary key, Sqoop will fail with an error stating that the only way to import from such a table is to explicitly set a single mapper.

Other options

Don't assume that Sqoop is all or nothing when it comes to importing data. Sqoop has several other options to specify, restrict, and alter the data extracted from the database. We will illustrate these in the following sections, where we discuss Hive, but bear in mind that most can also be used when exporting into HDFS.

Sqoop's architecture

Now that we have seen Sqoop in action, it is worthwhile taking a few moments to clarify its architecture and see how it works. In several ways, Sqoop interacts with Hadoop in much the same way that Hive does; both are single client programs that create one or more MapReduce jobs to perform their tasks.

Sqoop does not have any server processes; the command-line client we run is all there is to it. However, because it can tailor its generated MapReduce code to the specific tasks at hand, it tends to utilize Hadoop quite efficiently.

The preceding example of splitting a source RDBMS table on a primary key is a good example of this. Sqoop knows the number of mappers that will be configured in the MapReduce job—the default is four, as previously mentioned—and from this, it can do smart partitioning of the source table.

If we assume a table with 1 million records and four mappers, then each will process 2,50,000 records. With its knowledge of the primary key column, Sqoop can create four SQL statements to retrieve the data that each use the desired primary key column range as caveats. In the simplest case, this could be as straightforward as adding something like `WHERE id BETWEEN 1 and 250000` to the first statement and using different `id` ranges for the others.

We will see the reverse behavior when exporting data from Hadoop as Sqoop again parallelizes data retrieval across multiple mappers and works to optimize the insertion of this data into the relational database. However, all these smarts are pushed into the MapReduce jobs executed on Hadoop; the Sqoop command-line client's job is to generate this code as efficiently as possible and then get out of the way as the processing occurs.

Importing data into Hive using Sqoop

Sqoop has significant integration with Hive, allowing it to import data from a relational source into either new or existing Hive tables. There are multiple ways in which this process can be tailored, but again, let's start with the simple case.

Time for action – exporting data from MySQL into Hive

For this example, we'll export all the data from a single MySQL table into a correspondingly named table in Hive. You will need Hive installed and configured as detailed in the previous chapter.

1. Delete the output directory created in the previous section:

   ```
   $ hadoop fs -rmr employees
   ```

   You will receive the following response:

   ```
   Deleted hdfs://head:9000/user/hadoop/employees
   ```

2. Confirm Hive doesn't already contain an employees table:

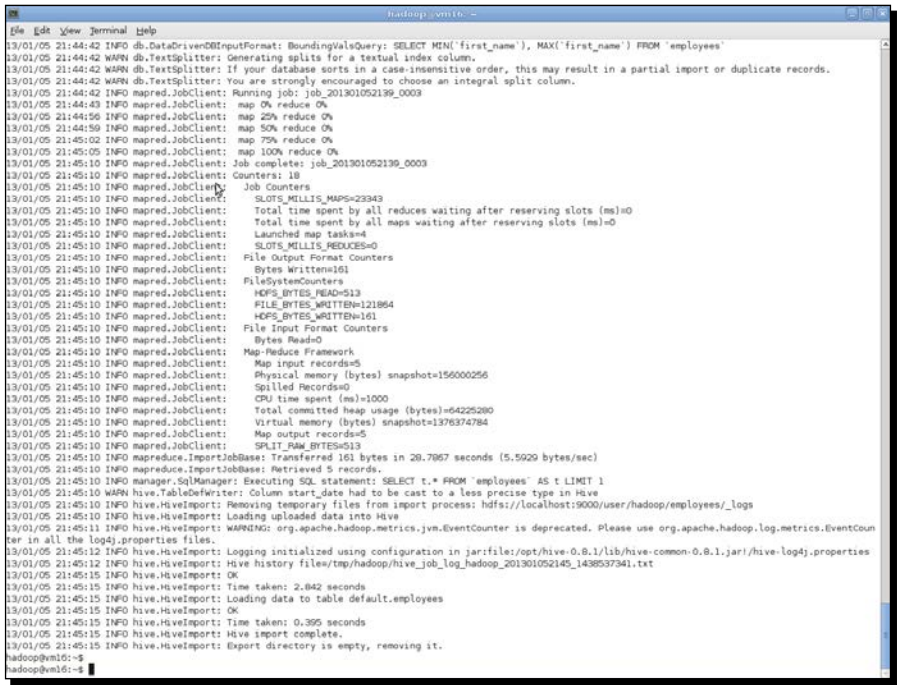
   ```
   $ hive -e "show tables like 'employees'"
   ```

   You will receive the following response:

   ```
   OK
   Time taken: 2.318 seconds
   ```

3. Perform the Sqoop import:

   ```
   $ sqoop import --connect jdbc:mysql://10.0.0.100/hadooptest
   --username hadoopuser -P
   --table employees --hive-import --hive-table employees
   ```



4. Check the contents in Hive:

```
$ hive -e "select * from employees"
```

You will receive the following response:

```
OK
Alice  Engineering  50000  2009-03-12
Camille  Marketing  40000  2003-04-20
David  Executive  75000  2001-03-20
Erica  Support  34000  2011-07-07
Time taken: 2.739 seconds
```

5. Examine the created table in Hive:

```
$ hive -e "describe employees"
```

You will receive the following response:

```
OK
first_name  string
dept  string
salary  int
start_date  string
Time taken: 2.553 seconds
```

What just happened?

Again, we use the Sqoop command with two new options, `--hive-import` to tell Sqoop the final destination is Hive and not HDFS, and `--hive-table` to specify the name of the table in Hive where we want the data imported.

In actuality, we don't need to specify the name of the Hive table if it is the same as the source table specified by the `--table` option. However, it does make things more explicit, so we will typically include it.

As before, do read the full Sqoop output as it provides great insight into what's going on, but the last few lines highlight the successful import into the new Hive table.

We see Sqoop retrieving five rows from MySQL and then going through the stages of copying them to HDFS and importing into Hive. We will talk about the warning re type conversions next.

After Sqoop completes the process, we use Hive to retrieve the data from the new Hive table and confirm that it is what we expected. Then, we examine the definition of the created table.

At this point, we do see one strange thing; the `start_date` column has been given a type string even though it was originally a SQL `DATE` type in MySQL.

The warning we saw during the Sqoop execution explains this situation:

```
12/05/23 13:06:33 WARN hive.TableDefWriter: Column start_date had to be
cast to a less precise type in Hive
```

The cause of this is that Hive does not support any temporal datatype other than `TIMESTAMP`. In those cases where imported data is of another type, relating to dates or times, Sqoop converts it to a string. We will look at a way of dealing with this situation a little later.

This example is a pretty common situation, but we do not always want to import an entire table into Hive. Sometimes, we want to only include particular columns or to apply a predicate to reduce the number of selected items. Sqoop allows us to do both.

Time for action – a more selective import

Let's see how this works by performing an import that is limited by a conditional expression.

1. Delete any existing employee import directory:

   ```
   $ hadoop fs -rmr employees
   ```

   You will receive the following response:

   ```
   Deleted hdfs://head:9000/user/hadoop/employees
   ```

2. Import selected columns with a predicate:

   ```
   sqoop import --connect jdbc:mysql://10.0.0.100/hadooptest
   --username hadoopuser -P
   --table employees --columns first_name,salary
    --where "salary > 45000"
   --hive-import --hive-table salary
   ```

   You will receive the following response:

   ```
   12/05/23 15:02:03 INFO hive.HiveImport: Hive import complete.
   ```

3. Examine the created table:

   ```
   $ hive -e "describe salary"
   ```

   You will receive the following response:

   ```
   OK
   first_name  string
   ```

```
salary  int
Time taken: 2.57 seconds
```

4. Examine the imported data:

```
$ hive -e "select * from salary"
```

You will see the following output:

```
OK
Alice  50000
David  75000
Time taken: 2.754 seconds
```

What just happened?

This time, our Sqoop command first added the `--columns` option that specifies which columns to include in the import. This is a comma-separated list.

We also used the `--where` option that allows the free text specification of a `WHERE` clause that is applied to the SQL used to extract data from the database.

The combination of these options is that our Sqoop command should import only the names and salaries of those with a salary greater than the threshold specified in the `WHERE` clause.

We execute the command, see it complete successfully, and then examine the table created in Hive. We see that it indeed only contains the specified columns, and we then display the table contents to verify that the where predicate was also applied correctly.

Datatype issues

In *Chapter 8*, *A Relational View on Data with Hive*, we mentioned that Hive does not support all the common SQL datatypes. The `DATE` and `DATETIME` types in particular are not currently implemented though they do exist as identified Hive issues; so hopefully, they will be added in the future. We saw this impact our first Hive import earlier in this chapter. Though the `start_date` column was of type `DATE` in MySQL, the Sqoop import flagged a conversion warning, and the resultant column in Hive was of type `STRING`.

Sqoop has an option that is of use here, that is, we can use `--map-column-hive` to explicitly tell Sqoop how to create the column in the generated Hive table.

Time for action – using a type mapping

Let's use a type mapping to improve our data import.

1. Delete any existing output directory:

```
$ hadoop fs -rmr employees
```

2. Execute Sqoop with an explicit type mapping:

```
sqoop import --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser
-P --table employees
--hive-import --hive-table employees
--map-column-hive start_date=timestamp
```

   You will receive the following response:

```
12/05/23 14:53:38 INFO hive.HiveImport: Hive import complete.
```

3. Examine the created table definition:

```
$ hive -e "describe employees"
```

   You will receive the following response:

```
OK
first_name  string
dept  string
salary  int
start_date  timestamp
Time taken: 2.547 seconds
```

4. Examine the imported data:

```
$ hive -e "select * from employees";
```

   You will receive the following response:

```
OK
Failed with exception java.io.IOException:java.lang.
IllegalArgumentException: Timestamp format must be yyyy-mm-dd
hh:mm:ss[.fffffffff]
Time taken: 2.73 seconds
```

What just happened?

Our Sqoop command line here is similar to our original Hive import, except for the addition of the column mapping specification. We specified that the `start_date` column should be of type `TIMESTAMP`, and we could have added other specifications. The option takes a comma-separated list of such mappings.

After confirming Sqoop executed successfully, we examined the created Hive table and verified that the mapping was indeed applied and that the `start_date` column has type `TIMESTAMP`.

We then tried to retrieve the data from the table and could not do so, receiving an error about type format mismatch.

On reflection, this should not be a surprise. Though we specified the desired column type was to be `TIMESTAMP`, the actual data being imported from MySQL was of type `DATE`, which does not contain the time component required in a timestamp. This is an important lesson. Ensuring that the type mappings are correct is only one part of the puzzle; we must also ensure the data is valid for the specified column type.

Time for action – importing data from a raw query

Let's see an example of an import where a raw SQL statement is used to select the data to be imported.

1. Delete any existing output directory:

```
$ hadoop fs –rmr employees
```

2. Drop any existing Hive employee table:

```
$ hive -e 'drop table employees'
```

3. Import data using an explicit query:

```
sqoop import --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser -P
--target-dir employees
--query 'select first_name, dept, salary,
timestamp(start_date) as start_date from employees where
$CONDITIONS'
--hive-import --hive-table employees
--map-column-hive start_date=timestamp -m 1
```

4. Examine the created table:

```
$ hive -e "describe employees"
```

You will receive the following response:

```
OK
first_name  string
dept  string
salary  int
start_date  timestamp
Time taken: 2.591 seconds
```

5. Examine the data:

```
$ hive -e "select * from employees"
```

You will receive the following response:

```
OK
Alice  Engineering  50000  2009-03-12 00:00:00
Bob    Sales  35000  2011-10-01 00:00:00
Camille  Marketing  40000  2003-04-20 00:00:00
David  Executive  75000  2001-03-20 00:00:00
Erica  Support  34000  2011-07-07 00:00:00
Time taken: 2.709 seconds
```

What just happened?

To achieve our goal, we used a very different form of the Sqoop import. Instead of specifying the desired table and then either letting Sqoop import all columns or a specified subset, here we use the `--query` option to define an explicit SQL statement.

In the statement, we select all the columns from the source table but apply the `timestamp()` function to convert the `start_date` column to the correct type. (Note that this function simply adds a `00:00` time element to the date). We alias the result of this function, which allows us to name it in the type mapping option.

Because we have no `--table` option, we have to add `--target-dir` to tell Sqoop the name of the directory it should create on HDFS.

The `WHERE` clause in the SQL is required by Sqoop even though we are not actually using it. Having no `--table` option does not just remove Sqoop's ability to auto-generate the name of the export directory, it also means that Sqoop does not know from where data is being retrieved, and hence, how to partition the data across multiple mappers. The `$CONDITIONS` variable is used in conjunction with a `--where` option; specifying the latter provides Sqoop with the information it needs to partition the table appropriately.

We take a different route here and instead explicitly set the number of mappers to `1`, which obviates the need for an explicit partitioning clause.

After executing Sqoop, we examine the table definition in Hive, which as before, has the correct datatypes for all columns. We then look at the data, and this is now successful, with the `start_date` column data being appropriately converted into the `TIMESTAMP` values.



When we mentioned in the *Sqoop and HDFS* section that Sqoop provided mechanisms to restrict the data extracted from the database, we were referring to the `query`, `where`, and `columns` options. Note that these can be used by any Sqoop import regardless of the destination.

Have a go hero

Though it truly is not needed for such a small data set, the `$CONDITIONS` variable is an important tool. Modify the preceding Sqoop statement to use multiple mappers with an explicit partitioning statement.

Sqoop and Hive partitions

In *Chapter 8*, *A Relational View on Data with Hive*, we talked a lot about Hive partitions and highlighted how important they are in allowing query optimization for very large tables. The good news is that Sqoop can support Hive partitions; the bad news is that the support is not complete.

To import data from a relational database into a partitioned Hive table, we use the `--hive-partition-key` option to specify the partition column and the `--hive-partition-value` option to specify the value for the partition into which this Sqoop command will import data.

This is excellent but does require each Sqoop statement to be imported into a single Hive partition; there is currently no support for Hive auto-partitioning. Instead, if a data set is to be imported into multiple partitions in a table, we need use a separate Sqoop statement for insertion into each partition.

Field and line terminators

Until now, we have been implicitly relying on some defaults but should discuss them at this point. Our original text file was tab separated, but you may have noticed that the data we exported onto HDFS was comma-separated. If you go look in the files under `/user/hive/warehouse/employees` (remember this is the default location on HDFS where Hive keeps its source files), the records use ASCII code 001 as the separator. What is going on?

In the first instance, we let Sqoop use its defaults, which in this case, means using a comma to separate fields and using `\n` for records. However, when Sqoop is importing into Hive, it instead employs the Hive defaults, which include using the 001 code (^A) to separate fields.

We can explicitly set separators using the following Sqoop options:

- `fields-terminated-by`: This is the separator between fields
- `lines-terminated-by`: The line terminator
- `escaped-by`: Used to escape characters (for example, \)
- `enclosed-by`: The character enclosing fields (for example, ")
- `optionally-enclosed-by`: Similar to the preceding option but not mandatory
- `mysql-delimiters`: A shortcut to use the MySQL defaults

This may look a little intimidating, but it's not as obscure as the terminology may suggest, and the concepts and syntax should be familiar to those with SQL experience. The first few options are pretty self-explanatory; where it gets less clear is when talking of enclosing and optionally enclosing characters.

This is really about (usually free-form) data where a given field may include characters that have special meanings. For example, a string column in a comma-separated file that includes commas. In such a case, we could enclose the string columns within quotes to allow the commas within the field. If all fields need such enclosing characters, we would use the first form; if it was only required for a subset of the fields, it could be specified as optional.

Getting data out of Hadoop

We said that the data flow between Hadoop and a relational database is rarely a linear single direction process. Indeed the situation where data is processed within Hadoop and then inserted into a relational database is arguably the more common case. We will explore this now.

Writing data from within the reducer

Thinking about how to copy the output of a MapReduce job into a relational database, we find similar considerations as when looking at the question of data import into Hadoop.

The obvious approach is to modify a reducer to generate the output for each key and its associated values and then to directly insert them into a database via JDBC. We do not have to worry about source column partitioning, as with the import case, but do still need to think about how much load we are placing on the database and whether we need to consider timeouts for long-running tasks. In addition, just as with the mapper situation, this approach tends to perform many single queries against the database, which is typically much less efficient than bulk operations.

Writing SQL import files from the reducer

Often, a superior approach is not to work around the usual MapReduce case of generating output files, as with the preceding example, but instead to exploit it.

All relational databases have the ability to ingest data from source files, either through custom tools or through the use of the `LOAD DATA` statement. Within the reducer, therefore, we can modify the data output to make it more easily ingested into our relational destination. This obviates the need to consider issues such as reducers placing load on the database or how to handle long-running tasks, but it does require a second step external to our MapReduce job.

A better way – Sqoop again

It probably won't come as a surprise—certainly not if you've looked at the output of Sqoop's inbuilt help or its online documentation—to learn that Sqoop can also be our tool of choice for data export from Hadoop.

Time for action – importing data from Hadoop into MySQL

Let's demonstrate this by importing data into a MySQL table from an HDFS file.

1. Create a tab-separated file named `newemployees.tsv` with the following entries:

```
Frances  Operations  34000  2012-03-01
Greg  Engineering  60000  2003-11-18
Harry  Intern  22000  2012-05-15
Iris  Executive  80000  2001-04-08
Jan  Support  28500  2009-03-30
```

2. Create a new directory on HDFS and copy the file into it:

```
$hadoop fs -mkdir edata
$ hadoop fs -put newemployees.tsv edata/newemployees.tsv
```

3. Confirm the current number of records in the employee table:

```
$ echo "select count(*) from employees" |
```

```
mysql –u hadoopuser –p hadooptest
```

You will receive the following response:

```
Enter password:
count(*)
5
```

4. Run a Sqoop export:

```
$ sqoop export --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser  -P --table employees
--export-dir edata --input-fields-terminated-by '\t'
```

You will receive the following response:

```
12/05/27 07:52:22 INFO mapreduce.ExportJobBase: Exported 5
records.
```

5. Check the number of records in the table after the export:

```
Echo "select count(*) from employees"
| mysql -u hadoopuser -p hadooptest
```

You will receive the following response:

```
Enter password:
count(*)
10
```

6. Check the data:

```
$ echo "select * from employees"
| mysql -u hadoopuser -p hadooptest
```

You will receive the following response:

```
Enter password:
first_name  dept  salary  start_date
Alice  Engineering  50000  2009-03-12
…
Frances  Operations  34000  2012-03-01
Greg  Engineering  60000  2003-11-18
Harry  Intern  22000  2012-05-15
Iris  Executive  80000  2001-04-08
Jan  Support  28500  2009-03-30
```

What just happened?

We first created a data file containing information on five more employees. We created a directory for our data on HDFS into which we copied the new file.

Before running the export, we confirmed that the table in MySQL contained the original five employees only.

The Sqoop command has a similar structure as before with the biggest change being the use of the `export` command. As the name suggests, Sqoop exported export data from Hadoop into a relational database.

We used several similar options as before, mainly to specify the database connection, the username and password needed to connect, and the table into which to insert the data.

Because we are exporting data from HDFS, we needed to specify the location containing any files to be exported which we do via the `--export-dir` option. All files contained within the directory will be exported; they do not need be in a single file; Sqoop will include all files within its MapReduce job. By default, Sqoop uses four mappers; if you have a large number of files it may be more effective to increase this number; do test, though, to ensure that load on the database remains under control.

The final option passed to Sqoop specified the field terminator used in the source files, in this case, the tab character. It is your responsibility to ensure the data files are properly formatted; Sqoop will assume there is the same number of elements in each record as columns in the table (though null is acceptable), separated by the specified field separator character.

After watching the Sqoop command complete successfully, we saw it reports that it exported five records. We check, using the `mysql` tool, the number of rows now in the database and then view the data to confirm that our old friends are now joined by the new employees.

Differences between Sqoop imports and exports

Though similar conceptually and in the command-line invocations, there are a number of important differences between Sqoop imports and exports that are worth exploring.

Firstly, Sqoop imports can assume much more about the data being processed; through either explicitly named tables or added predicates, there is much information about both the structure and type of the data. Sqoop exports, however, are given only a location of source files and the characters used to separate and enclose fields and records. While Sqoop imports into Hive can automatically create a new table based on the provided table name and structure, a Sqoop export must be into an existing table in the relational database.

Even though our earlier demonstration with dates and timestamps showed there are some sharp edges, Sqoop imports are also able to determine whether the source data complies with the defined column types; the data would not have been possible to insert into the database otherwise. Sqoop exports again only have access effectively to fields of characters with no understanding of the real datatype. If you have the luxury of very clean and well-formatted data, this may never matter, but for the rest of us, there will be a need to consider data exports and type conversions, particularly in terms of null and default values. The Sqoop documentation goes into these options in some detail and is worth a read.

Inserts versus updates

Our preceding example was very straightforward; we added an entire new set of data that can happily coexist with the existing contents of the table. Sqoop exports by default do a series of appends, adding each record as a new row in the table.

However, what if we later want to update data when, for example, our employees get increased salaries at the end of the year? With the database table defining `first_name` as a primary key, any attempt to insert a new row with the same name as an existing employee will fail with a failed primary key constraint.

In such cases, we can set the Sqoop `--update-key` option to specify the primary key, and Sqoop will generate `UPDATE` statements based on this key (it can be a comma-separated list of keys), as opposed to `INSERT` statements adding new rows.



In this mode, any record that does not match an existing key value will silently be ignored, and Sqoop will not flag errors if a statement updates more than one row.

If we also want the option of an update that adds new rows for non-existing data, we can set the `--update-mode` option to `allowinsert`.

Have a go hero

Create another data file that contains three new employees as well as updated salaries for two of the existing employees. Use Sqoop in import mode to both add the new employees as well as apply the needed updates.

Sqoop and Hive exports

Given the preceding example, it may not be surprising to learn that Sqoop does not currently have any direct support to export a Hive table into a relational database. More precisely, there are no explicit equivalents to the `--hive-import` option we used earlier.

However, in some cases, we can work around this. If a Hive table is storing its data in text format, we could point Sqoop at the location of the table data files on HDFS. In case of tables referring to external data, this may be straightforward, but once we start seeing Hive tables with complex partitioning, the directory structure becomes more involved.

Hive can also store tables as binary SequenceFiles, and a current limitation is that Sqoop cannot transparently export from tables stored in this format.

Time for action – importing Hive data into MySQL

Regardless of these limitations, let's demonstrate that, in the right situations, we can use Sqoop to directly export data stored in Hive.

1. Remove any existing data in the employee table:

```
$ echo "truncate employees" | mysql –u hadoopuser –p hadooptest
```

   You will receive the following response:

```
Query OK, 0 rows affected (0.01 sec)
```

2. Check the contents of the Hive warehouse for the employee table:

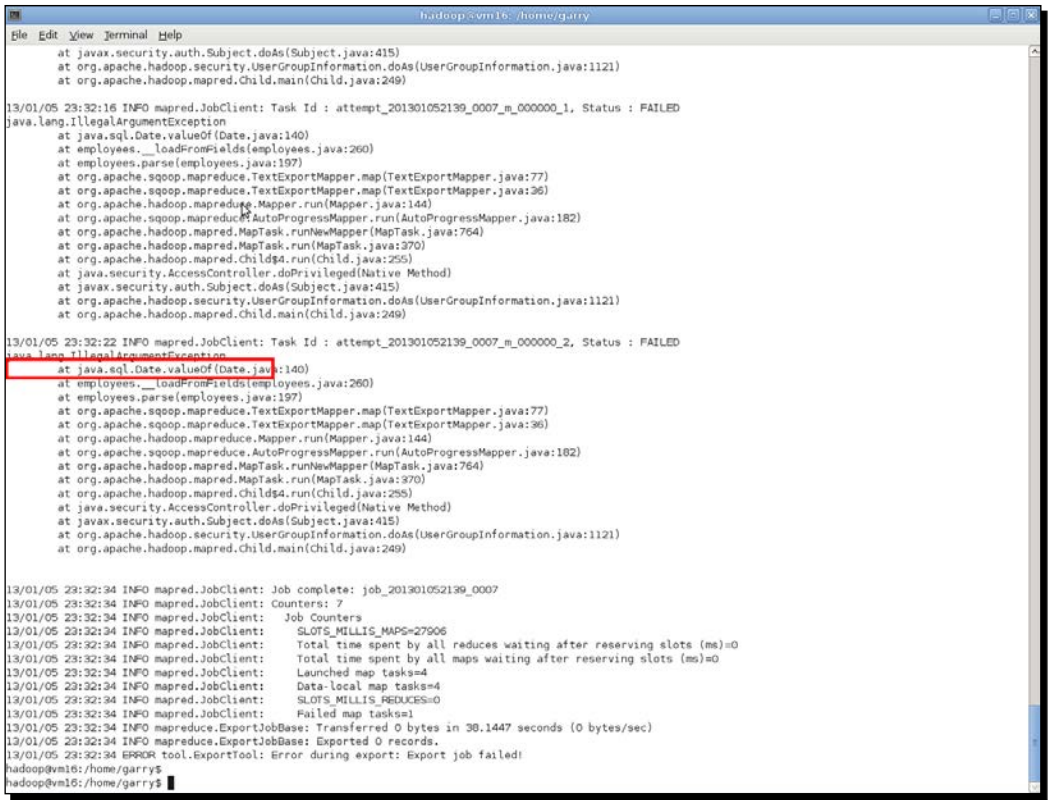
```
$ hadoop fs –ls /user/hive/warehouse/employees
```

   You will receive the following response:

```
Found 1 items
… /user/hive/warehouse/employees/part-m-00000
```

3. Perform the Sqoop export:

```
sqoop export --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser –P --table employees \
--export-dir /user/hive/warehouse/employees
--input-fields-terminated-by '\001'
--input-lines-terminated-by '\n'
```



What just happened?

Firstly, we truncated the `employees` table in MySQL to remove any existing data and then confirmed the employee table data was where we expected it to be.



Note that Sqoop may also create an empty file in this directory with the suffix `_SUCCESS`; if this is present it should be deleted before running the Sqoop export.

The Sqoop `export` command is like before; the only changes are the different source location for the data and the addition of explicit field and line terminators. Recall that Hive, by default, uses ASCII code 001 and `\n` for its field and line terminators, respectively (also recall, though, that we have previously imported files into Hive with other separators, so this is something that always needs to be checked).

We execute the Sqoop command and watch it fail due to `Java IllegalArgumentExceptions` when trying to create instances of `java.sql.Date`.

We are now hitting the reverse of the problem we encountered earlier; the original type in the source MySQL table had a datatype not supported by Hive, and we converted the data to match the available type of `TIMESTAMP`. When exporting data back again, however, we are now trying to create a `DATE` using a `TIMESTAMP` value, which is not possible without some conversion.

The lesson here is that our earlier approach of doing a one-way conversion only worked for as long as we only had data flowing in one direction. As soon as we need bi-directional data transfer, mismatched types between Hive and the relational store add complexity and require the insertion of conversion routines.

Time for action – fixing the mapping and re-running the export

In this case, however, let us do what probably makes more sense—modifying the definition of the employee table to make it consistent in both data sources.

1. Start the `mysql` utility:

```
$ mysql -u hadoopuser -p hadooptest
Enter password:
```

2. Change the type of the `start_date` column:

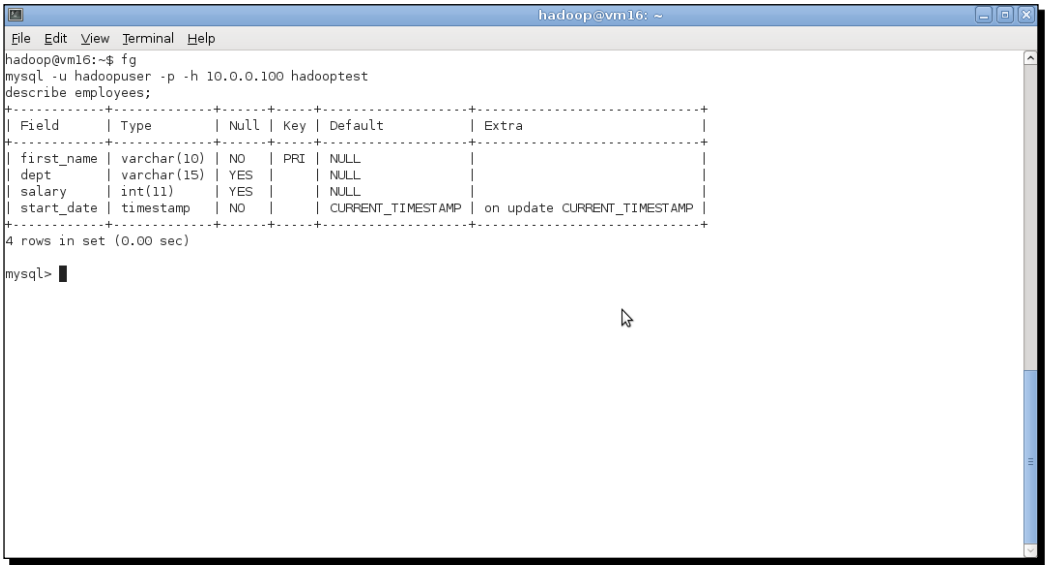
```
mysql> alter table employees modify column start_date timestamp;
```

   You will receive the following response:

```
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

3. Display the table definition:

```
mysql> describe employees;
```



4. Quit the `mysql` tool:

```
mysql> quit;
```

5. Perform the Sqoop export:

```
sqoop export --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser -P –table employees
--export-dir /user/hive/warehouse/employees
--input-fields-terminated-by '\001'
--input-lines-terminated-by '\n'
```

You will receive the following response:

```
12/05/27 09:17:39 INFO mapreduce.ExportJobBase: Exported 10
records.
```

6. Check the number of records in the MySQL database:

```
$ echo "select count(*) from employees"
| mysql -u hadoopuser -p hadooptest
```

You will receive the following output:

```
Enter password:
count(*)
10
```

What just happened?

Before trying the same Sqoop export as last time, we used the `mysql` tool to connect to the database and modify the type of the `start_date` column. Note, of course, that such changes should never be made casually on a production system, but given that we have a currently empty test table, there are no issues here.

After making the change, we re-ran the Sqoop export and this time it succeeded.

Other Sqoop features

Sqoop has a number of other features that we won't discuss in detail, but we'll highlight them so the interested reader can look them up in the Sqoop documentation.

Incremental merge

The examples we've used have been all-or-nothing processing that, in most cases, make the most sense when importing data into empty tables. There are mechanisms to handle additions, but if we foresee Sqoop performing ongoing imports, some additional support is available.

Sqoop supports the concept of incremental imports where an import task is additionally qualified by a date and only records more recent than that date are processed by the task. This allows the construction of long-running workflows that include Sqoop.

Avoiding partial exports

We've already seen how errors can occur when exporting data from Hadoop into a relational database. For us, it wasn't a significant problem as the issue caused all exported records to fail. But it isn't uncommon for only part of an export to fail and result in partially committed data in the database.

To mitigate this risk, Sqoop allows the use of a staging table; it loads all the data into this secondary table, and only after all data is successfully inserted, performs the move into the main table in a single transaction. This can be very useful for failure-prone workloads but does come with some important restrictions, such as the inability to support update mode. For very large imports, there are also performance and load impacts on the RDBMS of a single very long-running transaction.

Sqoop as a code generator

We've been ignoring an error during Sqoop processing that we casually brushed off a while ago—the exception thrown because the generated code required by Sqoop already exists.

When performing an import, Sqoop generates Java class files that provide a programmatic means of accessing the fields and records in the created files. Sqoop uses these classes internally, but they can also be used outside of a Sqoop invocation, and indeed, the Sqoop `codegen` command can regenerate the classes outside of an export task.

AWS considerations

We've not mentioned AWS so far in this chapter as there's been nothing in Sqoop that either supports or prevents its use on AWS. We can run Sqoop on an EC2 host as easily as on a local one, and it can access either a manually or EMR-created Hadoop cluster optionally running Hive. The only possible quirk when considering use in AWS is security group access as many default EC2 configurations will not allow traffic on the ports used by most relational databases (3306 by default for MySQL). But, that's no more of an issue than if our Hadoop cluster and MySQL database were to be located on different sides of a firewall or any other network security boundary.

Considering RDS

There is another AWS service that we've not mentioned before that does deserve an introduction now. Amazon **Relational Database Service** (**RDS**) offers hosted relational databases in the cloud and provides MySQL, Oracle, and Microsoft SQL Server options. Instead of having to worry about the installation, configuration, and management of a database engine, RDS allows an instance to be started from either the console or command-line tools. You then just point your database client tool at the database and start creating tables and manipulating data.

RDS and EMR are a powerful combination, providing hosted services that take much of the pain out of manually managing such services. If you need a relational database but don't want to worry about its management, RDS may be for you.

The RDS and EMR combination can be particularly powerful if you use EC2 hosts to generate data or store data in S3. Amazon has a general policy that there is no cost for data transfer from one service to another within a single region. Consequently, it's possible to have a fleet of EC2 hosts generating large data volumes that get pushed into a relational database in RDS for query access and are stored in EMR for archival and long-term analytics. Getting data into the storage and processing systems is often a technically challenging activity that can easily consume significant expense if the data needs be moved across commercial network links. Architectures built atop collaborating AWS services such as EC2, RDS, and EMR can minimize both these concerns.

Summary

In this chapter, we have looked at the integration of Hadoop and relational databases. In particular, we explored the most common use cases and saw that Hadoop and relational databases can be highly complimentary technologies. We considered ways of exporting data from a relational database onto HDFS files and realized that issues such as primary key column partitioning and long-running tasks make it harder than it first seems.

We then introduced Sqoop, a Cloudera tool now donated to the Apache Software Foundation and that provides a framework for such data migration. We used Sqoop to import data from MySQL into HDFS and then Hive, highlighting how we must consider aspects of datatype compatibility in such tasks. We also used Sqoop to do the reverse—copying data from HDFS into a MySQL database—and found out that this path has more subtle considerations than the other direction, briefly discussed issues of file formats and update versus insert tasks, and introduced additional Sqoop capabilities, such as code generation and incremental merging.

Relational databases are an important—often critical—part of most IT infrastructures. But, they aren't the only such component. One that has been growing in importance—often with little fanfare—is the vast quantities of log files generated by web servers and other applications. The next chapter will show how Hadoop is ideally suited to process and store such data.

10
Data Collection with Flume

In the previous two chapters, we've seen how Hive and Sqoop give a relational database interface to Hadoop and allow it to exchange data with "real" databases. Although this is a very common use case, there are, of course, many different types of data sources that we may want to get into Hadoop.

In this chapter, we will cover:

- An overview of data commonly processed in Hadoop
- Simple approaches to pull this data into Hadoop
- How Apache Flume can make this task a lot easier
- Common patterns for simple through sophisticated, Flume setups
- Common issues, such as the data lifecycle, that need to be considered regardless of technology

A note about AWS

This chapter will discuss AWS less than any other in the book. In fact, we won't even mention it after this section. There are no Amazon services akin to Flume so there is no AWS-specific product that we could explore. On the other hand, when using Flume, it works exactly the same, be it on a local host or EC2 virtual instance. The rest of this chapter, therefore, assumes nothing about the environment on which the examples are executed; they will perform identically on each.

Data data everywhere...

In discussions concerning integration of Hadoop with other systems, it is easy to think of it as a one-to-one pattern. Data comes out of one system, gets processed in Hadoop, and then is passed onto a third.

Things may be like that on day one, but the reality is more often a series of collaborating components with data flows passing back and forth between them. How we build this complex network in a maintainable fashion is the focus of this chapter.

Types of data

For the sake of the discussion, we will categorize data into two broad categories:

- **Network traffic**, where data is generated by a system and sent across a network connection
- **File data**, where data is generated by a system and written to files on a filesystem somewhere

We don't assume these data categories are different in any way other than how the data is retrieved.

Getting network traffic into Hadoop

When we say network data, we mean things like information retrieved from a web server via an HTTP connection, database contents pulled by a client application, or messages sent across a data bus. In each case, the data is retrieved by a client application that either pulls the data across the network or listens for its arrival.



In several of the following examples, we will use the `curl` utility to either retrieve or send network data. Ensure that it is installed on your system and install it if not.

Time for action – getting web server data into Hadoop

Let's take a look at how we can simplistically copy data from a web server onto HDFS.

1. Retrieve the text of the NameNode web interface to a local file:

```
$ curl localhost:50070 > web.txt
```

2. Check the file size:

```
$ ls -ldh web.txt
```

You will receive the following response:

```
-rw-r--r-- 1 hadoop hadoop 246 Aug 19 08:53 web.txt
```

3. Copy the file to HDFS:

```
$ hadoop fs -put web.txt web.txt
```

4. Check the file on HDFS:

```
$ hadoop fs -ls
```

You will receive the following response:

```
Found 1 items
-rw-r--r--   1 hadoop supergroup        246 2012-08-19 08:53 /
user/hadoop/web.txt
```

What just happened?

There shouldn't be anything that is surprising here. We use the `curl` utility to retrieve a web page from the embedded web server hosting the NameNode web interface and save it to a local file. We check the file size, copy it to HDFS, and verify the file has been transferred successfully.

The point of note here is not the series of actions—it is after all just another use of the `hadoop fs` command we have used since *Chapter 2*, *Getting Up and Running*—rather the pattern used is what we should discuss.

Though the data we wanted was in a web server and accessible via the HTTP protocol, the out of the box Hadoop tools are very file-based and do not have any intrinsic support for such remote information sources. This is why we need to copy our network data into a file before transferring it to HDFS.

We can, of course, write data directly to HDFS through the programmatic interface mentioned back in *Chapter 3*, *Writing MapReduce Jobs*, and this would work well. This would, however, require us to start writing custom clients for every different network source from which we need to retrieve data.

Have a go hero

Programmatically retrieving data and writing it to HDFS is a very powerful capability and worth some exploration. A very popular Java library for HTTP is the **Apache HTTPClient**, within the **HTTP Components** project found at `http://hc.apache.org/httpcomponents-client-ga/index.html`.

Use the HTTPClient and the Java HDFS interface to retrieve a web page as before and write it to HDFS.

Getting files into Hadoop

Our previous example showed the simplest method for getting file-based data into Hadoop and the use of the standard command-line tools or programmatic APIs. There is little else to discuss here, as it is a topic we have dealt with throughout the book.

Hidden issues

Though the preceding approaches are good as far as they go, there are several reasons why they may be unsuitable for production use.

Keeping network data on the network

Our model of copying network-accessed data to a file before placing it on HDFS will have an impact on performance. There is added latency due to the round-trip to disk, the slowest part of a system. This may not be an issue for large amounts of data retrieved in one call—though disk space potentially becomes a concern—but for small amounts of data retrieved at high speed, it may become a real problem.

Hadoop dependencies

For the file-based approach, it is implicit in the model mentioned before that the point at which we can access the file must have access to the Hadoop installation and be configured to know the location of the cluster. This potentially adds additional dependencies in the system; this could force us to add Hadoop to hosts that really need to know nothing about it. We can mitigate this by using tools like SFTP to retrieve the files to a Hadoop-aware machine and from there, copy onto HDFS.

Reliability

Notice the complete lack of error handling in the previous approaches. The tools we are using do not have built-in retry mechanisms which means we would need to wrap a degree of error detection and retry logic around each data retrieval.

Re-creating the wheel

This last point touches on perhaps the biggest issue with these ad hoc approaches; it is very easy to end up with a dozen different strings of command-line tools and scripts, each of which is doing very similar tasks. The potential costs in terms of duplicate effort and more difficult error tracking can be significant over time.

A common framework approach

Anyone with experience in enterprise computing will, at this point, be thinking that this sounds like a problem best solved with some type of common integration framework. This is exactly correct and is indeed a general type of product well known in fields such as **Enterprise Application Integration** (**EAI**).

What we need though is a framework that is Hadoop-aware and can easily integrate with Hadoop (and related projects) without requiring massive effort in writing custom adaptors. We could create our own, but instead let's look at **Apache Flume** which provides much of what we need.

Introducing Apache Flume

Flume, found at `http://flume.apache.org`, is another Apache project with tight Hadoop integration and we will explore it for the remainder of this chapter.

Before we explain what Flume can do, let's make it clear what it is not. Flume is described as a system for the retrieval and distribution of logs, meaning line-oriented textual data. It is not a generic data-distribution platform; in particular, don't look to use it for the retrieval or movement of binary data.

However, since the vast majority of the data processed in Hadoop matches this description, it is likely that Flume will meet many of your data retrieval needs.



Flume is also not a generic data serialization framework like **Avro** that we used in *Chapter 5*, *Advanced MapReduce Techniques*, or similar technologies such as **Thrift** and **Protocol Buffers**. As we'll see, Flume makes assumptions about the data format and provides no ways of serializing data outside of these.

Flume provides mechanisms for retrieving data from multiple sources, passing it to remote locations (potentially multiple locations in either a fan-out or pipeline model), and then delivering it to a variety of destinations. Though it does have a programmatic API that allows the development of custom sources and destinations, the base product has built-in support for many of the most common scenarios. Let's install it and take a look.

A note on versioning

Flume has gone through some major changes in recent times. The original Flume (now renamed **Flume OG** for Original Generation) is being superseded by **Flume NG** (**Next Generation**). Though the general principles and capabilities are very similar, the implementation is quite different.

Because Flume NG is the future, we will cover it in this book. For some time though, it will lack several of the features of the more mature Flume OG, so if you find a specific requirement that Flume NG doesn't meet then it may be worth looking at Flume OG.

Time for action – installing and configuring Flume

Let's get Flume downloaded and installed.

1. Retrieve the most recent Flume NG binary from `http://flume.apache.org/` and download and save it to the local filesystem.
2. Move the file to the desired location and uncompress it:

```
$ mv apache-flume-1.2.0-bin.tar.gz /opt
$ tar -xzf /opt/apache-flume-1.2.0-bin.tar.gz
```

3. Create a symlink to the installation:

```
$ ln -s /opt/apache-flume-1.2.0 /opt/flume
```

4. Define the `FLUME_HOME` environment variable:

```
Export FLUME_HOME=/opt/flume
```

5. Add the Flume `bin` directory to your path:

```
Export PATH=${FLUME_HOME}/bin:${PATH}
```

6. Verify that `JAVA_HOME` is set:

```
Echo ${JAVA_HOME}
```

7. Verify that the Hadoop libraries are in the classpath:

```
$ echo ${CLASSPATH}
```

8. Create the directory that will act as the Flume `conf` directory:

```
$ mkdir /home/hadoop/flume/conf
```

9. Copy the needed files into the `conf` directory:

```
$ cp /opt/flume/conf/log4j.properties /home/hadoop/flume/conf
$ cp /opt/flume/conf/flume-env.sh.sample /home/hadoop/flume/conf/
flume-env.sh
```

10. Edit `flume-env.sh` and set `JAVA_HOME`.

What just happened?

The Flume installation is straightforward and has similar prerequisites to previous tools we have installed.

Firstly, we retrieved the latest version of Flume NG (any version of 1.2.x or later will do) and saved it to the local filesystem. We moved it to the desired location, uncompressed it, and created a convenience symlink to the location.

We needed to define the `FLUME_HOME` environment variable and add the `bin` directory within the installation directory to our classpath. As before, this can be done directly on the command line or within convenience scripts.

Flume requires `JAVA_HOME` to be defined and we confirmed this is the case. It also requires Hadoop libraries, so we checked that the Hadoop classes are in the classpath.

The last steps are not strictly necessary for demonstration though will be used in production. Flume looks for a configuration directory within which are files defining the default logging properties and environment setup variables (such as `JAVA_HOME`). We find Flume performs most predictably when this directory is properly set up, so we did this now and don't need to change it much later.

We assumed `/home/hadoop/flume` is the working directory within which the Flume configuration and other files will be stored; change this based on what's appropriate for your system.

Using Flume to capture network data

Now that we have Flume installed, let's use it to capture some network data.

Time for action – capturing network traffic in a log file

In the first instance, let's use a simple Flume configuration that will capture the network data to the main Flume log file.

1. Create the following file as `agent1.conf` within your Flume working directory:

```
agent1.sources = netsource
agent1.sinks = logsink
agent1.channels = memorychannel

agent1.sources.netsource.type = netcat
agent1.sources.netsource.bind = localhost
agent1.sources.netsource.port = 3000
```

```
agent1.sinks.logsink.type = logger

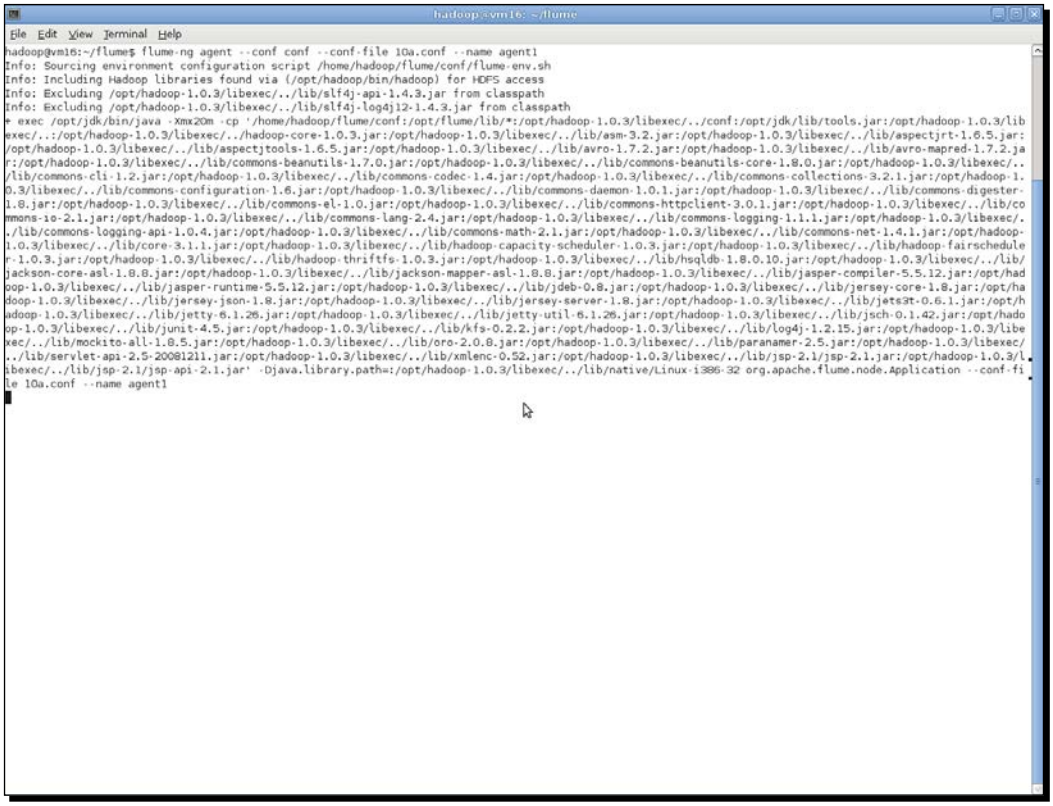
agent1.channels.memorychannel.type = memory
agent1.channels.memorychannel.capacity = 1000
agent1.channels.memorychannel.transactionCapacity = 100

agent1.sources.netsource.channels = memorychannel
agent1.sinks.logsink.channel = memorychannel
```

2. Start a Flume agent:

```
$ flume-ng agent --conf conf --conf-file 10a.conf  --name agent1
```

The output of the preceding command can be shown in the following screenshot:



3. In another window, open a telnet connection to port 3000 on the local host and then type some text:

```
$ curl telnet://localhost:3000
Hello
OK
Flume!
OK
```

4. Close the curl connection with *Ctrl* + *C*.

5. Look at the Flume log file:

```
$ tail flume.log
```

You will receive the following response:

```
2012-08-19 00:37:32,702 INFO sink.LoggerSink: Event: { headers:{}
body: 68 65 6C 6C 6F                                   Hello }
2012-08-19 00:37:32,702 INFO sink.LoggerSink: Event: { headers:{}
body: 6D 65                                            Flume }
```

What just happened?

Firstly, we created a Flume configuration file within our Flume working directory. We'll go into this in more detail later, but for now, think of Flume receiving data through a component called a **source** and writing it to a destination called a **sink**.

In this case, we create a **Netcat** source which listens on a port for network connections. You can see we configure it to bind to port 3000 on the local machine.

The configured sink is of the type `logger` which, not surprisingly, writes its output to a log file. The rest of the configuration file defines an **agent** called `agent1`, which uses this source and sink.

We then start a Flume agent by using the `flume-ng` binary. This is the tool we'll use to launch all Flume processes. Note that we give a few options to this command:

- The `agent` argument tells Flume to start an agent, which is the generic name for a running Flume process involved in data movement
- The `conf` directory, as mentioned earlier
- The particular configuration file for the process we are going to launch
- The name of the agent within the configuration file

The agent will start and no further output will appear on that screen. (Obviously, we would run the process in the background in a production setting.)

In another window, we open a telnet connection to port 3000 on the local machine using the `curl` utility. The traditional way of opening such sessions is of course the telnet program itself, but many Linux distributions have curl installed by default; almost none use the older `telnet` utility.

We type a word on each line and hit *Enter* then kill the session with a *Ctrl* + *C* command. Finally, we look at the `flume.log` file that is being written into the Flume working directory and see an entry for each of the words we typed in.

Time for action – logging to the console

It's not always convenient to have to look at log files, particularly when we already have the agent screen open. Let's modify the agent to also log events to the screen.

1. Restart the Flume agent with an additional argument:

```
$ flume-ng agent --conf conf --conf-file 10a.conf --name agent1
-Dflume.root.logger=INFO,console
```

   You will receive the following response:

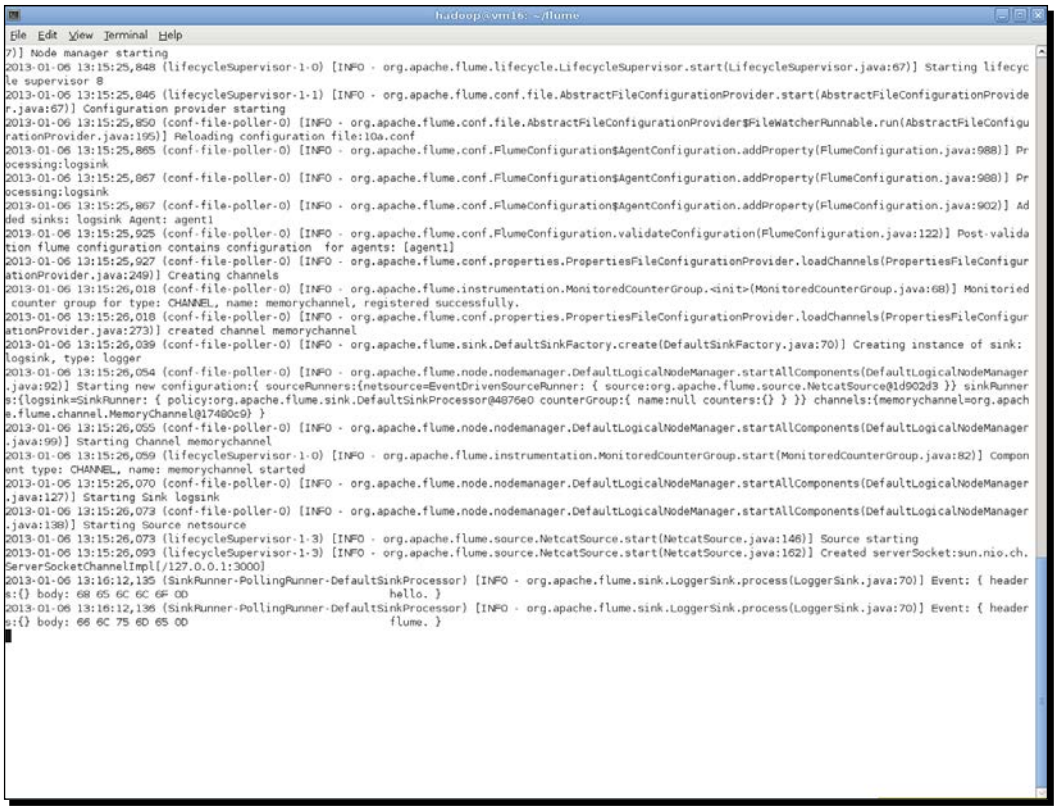
```
Info: Sourcing environment configuration script /home/hadoop/
flume/conf/flume-env.sh
…
org.apache.flume.node.Application --conf-file 10a.conf --name
agent1

2012-08-19 00:41:45,462 (main) [INFO - org.apache.flume.lifecycle.
LifecycleSupervisor.start(LifecycleSupervisor.java:67)] Starting
lifecycle supervisor 1
```

2. In another window, connect to the server via curl:

```
$ curl telnet://localhost:3000
```

3. Type in `Hello` and `Flume` on separate lines, hit *Ctrl* + *C*, and then check the agent window:



What just happened?

We added this example as it becomes very useful when debugging or creating new flows.

As seen in the previous example, Flume will, by default, write its logs to a file on the filesystem. More precisely, this is the default behavior as specified within the log4j property file within our `conf` directory. Sometimes we want more immediate feedback without constantly looking at log files or having to change the property file.

By explicitly setting the `flume.root.logger` variable on the command line, we can override the default logger configuration and have the output sent directly to the agent window. The logger is standard log4j, so the usual log levels such as `DEBUG` and `INFO` are supported.

Writing network data to log files

The default log sink behavior of Flume writing its received data into log files has some limitations, particularly if we want to use the captured data in other applications. By configuring a different type of sink, we can instead write the data into more consumable data files.

Time for action – capturing the output of a command to a flat file

Let's show this in action, along the way demonstrating a new kind of source as well.

1. Create the following file as `agent2.conf` within the Flume working directory:

```
agent2.sources = execsource
agent2.sinks = filesink
agent2.channels = filechannel

agent2.sources.execsource.type = exec
agent2.sources.execsource.command = cat /home/hadoop/message

agent2.sinks.filesink.type = FILE_ROLL
agent2.sinks.filesink.sink.directory = /home/hadoop/flume/files
agent2.sinks.filesink.sink.rollInterval = 0

agent2.channels.filechannel.type = file
agent2.channels.filechannel.checkpointDir = /home/hadoop/flume/fc/
checkpoint
agent2.channels.filechannel.dataDirs = /home/hadoop/flume/fc/data

agent2.sources.execsource.channels = filechannel
agent2.sinks.filesink.channel = filechannel
```

2. Create a simple test file in the home directory:

```
$ echo "Hello again Flume!" > /home/hadoop/message
```

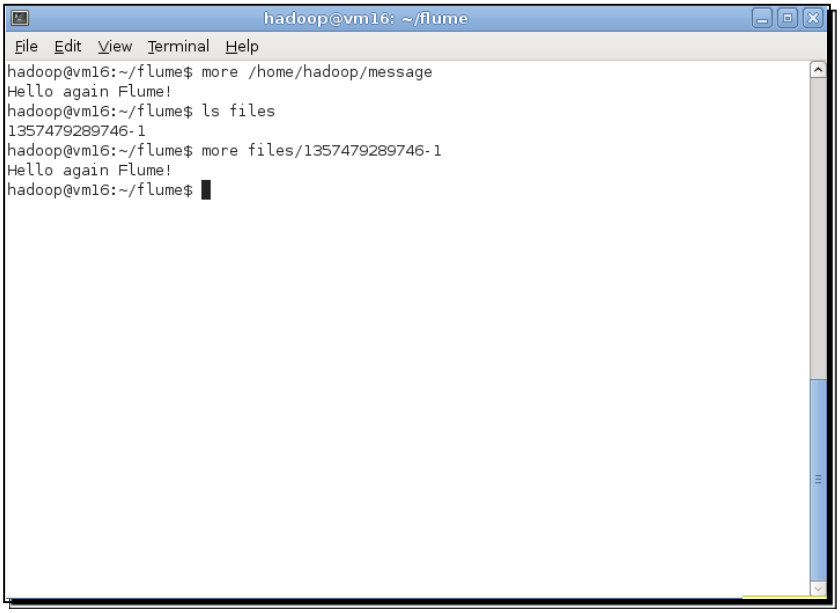
3. Start the agent:

```
$ flume-ng agent --conf conf --conf-file agent2.conf --name agent2
```

4. In another window, check file sink output directory:

```
$ ls files
$ cat files/*
```

The output of the preceding command can be shown in the following screenshot:



What just happened?

The previous example follows a similar pattern as before. We created the configuration file for a Flume agent, ran the agent, and then confirmed it had captured the data we expected.

This time we used an exec source and a `file_roll` sink. The former, as the name suggests, executes a command on the host and captures its output as the input to the Flume agent. Although, in the previous case, the command is executed only once, this was for illustration purposes only. More common uses will use commands that produce an ongoing stream of data. Note that the exec sink can be configured to restart the command if it does terminate.

The output of the agent is written to a file as specified in the configuration file. By default, Flume rotates (rolls) to a new file every 30 seconds; we disable this capability to make it easier to track what's going on in a single file.

We see the file does indeed contain the output of the specified `exec` command.

Logs versus files

It may not be immediately obvious why Flume has both log and file sinks. Conceptually both do the same thing, so what's the difference?

The logger sink in reality is more of a debug tool than anything else. It doesn't just record the information captured by the source, but adds a lot of additional metadata

and events. The file sink however records the input data exactly as it was received with no alteration—though such is possible if required as we will see later.

In most cases, you'll want the file sink to capture the input data but the log may also be of use in non-production situations depending on your needs.

Time for action – capturing a remote file in a local flat file

Let's show another example of capturing data to a file sink. This time we will use another Flume capability that allows it to receive data from a remote client.

1. Create the following file as `agent3.conf` in the Flume working directory:

```
agent3.sources = avrosource
agent3.sinks = filesink
agent3.channels = jdbcchannel

agent3.sources.avrosource.type = avro
agent3.sources.avrosource.bind = localhost
agent3.sources.avrosource.port = 4000
agent3.sources.avrosource.threads = 5

agent3.sinks.filesink.type = FILE_ROLL
agent3.sinks.filesink.sink.directory = /home/hadoop/flume/files
agent3.sinks.filesink.sink.rollInterval = 0

agent3.channels.jdbcchannel.type = jdbc

agent3.sources.avrosource.channels = jdbcchannel
agent3.sinks.filesink.channel = jdbcchannel
```

2. Create a new test file as `/home/hadoop/message2`:

```
Hello from Avro!
```

3. Start the Flume agent:

```
$ flume-ng agent –conf conf –conf-file agent3.conf –name agent3
```

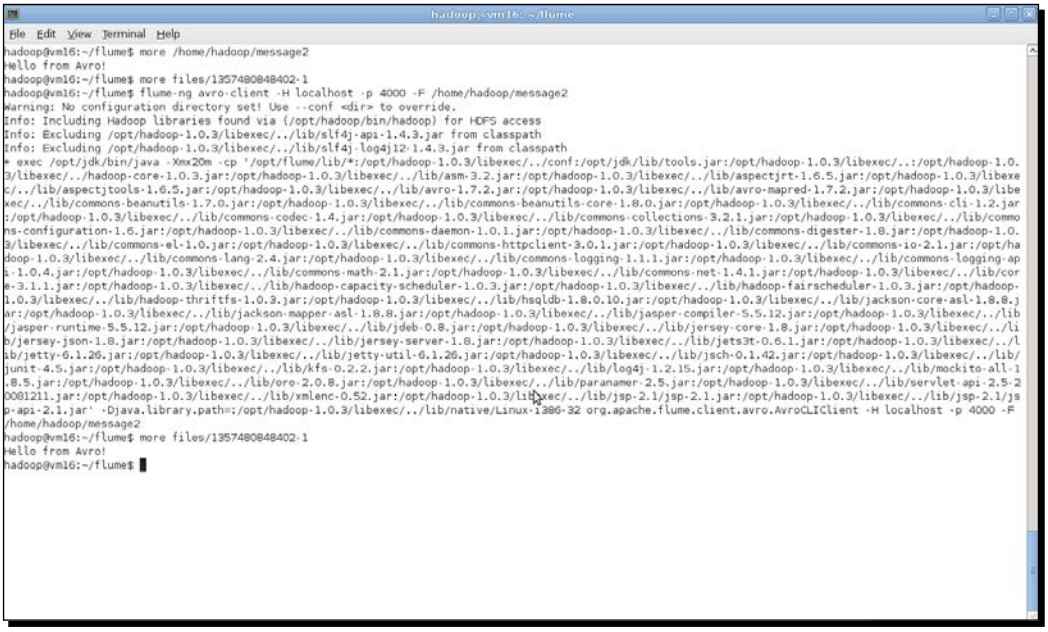
4. In another window, use the Flume Avro client to send a file to the agent:

```
$ flume-ng avro-client -H localhost -p 4000 -F /home/hadoop/
message
```

5. As before, check the file in the configured output directory:

```
$ cat files/*
```

The output of the preceding command can be shown in following screenshot:



What just happened?

As before, we created a new configuration file and this time used an Avro source for the agent. Recall from *Chapter 5*, *Advanced MapReduce Techniques*, that Avro is a data serialization framework; that is, it manages the packaging and transport of data from one point to another across the network. Similarly to the Netcat source, the Avro source requires configuration parameters that specify its network settings. In this case, it will listen on port 4000 on the local machine. The agent is configured to use the file sink as before and we start it up as usual.

Flume comes with both an Avro source and a standalone Avro client. The latter can be used to read a file and send it to an Avro source anywhere on the network. In our example, we just use the local machine, but note that the Avro client requires the explicit hostname and port of the Avro source to which it should send the file. So this is not a constraint; an Avro client can send files to a listening Flume Avro source anywhere on the network.

The Avro client reads the file, sends it to the agent, and this gets written to the file sink. We check this behavior by confirming that the file contents are in the file sink location as expected.

Sources, sinks, and channels

We intentionally used a variety of sources, sinks, and channels in the previous examples just to show how they can be mixed and matched. However, we have not explored them—especially channels—in much detail. Let's dig a little deeper now.

Sources

We've looked at three sources: Netcat, exec, and Avro. Flume NG also supports a sequence generator source (mostly for testing) as well as both TCP and UDP variants of a source that reads **syslogd** data. Each source is configured within an agent and after receiving enough data to produce a Flume event, it sends this newly created event to the channel to which the source is connected. Though a source may have logic relating to how it reads data, translates events, and handles failure situations, the source has no knowledge of how the event is to be stored. The source has the responsibility of delivering the event to the configured channel, and all other aspects of the event processing are invisible to the source.

Sinks

In addition to the logger and file roll sinks we used previously, Flume also supports sinks for HDFS, HBase (two types), Avro (for agent chaining), null (for testing), and IRC (for an Internet Relay Chat service). The sink is conceptually similar to the source but in reverse.

The sink waits for events to be received from the configured channel about whose inner workings it knows nothing. On receipt, the sink handles the output of the event to its particular destination, managing all issues around time outs, retries, and rotation.

Channels

So what are these mysterious channels that connect the source and sink? They are, as the name and configuration entries before suggest, the communication and retention mechanism that manages event delivery.

When we define a source and a sink, there may be significant differences in how they read and write data. An exec source may, for example, receive data much faster than a file roll sink can write it or the source may have times (such as when rolling to a new file or dealing with system I/O congestion) that writing needs be paused. The channel, therefore, needs buffer data between the source and sink to allow data to stream through the agent as efficiently as possible. This is why the channel configuration portions of our configuration files include elements such as capacity.

The **memory** channel is the easiest to understand as the events are read from the source into memory and passed to the sink as it is able to receive them. But if the agent process dies mid-way through the process (be it due to software or hardware failure), then all the events currently in the memory channel are lost forever.

The **file** and **JDBC** channels that we also used provide persistent storage of events to prevent such loss. After reading an event from a source, the file channel writes the contents to a file on the filesystem that is deleted only after successful delivery to the sink. Similarly, the JDBC channel uses an embedded Derby database to store events in a recoverable fashion.

This is a classic performance versus reliability trade-off. The memory channel is the fastest but has the risk of data loss. The file and JDBC channels are typically much slower but effectively provide guaranteed delivery to the sink. Which channel you choose depends on the nature of the application and the values of each event.



Don't worry too much about this trade-off; in the real world, the answer is usually obvious. Also be sure to look carefully at the reliability of the source and sink being used. If those are unreliable and you drop events anyway, do you gain much from a persistent channel?

Or roll your own

Don't feel limited by the existing collection of sources, sinks, and channels. Flume offers an interface to define your own implementation of each. In addition, there are a few components present in Flume OG that have not yet been incorporated into Flume NG but may appear in the future.

Understanding the Flume configuration files

Now that we've talked through sources, sinks, and channels, let's take a look at one of the configuration files from earlier in a little more detail:

```
agent1.sources = netsource
agent1.sinks = logsink
agent1.channels = memorychannel
```

These first lines name the agent and define the sources, sinks, and channels associated with it. We can have multiple values on each line; the values are space separated:

```
agent1.sources.netsource.type = netcat
agent1.sources.netsource.bind = localhost
agent1.sources.netsource.port = 3000
```

These lines specify the configuration for the source. Since we are using the Netcat source, the configuration values specify how it should bind to the network. Each type of source has its own configuration variables.

```
agent1.sinks.logsink.type = logger
```

This specifies the sink to be used is the logger sink which is further configured via the command line or the log4j property file.

```
agent1.channels.memorychannel.type = memory
agent1.channels.memorychannel.capacity = 1000
agent1.channels.memorychannel.transactionCapacity = 100
These lines specify the channel to be used and then add the type
specific configuration values.  In this case we are using the memory
channel and we specify its capacity but – since it is non-persistent –
no external storage mechanism.
agent1.sources.netsource.channels = memorychannel
agent1.sinks.logsink.channel = memorychannel
```

These last lines configure the channel to be used for the source and sink. Though we used different configuration files for our different agents, we could just as easily place all the elements in a single configuration file as the respective agent names provide the necessary separation. This can, however, produce a pretty verbose file which can be a little intimidating when you are just learning Flume. We can also have multiple flows within a given agent, we could, for example, combine the first two preceding examples into a single configuration file and agent.

Have a go hero

Do just that! Create a configuration file that specifies the capabilities of both our previous `agent1` and `agent2` from the preceding example in a single composite agent that contains:

- A Netcat source and its associated logger sink
- An exec source and its associated file sink
- Two memory channels, one for each of the source/sink pairs mentioned before

To get you started, here's how the component definitions could look:

```
agentx.sources = netsource execsource
agentx.sinks = logsink filesink
agentx.channels = memorychannel1 memorychannel2
```

It's all about events

Let's discuss one more definition before we try another example. Just what is an event?

Remember that Flume is explicitly based around log files, so in most cases, an event equates to a line of text followed by a new line character. That is the behavior we've seen with the sources and sinks we've used.

This isn't always the case, however, the UDP syslogd source, for example, treats each packet of data received as a single event, which gets passed through the system. When using these sinks and sources, however, these definitions of events are unchangeable and when reading files, for example, we have no choice but to use line-based events.

Time for action – writing network traffic onto HDFS

This discussion of Flume in a book about Hadoop hasn't actually used Hadoop at all so far. Let's remedy that by writing data onto HDFS via Flume.

1. Create the following file as `agent4.conf` within the Flume working directory:

```
agent4.sources = netsource
agent4.sinks = hdfssink
agent4.channels = memorychannel

agent4.sources.netsource.type = netcat
agent4.sources.netsource.bind = localhost
agent4.sources.netsource.port = 3000

agent4.sinks.hdfssink.type = hdfs
agent4.sinks.hdfssink.hdfs.path = /flume
agent4.sinks.hdfssink.hdfs.filePrefix = log
agent4.sinks.hdfssink.hdfs.rollInterval = 0
agent4.sinks.hdfssink.hdfs.rollCount = 3
agent4.sinks.hdfssink.hdfs.fileType = DataStream

agent4.channels.memorychannel.type = memory
agent4.channels.memorychannel.capacity = 1000
agent4.channels.memorychannel.transactionCapacity = 100

agent4.sources.netsource.channels = memorychannel
agent4.sinks.hdfssink.channel = memorychannel
```

2. Start the agent:

```
$ flume-ng agent –conf conf –conf-file agent4.conf –name agent4
```

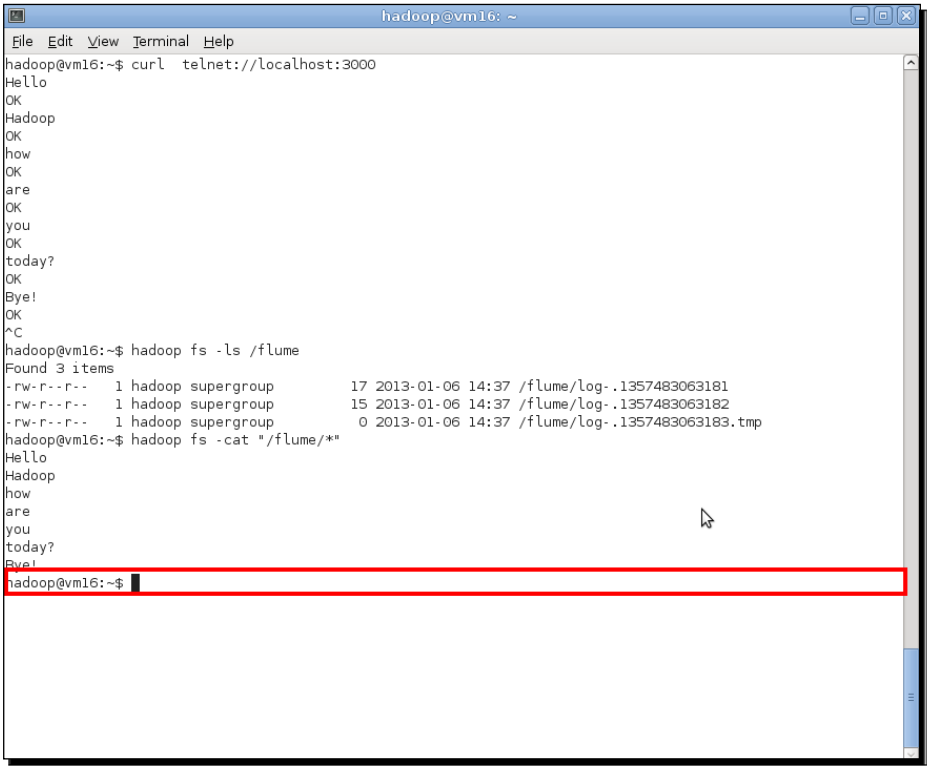
3. In another window, open a telnet connection and send seven events to Flume:

```
$ curl telnet://localhost:3000
```

4. Check the contents of the directory specified in the Flume configuration file and then examine the file contents:

```
$ hadoop fs -ls /flume
$ hadoop fs –cat "/flume/*"
```

The output of the preceding command can be shown in the following screenshot:



What just happened?

This time we paired a Netcat source with the HDFS sink. As can be seen from the configuration file, we need to specify aspects such as the location for the files, any file prefix, and the strategy for rolling from one file to another. In this case, we specified files within the `/flume` directory, each starting with `log-` and with a maximum of three entries in each file (obviously, such a low value is for testing only).

After starting the agent, we use curl once more to send seven single word events to Flume. We then used the Hadoop command-line utility to look at the directory contents and verify that our input data was being written to HDFS.

Note that the third HDFS file has a `.tmp` extension. Remember that we specified three entries per file but only input seven values. We, therefore, filled up two files and started on another. Flume gives the file currently being written a `.tmp` extension, which makes it easy to differentiate the completed files from in-progress files when specifying which files to process via MapReduce jobs.

Time for action – adding timestamps

We mentioned earlier that there were mechanisms to have file data written in slightly more sophisticated ways. Let's do something very common and write our data into a directory with a dynamically-created timestamp.

1. Create the following configuration file as `agent5.conf`:

```
agent5.sources = netsource
agent5.sinks = hdfssink
agent5.channels = memorychannel

agent5.sources.netsource.type = netcat
agent5.sources.netsource.bind = localhost
agent5.sources.netsource.port = 3000
agent5.sources.netsource.interceptors = ts

agent5.sources.netsource.interceptors.ts.type = org.apache.flume.
interceptor.TimestampInterceptor$Builder

agent5.sinks.hdfssink.type = hdfs
agent5.sinks.hdfssink.hdfs.path = /flume-%Y-%m-%d
agent5.sinks.hdfssink.hdfs.filePrefix = log-
agent5.sinks.hdfssink.hdfs.rollInterval = 0
agent5.sinks.hdfssink.hdfs.rollCount = 3
agent5.sinks.hdfssink.hdfs.fileType = DataStream

agent5.channels.memorychannel.type = memory
agent5.channels.memorychannel.capacity = 1000
agent5.channels.memorychannel.transactionCapacity = 100

agent5.sources.netsource.channels = memorychannel
agent5.sinks.hdfssink.channel = memorychannel
```

2. Start the agent:

```
$ flume-ng agent –conf conf –conf-file agent5.conf –name agent5
```

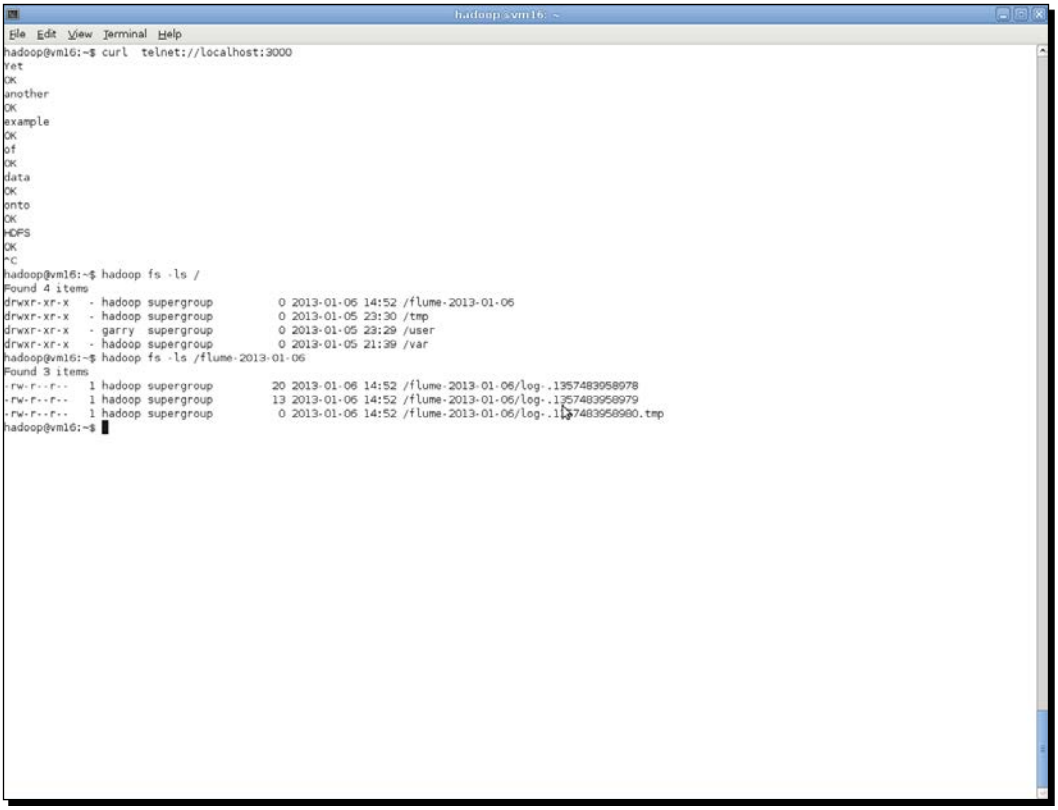
3. In another window, open up a telnet session and send seven events to Flume:

```
$ curl telnet://localhost:3000
```

4. Check the directory name on HDFS and the files within it:

```
$ hadoop fs -ls /
```

The output of the preceding code can be shown in the following screenshot:



What just happened?

We made a few changes to the previous configuration file. We added an `interceptor` specification to the Netcat source and gave its implementation class as `TimestampInterceptor`.

Flume interceptors are plugins that can manipulate and modify events before they pass from the source to the channel. Most interceptors either add metadata to the event (as in this case) or drop events based on certain criteria. In addition to several inbuilt interceptors, there is naturally a mechanism for user-defined interceptors.

We used the **timestamp** interceptor here which adds to the event metadata the Unix timestamp at the time the event is read. This allows us to extend the definition of the HDFS path into which events are to be written.

While previously we simply wrote all events to the `/flume` directory, we now specified the path as `/flume-%Y-%m-%d`. After running the agent and sending some data to Flume, we looked at HDFS and saw that these variables have been expanded to give the directory a year/month/date suffix.

The HDFS sink supports many other variables such as the hostname of the source and additional temporal variables that can allow precise partitioning to the level of seconds.

The utility here is plain; instead of having all events written into a single directory that becomes enormous over time, this simple mechanism can give automatic partitioning, making data management easier but also providing a simpler interface to the data for MapReduce jobs. If, for example, most of your MapReduce jobs process hourly data, then having Flume partition incoming events into hourly directories will make your life much easier.

To be precise, the event passing through Flume has had a complete Unix timestamp added, that is, accurate to the nearest second. In our example, we used only date-related variables in the directory specification, if hourly or finer-grained directory partitioning is required, then the time-related variables would be used.



This assumes that the timestamp at the point of processing is sufficient for your needs. If files are being batched and then fed to Flume, then a file's contents may have timestamps from the previous hour than when they are being processed. In such a case, you could write a custom interceptor to set the timestamp header based on the contents of the file.

To Sqoop or to Flume...

An obvious question is whether Sqoop or Flume is most appropriate if we have data in a relational database that we want to export onto HDFS. We've seen how Sqoop can perform such an export and we could do something similar using Flume, either with a custom source or even just by wrapping a call to the `mysql` command within an exec source.

A good rule of thumb is to look at the type of data and ask if it is log data or something more involved.

Flume was created in large part to handle log data and it excels in this area. But in most cases, Flume networks are responsible for delivering events from sources to sinks without any real transformation on the log data itself. If you have log data in multiple relational databases, then Flume is likely a great choice, though I would question the long-term scalability of using a database for storing log records.

Non-log data may require data manipulation that only Sqoop is capable of performing. Many of the transformations we performed in the previous chapter using Sqoop, such as specifying subsets of columns to be retrieved, are really not possible using Flume. It's also quite possible that if you are dealing with structured data that requires individual field processing, then Flume alone is not the ideal tool. If you want direct Hive integration then Sqoop is your only choice.

Remember, of course, that the tools can also work together in more complex workflows. Events could be gathered together onto HDFS by Flume, processed through MapReduce, and then exported into a relational database by Sqoop.

Time for action – multi level Flume networks

Let's put together a few pieces we touched on earlier and see how one Flume agent can use another as its sink.

1. Create the following file as `agent6.conf`:

```
agent6.sources = avrosource
agent6.sinks = avrosink
agent6.channels = memorychannel

agent6.sources.avrosource.type = avro
agent6.sources.avrosource.bind = localhost
agent6.sources.avrosource.port = 2000
agent6.sources.avrosource.threads = 5

agent6.sinks.avrosink.type = avro
agent6.sinks.avrosink.hostname = localhost
agent6.sinks.avrosink.port = 4000

agent6.channels.memorychannel.type = memory
agent6.channels.memorychannel.capacity = 1000
agent6.channels.memorychannel.transactionCapacity = 100

agent6.sources.avrosource.channels = memorychannel
agent6.sinks.avrosink.channel = memorychannel
```

2. Start an agent configured as per the `agent3.conf` file created earlier, that is, with an Avro source and a file sink:

```
$ flume-ng client –conf conf –conf-file agent3.conf agent3
```

3. In a second window, start another agent; this one configured with the preceding file:

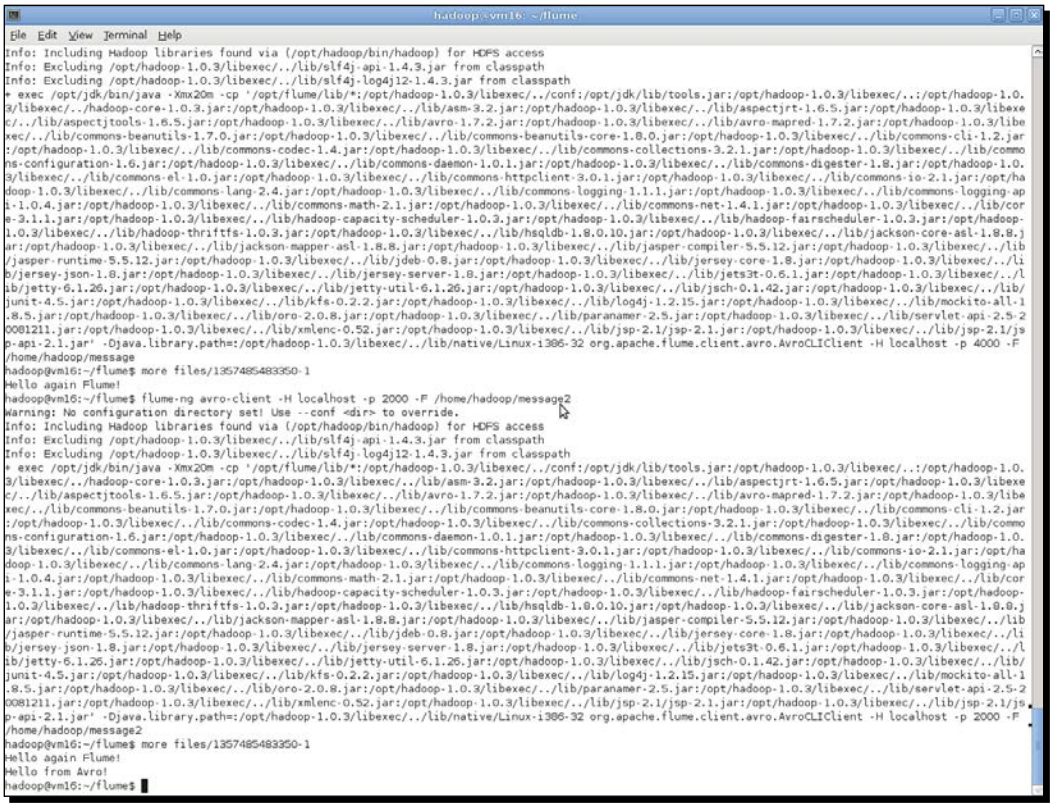
```
$ flume-ng client –conf conf –conf-file agent6.conf agent6
```

4. In a third window, use the Avro client to send a file to each of the Flume agents:

```
$ flume-ng avro-client -H localhost -p 4000 -F /home/hadoop/
message

$ flume-ng avro-client -H localhost -p 2000 -F /home/hadoop/
message2
```

5. Check the output directory for files and examine the file present:



What just happened?

Firstly, we defined a new agent with an Avro source and also an Avro sink. We've not used this sink before; instead of writing the events to a local location or HDFS, this sink sends the events to a remote source using Avro.

We start an instance of this new agent and then also start an instance of `agent3` we used earlier. Recall this agent has an Avro source and a file roll sink. We configure the Avro sink in the first agent to point to the host and port of the Avro sink in the second and by doing so, build a data-routing chain.

With both agents running, we then use the Avro client to send a file to each and confirm that both appear in the file location configured as the destination for the `agent3` sink.

This isn't just technical capability for its own sake. This capability is the building block that allows Flume to build arbitrarily complex distributed event collection networks. Instead of one copy of each agent, think of multiple agents of each type feeding events into the next link in the chain, which acts as an event aggregation point.

Time for action – writing to multiple sinks

We need one final piece of capability to build such networks, namely, an agent that can write to multiple sinks. Let's create one.

1. Create the following configuration file as `agent7.conf`:

```
agent7.sources = netsource
agent7.sinks = hdfssink filesink
agent7.channels = memorychannel1 memorychannel2

agent7.sources.netsource.type = netcat
agent7.sources.netsource.bind = localhost
agent7.sources.netsource.port = 3000
agent7.sources.netsource.interceptors = ts

agent7.sources.netsource.interceptors.ts.type = org.apache.flume.
interceptor.TimestampInterceptor$Builder

agent7.sinks.hdfssink.type = hdfs
agent7.sinks.hdfssink.hdfs.path = /flume-%Y-%m-%d
agent7.sinks.hdfssink.hdfs.filePrefix = log
agent7.sinks.hdfssink.hdfs.rollInterval = 0
agent7.sinks.hdfssink.hdfs.rollCount = 3
agent7.sinks.hdfssink.hdfs.fileType = DataStream

agent7.sinks.filesink.type = FILE_ROLL
agent7.sinks.filesink.sink.directory = /home/hadoop/flume/files
agent7.sinks.filesink.sink.rollInterval = 0

agent7.channels.memorychannel1.type = memory
agent7.channels.memorychannel1.capacity = 1000
agent7.channels.memorychannel1.transactionCapacity = 100

agent7.channels.memorychannel2.type = memory
```

```
agent7.channels.memorychannel2.capacity = 1000
agent7.channels.memorychannel2.transactionCapacity = 100

agent7.sources.netsource.channels = memorychannel1 memorychannel2
agent7.sinks.hdfssink.channel = memorychannel1
agent7.sinks.filesink.channel = memorychannel2

agent7.sources.netsource.selector.type = replicating
```

2. Start the agent:

```
$ flume-ng agent -conf conf -conf-file agent7.conf -name agent7
```

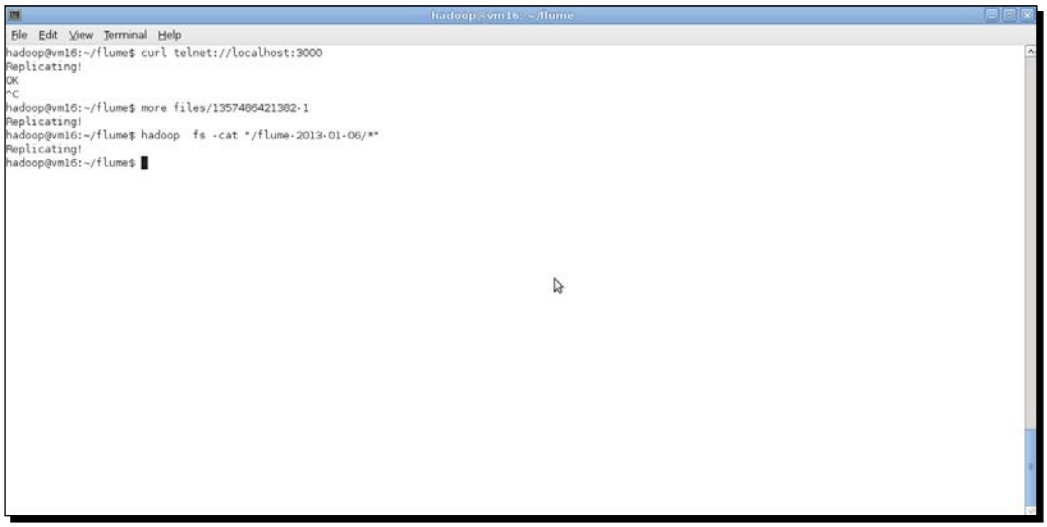
3. Open a telnet session and send an event to Flume:

```
$ curl telnet://localhost:3000
```

You will receive the following response:

```
Replicating!
Check the contents of the HDFS and file sinks:
$ cat files/*
$ hdfs fs -cat "/flume-*/*"
```

The output of the preceding command can be shown in the following screenshot:



What just happened?

We created a configuration file containing a single Netcat source and both the file and HDFS sink. We configured separate memory channels connecting the source to both sinks.

We then set the source selector type to `replicating`, which means events will be sent to all configured channels.

After starting the agent as normal and sending an event to the source, we confirmed that the event was indeed written to both the filesystem and HDFS sinks.

Selectors replicating and multiplexing

The source selector has two modes, replicating as we have seen here and multiplexing. A **multiplexing** source selector will use logic to determine to which channel an event should be sent, depending on the value of a specified header field.

Handling sink failure

By their nature of being output destinations, it is to be expected that sinks may fail or become unresponsive over time. As with any input/output device, a sink may be saturated, run out of space, or go offline.

Just as Flume associates selectors with sources to allow the replication and multiplexing behavior we have just seen it also supports the concept of sink processors.

There are two defined sink processors, namely, the **failover** sink processor and the **load balancing** sink processor.

The sink processors view the sinks as being within a group and, dependent on their type, react differently when an event arrives. The load balancing sink processor sends events to sinks one at a time, using either a round-robin or random algorithm to select which sink to use next. If a sink fails, the event is retried on another sink, but the failed sink remains in the pool.

The failover sink, in contrast, views the sinks as a prioritized list and only tries a lower priority sink if the ones above it have failed. Failed sinks are removed from the list and are only retried after a cooling-off period that increases with subsequent failures.

Have a go hero - Handling sink failure

Set up a Flume configuration that has three configured HDFS sinks, each writing to different locations on HDFS. Use the load balancer sink processor to confirm events are written to each sink, and then use the failover sink processor to show the prioritization.

Can you force the agent to select a processor other than the highest priority one?

Next, the world

We have now covered most of the key features of Flume. As mentioned earlier, Flume is a framework and this should be considered carefully; Flume has much more flexibility in its deployment model than any other product we have looked at.

It achieves its flexibility through a relatively small set of capabilities; the linking of sources to sinks via channels and multiple variations that allow multi-agent or multi-channel configurations. This may not seem like much, but consider that these building blocks can be composed to create a system such as the following where multiple web server farms feed their logs into a central Hadoop cluster:

- Each node in each farm runs an agent pulling each local log file in turn.
- These log files are sent to a highly-available aggregation point, one within each farm which also performs some processing and adds additional metadata to the events, categorizing them as three types of records.
- These first level aggregators then send the events to one of the series of agents that access the Hadoop cluster. The aggregators offer multiple access points and event types 1 and 2 are sent to the first, event type 3 to the second.
- Within the final aggregator, they write event type 1 and 2 to different locations on HDFS, with type 2 also being written to a local filesystem. Event type 3 is written directly into HBase.

It is amazing how simple primitives can be composed to build complex systems like this!

Have a go hero - Next, the world

As a thought experiment, try to work through the preceding scenario and determine what sort of Flume setup would be required at each step in the flow.

The bigger picture

It's important to realize that "simply" getting data from one point to another is rarely the extent of your data considerations. Terms such as **data lifecycle management** have become widely used recently for good reason. Let's briefly look at some things to consider, ideally before you have the data flooding across the system.

Data lifecycle

The main question to be asked in terms of data lifecycle is for how long will the value you gain from data storage be greater than the storage costs. Keeping data forever may seem attractive but the costs of holding more and more data will increase over time. These costs are not just financial; many systems see their performance degrade as volumes increase.

This question isn't—or at least rarely should be—decided by technical factors. Instead, you need the value and costs to the business to be the driving factors. Sometimes data becomes worthless very quickly, other times the business cannot delete it for either competitive or legal reasons. Determine the position and act accordingly.

Remember of course that it is not a binary decision between keeping or deleting data; you can also migrate data across tiers of storage that decrease in cost and performance as they age.

Staging data

On the other side of the process, it is often worthwhile to think about how data is fed into processing platforms such as MapReduce. With multiple data sources, the last thing you often want is to have all the data arrive on a single massive volume.

As we saw earlier, Flume's ability to parameterize the location to which it writes on HDFS is a great tool to aid this problem. However, often it is useful to view this initial drop-off point as a temporary staging area to which data is written prior to processing. After it is processed, it may be moved into the long-term directory structure.

Scheduling

At many points in the flows, we have discussed that there is an implicit need for an external task to do something. As mentioned before, we want MapReduce to process files once they are written to HDFS by Flume, but how is that task scheduled? Alternatively, how do we manage the post-processing, the archival or deletion of old data, even the removal of log files on the source hosts?

Some of these tasks, such as the latter, are likely managed by existing systems such as **logrotate** on Linux but the others may be things you need to build. Obvious tools such as **cron** may be good enough, but as system complexity increases, you may need to investigate more sophisticated scheduling systems. We will briefly mention one such system with tight Hadoop integration in the next chapter.

Summary

This chapter discussed the problem of how to retrieve data from across the network and make it available for processing in Hadoop. As we saw, this is actually a more general challenge and though we may use Hadoop-specific tools, such as Flume, the principles are not unique. In particular, we covered an overview of the types of data we may want to write to Hadoop, generally categorizing it as network or file data. We explored some approaches for such retrieval using existing command-line tools. Though functional, the approaches lacked sophistication and did not suit extension into more complex scenarios.

We looked at Flume as a flexible framework for defining and managing data (particularly from log files) routing and delivery, and learned the Flume architecture which sees data arrive at sources, be processed through channels, and then written to sinks.

We then explored many of Flume's capabilities such as how to use the different types of sources, sinks, and channels. We saw how the simple building blocks could be composed into very complex systems and we closed with some more general thoughts on data management.

This concludes the main content of this book. In the next chapter, we will sketch out a number of other projects that may be of interest and highlight some ways of engaging the community and getting support.

11
Where to Go Next

This book has, as the title suggests, sought to give a beginner to Hadoop in-depth knowledge of the technology and its application. As has been seen on several occasions, there is a lot more to the Hadoop ecosystem than the core product itself. We will give a quick highlight of some potential areas of interest in this chapter.

In this chapter we will discuss:

- What we covered in this book
- What we didn't cover in this book
- Upcoming Hadoop changes
- Alternative Hadoop distributions
- Other significant Apache projects
- Alternative programming abstractions
- Sources of information and help

What we did and didn't cover in this book

With our focus on beginners, the aim of this book was to give you a strong grounding in the core Hadoop concepts and tools. In addition, we provided experiences with some other tools that help you integrate the technology into your infrastructure.

Though Hadoop started as the single core product, it's fair to say that the ecosystem surrounding Hadoop has exploded in recent years. There are alternative distributions of the technology, some providing commercial custom extensions. There are a plethora of related projects and tools that build upon Hadoop and provide specific functionality or alternative approaches to existing ideas. It's a really exciting time to get involved with Hadoop; let's take a quick tour of what is out there.



Note, of course, that any overview of the ecosystem is both skewed by the author's interests and preferences and outdated the moment it is written. In other words, don't for a moment think this is all that's available; consider it a whetting of the appetite.

Upcoming Hadoop changes

Before discussing alternative Hadoop distributions, let's look at some changes to Hadoop itself in the near future. We've already discussed the HDFS changes coming in Hadoop 2.0, particularly the high availability of NameNode enabled by the new BackupNameNode and CheckpointNameNode services. This is a significant capability for Hadoop as it will make HDFS much more robust, greatly enhancing its enterprise credentials and streamlining cluster operations. The impact of NameNode HA is hard to exaggerate; it will almost certainly become one of those capabilities that no one will be able to remember how we lived without in a few years' time.

MapReduce is not standing still while all this is going on, and in fact, the changes being introduced may not have as much immediate impact but are actually much more fundamental.

These changes were initially developed under the name **MapReduce 2.0** or **MRV2**. However, the name now being used is **YARN** (**Yet Another Resource Negotiator**), which is more accurate as the changes are much more about Hadoop as a platform than MapReduce itself. The goal of YARN is to build a framework on Hadoop that allows cluster resources to be allocated to given applications and for MapReduce to be only one of these applications.

If you consider the JobTracker today, it is responsible for two quite different tasks: managing the progress of a given MapReduce job (but also identifying which cluster resources are available at any point in time) and allocating the resources to the various stages of the job. YARN separates these out into distinct roles; a global **ResourceManager** that uses NodeManagers on each host to manage the cluster's resources and a distinct **ApplicationManager** (the first example of which is MapReduce) that communicates with the ResourceManager to get the resources it needs for its job.

The MapReduce interface in YARN will be unchanged, so from a client perspective, all existing code will still run on the new platform. But as new ApplicationManagers are developed, we will start to see Hadoop being used more as a generic task processing platform with multiple types of processing models supported. Early examples of other models being ported to YARN are stream-based processing and a port of the **Message Passing Interface** (**MPI**), which is broadly used in scientific computing.

Alternative distributions

Way back in *Chapter 2*, *Getting Up and Running*, we went to the Hadoop homepage from which we downloaded the installation package. Odd as it may seem, this is far from the only way to get Hadoop. Odder still may be the fact that most production deployments don't use the Apache Hadoop distribution.

Why alternative distributions?

Hadoop is open source software. Anyone can, providing they comply with the Apache Software License that governs Hadoop, make their own release of the software. There are two main reasons alternative distributions have been created.

Bundling

Some providers seek to build a pre-bundled distribution containing not only Hadoop but also other projects, such as Hive, HBase, Pig, and many more. Though installation of most projects is rarely difficult—with the exception of HBase, which has historically been more difficult to set up by hand—there can be subtle version incompatibilities that don't arise until a particular production workload hits the system. A bundled release can provide a pre-integrated set of compatible versions that are known to work together.

The bundled release can also provide the distribution not only in a tarball file but also in packages that are easily installed through package managers such as RPM, Yum, or APT.

Free and commercial extensions

Being an open source project with a relatively liberal distribution license, creators are also free to enhance Hadoop with proprietary extensions that are made available either as free open source or commercial products.

This can be a controversial issue as some open source advocates dislike any commercialization of successful open source projects; to them it appears that the commercial entity is freeloading by taking the fruits of the open source community without having to build it for themselves. Others see this as a healthy aspect of the flexible Apache license; the base product will always be free and individuals and companies can choose to go with commercial extensions or not. We do not pass judgment either way, but be aware that this is a controversy you will almost certainly encounter.

Given the reasons for the existence of alternative distributions, let's look at a few popular examples.

Cloudera Distribution for Hadoop

The most widely used Hadoop distribution is the **Cloudera Distribution for Hadoop**, referred to as **CDH**. Recall that Cloudera is the company that first created Sqoop and contributed it back to the open source community and is where Doug Cutting now works.

The Cloudera distribution is available at `http://www.cloudera.com/hadoop` and contains a large number of Apache products, from Hadoop itself, Hive, Pig, and HBase through tools such as Sqoop and Flume, to other lesser-known products such as Mahout and Whir. We'll talk about some of these later.

CDH is available in several package formats and deploys the software in a ready-to-go fashion. The base Hadoop product, for example, is separated into different packages for the components such as NameNode, TaskTracker, and so on, and for each, there is integration with the standard Linux service infrastructure.

CDH was the first widely available alternative distribution, and its breadth of available software, proven level of quality, and free cost has made it a very popular choice.

Cloudera does also offer additional commercial-only products, such as a Hadoop management tool, in addition to training, support, and consultancy services. Details are available on the company webpage.

Hortonworks Data Platform

In 2011, the Yahoo division responsible for so much of the development of Hadoop was spun off into a new company called **Hortonworks**. They have also produced their own pre-integrated Hadoop distribution, called the **Hortonworks Data Platform** (**HDP**) and available at `http://hortonworks.com/products/hortonworksdataplatform/`.

HDP is conceptually similar to CDH, but both products have differences in their focus. Hortonworks makes much of the fact that HDP is fully open source, including the management tool. They also have positioned HDP as a key integration platform through support for tools such as Talend Open Studio. Hortonworks does not offer commercial software; its business model focuses instead on offering professional services and support for the platform.

Both Cloudera and Hortonworks are venture-backed companies with significant engineering expertise; both companies employ many of the most prolific contributors to Hadoop. The underlying technology is however the same Apache projects; the differences are how they are packaged, the versions employed, and the additional value-added offerings provided by the companies.

MapR

A different type of distribution is offered by **MapR Technologies**, though the company and distribution are usually referred to simply as **MapR**. Available at `http://www.mapr.com`, the distribution is based on Hadoop but has added a number of changes and enhancements.

One main MapR focus is on performance and availability, for example, it was the first distribution to offer a high-availability solution for the Hadoop NameNode and JobTracker, which you will remember (from *Chapter 7*, *Keeping Things Running*) is a significant weakness in core Hadoop. It also offers native integration with NFS file systems, which makes processing of existing data much easier; MapR replaced HDFS with a full POSIX-compliant filesystem that can easily be mounted remotely.

MapR provides both a community and enterprise edition of its distribution; not all the extensions are available in the free product. The company also offers support services as part of the enterprise product subscription, in addition to training and consultancy.

IBM InfoSphere Big Insights

The last distribution we'll mention here comes from IBM. The **IBM InfoSphere Big Insights** distribution is available at `http://www-01.ibm.com/software/data/infosphere/biginsights/` and (like MapR) offers commercial improvements and extensions to the open source Hadoop core.

Big Insights comes in two versions, the free IBM InfoSphere Big Insights Basic Edition and the commercial IBM InfoSphere Big Insights Enterprise Edition. Big Insights, big names! The basic edition is an enhanced set of Apache Hadoop products, adding some free management and deployment tools as well as integration with other IBM products.

The Enterprise Edition is actually quite different from the Basic Edition; it is more of a layer atop Hadoop, and in fact, can be used with other distributions such as CDH or HDP. The Enterprise Edition provides an array of data visualization, business analysis, and processing tools. It also has deep integration with other IBM products such as InfoSphere Streams, DB2, and GPFS.

Choosing a distribution

As can be seen, the available distributions (and we didn't cover them all) range from convenience packaging and integration of fully open source products through to entire bespoke integration and analysis layers atop them. There is no overall best distribution; think carefully about your needs and consider the alternatives. Since all the previous distributions offer a free download of at least a basic version, it's also good to simply have a try and experience the options for yourself.

Other Apache projects

Whether you use a bundled distribution or stick with the base Apache Hadoop download, you will encounter many references to other, related Apache projects. We have covered Hive, Sqoop, and Flume in this book; we'll now highlight some of the others.

Note that this coverage seeks to point out the highlights (from my perspective) as well as give a taste of the wide range of the types of projects available. As before, keep looking out; there will be new ones launching all the time.

HBase

Perhaps the most popular Apache Hadoop-related project that we didn't cover in this book is **HBase**; its homepage is at `http://hbase.apache.org`. Based on the BigTable model of data storage publicized by Google in an academic paper (sound familiar?), HBase is a non-relational data store sitting atop HDFS.

Whereas both MapReduce and Hive tasks focus on batch-like data access patterns, HBase instead seeks to provide very low latency access to data. Consequently, HBase can, unlike the already mentioned technologies, directly support user-facing services.

The HBase data model is not the relational approach we saw used in Hive and all other RDBMSs. Instead, it is a key-value, schemaless solution that takes a column-oriented view of data; columns can be added at run-time and depend on the values inserted into HBase. Each lookup operation is then very fast as it is effectively a key-value mapping from the row key to the desired column. HBase also treats timestamps as another dimension on the data, so one can directly retrieve data from a point in time.

The data model is very powerful but does not suit all use cases, just as the relational model isn't universally applicable. But if you have a need for structured low-latency views on large-scale data stored in Hadoop, HBase is absolutely something you should look at.

Oozie

We have said many times that Hadoop clusters do not live in a vacuum and need to integrate with other systems and into broader workflows. **Oozie**, available at `http://oozie.apache.org`, is a Hadoop-focused workflow scheduler that addresses this latter scenario.

In its simplest form, Oozie provides mechanisms to schedule the execution of MapReduce jobs based either on a time-based criteria (for example, do this every hour) or on data availability (for example, do this when new data arrives in this location). It allows the specification of multi-stage workflows that can describe a complete end-to-end process.

In addition to straight-forward MapReduce jobs, Oozie can also schedule jobs that run Hive or Pig commands as well as tasks entirely outside of Hadoop (such as sending emails, running shell scripts, or running commands on remote hosts).

There are many ways of building workflows; a common approach is with **Extract Transform and Load** (**ETL**) tools such as **Pentaho Kettle** (`http://kettle.pentaho.com`) and **Spring Batch** (`http://static.springsource.org/spring-batch`). These, for example, do include some Hadoop integration but the traditional dedicated workflow engines may not. Consider Oozie if you are building workflows with significant Hadoop interaction and you do not have an existing workflow tool with which you have to integrate.

Whir

When looking to use cloud services such as Amazon AWS for Hadoop deployments, it is usually a lot easier to use a higher-level service such as ElasticMapReduce as opposed to setting up your own cluster on EC2. Though there are scripts to help, the fact is that the overhead of Hadoop-based deployments on cloud infrastructures can be involved. That is where Apache **Whir** from `http://whir.apache.org` comes in.

Whir is not focused on Hadoop; it is about supplier-independent instantiation of cloud services of which Hadoop is a single example. Whir provides a programmatic way of specifying and creating Hadoop-based deployments on cloud infrastructures in a way that handles all the underlying service aspects for you. And it does this in a provider-independent fashion so that once you've launched on, say, EC2, you can use the same code to create the identical setup on another provider such as Rackspace or Eucalyptus. This makes vendor lock-in—often a concern with cloud deployments—less of an issue.

Whir is not quite there yet. Today it is limited in what services it can create and only supports a single provider, AWS. However, if you are interested in cloud deployment with less pain, it is worth watching its progress.

Mahout

The previous projects are all general-purpose in that they provide a capability that is independent of any area of application. Apache **Mahout**, located at `http://mahout.apache.org`, is instead a library of machine learning algorithms built atop Hadoop and MapReduce.

The Hadoop processing model is often well suited for machine learning applications where the goal is to extract value and meaning from a large dataset. Mahout provides implementations of such common ML techniques as clustering and recommenders.

If you have a lot of data and need help finding the key patterns, relationships, or just the needles in the haystack, Mahout may be able to help.

MRUnit

The final Apache Hadoop project we will mention also highlights the wide range of what is available. To a large extent, it does not matter how many cool technologies you use and which distribution if your MapReduce jobs frequently fail due to latent bugs. The recently promoted MRUnit from `http://mrunit.apache.org` can help here.

Developing MapReduce jobs can be difficult, especially in the early days, but testing and debugging them is almost always hard. MRUnit takes the unit test model of its namesakes such as JUnit and DBUnit and provides a framework to help write and execute tests that can help improve the quality of your code. Build up a test suite, integrate with automated test, and build tools, and suddenly, all those software engineering best practices that you wouldn't dream of not following when writing non-MapReduce code are available here also.

MRUnit may be of interest, well, if you ever write any MapReduce jobs. In my humble opinion, it's a really important project; please check it out.

Other programming abstractions

Hadoop is not just extended by additional functionality; there are tools to provide entirely different paradigms for writing the code used to process your data within Hadoop.

Pig

We mentioned **Pig** (`http://pig.apache.org`) in *Chapter 8*, *A Relational View on Data with Hive*, and won't say much else about it here. Just remember that it is available and may be useful if you have processes or people for whom a data flow definition of the Hadoop processes is a more intuitive or better fit than writing raw MapReduce code or HiveQL scripts. Remember that the major difference is that Pig is an imperative language (it defines how the process will be executed), while Hive is more declarative (defines the desired results but not how they will be produced).

Cascading

Cascading is not an Apache project but is open source and is available at `http://www.cascading.org`. While Hive and Pig effectively define different languages with which to express data processing, Cascading provides a set of higher-level abstractions.

Instead of thinking of how multiple MapReduce jobs may process and share data with Cascading, the model is a data flow using pipes and multiple joiners, taps, and similar constructs. These are built programmatically (the core API was originally Java, but there are numerous other language bindings), and Cascading manages the translation, deployment, and execution of the workflow on the cluster.

If you want a higher-level interface to MapReduce and the declarative style of Pig and Hive doesn't suit, the programmatic model of Cascading may be what you are looking for.

AWS resources

Many Hadoop technologies can be deployed on AWS as part of a self-managed cluster. But just as Amazon offers support for Elastic MapReduce, which handles Hadoop as a managed service, there are a few other services that are worth mentioning.

HBase on EMR

This isn't really a distinct service per se, but just as EMR has native support for Hive and Pig, it also now offers direct support for HBase clusters. This is a relatively new capability, and it will be interesting to see how well it works in practice; HBase has historically been quite sensitive to the quality of the network and system load.

SimpleDB

Amazon SimpleDB (`http://aws.amazon.com/simpledb`) is a service offering an HBase-like data model. This isn't actually implemented atop Hadoop, but we'll mention this and the following service as they do provide hosted alternatives worth considering if a HBase-like data model is of interest. The service has been around for several years and is very mature with well understood use cases.

SimpleDB does have some limitations, particularly on table size and the need to manually partition large datasets, but if you have a need for an HBase-type store at smaller volumes, it may be a good fit. It's also easy to set up and can be a nice way of having a go at the column-based data model.

DynamoDB

A more recent service from AWS is **DynamoDB**, available at `http://aws.amazon.com/dynamodb`. Though its data model is again very similar to that of SimpleDB and HBase, it is aimed at a very different type of application. Where SimpleDB has quite a rich search API but is very limited in terms of size, DynamoDB provides a more constrained API but with a service guarantee of near-unlimited scalability.

The DynamoDB pricing model is particularly interesting; instead of paying for a certain number of servers hosting the service, you allocate a certain read/write capacity and DynamoDB manages the resources required to meet this provisioned capacity. This is an interesting development as it is a purer service model, where the mechanism of delivering the desired performance is kept completely opaque to the service user. Look at DynamoDB if you need a much larger scale of data store than SimpleDB can offer, but do consider the pricing model carefully as provisioning too much capacity can become very expensive very quickly.

Sources of information

You don't just need new technologies and tools, no matter how cool they are. Sometimes, a little help from a more experienced source can pull you out of a hole. In this regard you are well covered; the Hadoop community is extremely strong in many areas.

Source code

It's sometimes easy to overlook, but Hadoop and all the other Apache projects are after all fully open source. The actual source code is the ultimate source (pardon the pun) of information about how the system works. Becoming familiar with the source and tracing through some of the functionality can be hugely informative, not to mention helpful, when you hit unexpected behavior.

Mailing lists and forums

Almost all the projects and services listed earlier have their own mailing lists and/or forums; check out the homepages for the specific links. If using AWS, make sure to check out the AWS developer forums at `https://forums.aws.amazon.com`.

Remember to always read posting guidelines carefully and understand the expected etiquette. These are tremendous sources of information, and the lists and forums are often frequently visited by the developers of the particular project. Expect to see the core Hadoop developers on the Hadoop lists, Hive developers on the Hive list, EMR developers on the EMR forums, and so on.

LinkedIn groups

There are a number of Hadoop and related groups on the professional social network, LinkedIn. Do a search for your particular areas of interest, but a good starting point may be the general Hadoop Users group at `http://www.linkedin.com/groups/Hadoop-Users-988957`.

HUGs

If you want more face-to-face interaction, look for a **Hadoop User Group** (**HUG**) in your area; most should be listed at `http://wiki.apache.org/hadoop/HadoopUserGroups`. These tend to arrange semi-regular get-togethers that combine things such as quality presentations, the ability to discuss technology with like-minded individuals, and often pizza and drinks.

No HUG near where you live? Consider starting one!

Conferences

Though it's a relatively new technology, Hadoop already has some significant conference action involving the open source, academic, and commercial worlds. Events such as the **Hadoop Summit** are pretty big; it and and other events are linked to via `http://wiki.apache.org/hadoop/Conferences`.

Summary

In this chapter, we took a quick gallop around the broader Hadoop ecosystem. We looked at the upcoming changes in Hadoop, particularly HDFS high availability and YARN, why alternative Hadoop distributions exist and some of the more popular ones, and other Apache projects that provide capabilities, extensions, or Hadoop supporting tools.

We also looked at the alternative ways of writing or creating Hadoop jobs and sources of information and how to connect with other enthusiasts.

Now go have fun and build something amazing!

Pop Quiz Answers

Chapter 3, Understanding MapReduce

Pop quiz – key/value pairs

Q1	2
Q2	3

Pop quiz – walking through a run of WordCount

Q1	1
Q2	3
Q3	2. Reducer C cannot be used because if such reduction were to occur, the final reducer could receive from the combiner a series of means with no knowledge of how many items were used to generate them, meaning the overall mean is impossible to calculate. Reducer D is subtle as the individual tasks of selecting a maximum or minimum are safe for use as combiner operations. But if the goal is to determine the overall variance between the maximum and minimum value for each key, this would not work. If the combiner that received the maximum key had values clustered around it, this would generate small results; similarly for the one receiving the minimum value. These subranges have little value in isolation and again the final reducer cannot construct the desired result.

Chapter 7, Keeping Things Running

Pop quiz – setting up a cluster

Q1	5. Though some general guidelines are possible and you may need to generalize whether your cluster will be running a variety of jobs, the best fit depends on the anticipated workload.
Q2	4. Network storage comes in many flavors but in many cases you may find a large Hadoop cluster of hundreds of hosts reliant on a single (or usually a pair) of storage devices. This adds a new failure scenario to the cluster and one with a less uncommon likelihood than many others. Where storage technology does look to address failure mitigation it is usually through disk-level redundancy. These disk arrays can be highly performant but will usually have a penalty on either reads or writes. Giving Hadoop control of its own failure handling and allowing it full parallel access to the same number of disks is likely to give higher overall performance.
Q3	3. Probably! We would suggest avoiding the first configuration as, though it has just enough raw storage and is far from underpowered, there is a good chance the setup will provide little room for growth. An increase in data volumes would immediately require new hosts and additional complexity in the MapReduce job could require additional processor power or memory. Configurations B and C both look good as they have surplus storage for growth and provide similar head-room for both processor and memory. B will have the higher disk I/O and C the better CPU performance. Since the primary job is involved in financial modelling and forecasting, we expect each task to be reasonably heavyweight in terms of CPU and memory needs. Configuration B may have higher I/O but if the processors are running at 100 percent utilization it is likely the extra disk throughput will not be used. So the hosts with greater processor power are likely the better fit. Configuration D is more than adequate for the task and we don't choose it for that very reason; why buy more capacity than we know we need?

Module 2

Hadoop Real-World Solutions Cookbook, Second Edition

Over 90 hands-on recipes to help you learn and master the intricacies of Apache Hadoop 2.X, Yarn, Hive, Pig, Oozie, Flume, Sqoop, Apache Spark, and Mahout

1
Getting Started with Hadoop 2.X

This chapter covers the following topics:

- Installing a single-node Hadoop cluster
- Installing a multi-node Hadoop cluster
- Adding new nodes to existing Hadoop clusters
- Executing the balancer command for uniform data distribution
- Entering and exiting from the safe mode in a Hadoop cluster
- Decommissioning `DataNodes`
- Performing benchmarking on a Hadoop cluster

Introduction

Hadoop has been the primary platform for many people who deal with big data problems. It is the heart of big data. Hadoop was developed way back between 2003 and 2004 when Google published research papers on **Google File System (GFS)** and **Map Reduce**. Hadoop was structured around the crux of these research papers, and thus derived its shape. With the advancement of the Internet and social media, people slowly started realizing the power that Hadoop had, and it soon became the top platform used to handle big data. With a lot of hard work from dedicated contributors and open source groups to the project, Hadoop 1.0 was released and the IT industry welcomed it with open arms.

A lot of companies started using Hadoop as the primary platform for their Data Warehousing and **Extract-Transform-Load** (**ETL**) needs. They started deploying thousands of nodes in a Hadoop cluster and realized that there were scalability issues beyond the 4000+ node clusters that were already present. This was because JobTracker was not able to handle that many Task Trackers, and there was also the need for high availability in order to make sure that clusters were reliable to use. This gave birth to Hadoop 2.0.

In this introductory chapter, we are going to learn interesting recipes such as installing a single/multi-node Hadoop 2.0 cluster, its benchmarking, adding new nodes to existing clusters, and so on. So, let's get started.

Installing a single-node Hadoop Cluster

In this recipe, we are going to learn how to install a single-node Hadoop cluster, which can be used for development and testing.

Getting ready

To install Hadoop, you need to have a machine with the UNIX operating system installed on it. You can choose from any well known UNIX OS such as Red Hat, CentOS, Ubuntu, Fedora, and Amazon Linux (this is in case you are using Amazon Web Service instances).

Here, we will be using the Ubuntu distribution for demonstration purposes.

How to do it...

Let's start installing Hadoop:

1. First of all, you need to download the required installers from the Internet. Here, we need to download Java and Hadoop installers. The following are the links to do this:

   For the Java download, choose the latest version of the available JDK from `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`.

   You can also use Open JDK instead of Oracle.

   For the Hadoop 2.7 Download, go to

   `http://www.eu.apache.org/dist/hadoop/common/hadoop-2.7.0/hadoop-2.7.0.tar.gz`.

2. We will first install Java. Here, I am using `/usr/local` as the installation directory and the `root` user for all installations. You can choose a directory of your choice.

Extract `tar.gz` like this:

```
tar -xzf java-7-oracle.tar.gz
```

Rename the extracted folder to give the shorter name Java instead of java-7-oracle. Doing this will help you remember the folder name easily.

Alternately, you can install Java using the `apt-get` package manager if your machine is connected to the Internet:

```
sudo apt-get update
sudo apt-get install openjdk-7-jdk
```

3. Similarly, we will extract and configure Hadoop. We will also rename the extracted folder for easier accessibility. Here, we will extract Hadoop to `path /usr/local`:

```
tar –xzf hadoop-2.7.0.tar.gz
mv hadoop-2.7.0 hadoop
```

4. Next, in order to use Java and Hadoop from any folder, we would need to add these paths to the `~/.bashrc` file. The contents of the file get executed every time a user logs in:

```
cd ~
vi .bashrc
```

Once the file is open, append the following environment variable settings to it. These variables are used by Java and Hadoop at runtime:

```
export JAVA_HOME=/usr/local/java
export PATH=$PATH:$JAVA_HOME/bin
export HADOOP_INSTALL=/usr/local/hadoop
export PATH=$PATH:$HADOOP_INSTALL/bin
export PATH=$PATH:$HADOOP_INSTALL/sbin
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_HOME=$HADOOP_INSTALL
export HADOOP_HDFS_HOME=$HADOOP_INSTALL
export YARN_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_INSTALL/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_INSTALL/lib"
```

5. In order to verify whether our installation is perfect, close the terminal and restart it again. Also, check whether the Java and Hadoop versions can be seen:

```
$java -version
java version "1.7.0_45"
Java(TM) SE Runtime Environment (build 1.7.0_45-b18)
Java HotSpot(TM) Server VM (build 24.45-b08, mixed mode)
```

```
$ hadoop version
Hadoop 2.7.0
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r
d4c8d4d4d203c934e8074b31289a28724c0842cf
Compiled by jenkins on 2015-04-10T18:40Z
Compiled with protoc 2.5.0
From source with checksum a9e90912c37a35c3195d23951fd18f
```

This command was run using `/usr/local/hadoop/share/hadoop/common/hadoop-common-2.7.0.jar`.

6. Now that Hadoop and Java are installed and verified, we need to install `ssh` (Secure Shell) if it's not already available by default. If you are connected to the Internet, execute the following commands. `SSH` is used to secure data transfers between nodes:

   ```
   sudo apt-get install openssh-client
   ```

   ```
   sudo apt-get install openssh-server
   ```

7. Once the `ssh` installation is done, we need to execute the `ssh` configuration in order to avail a passwordless access to remote hosts. Note that even though we are installing Hadoop on a single node, we need to perform an `ssh` configuration in order to securely access the localhost.

   First of all, we need to generate public and private keys by executing the following command:

   ```
   ssh-keygen -t rsa -P ""
   ```

   This will generate the private and public keys by default in the `$HOME/.ssh` folder. In order to provide passwordless access, we need to append the public key to `authorized_keys` file:

   ```
   cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
   ```

   Let's check whether the `ssh` configuration is okay or not. To test it, execute and connect to the localhost like this:

   ```
   ssh localhost
   ```

   This will prompt you to confirm whether to add this connection to the `known_hosts` file. Type `yes`, and you should be connected to `ssh` without prompting for the password.

8. Once the `ssh` configuration is done and verified, we need to configure Hadoop. The Hadoop configuration begins with adding various configuration parameters to the following default files:

   - `hadoop-env.sh`: This is where we need to perform the Java environment variable configuration.
   - `core-site.xml`: This is where we need to perform `NameNode`-related configurations.

- `yarn-site.xml`: This is where we need to perform configurations related to **Yet Another Resource Negotiator** (**YARN**).
- `mapred-site.xml`: This is where we need to the map reduce engine as YARN.
- `hdfs-site.xml`: This is where we need to perform configurations related to **Hadoop Distributed File System** (**HDFS**).

These configuration files can be found in the `/usr/local/hadoop/etc/hadoop` folder. If you install Hadoop as the root user, you will have access to edit these files, but if not, you will first need to get access to this folder before editing.

So, let's take a look at the configurations one by one.

Configure `hadoop-env.sh` and update the Java path like this:

1. Export `JAVA_HOME=/usr/local/java`.
2. Edit `core-site.xml`, and add the host and port on which you wish to install `NameNode`. Here is the single node installation that we would need in order to add the localhost:

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

3. Edit `yarn-site.xml`, add the following properties to it:

```
<configuration>
   <property>
      <name>yarn.nodemanager.aux-services</name>
      <value>mapreduce_shuffle</value>
   </property>
   <property>
      <name>yarn.nodemanager.aux-services.mapreduce.shuffle.
class</name>
      <value>org.apache.hadoop.mapred.ShuffleHandler</value>
   </property>
</configuration>
```

The `yarn.nodemanager.aux-services` property tells `NodeManager` that an auxiliary service named `mapreduce.shuffle` is present and needs to be implemented. The second property tells `NodeManager` about the class by which means it needs to implement the shuffle auxiliary service. This specific configuration is needed as the `MapReduce` job involves shuffling of key value pairs.

4. Next, edit `mapred-site.xml` to set the map reduce processing engine as YARN:

```
<configuration>
   <property>
      <name>mapreduce.framework.name</name>
      <value>yarn</value>
   </property>
</configuration>
```

5. Edit `hdfs-site.xml` to set the folder paths that can be used by `NameNode` and `datanode`:

```
<configuration>
   <property>
      <name>yarn.nodemanager.aux-services</name>
      <value>mapreduce_shuffle</value>
   </property>
   <property>
      <name>yarn.nodemanager.aux-services.mapreduce.shuffle.
class</name>
      <value>org.apache.hadoop.mapred.ShuffleHandler</value>
   </property>
</configuration>
```

6. I am also setting the HDFS block replication factor to `1` as this is a single node cluster installation.

   We also need to make sure that we create the previously mentioned folders and change their ownership to suit the current user. To do this, you can choose a folder path of your own choice:

```
sudo mkdir –p /usr/local/store/hdfs/namenode
sudo mkdir –p /usr/local/store/hdfs/datanode
sudo chown root:root –R /usr/local/store
```

7. Now, it's time to format `namenode` so that it creates the required folder structure by default:

```
hadoop namenode -format
```

8. The final step involves starting Hadoop daemons; here, we will first execute two scripts to start HDFS daemons and then start YARN daemons:

```
/usr/local/hadoop/sbin/start-dfs.sh
```

This will start `NameNode`, the secondary `NameNode`, and then `DataNode` daemons:

```
/usr/local/hadoop/sbin/start-yarn.sh
```

This will start `NodeManager` and `ResourceManager`. You can execute the `jps` command to take a look at the running daemons:

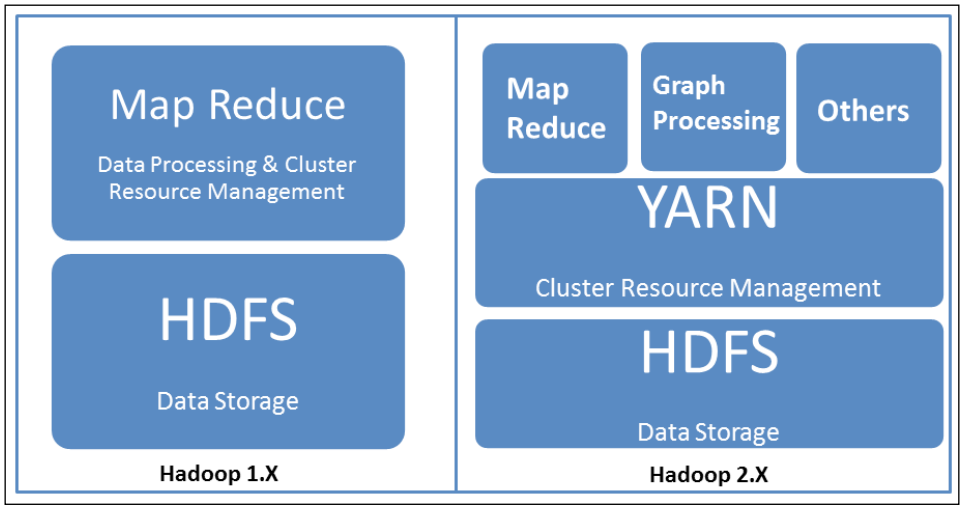
```
$jps
2184 DataNode
2765 NodeManager
2835 Jps
2403 SecondaryNameNode
2025 NameNode
2606 ResourceManager
```

We can also access the web portals for HDFS and YARN by accessing the following URLs:

- For HDFS: `http://<hostname>:50070/`
- For YARN: `http://<hostname>:8088/`

How it works...

Hadoop 2.0 has been majorly reformed in order to solve issues of scalability and high-availability. Earlier in Hadoop 1.0, Map Reduce was the only means of processing data stored in HDFS. With advancement of YARN, Map Reduce is one of the ways of processing data on Hadoop. Here is a pictorial difference between Hadoop 1.x and Hadoop 2.x:



Now, let's try to understand how HDFS and YARN works.

Hadoop Distributed File System (HDFS)

HDFS is a redundant, reliable storage for Hadoop. It consists of three important parts: `NameNode`, the secondary `NameNode`, and `DataNodes`. When a file needs to be processed on Hadoop, it first needs to be saved on HDFS. HDFS distributes the file in chunks of 64/128 MB data blocks across the data nodes. The blocks are replicated across data nodes for reliability. `NameNode` stores the metadata in the blocks and replicas. After a certain period of time, the metadata is backed up on the secondary `NameNode`. The default time is 60 seconds. We can modify this by setting a property called `dfs.namenode.checkpoint.check.period in hdfs-site.xml`.

Yet Another Resource Negotiator (YARN)

YARN has been developed to address scalability issues and for the better management of jobs in Hadoop; till date, it has proved itself to be the perfect solution. It is responsible for the management of resources available in clusters. It consists of two important components: `ResouceManager`(Master) and `NodeManager`(Worker). `NodeManager` provides a node-level view of the cluster, while `ResourceManager` takes a view of a cluster. When an application is submitted by an application client, the following things happen:

- The application talks to `ResourceManager` and provides details about it.
- `ResourceManager` makes a container request on behalf of an application to any of the worker nodes and `ApplicationMaster` starts running within that container.
- `ApplicationMaster` then makes subsequent requests for the containers to execute tasks on other nodes.
- These tasks then take care of all the communication. Once all the tasks are complete, containers are deallocated and `ApplicationMaster` exits.
- After this, the application client also exits.

There's more

Now that your single node Hadoop cluster is up and running, you can try some HDFS file operations on it, such as creating a directory, copying a file from a local machine to HDFS, and so on. Here some sample commands.

To list all the files in the HDFS root directory, take a look at this:

```
hadoop fs –ls /
```

To create a new directory, take a look at this:

```
hadoop fs –mkdir /input
```

To copy a file from the local machine to HDFS, take a look at this:

```
hadoop fs –copyFromLocal /usr/local/hadoop/LICENSE.txt /input
```

In order to access all the command options that are available, go to `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html`.

Installing a multi-node Hadoop cluster

Now that we are comfortable with a single-node Hadoop installation, it's time to learn about a multi-node Hadoop installation.

Getting ready

In the previous recipe, we used a single Ubuntu machine for installation; in this recipe, we will be using three Ubuntu machines. If you are an individual trying to install Hadoop for your own purposes and you don't have three machines to try this recipe, I would suggest that you get three AWS EC2 Ubuntu machines. I am using the `t2.small` type of EC2 instances. For more information on this, go to `https://aws.amazon.com/ec2/`.

Apart from this, I've also performed the following configurations on all the EC2 instances that I have been using:

1. Create an AWS security group to allow access to traffic to EC2 instances, and add EC2 instances to this security group.
2. Change the hostname of EC2 instances to their public hostnames like this:

   ```
   sudo hostname ec2-52-10-22-65.us-west-2.compute.amazonaws.com
   ```

3. Disable firewalls for EC2 Ubuntu instances:

   ```
   sudo ufw disable
   ```

How to do it...

There are a lot of similarities between single-node and multi-node Hadoop installations, so instead of repeating the steps, I would suggest that you refer to earlier recipes as and when they're mentioned. So, let's start installing a multi-node Hadoop cluster:

1. Install Java and Hadoop, as discussed in the previous recipe, on the master and slave nodes. Refer to steps 1-5 in the previous recipe.
2. AWS EC2 has a built-in installation of `ssh` so there's no need to install it again. To configure it, we need to perform the following steps.

First, copy the PEM key with which you initiated EC2 instances to the master node. Next, you need to execute the following set of commands that will add an identity into the master's `ssh` configurations, which can be used to perform passwordless logins to slave machines:

```
eval `ssh-agent -s`
chmod 644 $HOME/.ssh/authorized_keys
chmod 400 <my-pem-key>.pem
ssh-add <my-pem-key>.pem
```

But if you are NOT using AWS EC2, then you need to generate the `ssh` key on the master, and this key needs to be copied to slave machines. Here is a sample command to do this:

```
ssh-keygen -t rsa -P ""
ssh-copy-id -i $HOME/.ssh/id_rsa.pub ubuntu@slave1
ssh-copy-id -i $HOME/.ssh/id_rsa.pub ubuntu@slave2
```

3. Next, we need to perform the Hadoop configurations—most of the configuration files will be same as they were in the case of the single-node installation. These configurations are the same for all the nodes in the cluster. Refer to step 8 from the previous recipe for `hadoop-env.sh`, `mapred-site.xml`, and `hdfs-site.xml`. For `core-site.xml` and `yarn-site.xml`, we need to add some more properties, as shown here:

   Edit `core-site.xml` and add the host and port on which you wish to install `NameNode`. As this is a multi-node Hadoop cluster installation, we will add the master's hostname instead of the localhost:

```
<configuration>
<property>
    <name>fs.default.name</name>
    <value>hdfs://<master's-host-name>:9000/</value>
</property>
</configuration>
```

   Edit `yarn-site.xml` and add the following properties. As this is a multi-node installation, we also need to provide the address of the machine where `ResourceManager` is running:

```
<configuration>
    <property>
      <name>yarn.nodemanager.aux-services</name>
      <value>mapreduce_shuffle</value>
    </property>
    <property>
```

```
      <name>yarn.nodemanager.aux-services.mapreduce.shuffle.
class</name>
      <value>org.apache.hadoop.mapred.ShuffleHandler</value>
    </property>
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value><master's-host-name></value>
    </property>
</configuration>
```

In the case of `hdfs-site.xml`, in the previous recipe, we set the replication factor to 1. As this is a multi-node cluster, we set it to 3. Don't forget to create storage folders configured in `hdfs-site.xml`.

These configurations need to be made on all the machines of the cluster.

4. Now that we are done with configurations, execute the `namenode` format command so that it creates the required subfolder structure:

   `hadoop namenode -format`

5. Now, we need to start specific services on specific nodes in order to start the cluster.

   On the master node, execute following:

   `/usr/local/hadoop/sbin/hadoop-daemon.sh start namenode`

   `/usr/local/hadoop/sbin/hadoop-daemon.sh start secondarynamenode`

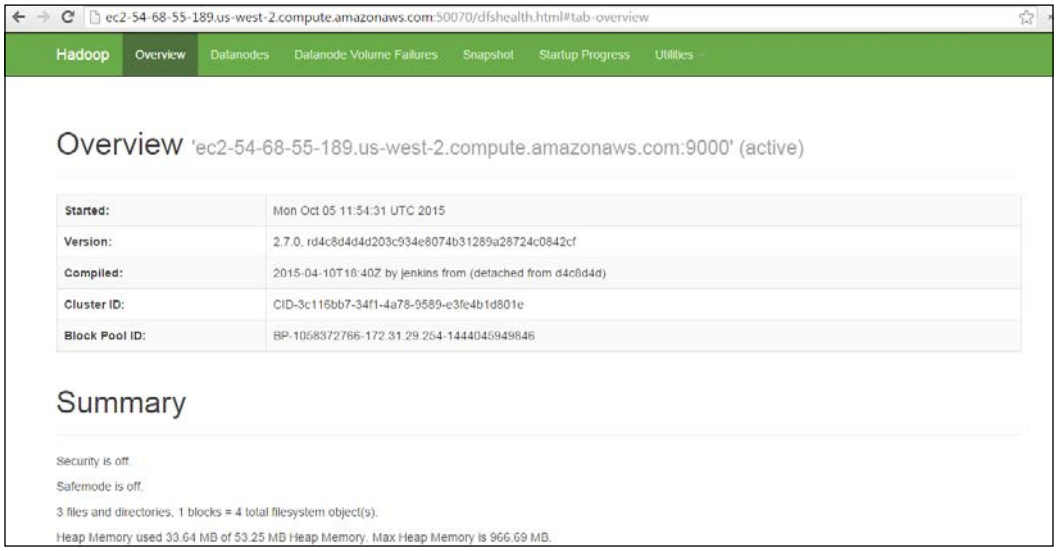
   `/usr/local/hadoop/sbin/yarn-daemon.sh start resourcemanager`

   On all slave nodes, execute following:

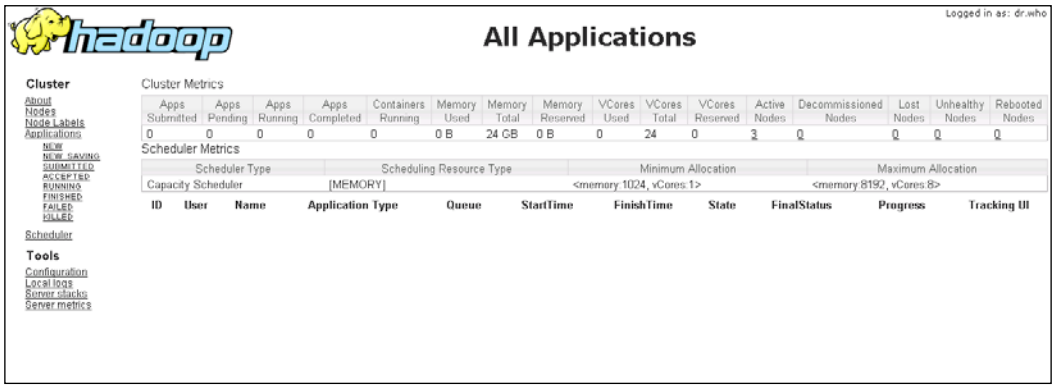
   `/usr/local/hadoop/sbin/hadoop-daemon.sh start datanode`

   `/usr/local/hadoop/sbin/yarn-daemon.sh start nodemanager`

If everything goes well, you should be able to see the cluster running properly. You can also check out the web interfaces for `NameNode` and Resource Managers, for example, by going to `http://<master-ip-hostname>:50070/`.



For `ResourceManager`, go to `http://<master-ip-hostname>/8088`



How it works...

Refer to the *How it works* section from the previous recipe.

Adding new nodes to existing Hadoop clusters

Sometimes, it may happen that an existing Hadoop cluster's capacity is not adequate enough to handle all the data you may want to process. In this case, you can add new nodes to the existing Hadoop cluster without any downtime for the existing cluster. Hadoop supports horizontal scalability.

Getting ready

To perform this recipe, you should have a Hadoop cluster running. Also, you will need one more machine. If you are using AWS EC2, then you can launch an EC2 instance that's similar to what we did in the previous recipes. You will also need the same security group configurations in order to make the installation process smooth.

How to do it...

To add a new instance to an existing cluster, simply install and configure Hadoop the way we did for the previous recipe. Make sure that you put the same configurations in `core-site.xml` and `yarn-site.xml`, which will point to the correct master node.

Once all the configurations are done, simply execute commands to start the newly added `datanode` and `nodemanager`:

```
/usr/local/hadoop/sbin/hadoop-daemon.sh start datanode
/usr/local/hadoop/sbin/yarn-daemon.sh start nodemanager
```

If you take a look at the cluster again, you will find that the new node is registered. You can use the `dfsadmin` command to take a look at the number of nodes and amount of capacity that's been used:

```
hdfs dfsadmin -report
```

Here is a sample output for the preceding command:

```
ubuntu@ec2-52-10-22-65:~$ hdfs dfsadmin -report
15/10/08 08:57:24 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Configured Capacity: 33239728128 (30.96 GB)
Present Capacity: 23809320097 (22.17 GB)
DFS Remaining: 23605534720 (21.98 GB)
DFS Used: 203785377 (194.34 MB)
DFS Used%: 0.86%
Under replicated blocks: 8
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0

-------------------------------------------------
Live datanodes (4):

Name: 172.31.18.55:50010 (ip-172-31-18-55.us-west-2.compute.internal)
Hostname: ip-172-31-18-55.us-west-2.compute.internal
Decommission Status : Normal
Configured Capacity: 8309932032 (7.74 GB)
DFS Used: 1127585 (1.08 MB)
Non DFS Used: 2372033375 (2.21 GB)
DFS Remaining: 5936771072 (5.53 GB)
DFS Used%: 0.01%
DFS Remaining%: 71.44%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Thu Oct 08 08:57:25 UTC 2015


Name: 172.31.0.9:50010 (ip-172-31-0-9.us-west-2.compute.internal)
Hostname: ip-172-31-0-9.us-west-2.compute.internal
Decommission Status : Normal
Configured Capacity: 8309932032 (7.74 GB)
DFS Used: 67551232 (64.42 MB)
Non DFS Used: 2193256448 (2.04 GB)
```

How it works...

Hadoop supports horizontal scalability. If the resources that are being used are not enough, we can always go ahead and add new nodes to the existing cluster without hiccups. In Hadoop, it's always the slave that reports to the master. So, while making configurations, we always configure the details of the master and do nothing about the slaves. This architecture helps achieve horizontal scalability as at any point of time, we can add new nodes by only providing the configurations of the master, and everything else is taken care of by the Hadoop cluster. As soon as the daemons start, the master node realizes that a new node has been added and it becomes part of the cluster.

Executing the balancer command for uniform data distribution

Data in HDFS may not always be placed uniformly. There can be numerous reasons for this. One of the major reasons is the addition of new nodes to the cluster. In such a case, it's the Hadoop administrator's job to make sure that they execute the balancer command to rebalance the data load.

Getting ready

To perform this recipe, you should have performed earlier recipes.

How to do it...

In the previous recipe, we added a new node to the cluster while the other three nodes were already part of the cluster. When you execute the `dfsadmin report` command, you would have noticed that the data is not uniformly balanced because of the addition of a new node. In my case, here is the state of the new node versus the old node.

This is the code for the old node:

```
Name: 172.31.0.9:50010 (ip-172-31-0-9.us-west-2.compute.internal)
Hostname: ip-172-31-0-9.us-west-2.compute.internal
Decommission Status : Normal
Configured Capacity: 8309932032 (7.74 GB)
DFS Used: 67551232 (64.42 MB)
Non DFS Used: 2193256448 (2.04 GB)
DFS Remaining: 6049124352 (5.63 GB)
DFS Used%: 0.81%
DFS Remaining%: 72.79%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Thu Oct 08 08:57:23 UTC 2015
```

This is the code for the new node:

```
Name: 172.31.18.55:50010 (ip-172-31-18-55.us-west-2.compute.internal)
Hostname: ip-172-31-18-55.us-west-2.compute.internal
Decommission Status : Normal
Configured Capacity: 8309932032 (7.74 GB)
DFS Used: 1127585 (1.08 MB)
Non DFS Used: 2372033375 (2.21 GB)
DFS Remaining: 5936771072 (5.53 GB)
DFS Used%: 0.01%
DFS Remaining%: 71.44%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
```

```
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Thu Oct 08 08:57:25 UTC 2015
```

This means that the load on the cluster is not uniform. In this case, we can execute the balancer command to distribute the data uniformly throughout the data nodes:

```
hdfs balancer
```

This will initiate the block balancing activity across the cluster. By default, it will run the balancing activity to make sure that the block storage in the nodes does not differ by more than 10%. You can also decide on the threshold limit by setting an optional parameter called `threshold`:

```
hdfs balancer -threshold 5
```

This will execute the balancer command with 5% threshold. This is how the sample execution looks:

```
ubuntu@ec2-52-10-22-65:~$ hdfs balancer
15/10/08 09:29:07 INFO balancer.Balancer: namenodes  = [hdfs://ec2-52-10-22-65.us-west-2.compute.amazonaws.com:9000]
15/10/08 09:29:07 INFO balancer.Balancer: parameters = Balancer.Parameters[BalancingPolicy.Node, threshold=10.0, max idle iteration = 5, number of nodes to b
e excluded = 0, number of nodes to be included = 0]
Time Stamp               Iteration#  Bytes Already Moved  Bytes Left To Move  Bytes Being Moved
15/10/08 09:29:08 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
15/10/08 09:29:09 INFO net.NetworkTopology: Adding a new node: /default-rack/172.31.0.9:50010
15/10/08 09:29:09 INFO net.NetworkTopology: Adding a new node: /default-rack/172.31.0.8:50010
15/10/08 09:29:09 INFO net.NetworkTopology: Adding a new node: /default-rack/172.31.9.47:50010
15/10/08 09:29:09 INFO net.NetworkTopology: Adding a new node: /default-rack/172.31.18.55:50010
15/10/08 09:29:09 INFO balancer.Balancer: 0 over-utilized: []
15/10/08 09:29:09 INFO balancer.Balancer: 0 underutilized: []
The cluster is balanced. Exiting...
Oct 8, 2015 9:29:09 AM           0                 0 B                 0 B                -1 B
Oct 8, 2015 9:29:09 AM   Balancing took 2.54 seconds
ubuntu@ec2-52-10-22-65:~$
```

How it works...

The balancer command provides instructions to `namenode` so that it can rebalance the data uniformly across `datanode`. This balancing is done by repositioning the blocks placed in `datanode`. So, if a data node is over utilized, some the blocks from that node would be repositioned to the node that is underutilized.

There's more...

There are some options you can provide as arguments to this command:

```
Usage: hdfs balancer
        [-policy <policy>]      the balancing policy: datanode or
blockpool
        [-threshold <threshold>]        Percentage of disk capacity
```

```
        [-exclude [-f <hosts-file> | <comma-separated list of hosts>]]
Excludes the specified datanodes.
        [-include [-f <hosts-file> | <comma-separated list of hosts>]]
Includes only the specified datanodes.
        [-idleiterations <idleiterations>]      Number of consecutive
idle iterations (-1 for Infinite) before exit.
```

Entering and exiting from the safe mode in a Hadoop cluster

Sometimes, due to an unclear filesystem image state, the Hadoop cluster goes into `safemode`. In this recipe, we will see how to enter and exit from `safemode`.

How to do it...

`Safemode` is an HDFS state that does not allow any new writes to the filesystem. It also does not replicate or delete any blocks from the filesystem. In case you want to make any configuration changes to your cluster, you can put the system into `safemode`. Here is a command to enter the system into `safemode`:

```
hdfs dfsadmin -safemode enter
```

Now, if you try to make any writes to HDFS, it will not allow you do so. You can perform cluster maintenance and once this is done, you can switch off `safemode`:

```
hdfs dfsadmin -safemode leave
```

In case you are not aware of whether `safemode` is `ON` or `OFF`, you can get its status by executing the following command:

```
hdfs dfsadmin -safemode get
```

How it works...

Generally, `safemode` is enabled automatically for `NameNode` on startup. It then tries to get the state of the filesystem from `FSImage` and `EditLogs`. `Namenode` waits until `datanodes` start reporting the block status of individual nodes. `safemode` is automatically disabled when `datanodes` report the availability of most of the blocks.

If we enter into `safemode` manually, we need to disable it manually. It won't be disabled automatically.

Decommissioning DataNodes

The Hadoop framework provides us with the option to remove certain nodes from the cluster if they are not needed any more. Here, we cannot simply shutdown the nodes that need to be removed as we might lose some of our data. They need to be decommissioned properly. In this recipe, we are going to learn how to decommission nodes from the Hadoop cluster.

Getting ready

To perform this recipe, you should have a Hadoop cluster, and you should have decided which node to decommission.

How to do it...

To decommission a node from the HDFS cluster, we need to perform the following steps:

1. Create a `dfs.exclude` file in a folder, say `/usr/local/hadoop/etc/hadoop`, and add the hostname of the node you wish to decommission.
2. Edit `hdfs-site.xml` on `NameNode` to append the following property:

```
<property>
    <name>dfs.hosts.exclude</name>
    <value>/usr/local/hadoop/etc/hadoop/dfs.exclude</value>
</property>
```

3. Next, we need to execute the `refreshNodes` command so that it rereads the HDFS configuration in order to start the decommissioning:

```
hdfs dfsadmin -refreshNodes
```

This will start the decommissioning, and once successful execution of the `dfsadmin` report command, you will see that the node's status is changed to `Decommissioned` from `Normal`:

```
hdfs dfsadmin -report
Name: 172.31.18.55:50010 (ip-172-31-18-55.us-west-2.compute.internal)
Hostname: ip-172-31-18-55.us-west-2.compute.internal
Decommission Status : Decommissioned
Configured Capacity: 8309932032 (7.74 GB)
DFS Used: 1179648 (1.13 MB)
Non DFS Used: 2371989504 (2.21 GB)
DFS Remaining: 5936762880 (5.53 GB)
DFS Used%: 0.01%
```

```
DFS Remaining%: 71.44%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Thu Oct 08 10:56:49 UTC 2015
```

Generally, the decommissioning takes time as it requires block replications on other nodes. Once the decommissioning is complete, the node will be added to the decommissioned nodes list.

How it works...

HDFS/`Namenode` reads the configurations from `hdfs-site.xml`. You can configure a file with the list of nodes to decommission and execute the `refreshNodes` command; it then rereads the configuration file. While doing this, it reads the configuration about the decommissioned nodes and will start rereplicating blocks to other available `datanode`. Depending on the size of `datanode` getting decommissioned, the time varies. Unless the completed decommissioning is not completed, it advisable for you to touch `datanode`.

Performing benchmarking on a Hadoop cluster

The Hadoop framework supports built-in libraries so that we can perform benchmarking in order to take a look at how the Hadoop cluster configurations/hardware are performing. There are plenty of tests available that will perform the benchmarking of various aspects of the Hadoop cluster. In this recipe, we are going to take a look at how to perform benchmarking and read the results.

Getting ready

To perform this recipe, you should have a Hadoop cluster up and running.

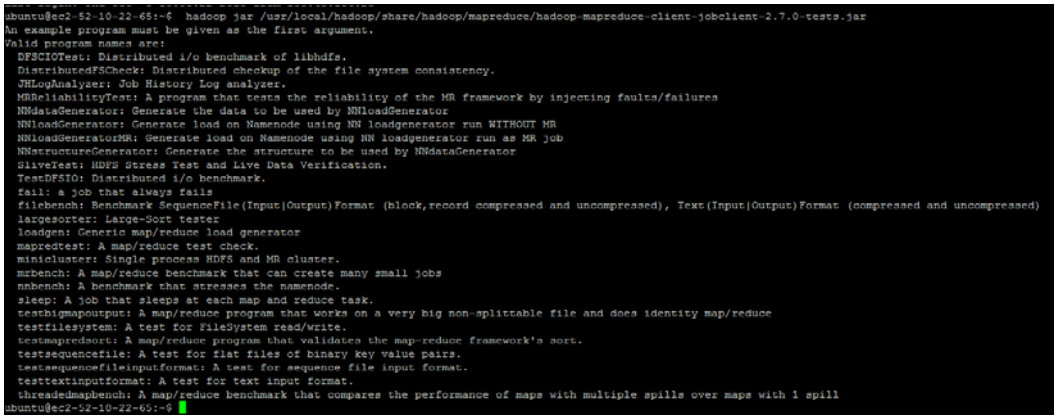
How to do it...

The Hadoop framework supports built-in support to benchmark various aspects. These tests are written in a library called `hadoop-mapreduce-client-jobclient-2.7.0-tests.jar`

To know the list of all the supported tests, you can execute the following command:

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-2.7.0-tests.jar
```

The result of the command will be similar to what is shown in this screenshot:



TestDFSIO

This is one the major tests that you may want to do in order to see how DFS is performing. So, we are now going to take a look at how to use these tests to know how efficiently HDFS is able to write and read data.

As seen in the preceding screenshot, the library provides tools to test DFS through an option called `TestDFSIO`. Now, let's execute the write test in order to understand how efficiently HDFS is able to write big files. The following is the command to execute the write test:

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-2.7.0-tests.jar TestDFSIO -write -nrFiles 2 -fileSize 1GB -resFile /tmp/TestDFSIOwrite.txt
```

Once you initiate the preceding command, a map reduce job will start, which will write two files to HDFS that are 1GB in size . You can choose any numbers based on your cluster size. These tests create data in HDFS under the `/benchmarks` directory. Once the execution is complete, you will see these results:

```
15/10/08 11:37:23 INFO fs.TestDFSIO: ----- TestDFSIO ----- : write
15/10/08 11:37:23 INFO fs.TestDFSIO:            Date & time: Thu Oct 08 11:37:23 UTC 2015
15/10/08 11:37:23 INFO fs.TestDFSIO:        Number of files: 2
15/10/08 11:37:23 INFO fs.TestDFSIO: Total MBytes processed: 2048.0
15/10/08 11:37:23 INFO fs.TestDFSIO:      Throughput mb/sec: 26.637185406776354
```

```
15/10/08 11:37:23 INFO fs.TestDFSIO: Average IO rate mb/sec:
26.63718605041504
15/10/08 11:37:23 INFO fs.TestDFSIO:  IO rate std deviation:
0.00829867575568246
15/10/08 11:37:23 INFO fs.TestDFSIO:     Test exec time sec: 69.023
```

The preceding data is calculated from the RAW data generated by the Map Reduce program. You can also view the raw data as follows:

```
hdfs dfs -cat /benchmarks/TestDFSIO/io_read/part*
f:rate  53274.37
f:sqrate        1419079.2
l:size  2147483648
l:tasks 2
l:time  76885
```



The following formulae are used to calculate throughput, the average IO rate, and standard deviation.

*Throughput = size * 1000/time * 1048576*

Average IO rate = rate/1000/tasks

*Standard deviation = square root of (absolute value(sqrate/1000/ tasks – Average IO Rate * Average IO Rate))*

Similarly, you can perform benchmarking of HDFS read operations as well:

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-
client-jobclient-2.7.0-tests.jar TestDFSIO -read -nrFiles 2 -fileSize 1GB
-resFile /tmp/TestDFSIOread.txt
```

At the end of the execution, a reducer will collect the data from the RAW results, and you will see calculated numbers for the DFSIO reads:

```
15/10/08 11:41:01 INFO fs.TestDFSIO: ----- TestDFSIO ----- : read
15/10/08 11:41:01 INFO fs.TestDFSIO:            Date & time: Thu Oct 08
11:41:01 UTC 2015
15/10/08 11:41:01 INFO fs.TestDFSIO:        Number of files: 2
15/10/08 11:41:01 INFO fs.TestDFSIO: Total MBytes processed: 2048.0
15/10/08 11:41:01 INFO fs.TestDFSIO:      Throughput mb/sec:
33.96633220001659
15/10/08 11:41:01 INFO fs.TestDFSIO: Average IO rate mb/sec:
33.968116760253906
15/10/08 11:41:01 INFO fs.TestDFSIO:  IO rate std deviation:
0.24641533955938721
15/10/08 11:41:01 INFO fs.TestDFSIO:     Test exec time sec: 59.343
```

Here, we can take a look at the RAW data as well:

```
hdfs dfs -cat /benchmarks/TestDFSIO/io_read/part*
f:rate  67936.234
f:sqrate        2307787.2
l:size  2147483648
l:tasks 2
l:time  60295
```

The same formulae are used to calculate the throughput, average IO rate, and standard deviation.

This way, you can benchmark the DFSIO reads and writes.

NNBench

Similar to DFS IO, we can also perform benchmarking for `NameNode`:

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-
client-jobclient-2.7.0-tests.jar nnbench -operation create_write
```

MRBench

MRBench helps us understand the average time taken for a job to execute for a given number of mappers and reducers. The following is a sample command to execute MRBench with default parameters:

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-
client-jobclient-2.7.0-tests.jar mrbench
```

How it works...

Hadoop benchmark tests use the parameters and conditions provided by users. For every test, it executes a map reduce job and once complete, it displays the results on the screen. Generally, it is recommended that you run the benchmarking tests as soon as you have installed the Hadoop cluster in order to predict the performance of HDFS/Map Reduce and so on.

Most of the tests require a sequence in which they should be executed, for example, all write tests should be executed first, then read/delete, and so on.

Once the complete execution is done, make sure you clean up the data in the `/benchmarks` directory in HDFS.

Here is an example command to clean up the data generated by the `TestDFSIO` tests:

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-
client-jobclient-2.7.0-tests.jar TestDFSIO -clean
```

2
Exploring HDFS

In this chapter, we'll take a look at the following recipes:

- Loading data from a local machine to HDFS
- Exporting HDFS data to a local machine
- Changing the replication factor of an existing file in HDFS
- Setting the HDFS block size for all the files in a cluster
- Setting the HDFS block size for a specific file in a cluster
- Enabling transparent encryption for HDFS
- Importing data from another Hadoop cluster
- Recycling deleted data from trash to HDFS
- Saving compressed data in HDFS

Introduction

In the previous chapter, we discussed the installation and configuration details of a Hadoop cluster. In this chapter, we are going to explore the details of HDFS. As we know, Hadoop has two important components:

- **Storage**: This includes HDFS
- **Processing**: This includes Map Reduce

HDFS takes care of the storage part of Hadoop. So, let's explore the internals of HDFS through various recipes.

Loading data from a local machine to HDFS

In this recipe, we are going to load data from a local machine's disk to HDFS.

Getting ready

To perform this recipe, you should have an already Hadoop running cluster.

How to do it...

Performing this recipe is as simple as copying data from one folder to another. There are a couple of ways to copy data from the local machine to HDFS.

- Using the `copyFromLocal` command
  - To copy the file on HDFS, let's first create a directory on HDFS and then copy the file. Here are the commands to do this:

```
hadoop fs -mkdir /mydir1
hadoop fs -copyFromLocal /usr/local/hadoop/LICENSE.txt /
mydir1
```

- Using the `put` command
  - We will first create the directory, and then put the local file in HDFS:

```
hadoop fs -mkdir /mydir2
hadoop fs -put /usr/local/hadoop/LICENSE.txt /mydir2
```

You can validate that the files have been copied to the correct folders by listing the files:

```
hadoop fs -ls /mydir1
hadoop fs -ls /mydir2
```

How it works...

When you use HDFS `copyFromLocal` or the `put` command, the following things will occur:

1. First of all, the HDFS client (the command prompt, in this case) contacts `NameNode` because it needs to copy the file to HDFS.
2. `NameNode` then asks the client to break the file into chunks of different cluster block sizes. In Hadoop 2.X, the default block size is 128MB.
3. Based on the capacity and availability of space in `DataNodes`, `NameNode` will decide where these blocks should be copied.

4. Then, the client starts copying data to specified `DataNodes` for a specific block. The blocks are copied sequentially one after another.
5. When a single block is copied, the block is sent to `DataNode` in packets that are 4MB in size. With each packet, a checksum is sent; once the packet copying is done, it is verified with checksum to check whether it matches. The packets are then sent to the next `DataNode` where the block will be replicated.
6. The HDFS client's responsibility is to copy the data to only the first node; the replication is taken care by respective `DataNode`. Thus, the data block is pipelined from one `DataNode` to the next.
7. When the block copying and replication is taking place, metadata on the file is updated in `NameNode` by `DataNode`.

Exporting HDFS data to a local machine

In this recipe, we are going to export/copy data from HDFS to the local machine.

Getting ready

To perform this recipe, you should already have a running Hadoop cluster.

How to do it...

Performing this recipe is as simple as copying data from one folder to the other. There are a couple of ways in which you can export data from HDFS to the local machine.

- Using the `copyToLocal` command, you'll get this code:

  ```
  hadoop fs -copyToLocal /mydir1/LICENSE.txt /home/ubuntu
  ```

- Using the `get` command, you'll get this code:

  ```
  hadoop fs -get/mydir1/LICENSE.txt /home/ubuntu
  ```

How it works...

When you use HDFS `copyToLocal` or the `get` command, the following things occur:

1. First of all, the client contacts `NameNode` because it needs a specific file in HDFS.
2. `NameNode` then checks whether such a file exists in its `FSImage`. If the file is not present, the error code is returned to the client.
3. If the file exists, `NameNode` checks the metadata for blocks and replica placements in `DataNodes`.

4. `NameNode` then directly points `DataNode` from where the blocks would be given to the client one by one. The data is directly copied from `DataNode` to the client machine. and it never goes through `NameNode` to avoid bottlenecks.
5. Thus, the file is exported to the local machine from HDFS.

Changing the replication factor of an existing file in HDFS

In this recipe, we are going to take a look at how to change the replication factor of a file in HDFS. The default replication factor is 3.

Getting ready

To perform this recipe, you should already have a running Hadoop cluster.

How to do it...

Sometimes. there might be a need to increase or decrease the replication factor of a specific file in HDFS. In this case, we'll use the `setrep` command.

This is how you can use the command:

```
hadoop fs -setrep [-R] [-w] <noOfReplicas><path> ...
```

In this command, a path can either be a file or directory; if its a directory, then it recursively sets the replication factor for all replicas.

- The `w` option flags the command and should wait until the replication is complete
- The `r` option is accepted for backward compatibility

First, let's check the replication factor of the file we copied to HDFS in the previous recipe:

```
hadoop fs -ls /mydir1/LICENSE.txt
-rw-r--r--   3 ubuntu supergroup      15429 2015-10-29 03:04 /mydir1/
LICENSE.txt
```

Once you list the file, it will show you the read/write permissions on this file, and the very next parameter is the replication factor. We have the replication factor set to 3 for our cluster, hence, you the number is `3` .

Let's change it to `2` using this command:

```
hadoop fs -setrep -w 2 /mydir1/LICENSE.txt
```

It will wait till the replication is adjusted. Once done, you can verify this again by running the ls command:

```
hadoop fs -ls /mydir1/LICENSE.txt
-rw-r--r--   2 ubuntu supergroup      15429 2015-10-29 03:04 /mydir1/
LICENSE.txt
```

How it works...

Once the `setrep` command is executed, `NameNode` will be notified, and then `NameNode` decides whether the replicas need to be increased or decreased from certain `DataNode`. When you are using the `–w` command, sometimes, this process may take too long if the file size is too big.

Setting the HDFS block size for all the files in a cluster

In this recipe, we are going to take a look at how to set a block size at the cluster level.

Getting ready

To perform this recipe, you should already have a running Hadoop cluster.

How to do it...

The HDFS block size is configurable for all files in the cluster or for a single file as well. To change the block size at the cluster level itself, we need to modify the `hdfs-site.xml` file.

By default, the HDFS block size is 128MB. In case we want to modify this, we need to update this property, as shown in the following code. This property changes the default block size to 64MB:

```
<property>
<name>dfs.block.size</name>
    <value>67108864</value>
    <description>HDFS Block size</description>
</property>
```

If you have a multi-node Hadoop cluster, you should update this file in the nodes, that is, `NameNode` and `DataNode`. Make sure you save these changes and restart the HDFS daemons:

```
/usr/local/hadoop/sbin/stop-dfs.sh
/usr/local/hadoop/sbin/start-dfs.sh
```

This will set the block size for files that will now get added to the HDFS cluster. Make sure that this does not change the block size of the files that are already present in HDFS. There is no way to change the block sizes of existing files.

How it works...

By default, the HDFS block size is 128MB for Hadoop 2.X. Sometimes, we may want to change this default block size for optimization purposes. When this configuration is successfully updated, all the new files will be saved into blocks of this size. Ensure that these changes do not affect the files that are already present in HDFS; their block size will be defined at the time being copied.

Setting the HDFS block size for a specific file in a cluster

In this recipe, we are going to take a look at how to set the block size for a specific file only.

Getting ready

To perform this recipe, you should already have a running Hadoop cluster.

How to do it...

In the previous recipe, we learned how to change the block size at the cluster level. But this is not always required. HDFS provides us with the facility to set the block size for a single file as well. The following command copies a file called `myfile` to HDFS, setting the block size to 1MB:

```
hadoop fs -Ddfs.block.size=1048576  -put /home/ubuntu/myfile /
```

Once the file is copied, you can verify whether the block size is set to 1MB and has been broken into exact chunks:

```
hdfs fsck -blocks /myfile
    Connecting to namenode via
    http://localhost:50070/fsck?ugi=ubuntu&blocks=1&path=%2Fmyfile
    FSCK started by ubuntu (auth:SIMPLE) from /127.0.0.1 for path
    /myfile at Thu Oct 29 14:58:00 UTC 2015
    .Status: HEALTHY
    Total size:    17276808 B
    Total dirs:    0
```

```
Total files:   1
Total symlinks:                0
Total blocks (validated):      17 (avg. block size 1016282 B)
Minimally replicated blocks:   17 (100.0 %)
Over-replicated blocks:        0 (0.0 %)
Under-replicated blocks:       0 (0.0 %)
Mis-replicated blocks:         0 (0.0 %)
Default replication factor:    1
Average block replication:     1.0
Corrupt blocks:                0
Missing replicas:              0 (0.0 %)
Number of data-nodes:          3
Number of racks:               1
FSCK ended at Thu Oct 29 14:58:00 UTC 2015 in 2 milliseconds

The filesystem under path '/myfile' is HEALTHY
```

How it works...

When we specify the block size at the time of copying a file, it overwrites the default block size and copies the file to HDFS by breaking the file into chunks of a given size. Generally, these modifications are made in order to perform other optimizations. Make sure you make these changes, and you are aware of their consequences. If the block size isn't adequate enough, it will increase the parallelization, but it will also increase the load on NameNode as it would have more entries in `FSImage`. On the other hand, if the block size is too big, then it will reduce the parallelization and degrade the processing performance.

Enabling transparent encryption for HDFS

When handling sensitive data, it is always important to consider the security measures. Hadoop allows us to encrypt sensitive data that's present in HDFS. In this recipe, we are going to see how to encrypt data in HDFS.

Getting ready

To perform this recipe, you should already have a running Hadoop cluster.

How to do it...

For many applications that hold sensitive data, it is very important to adhere to standards such as PCI, HIPPA, FISMA, and so on. To enable this, HDFS provides a utility called encryption zone in which we can create a directory so that data is encrypted on writes and decrypted on read.

To use this encryption facility, we first need to enable Hadoop **Key Management Server** (**KMS**):

```
/usr/local/hadoop/sbin/kms.sh start
```

This would start KMS in the Tomcat web server.

Next, we need to append the following properties in `core-site.xml` and `hdfs-site.xml`.

In `core-site.xml`, add the following property:

```
<property>
    <name>hadoop.security.key.provider.path</name>
    <value>kms://http@localhost:16000/kms</value>
</property>
```

In `hds-site.xml`, add the following property:

```
<property>
    <name>dfs.encryption.key.provider.uri</name>
    <value>kms://http@localhost:16000/kms</value>
</property>
```

Restart the HDFS daemons:

```
/usr/local/hadoop/sbin/stop-dfs.sh
/usr/local/hadoop/sbin/start-dfs.sh
```

Now, we are all set to use KMS. Next, we need to create a key that will be used for the encryption:

```
hadoop key create mykey
```

This will create a key, and then, save it on KMS. Next, we have to create an encryption zone, which is a directory in HDFS where all the encrypted data is saved:

```
hadoop fs -mkdir /zone
hdfs crypto -createZone -keyName mykey -path /zone
```

We will change the ownership to the current user:

```
hadoop fs -chown ubuntu:ubuntu /zone
```

If we put any file into this directory, it will encrypt and would decrypt at the time of reading:

```
hadoop fs -put myfile /zone
hadoop fs -cat /zone/myfile
```

How it works...

There can be various types of encryptions one can do in order to comply with security standards, for example, application-level encryption, database-level, file-level, and disk-level encryption.

The HDFS transparent encryption sits between the database and file-level encryptions. KMS acts like a proxy between HDFS clients and HDFS's encryption provider via HTTP REST APIs. There are two types of keys used for encryption: **Encryption Zone Key**(**EZK**) and **Data Encryption Key** (**DEK**). EZK is used to encrypt DEK, which is also called **Encrypted Data Encryption Key**(**EDEK**). This is then saved on `NameNode`.

When a file needs to be written to the HDFS encryption zone, the client gets EDEK from `NameNode` and EZK from KMS to form DEK, which is used to encrypt data and store it in HDFS (the encrypted zone).

When an encrypted file needs to be read, the client needs DEK, which is formed by combining EZK and EDEK. These are obtained from KMS and `NameNode`, respectively. Thus, encryption and decryption is automatically handled by HDFS. and the end user does not need to worry about executing this on their own.



You can read more on this topic at `http://blog.cloudera.com/blog/2015/01/new-in-cdh-5-3-transparent-encryption-in-hdfs/`.

Importing data from another Hadoop cluster

Sometimes, we may want to copy data from one HDFS to another either for development, testing, or production migration. In this recipe, we will learn how to copy data from one HDFS cluster to another.

Getting ready

To perform this recipe, you should already have a running Hadoop cluster.

How to do it...

Hadoop provides a utility called `DistCp`, which helps us copy data from one cluster to another. Using this utility is as simple as copying from one folder to another:

```
hadoop distcp hdfs://hadoopCluster1:9000/source hdfs://
hadoopCluster2:9000/target
```

This would use a Map Reduce job to copy data from one cluster to another. You can also specify multiple source files to be copied to the target. There are a couple of other options that we can also use:

- `-update`: When we use `DistCp` with the update option, it will copy only those files from the source that are not part of the target or differ from the target.
- `-overwrite`: When we use `DistCp` with the overwrite option, it overwrites the target directory with the source.

How it works...

When `DistCp` is executed, it uses map reduce to copy the data and also assists in error handling and reporting. It expands the list of source files and directories and inputs them to map tasks. When copying from multiple sources, collisions are resolved in the destination based on the option (update/overwrite) that's provided. By default, it skips if the file is already present at the target. Once the copying is complete, the count of skipped files is presented.



You can read more on `DistCp` at `https://hadoop.apache.org/docs/current/hadoop-distcp/DistCp.html`.

Recycling deleted data from trash to HDFS

In this recipe, we are going to see how to recover deleted data from the trash to HDFS.

Getting ready

To perform this recipe, you should already have a running Hadoop cluster.

How to do it...

To recover accidently deleted data from HDFS, we first need to enable the trash folder, which is not enabled by default in HDFS. This can be achieved by adding the following property to `core-site.xml`:

```
<property>
    <name>fs.trash.interval</name>
    <value>120</value>
</property>
```

Then, restart the HDFS daemons:

```
/usr/local/hadoop/sbin/stop-dfs.sh
/usr/local/hadoop/sbin/start-dfs.sh
```

This will set the deleted file retention to 120 minutes.

Now, let's try to delete a file from HDFS:

```
hadoop fs -rmr /LICENSE.txt
    15/10/30 10:26:26 INFO fs.TrashPolicyDefault: Namenode trash
    configuration: Deletion interval = 120 minutes, Emptier interval
    = 0 minutes.
    Moved: 'hdfs://localhost:9000/LICENSE.txt' to trash at:
    hdfs://localhost:9000/user/ubuntu/.Trash/Current
```

We have 120 minutes to recover this file before it is permanently deleted from HDFS. To restore the file to its original location, we can execute the following commands.

First, let's confirm whether the file exists:

```
hadoop fs -ls /user/ubuntu/.Trash/Current
    Found 1 items
    -rw-r--r--   1 ubuntu supergroup      15429 2015-10-30 10:26
    /user/ubuntu/.Trash/Current/LICENSE.txt
```

Now, restore the deleted file or folder; it's better to use the `distcp` command instead of copying each file one by one:

hadoop `distcp` hdfs

```
//localhost:9000/user/ubuntu/.Trash/Current/LICENSE.txt hdfs://
localhost:9000/
```

This will start a map reduce job to restore data from the trash to the original HDFS folder. Check the HDFS path; the deleted file should be back to its original form.

How it works...

Enabling trash enforces the file retention policy for a specified amount of time. So, when trash is enabled, HDFS does not execute any blocks deletions or movements immediately but only updates the metadata of the file and its location. This way, we can accidently stop deleting files from HDFS; make sure that trash is enabled before experimenting with this recipe.

Saving compressed data in HDFS

In this recipe, we are going to take a look at how to store and process compressed data in HDFS.

Getting ready

To perform this recipe, you should already have a running Hadoop.

How to do it...

It's always good to use compression while storing data in HDFS. HDFS supports various types of compression algorithms such as LZO, BIZ2, Snappy, GZIP, and so on. Every algorithm has its own pros and cons when you consider the time taken to compress and decompress and the space efficiency. These days people prefer Snappy compression as it aims to achieve a very high speed and a reasonable amount of compression.

We can easily store and process any number of files in HDFS. To store compressed data, we don't need to specifically make any changes to the Hadoop cluster. You can simply copy the compressed data in the same way it's in HDFS. Here is an example of this:

```
hadoop fs -mkdir /compressed
hadoop fs –put file.bz2 /compressed
```

Now, we'll run a sample program to take a look at how Hadoop automatically uncompresses the file and processes it:

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-
examples-2.7.0.jar wordcount /compressed /compressed_out
```

Once the job is complete, you can verify the output.

How it works...

Hadoop explores native libraries to find the support needed for various codecs and their implementations. Native libraries are specific to the platform that you run Hadoop on. You don't need to make any configuration changes to enable compression algorithms. As mentioned earlier, Hadoop supports various compression algorithms that are already familiar to the computer world. Based on your needs and requirements (more space or more time), you can choose your compression algorithm.

Take a look at `http://comphadoop.weebly.com/` for more information on this.



Downloading the example code

You can download the example code files for this book 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.

You can download the code files by following these steps:

- Log in or register to our website using your e-mail address and password
- Hover the mouse pointer on the SUPPORT tab at the top.
- Click on Code Downloads & Errata
- Enter the name of the book in the Search box
- Select the book for which you're looking to download the code files
- Choose from the drop-down menu where you purchased this book from
- Click on Code Download

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

3
Mastering Map Reduce Programs

In this chapter, we'll cover the following recipes:

- Writing the Map Reduce program in Java to analyze web log data
- Executing the Map Reduce program in a Hadoop cluster
- Adding support for a new writable data type in Hadoop
- Implementing a user-defined counter in a Map Reduce program
- Map Reduce program to find the top X
- Map Reduce program to find distinct values
- Map Reduce program to partition data using a custom partitioner
- Writing Map Reduce results to multiple output files
- Performing Reduce side Joins using Map Reduce
- Unit testing the Map Reduce code using MRUnit

Introduction

Hadoop consists of two important components: HDFS and Map Reduce. In the previous chapter, we talked about various recipes one can perform to use and maintain HDFS in a good state. In this chapter, we are going to explore the details of the Map Reduce framework. Hadoop itself is written in Java, and Java is, of course, a preferred way to write Map Reduce programs, but this does not restrict you to only using Java. It provides libraries, such as Hadoop-Streaming, Hadoop Pipes, and so on so that you can write map reduce programs in most popular languages. These include C++, C#, Shell Scripting using Hadoop-Streaming, and so on.

Writing the Map Reduce program in Java to analyze web log data

In this recipe, we are going to take a look at how to write a map reduce program to analyze web logs. Web logs are data that is generated by web servers for requests they receive. There are various web servers such as Apache, Nginx, Tomcat, and so on. Each web server logs data in a specific format. In this recipe, we are going to use data from the Apache Web Server, which is in combined access logs.



To read more on combined access logs, refer to `http://httpd.apache.org/docs/1.3/logs.html#combined`.

Getting ready

To perform this recipe, you should already have a running Hadoop cluster as well as an eclipse similar to an IDE.

How to do it...

We can write map reduce programs to analyze various aspects of web log data. In this recipe, we are going to write a map reduce program that reads a web log file, results pages, views, and their counts. Here is some sample web log data we'll consider as input for our program:

```
106.208.17.105 - - [12/Nov/2015:21:20:32 -0800] "GET /tutorials/
mapreduce/advanced-map-reduce-examples-1.html HTTP/1.1" 200 0
"https://www.google.co.in/" "Mozilla/5.0 (Windows NT 6.3; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.86 Safari/537.36"

60.250.32.153 - - [12/Nov/2015:21:42:14 -0800] "GET /tutorials/
elasticsearch/install-elasticsearch-kibana-logstash-on-windows.html
HTTP/1.1" 304 0 - "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/46.0.2490.86 Safari/537.36"

49.49.250.23 - - [12/Nov/2015:21:40:56 -0800] "GET /tutorials/hadoop/
images/internals-of-hdfs-file-read-operations/HDFS_Read_Write.png
HTTP/1.1" 200 0 "http://hadooptutorials.co.in/tutorials/spark/install-
apache-spark-on-ubuntu.html" "Mozilla/5.0 (Windows NT 10.0; WOW64;
Trident/7.0; Touch; LCTE; rv:11.0) like Gecko"

60.250.32.153 - - [12/Nov/2015:21:36:01 -0800] "GET /tutorials/
elasticsearch/install-elasticsearch-kibana-logstash-on-windows.html
HTTP/1.1" 200 0 - "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/46.0.2490.86 Safari/537.36"
```

```
91.200.12.136 - - [12/Nov/2015:21:30:14 -0800] "GET /tutorials/hadoop/
hadoop-fundamentals.html HTTP/1.1" 200 0 "http://hadooptutorials.co.in/
tutorials/hadoop/hadoop-fundamentals.html" "Mozilla/5.0 (Windows NT 6.1)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.99 Safari/537.36"
```

These combined Apache Access logs are in a specific format. Here is the sequence and meaning of each component in each access log:

- **%h**: This is the remote host (that is, the IP client)
- **%l**: This is the identity of the user determined by an identifier (this is not usually used since it's not reliable)
- **%u**: This is the username determined by the HTTP authentication
- **%t**: This is the time the server takes to finish processing a request
- **%r**: This is the request line from the client ("`GET / HTTP/1.0`")
- **%>s**: This is the status code sent from a server to a client (200, 404, and so on)
- **%b**: This is the size of the response given to a client (in bytes)
- **Referrer**: This is the page that is linked to this URL
- **User agent**: This is the browser identification string

Now, let's start a writing program in order to get to know the page views of each unique URL that we have in our web logs.

First, we will write a mapper class where we will read each and every line and parse it to the extract page URL. Here, we will use a Java pattern that matches a utility in order to extract information:

```
public static class PageViewMapper extends Mapper<Object, Text, Text,
IntWritable> {
        public static String APACHE_ACCESS_LOGS_PATTERN = "^(\\S+) (\\
S+) (\\S+) \\[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(\\S+) (\\S+) (\\S+)\"
(\\d{3}) (\\d+) (.+?) \"([^\"]+|(.+?))\"";

        public static Pattern pattern = Pattern.compile(APACHE_ACCESS_
LOGS_PATTERN);

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

        public void map(Object key, Text value, Mapper<Object, Text,
Text, IntWritable>.Context context)
                throws IOException, InterruptedException {
        Matcher matcher = pattern.matcher(value.toString());
            if (matcher.matches()) {
                // Group 6 as we want only Page URL
```

```
                url.set(matcher.group(6));
                System.out.println(url.toString());
                context.write(this.url, one);
            }

        }
    }
```

In the preceding mapper class, we read key value pairs from the text file. By default, the key is a byte offset (the number of characters in a line), and the value is an actual line in a text file. Next, we match the line with the Apache Access Log regex pattern so that we can extract the exact information we need. For a page view counter, we only need a URL. Mapper outputs the URL as a key and 1 as the value. So, we can count these URL in reducer.

Here is the reducer class that sums up the output values of the mapper class:

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

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

Now, we just need a driver class to call these mappers and reducers:

```
public class PageViewCounter {

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        if (args.length != 2) {
            System.err.println("Usage: PageViewCounter <in><out>");
            System.exit(2);
        }
```

```
        Job job = Job.getInstance(conf, "Page View Counter");
        job.setJarByClass(PageViewCounter.class);
        job.setMapperClass(PageViewMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));

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

As the operation we are performing is aggregation, we can also use a combiner here to optimize the results. Here, the same reducer logic is being used as the one used for the combiner.

To compile your program properly, you need to add two external JARs, `hadoop-common-2.7.jar`, which can be found in the `/usr/local/hadoop/share/hadoop/common` folder and `hadoop-mapreduce-client-core-2.7.jar`, which can be found in the `/usr/local/hadoop/share/hadoop/mapreduce` path.

Make sure you add these two JARs in your build path so that your program can be compiled easily.

How it works...

The page view counter program helps us find the most popular pages, least accessed pages, and so on. Such information helps us make decisions about the ranking of pages, frequency of visits, and the relevance of a page. When a program is executed, each line of the HDFS block is read individually and then sent to Mapper. Mapper matches the input line with the log format and extracts its page URL. Mapper emits the (URL,1) type of key value pairs. These pairs are shuffled across nodes and partitioners to make sure that a similar URL goes to only one reducer. Once received by the reducers, we add up all the values for each key and emit them. This way, we get results in the form of a URL and the number of times it was accessed.

Executing the Map Reduce program in a Hadoop cluster

In the previous recipe, we took a look at how to write a map reduce program for a page view counter. In this recipe, we will explore how to execute this in a Hadoop cluster.

Getting ready

To perform this recipe, you should already have a running Hadoop cluster as well as an eclipse similar to an IDE.

How to do it

To execute the program, we first need to create a JAR file of it. JAR stands for Java Archive file, which contains compiled class files. To create a JAR file in eclipse, we need to perform the following steps:

1. Right-click on the project where you've written your Map Reduce Program. Then, click on **Export**.
2. Select **Java**->**Jar File** and click on the **Next** button. Browse through the path where you wish to export the JAR file, and provide a proper name to the jar file. Click on **Finish** to complete the creation of the JAR file.
3. Now, copy this file to the Hadoop cluster. If you have your Hadoop cluster running in the AWS EC2 instance, you can use the following command to copy the JAR file:

   ```
   scp –i mykey.pem logAnalyzer.jar ubuntu@ec2-52-27-157-247.us-west-2.compute.amazonaws.com:/home/ubuntu
   ```

4. If you don't already have your input log files in HDFS, use following commands:

   ```
   hadoop fs –mkdir /logs

   hadoop fs –put web.log /logs
   ```

5. Now, it's time to execute the map reduce program. Use the following command to start the execution:

   ```
   hadoop jar logAnalyzer.jar com.demo.PageViewCounter /logs /pageview_output
   ```

6. This will start the Map Reduce execution on your cluster. If everything goes well, you should be able to see output in the `pageview_output` folder in HDFS. Here, `logAnalyzer` is the name of the JAR file we created through eclipse. `logs` is the folder we have our input data in, while `pageview_output` is the folder that will first be created, and then results will be saved into. It is also important to provide a fully qualified name to the class along with its package name.

How it works...

Once the job is submitted, it first creates the Application Client and Application Master in the Hadoop cluster. The application tasks for Mapper are initiated in each node where data blocks are present in the Hadoop cluster. Once the Mapper phase is complete, the data is locally reduced by a combiner. Once the combiner finishes, the data is shuffled across the nodes in the cluster. Unless all the mappers have finished, reducers cannot be started. Output from the reducers is also written to HDFS in a specified folder.



The output folder to be specified should be a nonexisting folder in HDFS. If the folder is already present, then the program will give you an error.

When all the tasks are finished for the application, you can take a look at the output in HDFS. The following are the commands to do this:

```
hadoop fs –ls /pageview_output
hadoop fs –cat /pageview_output/part-m-00000
```

This way, you can write similar programs for the following:

- Most number of referral sites (hint: use a referral group from the matcher)
- Number of client errors (with the Http status of 4XX)
- Number of of server errors (with the Http status of 5XX)

Adding support for a new writable data type in Hadoop

In this recipe, we are going to learn how to introduce a new data type in Map Reduce programs and then use it.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as an eclipse that's similar to an IDE.

How to do it...

Hadoop allows us to add new custom data types ,which are made up of one or more primary data types. In the previous recipe, you must have noticed that when you handled the log data structure, you had to remember the sequence in which each data component was placed. This can get very nasty when it comes to complex programs. To avoid this, we will introduce a new data type in which `WritableComparable` can be used efficiently.

To add a new data type, we need to implement the `WritableComparable` interface, which is provided by Hadoop. This interface provides three methods, `readFields(DataInput in)`, `write(DataOut out)`, and `compareTo(To)`, which we will need to override with our own custom implementation. Here, we are going to abstract the log parsing and pattern matching logic from the user of this data type by providing a method that returns parsed objects:

```
public class ApacheAccessLogWritable implements WritableComparable<Apa
cheAccessLogWritable> {

    public static String APACHE_ACCESS_LOGS_PATTERN = "^(\\S+) (\\
S+) (\\S+) \\[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(\\S+) (\\S+) (\\S+)\"
(\\d{3}) (\\d+) (.+?) \"([^\"]+|(.+?))\"";
    public static Pattern pattern = Pattern.compile(APACHE_ACCESS_
LOGS_PATTERN);

    private Text clientIp, id, username, dateString, httpMethod, url,
httpVersion, referral, browserString;
    private IntWritable httpStatus, requestSize;

    @Override
    public void readFields(DataInput in) throws IOException {
        clientIp.readFields(in);
        id.readFields(in);
        username.readFields(in);
        dateString.readFields(in);
        httpMethod.readFields(in);
        url.readFields(in);
        httpVersion.readFields(in);
        referral.readFields(in);
        browserString.readFields(in);
        httpStatus.readFields(in);
        requestSize.readFields(in);

    }
```

```
    @Override
    public void write(DataOutput out) throws IOException {
        clientIp.write(out);
        id.write(out);
        username.write(out);
        dateString.write(out);
        httpMethod.write(out);
        url.write(out);
        httpVersion.write(out);
        referral.write(out);
        browserString.write(out);
        httpStatus.write(out);
        requestSize.write(out);

    }

    private ApacheAccessLogWritable(Text clientIp, Text id, Text
username, Text dateString, Text httpMethod, Text url,
            Text httpVersion, IntWritable httpStatus, IntWritable
requestSize, Text referral, Text browserString) {

        this.clientIp = clientIp;
        this.id = id;
        this.username = username;
        this.dateString = dateString;
        this.httpMethod = httpMethod;
        this.url = url;
        this.httpVersion = httpVersion;
        this.referral = referral;
        this.browserString = browserString;
        this.httpStatus = httpStatus;
        this.requestSize = requestSize;
    }

    public static ApacheAccessLogWritable parseFromLogLine(String
logline) {
        Matcher m = pattern.matcher(logline);
        if (!m.find()) {

            throw new RuntimeException("Error parsing logline");
        }

        return new ApacheAccessLogWritable(new Text(m.group(1)), new
Text(m.group(2)), new Text(m.group(3)),
```

```
                new Text(m.group(4)), new Text(m.group(5)), new
Text(m.group(6)), new Text(m.group(7)),
                new IntWritable(Integer.parseInt(m.group(8))), new
IntWritable(Integer.parseInt(m.group(9))),
                new Text(m.group(10)), new Text(m.group(11)));
    }

    @Override
    public int compareTo(ApacheAccessLogWritable o) {
        // TODO Auto-generated method stub
        return 0;
    }
    // Getter and Setter methods
    ..
}
```

The following piece of code shows us how we can use the data type in our map reduce program; here, I am going to update the same program that we used in the previous recipe. So, the mapper code will be updated as follows:

```
public static class PageViewMapper extends Mapper<Object, Text, Text,
IntWritable> {

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

        public void map(Object key, Text value, Mapper<Object, Text,
Text, IntWritable>.Context context)
                throws IOException, InterruptedException {
            ApacheAccessLogWritable log = ApacheAccessLogWritable.
parseFromLogLine(value.toString());
            context.write(log.getUrl(), one);

        }
    }
```

The highlighted code shows you where we have used our own custom data type. Here, the reducer and driver code remain as it is. Refer to the previous recipe to know more about these two.

To execute this code, we need to bundle the datatype class and map reduce program into a single JAR itself so that at runtime the map reduce code is able to find our newly introduced data type.

How it works...

We know that when we execute the map reduce code, a lot of data gets transferred over the network when this shuffle takes place. Sometimes, the size of the keys and values that are transferred can be huge, which might affect network traffic. To avoid congestion, it's very important to send the data in a serialized format. To abstract the pain of serialization and deserialization from the map reduce programmer, Hadoop has introduced a set of box/wrapper classes such as `IntWritable`, `LongWritable`, `Text`, `FloatWritable`, `DoubleWritable`, and so on. These are wrapper classes on top of primitive data types, which can be serialized and deserialized easily. The keys need to be `WritableComparable`, while the values need to be `Writable`. Technically, both keys and values are `WritableComparable`.

Apart from the set of built-in data types, Hadoop also supports the introduction of custom and new data types that are `WritableComparable`. This is done so that the handling of complex data becomes easy and serialization and deserialization is taken care of automatically. `WritableComparable` are data types that are easy to serialize and deserialize and can be compared in order to decide what their order is.

Implementing a user-defined counter in a Map Reduce program

In this recipe, we are going to learn how to add a user-defined counter so that we can keep track of certain events easily.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as an eclipse that's similar to an IDE.

How to do it...

After every map reduce execution, you will see a set of system defined counters getting published, such as File System counters, Job counters, and Map Reduce Framework counters. These counters help us understand the execution in detail. They give very detailed information about the number of bytes written to HDFS, read from HDFS, the input given to a map, the output received from a map, and so on. Similar to this information, we can also add our own user-defined counters, which will help us track the execution in a better manner.

In earlier recipes, we considered the use case of log analytics. There can be chances that the input we receive might always not be in the same format as we expect it to be. So, its very important to track such bad records and also avoid any failures because of them. In order to achieve this, in this recipe, we are going to add one custom counter that keeps track of such bad records without failing the task.

First of all, we have to define the counter as `enum` in our program:

```
private enum COUNTERS {
        INVALID_RECORD_COUNT
    }
```

Next, we will update our mapper code to use the defined counter, as shown here:

```
public static class PageViewMapper extends Mapper<Object, Text, Text,
IntWritable> {

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

        public void map(Object key, Text value, Mapper<Object, Text,
Text, IntWritable>.Context context)
                throws IOException, InterruptedException {
            ApacheAccessLogWritable log = null;
            try {
                // If record is in expected format, do normal
processing
                log = ApacheAccessLogWritable.parseFromLogLine(value.
toString());
                context.write(log.getUrl(), one);
                } catch (Exception e) {
                // if not, increment the invalid record counter
                System.out.println("Invalid record found");
                context.getCounter(COUNTERS.INVALID_RECORD_COUNT).
increment(1L);

            }

        }
    }
```

The reducer code will remain as it is while we will update the driver code to print the final count of invalid records, as shown here:

```
public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        if (args.length != 2) {
            System.err.println("Usage: PageViewCounter <in><out>");
            System.exit(2);
        }
        Job job = Job.getInstance(conf, "Page View Counter");
        job.setJarByClass(PageViewCounter.class);
        job.setMapperClass(PageViewMapper.class);
```

```
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
        org.apache.hadoop.mapreduce.Counters counters = job.
getCounters();
        System.out.println("No. of Invalid Records :" + counters.
findCounter(COUNTERS.INVALID_RECORD_COUNT).getValue());
    }
```

Now, to demonstrate, I've added a few invalid records (records with fewer columns than expected) and added the log file to HDFS. So, when I execute the program, I can see the invalid record count getting printed at the end of the execution:

```
$hadoop jar logAnalyzer.jar com.demo.PageViewCounter /log-input /log-
output
15/11/15 08:44:37 INFO client.RMProxy: Connecting to ResourceManager
at /0.0.0.0:8032
15/11/15 08:44:37 WARN mapreduce.JobResourceUploader: Hadoop command-
line option parsing not performed. Implement the Tool interface and
execute your application with ToolRunner to remedy this.
15/11/15 08:44:37 INFO input.FileInputFormat: Total input paths to
process : 2
15/11/15 08:44:37 INFO mapreduce.JobSubmitter: number of splits:2
15/11/15 08:44:38 INFO mapreduce.JobSubmitter: Submitting tokens for
job: job_1447554086128_0003
15/11/15 08:44:38 INFO impl.YarnClientImpl: Submitted application
application_1447554086128_0003
15/11/15 08:44:38 INFO mapreduce.Job: The url to track the job:
http://admin1:8088/proxy/application_1447554086128_0003/
15/11/15 08:44:38 INFO mapreduce.Job: Running job:
job_1447554086128_0003
15/11/15 08:44:43 INFO mapreduce.Job: Job job_1447554086128_0003
running in uber mode : false
15/11/15 08:44:43 INFO mapreduce.Job:  map 0% reduce 0%
15/11/15 08:44:50 INFO mapreduce.Job:  map 100% reduce 0%
15/11/15 08:44:55 INFO mapreduce.Job:  map 100% reduce 100%
15/11/15 08:44:55 INFO mapreduce.Job: Job job_1447554086128_0003
completed successfully
15/11/15 08:44:55 INFO mapreduce.Job: Counters: 50
    File System Counters
        FILE: Number of bytes read=580
```

```
        FILE: Number of bytes written=345070
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=3168
        HDFS: Number of bytes written=271
        HDFS: Number of read operations=9
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2
    Job Counters
        Launched map tasks=2
        Launched reduce tasks=1
        Data-local map tasks=2
        Total time spent by all maps in occupied slots (ms)=8542
        Total time spent by all reduces in occupied slots (ms)=3046
        Total time spent by all map tasks (ms)=8542
        Total time spent by all reduce tasks (ms)=3046
        Total vcore-seconds taken by all map tasks=8542
        Total vcore-seconds taken by all reduce tasks=3046
        Total megabyte-seconds taken by all map tasks=8747008
        Total megabyte-seconds taken by all reduce tasks=3119104
    Map-Reduce Framework
        Map input records=13
        Map output records=10
        Map output bytes=724
        Map output materialized bytes=586
        Input split bytes=201
        Combine input records=10
        Combine output records=8
        Reduce input groups=4
        Reduce shuffle bytes=586
        Reduce input records=8
        Reduce output records=4
        Spilled Records=16
        Shuffled Maps =2
        Failed Shuffles=0
        Merged Map outputs=2
        GC time elapsed (ms)=69
        CPU time spent (ms)=1640
        Physical memory (bytes) snapshot=591077376
        Virtual memory (bytes) snapshot=1219731456
        Total committed heap usage (bytes)=467140608
    Shuffle Errors
        BAD_ID=0
```

```
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    com.demo.PageViewCounter$COUNTERS
        INVALID_RECORD_COUNT=3
    File Input Format Counters
        Bytes Read=2967
    File Output Format Counters
        Bytes Written=271
```

How it works...

Custom counters are helpful in various situations such as keeping track of bad records, count outliers in the form of maximum and minimum values, summations, and so on. The Hadoop framework imposes an upper limit on using these counters. They can be incremented/decremented globally, or you may also update them in mappers or reducers. In either case, they are referred to using the group and counter names. All the counters are managed at the Application Master level. Information about each increment or decrement is passed to the Application Master via heartbeat messages between the containers that run mappers and reducers.

It is better to keep the counters to a limited number as this causes an overhead on the processing framework. The best thing to do is to remember a thumb rule: do not let the number of counters go beyond 100.

Map Reduce program to find the top X

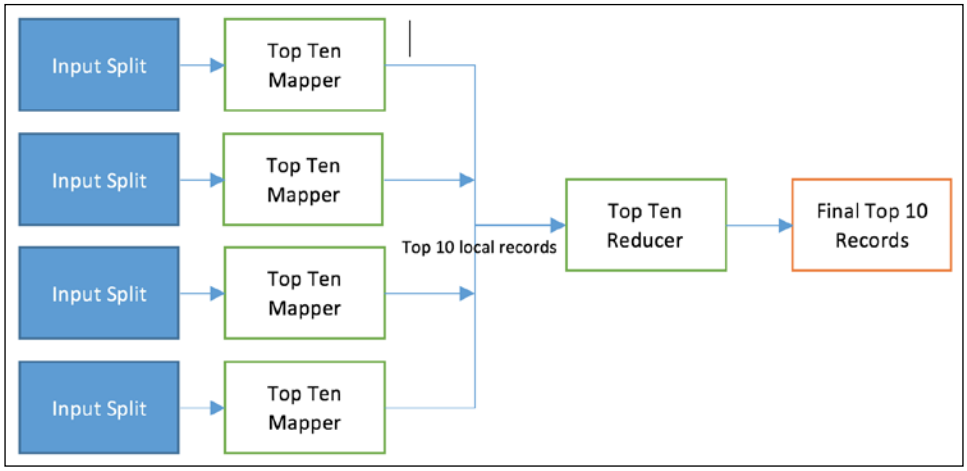
In this recipe, we are going to learn how to write a map reduce program to find the top X records from the given set of values.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as an eclipse that's similar to an IDE.

How to do it...

A lot of the time, we might need to find the top X values from the given set of values. A simple example could be to find the top 10 trending topics from a Twitter dataset. In this case, we will need to use two map reduce jobs. First of all, find out all the words that start with # and the number of times each hashtag has occurred in a given set of data. The first map reduce program is quite simple, which is pretty similar to the word count program. But for the second program, we need to use some logic. In this recipe, we'll explore how we can write a map reduce program to find the top X values from the given set. Now, though, lets try to understand the logic behind this.



As shown in the preceding figure, our logic includes finding the top 10 words from each input split and then sending these records to only one reducer. In the reducer, we will again find the top 10 words to get the final result of the top 10 records. Now, let's understand the execution.

First, let's prepare the input. Here, we will use the word count program provided along with Hadoop binaries:

First of all, let's put data in HDFS to be processed.

```
hadoop fs -mkdir /input
hadoop fs -put /usr/local/hadoop/LICENSE.txt /input
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.0.jar /input /word_count_output
```

This will execute the word count program and store the output in the HDFS folder called `/word_count_output`. This output will be used as the input for our top 10 map reduce program.

Let's take a look at the mapper code:

```
public static class TopTenMapper extends Mapper<Object, Text, Text,
IntWritable> {

        // Tree map keeps records sorted by key
        private TreeMap<Integer, String> countWordMap = new
TreeMap<Integer, String>();

        public void map(Object key, Text value, Mapper<Object, Text,
Text, IntWritable>.Context context)
                throws IOException, InterruptedException {

            String[] words = value.toString().split("[\t]");
            int count = Integer.parseInt(words[1]);
            String word = words[0];
            countWordMap.put(count, word);
            if (countWordMap.size() > 10) {

                countWordMap.remove(countWordMap.firstKey());
            }

        }

        @Override
        protected void cleanup(Context context) throws IOException,
InterruptedException {
            for (Entry<Integer, String> entry : countWordMap.
entrySet()) {
                context.write(new Text(entry.getValue()), new
IntWritable(entry.getKey()));

            }

        }
    }
```

In the preceding code, we are using `TreeMap` to store the words and their count. `TreeMap` helps store keys and values sorted order by the key. Here, we are using the count as the key and words as values. In each Mapper iteration, we check whether the size is greater than 10. If it is, we remove the first key from the key map, which would be the lowest count of the set. This way, at the end of each mapper, we will emit the top 10 words of the reducer.

You can read more about `TreeMap` at `http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html`.

Now, let's take a look at the reducer code:

```
public static class TopTenReducer extends Reducer<Text, IntWritable,
Text, IntWritable> {

        // Tree map keeps records sorted by key
        private TreeMap<IntWritable, Text> countWordMap = new
TreeMap<IntWritable, Text>();

        public void reduce(Text key, Iterable<IntWritable> values,
                Reducer<Text, IntWritable, Text, IntWritable>.Context
context)
                        throws IOException, InterruptedException {

            for (IntWritable value : values) {

                countWordMap.put(value, key);

            }
            if (countWordMap.size() > 10) {

                countWordMap.remove(countWordMap.firstKey());
            }
            for (Entry<IntWritable, Text> entry : countWordMap.
descendingMap().entrySet()) {
                context.write(entry.getValue(), entry.getKey());

            }

        }

    }
```

In the reducer, we will again use `TreeMap` to find the top 10 of all the collected records from each Mapper. Here, is it very important to use only one reducer for the complete processing; hence, we need to set this in the Driver class, as shown here:

```
public class TopTenWordsByOccurence {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        if (args.length != 2) {
            System.err.println("Usage: toptencounter <in><out>");
            System.exit(2);
        }
        Job job = Job.getInstance(conf, "Top Ten Word By Occurence
Counter");
        job.setJarByClass(TopTenWordsByOccurence.class);
```

```
            job.setMapperClass(TopTenMapper.class);
            job.setCombinerClass(TopTenReducer.class);
            job.setReducerClass(TopTenReducer.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);
            job.setNumReduceTasks(1);
            FileInputFormat.addInputPath(job, new Path(args[0]));

            FileOutputFormat.setOutputPath(job, new Path(args[1]));

            System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
    }
```

Now, when you execute the preceding code, as a result, you will see the output in the form of the top 10 words due to their frequencies in the document.

You can modify the same program to get the top 5, 20, or any number.

How it works

Here, the logic is quite straightforward, as shown in the preceding diagram. The trick is using `TreeMap`, which stores data in a sorted key order. It is also important to use only one reducer, and if we can't, we will again get the number of sets of the top records from each reducer, which will not show you the correct output.

Map Reduce program to find distinct values

In this recipe, we are going to learn how to write a map reduce program to find distinct values from a given set of data.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as an eclipse that is similar to an IDE.

How to do it

Sometimes, there may be a chance that the data you have contains some duplicate values. In SQL, we have something called a distinct function, which helps us get distinct values. In this recipe, we are going to take a look at how we can get distinct values using map reduce programs.

Let's consider a use case where we have some user data with us, which contains two columns: `userId` and `username`. Let's assume that the data we have contains duplicate records, and for our processing needs, we only need distinct records through user IDs. Here is some sample data that we have where columns are separated by '|':

```
1|Tanmay
2|Ramesh
3|Ram
1|Tanmay
2|Ramesh
6|Rahul
6|Rahul
4|Sneha
4|Sneha
```

The idea here is to use the default reducer behavior where the same keys are sent to one reducer. In this case, we will make `userId` the key and emit it to the reducer. In the reducer, the same keys will be reduced together, which will avoid duplicates.

Let's look at the Mapper Code.

```
public static class DistinctUserMapper extends Mapper<Object, Text,
Text, NullWritable> {
        private Text userId = new Text();

        public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {

            String words[] = value.toString().split("[|]");
            userId.set(words[0]);
            context.write(userId, NullWritable.get());
        }
    }
```

We only want distinct user IDs, hence, we emit only user IDs as keys and nulls as values.

Now, let's look at the reducer code:

```
public static class DistinctUserReducer extends Reducer<Text,
NullWritable, Text, NullWritable> {
        public void reduce(Text key, Iterable<NullWritable> values,
Context context)
                throws IOException, InterruptedException {
            context.write(key, NullWritable.get());
        }
}
```

Here, we only emit user IDs as they come. This step removes duplicates as the reducer only treats the records by their keys and only one record per key is kept.

The driver code remains simple, as shown here:

```
public class DistinctValues {

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        if (args.length != 2) {
            System.err.println("Usage: DistinctValues <in><out>");
            System.exit(2);
        }
        Job job = Job.getInstance(conf, "Distinct User Id finder");
        job.setJarByClass(DistinctValues.class);
        job.setMapperClass(DistinctUserMapper.class);
        job.setReducerClass(DistinctUserReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);

    }
}
```

Now, when we execute the code, we will see the following output:

```
hadoop jar distinct.jar com.demo.DistinctValues /users /distinct_user_ids
hadoop fs -cat /distinct_user_ids/part-r-00000
    1
    2
    3
    4
    6
```

How it works...

When mapper emits keys and values, the output is shuffled across the nodes in the cluster. Here, the partitioner decides which keys should be reduced and on which node. On all the nodes, the same partitioning logic is used, which makes sure that the same keys are grouped together. In the preceding code, we use this default behavior to find distinct user IDs.

Map Reduce program to partition data using a custom partitioner

In this recipe, we are going to learn how to write a map reduce program to partition data using a custom partitioner.

Getting ready

To perform this recipe, you should have a running Hadoop cluster running as well as an eclipse that's similar to an IDE.

How to do it...

During the shuffle and sort, if it's not specified, Hadoop by default uses a hash partitioner. We can also write our own custom partitioner with custom partitioning logic, such that we can partition the data into separate files.

Let's consider one example where we have user data with us along with the year of joining. Now, assume that we have to partition the users based on the year of joining that's specified in the record. The sample input data looks like this:

```
User_id|user_name|yoj
1|Tanmay|2010
2|Sneha|2015
3|Sakalya|2020
4|Manisha|2011
5|Avinash|2012
6|Vinit|2022
```

To get this data partitioned based on `YOJ`, we will have to write a custom partitioner:

```
public class YearOfJoiningPartitioner extends Partitioner<IntWritable,
Text> implements Configurable {

    private Configuration conf = null;

    @Override
    public int getPartition(IntWritable key, Text value, int
numPartitions) {
        return key.get() % 10;
    }
```

```
    @Override
    public Configuration getConf() {

        return conf;
    }

    @Override
    public void setConf(Configuration conf) {
        this.conf = conf;

    }
}
```

In the custom partitioner, we need to override the `getPartition()` method where we will write our own custom partitioning logic. Here, we will be using `YOJ` as the key and partition the data based on the modulo 10 value of year. The value given by this method will be in the form of a reducer where the key will be reduced.

Now, to use this partitioner, we need to set this in the driver class, as shown here:

```
public class YOJPartitioner {

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        if (args.length != 2) {
            System.err.println("Usage: YOJPartitioner <in><out>");
            System.exit(2);
        }
        Job job = Job.getInstance(conf, "YOJPartitioner");
        job.setJarByClass(YOJPartitioner.class);
        job.setMapperClass(YOJPartitionerMapper.class);
        job.setReducerClass(YOJReducer.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(Text.class);
        job.setPartitionerClass(YearOfJoiningPartitioner.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));

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

    }
}
```

Now that we want to partition the data based on `YOJ`, we will have to set `YOJ` as the key while emitting data from the mapper, as shown here:

```
public static class YOJPartitionerMapper extends Mapper<Object, Text,
IntWritable, Text> {

        public void map(Object key, Text value, Mapper<Object, Text,
IntWritable, Text>.Context context)
                throws IOException, InterruptedException {
            String words[] = value.toString().split("[|]");
            context.write(new IntWritable(Integer.parseInt(words[2])),
value);

        }
    }
```

Next, we will write the identity reducer, which emits keys and values as they come:

```
public static class YOJReducer extends Reducer<IntWritable, Text,
Text, NullWritable> {
        protected void reduce(IntWritable key, Iterable<Text> values,
Context context)
                throws IOException, InterruptedException {
            for (Text t : values) {
                context.write(t, NullWritable.get());
            }
        }
    }
```

When we execute this code in a multimode cluster, the records will be partitioned into different files based on the logic we wrote.

How it works...

When we provide the custom partitioner to the map reduce code and the mappers have finished their work, the custom partitioner will come into the picture. The keys are sent through the `getPartition()` method and the resultant value will be in the form of the reducer number where the keys are sent. Here, we are executing modulo 10 to the year value which would partition the data into 10 partitions. This way, by controlling the logic of the `getPartition()` method, we can decide which partitioning strategy to use.

Writing Map Reduce results to multiple output files

In this recipe, we are going to learn how to write a map reduce output to multiple output files. This will be useful when we need to use classified output for different purposes.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as an eclipse similar to an IDE.

How to do it...

Hadoop supports a class called `MultipleOutputs`, which allows us to write out of a map reduce program to multiple files. We can write output to different files, file types, and different locations with it. You can also choose the filename with this API. To use this, we will take a look at a simple word count program and write out of this program to multiple output files.

To do so, we need to add the named output files and their types in our Driver code, as shown here:

```
public class WordCount {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        if (args.length != 2) {
            System.err.println("Usage: wordcount <in><out>");
            System.exit(2);
        }
        Job job = Job.getInstance(conf, "WordCount");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        MultipleOutputs.addNamedOutput(job, "text", TextOutputFormat.
class, Text.class, IntWritable.class);
        MultipleOutputs.addNamedOutput(job, "seq",
SequenceFileOutputFormat.class, Text.class, IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));

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

The highlighted code shows that we have two named output files, that is, the text and seq. This will generate two more files in addition to the default output files. Next, we need to update the reducer code to write output key values to these new files as well:

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

        public void setup(Context context) {

            mos = new MultipleOutputs(context);
        }

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

            context.write(key, this.result);
        }

        public void cleanup(Context context) throws IOException,
InterruptedException {
            mos.close();

        }
    }
```

In the setup method, we need to initiate the object of the `MultipleOutputs` class. In the reduce method, we have to add the `write` method, which would write the desired key values to two files, one of the text type and the other of the sequence file type.

Additionally, you can also specify the complete path apart from the default output path in the reducer code where you wish to see to your file:

```
mos.write(key, this.result, "/my_new_output_path_up/file");
```

It is very important to close the `MultipleOutputs` stream in the `cleanup` method.

How it works...

`MulitpleOutputs` is a great help in two conditions: when you wish to see multiple output files in different file formats and when you wish to write the output of your map reduce program in order to use specified files. `MultipleOutputs` supports counters but they are disabled by default. Once specified, `MultipleOutputs` will first create a file with a specific name, and then it will start writing any data if available. If no data is generated, it will still create zero-sized files. To avoid this, we can use `LazyOutputFormat`, as shown in the following. Details on `LazyOutputFormat` are available at `https://hadoop.apache.org/docs/r2.6.1/api/org/apache/hadoop/mapreduce/lib/output/LazyOutputFormat.html`. Here is the code for `LazyOutputFormat`:

```
LazyOutputFormat.setOutputFormatClass(job, TextOutputFormat.class);
```

Performing Reduce side Joins using Map Reduce

In this recipe, we are going to learn how to write a map reduce, which will join records from two tables.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as an eclipse that's similar to an IDE.

How to do it

We are aware of the various types of joins that are available in SQL—Inner Join, Left outer join, right outer join, full outer join, and so on. Performing joins in SQL is quite easy, but when it comes to MapReduce, this is a little tricky. In this recipe, we will be try to perform various join operations using the Map Reduce program in the following dataset.

Consider two datasets: the `Users` table, which has information about `userId`, username, and `deptId`. We also have data on the `Department` table where we have `deptId` and `deptName` as columns. If we place our data in a table, it would look like this:

Users' table:

User ID	Username	Department ID
1	Tanmay	1
2	Sneha	1
3	Sakalya	2

User ID	Username	Department ID
4	Manisha	2
5	Avinash	3

Department table:

Department ID	Department name
1	Engineering
2	Sales
3	Production

Here, `deptId` is the foreign key between two tables, and we will join the two tables based on this key.

To start with, we need to have a driver code where we will use the `MultipleInputs` API, which allows us to take the two input paths where the users and department table is stored in HDFS. We also need to accept the join type from the user, and lastly, we need to accept the output path where we will save our final result:

```
public class ReduceSideJoin {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        if (args.length != 4) {
            System.err.println("Usage: join <input-table1><input-table
2><jointype:inner|leftouter|rightouter|fullouter><out>");
            System.exit(2);
        }
        Job job = Job.getInstance(conf, "Reduce Side Join");
        job.setJarByClass(ReduceSideJoin.class);

        job.setReducerClass(DeptJoinReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        MultipleInputs.addInputPath(job, new Path(args[0]),
TextInputFormat.class, UserJoinMapper.class);
        MultipleInputs.addInputPath(job, new Path(args[1]),
TextInputFormat.class, DeptJoinMapper.class);
        job.getConfiguration().set("join.type", args[2]);
        FileOutputFormat.setOutputPath(job, new Path(args[3]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Next, we need to write two mappers that deal with each table. We'll first look at the users table. We'll join the records with `deptId`, hence we need to emit keys as `deptIds`. Also, we are going to append an identifier, which will help us classify records based on the table that they are from:

```
public static class UserJoinMapper extends Mapper<Object, Text, Text,
Text> {
        private Text outkey = new Text();
        private Text outvalue = new Text();

        public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
            String attrs[] = value.toString().split("[\t]");
            String deptId = attrs[2];
            // The foreign join key is the dept ID
            outkey.set(deptId);
            // flag this each record with prefixing it with 'A'
            outvalue.set("A" + value.toString());
            context.write(outkey, outvalue);
        }
    }
```

Similarly, we'll write a mapper class for the department table as well:

```
public static class DeptJoinMapper extends Mapper<Object, Text, Text,
Text> {
        private Text outkey = new Text();
        private Text outvalue = new Text();

        public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
            String attrs[] = value.toString().split("[\t]");
            String deptId = attrs[0];
            // The foreign join key is the dept ID
            outkey.set(deptId);
            // flag this each record with prefixing it with 'B'
            outvalue.set("B" + value.toString());
            context.write(outkey, outvalue);
        }
    }
```

Next, we need to write a reducer class where we will write our logic to join the records:

```
public static class DeptJoinReducer extends Reducer<Text, Text, Text,
Text> {
        private static final Text EMPTY_TEXT = new Text("");
        private Text tmp = new Text();
        private ArrayList<Text> listA = new ArrayList<Text>();
        private ArrayList<Text> listB = new ArrayList<Text>();
        private String joinType = null;

        public void setup(Context context) {
            // set up join configuration based on input
            joinType = context.getConfiguration().get("join.type");
        }

        public void reduce(Text key, Iterable<Text> values, Context
context) throws IOException, InterruptedException {
            // Clear the lists
            listA.clear();
            listB.clear();
            // Put records from each table into correct lists, remove
the prefix
            for (Text t : values) {
                tmp = t;
                if (tmp.charAt(0) == 'A') {
                    listA.add(new Text(tmp.toString().substring(1)));
                } else if (tmp.charAt(0) == 'B') {
                    listB.add(new Text(tmp.toString().substring(1)));
                }
            }
            // Execute joining logic based on its type
            executeJoinLogic(context);
        }
…
}
```

In the preceding code, we first initialize the two empty lists where we keep our records. There's one list for each table. We segregate the records based on their prefix. In the setup method, we also set up the join type that's been provided by the user in the job configuration.

Next, we need to write the join logic based on the join type. The following code shows the logic for the inner, left outer, right outer, and the full outer join:

```
private void executeJoinLogic(Context context) throws IOException,
InterruptedException {
            if (joinType.equalsIgnoreCase("inner")) {

                if (!listA.isEmpty() && !listB.isEmpty()) {
                    for (Text A : listA) {
                        for (Text B : listB) {
                            context.write(A, B);
                        }
                    }
                } else if (joinType.equalsIgnoreCase("leftouter")) {

                    for (Text A : listA) {

                        if (!listB.isEmpty()) {
                            for (Text B : listB) {
                                context.write(A, B);
                            }
                        } else {

                            context.write(A, EMPTY_TEXT);
                        }
                    }
                } else if (joinType.equalsIgnoreCase("rightouter")) {

                    for (Text B : listB) {

                        if (!listA.isEmpty()) {
                            for (Text A : listA) {
                                context.write(A, B);
                            }
                        } else {

                            context.write(EMPTY_TEXT, B);
                        }
                    }
                } else if (joinType.equalsIgnoreCase("fullouter")) {

                    if (!listA.isEmpty()) {
```

```
                    for (Text A : listA) {

                        if (!listB.isEmpty()) {
                            for (Text B : listB) {
                                context.write(A, B);
                            }
                        } else {

                            context.write(A, EMPTY_TEXT);
                        }
                    }
                } else {

                    for (Text B : listB) {
                        context.write(EMPTY_TEXT, B);
                    }
                }
            }
        }
    }
```

Now, when we compile this code and bundle it into jar, it is ready for execution. Before the execution, we need to prepare our input:

```
hadoop fs –mkdir /users
hadoop fs –put users.txt /users
hadoop fs –mkdir /dept
hadoop fs –put dept.txt /dept
hadoop jar join.jar com.demo.ReduceSideJoin /users /dept inner /inner-
join-output
```

Once the execution is complete, you can take a look at the output file:

```
hadoop fs –cat /inner-join-output/part-r-00000
    2   Sneha 1      1      Engineering
    1   Tanmay 1     1      Engineering
    4   Manisha      2      2      Sales
    3   Sakalya      2      2      Sales
    5   Avinash      3      3      Production
```

How it works...

`MulitpleInputs` works with each mapper and sends data from the respective directories. Inside mappers, the keys and values are emitted based on the joining foreign key. When it comes to reducers, the actual join logic is executed based on the join type. Generally, reduce side joins consume a lot of resources and take more time than map side joins. But when you have huge data at hand, reduce side joins are the only options we have. Reduce side joins also cause huge network traffic as each and every record is sent to the reducers. Based on the foreign key to join, the keys should be changed. This way, you can update the code in the preceding section to join any two datasets easily.

Unit testing the Map Reduce code using MRUnit

In this recipe, we are going to learn how to unit test the map reduce code using a library called MRUnit.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as an eclipse similar to an IDE.

How to do it...

Sometimes, it is very difficult to develop, compile, deploy and execute the map reduce program and then figure out whether the code is correct or not. In order to avoid this deploying and testing technique, we can use a unit testing framework that's been built specially for Map Reduce called, MRUnit.

Let's assume that we have a map reduce program that emits and counts words starting with '#', such as a Twitter hashtag counter. The Mapper code for this hashtag counter looks like this:

```
public static class TokenizerMapper extends
            Mapper<Object, Text, Text, IntWritable> {
        private static final IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value,
            Mapper<Object, Text, Text, IntWritable>.Context context)
                throws IOException, InterruptedException {
```

```
            StringTokenizer itr = new StringTokenizer(value.
toString());
            while (itr.hasMoreTokens()) {
                this.word.set(itr.nextToken());
                if (word.toString().startsWith("#"))
                    context.write(this.word, one);
            }
        }
    }
```

The Reduce code is a simple sum reducer code:

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

        public void reduce(Text key, Iterable<IntWritable> values,
                Reducer<Text, IntWritable, Text, IntWritable>.Context
context)
                throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            this.result.set(sum);
            if (result.get() >= 2)
                context.write(key, this.result);
        }
  -  }
```

Now, to unit test this code, we will first add dependencies that are required for MRUnit. We need the following dependencies to be added to `POM.xml`:

```
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-all</artifactId>
            <version>2.0.2-beta</version>
        </dependency>
        <dependency>
            <groupId>org.apache.mrunit</groupId>
```

```
            <artifactId>mrunit</artifactId>
            <version>1.1.0</version>
            <classifier>hadoop1</classifier>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-core</artifactId>
            <version>2.7.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.7.1</version>
        </dependency>
```

We need to write a test class that would test the methods from our Map Reduce code:

First we write a setup method which would initialize the objects:

```
MapDriver<Object, Text, Text, IntWritable> mapDriver;
    ReduceDriver<Text, IntWritable, Text, IntWritable> reduceDriver;
    MapReduceDriver<Object, Text, Text, IntWritable, Text,
IntWritable> mapReduceDriver;

    @Before
    public void setUp() {
        TokenizerMapper mapper = new TokenizerMapper();
        IntSumReducer reducer = new IntSumReducer();
        mapDriver = MapDriver.newMapDriver(mapper);
        reduceDriver = ReduceDriver.newReduceDriver(reducer);
        mapReduceDriver = MapReduceDriver.newMapReduceDriver(mapper,
reducer);
    }
```

Here, we are initiating the mapper, reducer, and drivers so that they can be accessed in test cases.

Now, let's write a test case for the mapper class:

```
@Test
    public void testMapper() throws IOException {
          mapDriver.withInput(new LongWritable(), new Text("Hello
World #mrunit"));
        mapDriver.withOutput(new Text("#mrunit"), new IntWritable(1));
        mapDriver.runTest();
    }
```

Here, we are testing whether the mapper only emits words starting with '#'.

Next, we will write a test case to check whether the reducer properly sums up the values:

```
@Test
    public void testReducer() throws IOException {
        List<IntWritable> values = new ArrayList<IntWritable>();
        values.add(new IntWritable(1));
        values.add(new IntWritable(1));
        reduceDriver.withInput(new Text("#mrunit"), values);
        reduceDriver.withOutput(new Text("#mrunit"), new
IntWritable(2));
        reduceDriver.runTest();
    }
```

We can also write a combined test for the mapper and reducer:

```
    @Test
    public void testMapReduce() throws IOException {
        mapReduceDriver.withInput(new LongWritable(), new Text("Hello
World #mrunit"));
        List<IntWritable> values = new ArrayList<IntWritable>();
        values.add(new IntWritable(1));
        values.add(new IntWritable(1));
        mapReduceDriver.withOutput(new Text("#mrunit"), new
IntWritable(2));
        mapReduceDriver.runTest();
    }
```

This way, you can test any important logic that you have written in your map reduce code.

How it works...

`MRUnit` works exactly like any other testing framework, such as `JUnit/TestNG`. It only unit tests the logic. `Mockito` is a framework that helps mocking up an object, which is a part of the Hadoop/Map Reduce framework.

It is always good practice to unit test your code and only then deploy your map reduce jobs in production clusters. This also helps us with continuous integration to get to know any braking changes because any code update.

4

Data Analysis Using Hive, Pig, and Hbase

- Storing and processing Hive data in a sequential file format
- Storing and processing Hive data in the RC file format
- Storing and processing Hive data in the ORC file format
- Storing and processing Hive data in the Parquet file format
- Performing `FILTER By` queries in Pig
- Performing Group By queries in Pig
- Performing Order By queries in Pig
- Performing JOINS in Pig
- Writing a user-defined function in Pig
- Analyzing web log data using Pig
- Performing the Hbase operation in CLI
- Performing Hbase operations in Java
- Executing the MapReduce programming Hbase Table

Introduction

In the previous chapter, we discussed how to write MapReduce programs in various ways in order to analyze data. Earlier, MapReduce was the only means of processing data in Hadoop, but with the passage of time, growing popularity, and the ease of using the Hadoop platform, various subprojects joined its league, which helps users write logic in SQL, scripts, and so on, and analyze the data. Projects, such as Apache Hive, Apache Pig, and Apache Hbase, are well accepted by users, so the majority of development these days takes place using any of these. In this chapter, we are going to take a look at how to use these tools to perform various advanced operations. If you are new to these terms, I would recommend that you read through these links first.

- For Hive resources, take a look a these links:
- `https://hive.apache.org/`
- `http://hadooptutorials.co.in/tutorials/hive/introduction-to-apache-hive.html`
- `http://hadooptutorials.co.in/tutorials/hive/hive-data-units.html`
- `http://hadooptutorials.co.in/tutorials/hive/hive-best-practices.html`

For Pig resources, take a look at these links:

- `https://pig.apache.org/`

For Hbase resources, take a look at these links:

- `https://hbase.apache.org/`

We won't be going through the installations of the above in this book as the installations are quite easy if your Hadoop cluster is already installed; just download the binaries and set the installation paths with few or no configurations as you'll have to use all of them.

Let's get started with our first recipe now.

Storing and processing Hive data in a sequential file format

I'm sure that most of the time, you would have created Hive tables and stored data in a text format; in this recipe, we are going to store data in sequential files.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am going to use Hive 1.2.1.

How to do it...

Hive 1.2.1 supports various different types of files, which help process data in a faster manner. In this recipe, we are going to use sequential files to store data in Hive. To store data in sequential files, we first need to create a Hive table that stores the data in a textual format:

```
create table employee(
id int, name string)
row format delimited
fields terminated by '|'
 stored as textfile;
```

The preceding statement will create a table in Hive along with the employee name. Now, let's load data into this table:

```
load data local inpath '/usr/local/hive/examples/files/employee.dat'
into table employee;
```

Here, I am using an example data file that's been provided by the Hive distribution itself. The location where Hive is installed is `/usr/local/hive`.

Now, we can execute a select query to check whether the data is correct:

```
select * from employee;
```

You should see this output:

```
16      john
17      robert
18      andrew
19      katty
21      tom
22      tim
```

```
23      james
24      paul
27      edward
29      alan
31      kerry
34      terri
```

Now, to store data in a sequential file format, we will create one more table and insert data from this table to the new table:

```
create table employee_seq(
id int, name string)
row format delimited
fields terminated by '|'
stored as SEQUENCEFILE;
```

This will create a table in Hive that will read data in the sequential file format. Let's insert data into this table:

```
insert into employee_seq
select * from employee;
```

This will start a MapReduce program, which will read the data from the text table and write it to files in the sequential file format.

We can again verify that it's been stored properly by executing this code:

```
select * from employee_seq;
```

This will give you all the records from the sequential file.

Just to check whether the data is stored in sequential files, you can check the HDFS file directly, which gives you unreadable text on your screen:

```
hadoop fs -cat /user/hive/warehouse/employee_seq/000000_0
```

How it works...

A sequential file is a flat file consisting of key value pairs. It is one of the most commonly used file formats in the Hadoop ecosystem. It's worth noting that the temporary output of maps is stored in the sequential file format. The sequential file provides a writer, reader, and sorter classes to read, write, and sort data. There are three types of sequential files:

- Uncompressed key value records
- Record compressed key value records, where only values are compressed

- Block compressed key value records, where keys and values are stored in blocks, and the blocks are compressed.

You can select the type of sequential file you want by setting the following property in either `mapred-site.xml`, `hive-site.xml`, or setting it before a job or query execution:

```
<property>
    <name>mapred.output.compression.type</name>
    <value>BLOCK</value>
    <description>If the job outputs are to compressed as
SequenceFiles, how should they be compressed? Should be one of NONE,
RECORD or BLOCK.
    </description>
</property>
```

This way, you can use sequential files to store your data. Processing these files is the same as processing any other Hive query.

Storing and processing Hive data in the RC file format

I'm sure that most of the time, you would have created Hive tables and stored data in a text format; in this recipe, we are going to store data in RC files.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am going to use Hive 1.2.1.

How to do it...

Hive 1.2.1 supports different types of files, which help process data in a fast manner. In this recipe, we are going to use RC files to store data in Hive. To store data in RC files, we first need to create a Hive table that stores this data in a textual format. We will use the same table that we created in the first recipe.

Creating a table to store `RCFILE` is very easy, as shown here:

```
create table employee_rc(
id int, name string)
row format delimited
fields terminated by '|'
stored as RCFILE;
```

To insert data into the table from our text table, we need to execute the following query, which would start the MapReduce program that reads the data from one table in order to write it using the `RCFILE` writer:

```
insert into table employee_rc
select * from employee;
```

We can verify the data import by selecting the records from the table and displaying it on the console:

```
select * from employee_rc;
```

We can also look at the actual file stored in the Hive warehouse folder, which won't be as readable:

```
hadoop fs -cat /user/hive/warehouse/employee_rc/000000_0
```

How it works...

RC files are Record Columnar files that are flat and store the data in binary key value pairs, which is similar to sequence files. An RC file stores records in a columnar manner. Let's assume that we have the following table data:

C1	C2	C3	C4
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20

To serialize this data, it first horizontally partitions the data into multiple row groups, which looks like this:

Row group 1:

C1	C2	C3	C4
1	2	3	4
5	6	7	8
9	10	11	12

Row group 2:

C1	C2	C3	C4
13	14	15	16
17	18	19	20

Next, it partitions the data within the row group vertically similar to a column store:

Row group 1:

1;5;9

2;6;10

3;7;11

4;8;12

Row group 2

13;17

14;18

15;19

16;20

Within row groups, columns are compressed to reduce the storage.

These file storages are more efficient when a query only requires a subset of columns as the output at which time, it only reads the required output, whereas in a row store, it needs to scan the complete row itself, which consumes more I/O. This way, we can store the retrieved data more efficiently using the RC file.

Storing and processing Hive data in the ORC file format

I'm sure that most of the time, you would have created Hive tables and stored data in a text format; in this recipe, we are going store data in ORC files.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am going to use Hive 1.2.1.

How to do it...

Hive 1.2.1 supports different types of files, which help process data in a fast manner. In this recipe, we are going to use ORC files to store data in Hive. To store the data in ORC files, we first need to create a Hive table that stores the data in a textual format. We will use the same table that we created in the first recipe.

Creating a table to store `ORCFILE` is very easy, as shown here:

```
create table employee_orc(
id int, name string)
row format delimited
fields terminated by '|'
stored as ORC;
```

To insert data into the table from our text table, we need to execute the following query, which would start the Map Reduce program that reads the data from one table in order to write it using the ORC writer:

```
insert into table employee_orc
select * from employee;
```

We can verify the data import by selecting the records from the table and displaying it in the console:

```
select * from employee_orc;
```

We can also look at the actual file stored in the Hive warehouse folder, which won't be as readable:

```
hadoop fs -cat /user/hive/warehouse/employee_orc/000000_0
```

How it works...

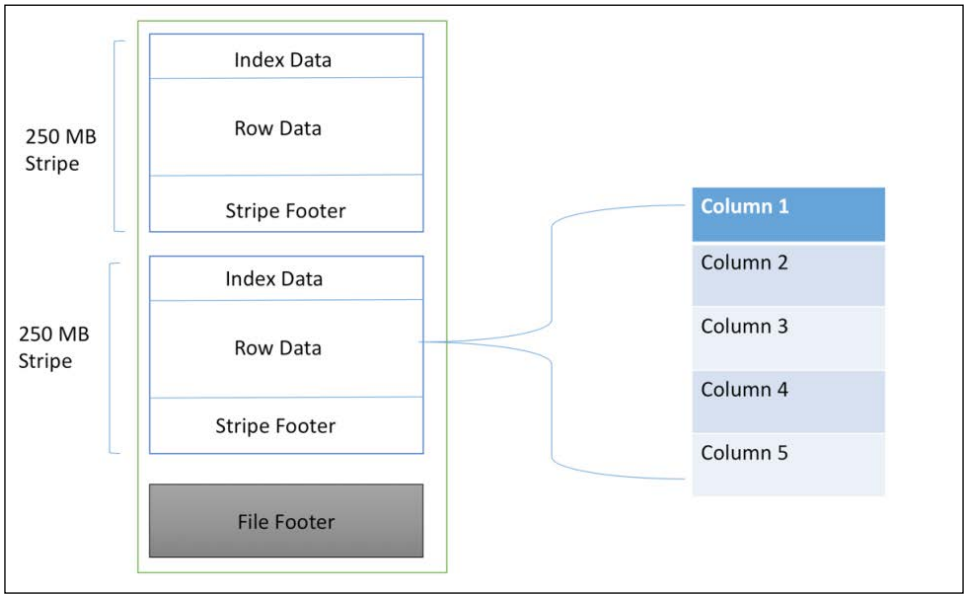
ORC files stand for Optimized Record Columnar files, which are specially designed to improve data reading, writing, and processing in Hadoop-based systems. An ORC file will give you the following advantages over other file formats such as RC and sequential:

- A single file as the output of each task
- Supports data types such as date and time, decimals and complex types, and so on
- Lightweight indexes that are included in the file itself
- Concurrent reads of the same file using individual RecordReaders
- It's able to scan files without scanning for markers
- Metadata is stored using Protocol Buffers, which help add and remove fields

The data storage, partitions, and so on are the same as the RC file, but apart from this, it also contains additional fields/information that help achieve the preceding advantages over other files.

The ORC file contains groups of row data, which are called stripes. The default size of each stripe is 250MB; this size enables the efficient reading of data from HDFS. The auxiliary information is kept in a file footer. It contains information like the number of stripes in a file, rows per stripe, and each column's data type. It also contains column-level aggregations such as count, min, max, and sum.

Here's the ORC file structure:



Storing and processing Hive data in the Parquet file format

I'm sure that most of the time, you would have created Hive tables and stored data in a text format; in this recipe, we are going to learn how to store data in PARQUET files.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am going to use Hive 1.2.1.

How to do it...

Hive 1.2.1 supports various types of files, which help process data more efficiently. Parquet is an ecosystem-wide accepted file format and can be used in Hive, Map Reduce, Pig, Impala, and so on. To store the data in Parquet files, we first need to create one Hive table, which will store the data in a textual format. We will use the same table that we created in the first recipe.

Creating a table to store Parquet is very easy, as shown here:

```
create table employee_par(
id int, name string)
row format delimited
fields terminated by '|'
stored as PARQUET;
```

To insert data into the table from our text table, we need to execute the following query, which would start the Map Reduce program so that it reads the data from one table in order to write it using the ORC writer:

```
insert into table employee_par
select * from employee;
```

We can verify the data import by selecting the records from the table and displaying it in the console:

```
select * from employee_par;
```

We can also take a look at the actual file stored in the Hive warehouse folder, which will not be as readable:

```
hadoop fs -cat /user/hive/warehouse/employee_par/000000_0
```

How it works...

Parquet files were designed keeping in mind the best compressed and columnar data files. They are built to support very efficient compressions and encoding schemes. A lot of projects have demonstrated the impact of efficient encoding and compression schemes. Parquet allows us to specify the compressions scheme per column level. It is also built to accommodate new compression and encoding schemes for future needs and extensibility. A detailed description of the Parquet storage format is available at `https://parquet.apache.org/documentation/latest/`.

Performing FILTER By queries in Pig

After going through the various file formats that we can use to store data in HDFS, it's time to take a look at how to execute various operations in Pig. Pig is a data flow language and works in the same way as Hive by transforming the instructions given in Pig Latin to Map Reduce programs.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Pig installed on it. Here, I am going to use Pig 0.15. In case you don't have the installation already, you can refer to `https://pig.apache.org/docs/r0.15.0/start.html#Pig+Setup`.

How to do it...

In this recipe, you will learn how to use `FILTER BY` in the Pig script. To do so, let's assume that we have an employee dataset that is stored in the following format (ID, name, department, and salary):

```
1       Tanmay ENGINEERING   5000
2       Sneha  PRODUCTION    8000
3       Sakalya       ENGINEERING   7000
4       Avinash       SALES  6000
5       Manisha       SALES  5700
6       Vinit  FINANCE       6200
```

Here the columns are delimited by \+*t* (*Tab*).

Now, let's put this data into HDFS and load it into Pig bag:

```
hadoop fs -mkdir /pig/emps_data
hadoop fs -put emps.txt /pig/emps_data
```

Apart from regular daemons, Pig needs one more daemon to be running on Hadoop called a Job History Server. If you have not started it earlier, execute the following command:

```
/usr/local/hadoop/sbin/mr-jobhistory-daemon.sh start historyserver
```

Start the Pig interactive shell to perform the processing. Pig's shell is called grunt. Now, let's load the data we have put in HDFS into one bag, say `emps`:

```
emps = LOAD '/pig/emps_data/emps.txt' AS (id, name, dept, salary);
```

This will load the data into the `emps` bag through the columns, as shown in the preceding command. The default delimiter is \+*t* (*Tab*).

Next, we filter some databases on the basis of salary and load into a new bag called `rich`:

```
rich = FILTER emps BY salary > 6000;
```

Pig scripts are lazy executors, which means that unless they don't get any actionable command, Pig will keep on stacking up the instructions. In order to perform the processing, we need to provide an actionable command, such as `DUMP`, which asks Pig to print the tuples of the bag in the terminal:

```
DUMP rich;
```

This will invoke the Map Reduce job as long as all the commands are received by grunt. On completion, you should be able to see results on your screen like this:

```
(2,Sneha,PRODUCTION,8000)
(3,Sakalya,ENGINEERING,7000)
(6,Vinit,FINANCE,6200)
```

How it works...

Pig has some very interesting terminology. In Pig, an item is called an Atom. A collection of Atoms is called a tuple. Collections of tuples are called bags. When we load data into a bag, we don't always need to provide data types for the atoms unlike Hive. Depending on the actions submitted by users, grunt explicitly casts the data into specific data types. It also informs the user that it has performed the type casting. When we use the `FILTER BY` command in a Pig script, it converts it into a Map Reduce program where the filtering logic is mostly applied in a mapper function.

After map, only those records are sent to reducers that are in line with the filter criteria. This way, we can filter out things we wish to.

Performing Group By queries in Pig

In this recipe, we will use the Group By operator in Pig scripts to get the desired output.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Pig installed on it.

How to do it...

Group By is a very useful operator for data analysis. Pig supports this operator so that we can perform aggregations at the group level. Take the same data that we used in the previous recipe where we have this employee dataset:

```
1      Tanmay ENGINEERING   5000
2      Sneha  PRODUCTION    8000
3      Sakalya       ENGINEERING   7000
4      Avinash       SALES  6000
5      Manisha       SALES  5700
6      Vinit  FINANCE       6200
```

First of all, load the data into HDFS:

```
hadoop fs -mkdir /pig/emps_data
hadoop fs -put emps.txt /pig/emps_data
```

Next, we load the data into a bag called `emps`, and then perform the Group By operation on this data by the department:

```
emps = LOAD '/pig/emps_data/emps.txt' AS (id, name, dept, salary);
by_dept = GROUP emps BY dept;
DUMP by_dept;
```

This will start a MapReduce job, which groups tuples by the department. As a result, you will see these details on your screen:

```
(SALES,{(5,Manisha,SALES,5700),(4,Avinash,SALES,6000)})
(FINANCE,{(6,Vinit,FINANCE,6200)})
(PRODUCTION,{(2,Sneha,PRODUCTION,8000)})
(ENGINEERING,{(3,Sakalya,ENGINEERING,7000),(1,Tanmay,ENGINEERI
NG,5000)})
```

We can also get the count of the employees by executing the following query:

```
dept_counts = FOREACH by_dept GENERATE
group as dept,
COUNT(emps);
DUMP dept_counts;
```

Use the `aggregation` function called `count` to count the number of employees per department. You will see the following results on completion:

```
(SALES,2)
(FINANCE,1)
(PRODUCTION,1)
(ENGINEERING,2)
```

We can also perform Group By on multiple attributes together. We just need to specify the attribute name one by one after the `by` clause.

How it works...

Pig converts the actions provided in the pig script form into MapReduce jobs. To perform Group By, it first maps the data by the key and then reduces the records with the help of this key. The group operator groups together all the tuples by the key. If the `by` clause has multiple keys, then the group key will be a tuple. You can get more information about the Group operator at `https://pig.apache.org/docs/r0.7.0/piglatin_ref2.html#GROUP`.

Performing Order By queries in Pig

In this recipe, we will use the Order By operator in Pig scripts to get the desired output.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Pig installed on it.

How to do it..

Order By is a very useful operator for data analysis when it comes to sequencing data records based on their values for certain attributes. In order to sequence the records in the proper order, Pig supports Order By.

To learn its usage, we will use the dataset that we took a look at in the previous recipe; in case you don't have the employee dataset, you can perform the following actions.

First of all, load the data in HDFS:

```
hadoop fs -mkdir /pig/emps_data
hadoop fs -put emps.txt /pig/emps_data
```

Next, we load data into a bag called `emps`, and then perform the Order By operation on this data on the basis of salary:

```
emps = LOAD '/pig/emps_data/emps.txt' AS (id, name, dept, salary);
```

Next, we will sequence the data by salary. We can also specify the order type, that is, whether it should be in ascending or descending form:

```
sorted_emps = ORDER emps by salary DESC;
DUMP sorted_emps;
```

At the end of the execution, you will see the following results:

```
(2,Sneha,PRODUCTION,8000)
(3,Sakalya,ENGINEERING,7000)
(6,Vinit,FINANCE,6200)
(4,Avinash,SALES,6000)
(5,Manisha,SALES,5700)
(1,Tanmay,ENGINEERING,5000)
```

How it works...

Pig converts the actions provided in the Pig script form into MapReduce jobs. When we execute the order through this command, a MapReduce job will get invoked in order to sort the data on the basis of the provided attribute. Pig performs a natural sort on the attribute, that is, here, we need to order employees by their salary. Therefore, the MapReduce job will get triggered so that it can sort the salary, which is of the data type integer. Similarly, if we order the record by a string, it sequences the string by its natural order.

Performing JOINS in Pig

In this recipe, we will learn how to perform various joins in Pig in order to join datasets.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Pig installed on it.

How to do it...

JOIN operations are very famous in SQL. Pig Latin also supports joining datasets based on a common attribute between them. Pig supports both Inner and Outer joins. Let's understand these syntaxes one by one.

In order to learn about Joins in Pig, we'll need two datasets. The first one is the employee dataset, which we have been using in earlier recipes, the second is the ID location dataset, which contains information about the ID of an employee and their location.

The employee dataset will look like this:

```
1       Tanmay ENGINEERING   5000
2       Sneha  PRODUCTION    8000
3       Sakalya       ENGINEERING   7000
4       Avinash       SALES  6000
5       Manisha       SALES  5700
6       Vinit  FINANCE       6200
```

The ID location dataset will look like this:

```
1       Pune
2       London
3       Mumbai
4       Pune
```

Like the `emps` data, we also loaded the ID locations data in HDFS and then into the `id_locations` bag:

```
hadoop fs -put id_locations /pig/emps_data
```

On grunt:

```
id_location = load '/pig/emps_data/id_locations' AS (emp_id,
location);
```

Now, to perform inner joins on these datasets, we'll execute the following command:

```
emps_location = JOIN emps BY id, id_location BY emp_id;
DUMP emps_location;
```

As result, you will get the following displayed on your screen:

```
(1,Tanmay,ENGINEERING,5000,1,Pune)
(2,Sneha,PRODUCTION,8000,2,London)
(3,Sakalya,ENGINEERING,7000,3,Mumbai)
(4,Avinash,SALES,6000,4,Pune)
```

As this is the inner join, only matching ID records will make it to the final results.

Next, we can execute OUTER Joins on the same dataset; to perform the LEFT OUTER join, we execute the following command:

```
emps_location_left_outer = JOIN emps BY id LEFT OUTER, id_location BY
emp_id;
DUMP emps_location_left_outer;
```

On successful execution, you will see the following results:

```
(1,Tanmay,ENGINEERING,5000,1,Pune)
(2,Sneha,PRODUCTION,8000,2,London)
(3,Sakalya,ENGINEERING,7000,3,Mumbai)
(4,Avinash,SALES,6000,4,Pune)
(5,Manisha,SALES,5700,,)
(6,Vinit,FINANCE,6200,,)
```

Here, all records from the LHS of the command will make it to the final table, and all the nonmatching records will be kept blank.

Similarly, we can perform RIGHT OUTER and FULL OUTER joins as well.

We can also specify the type of specialized joins supported by Pig, that is, Replicated Joins, Skewed Joins, and Merge Joins:

```
emps_location_left_outer = JOIN emps BY id LEFT OUTER, id_location BY
emp_id using 'replicated';
DUMP emps_location_left_outer;
```

The results are the same but Pig performs certain optimizations when we mention these types. Similarly, we can use skewed and merge joins as well.

How it works

Pig converts the actions provided in the Pig script form into Map Reduce jobs. In the previous chapter, we spoke about how to write a Map Reduce program to perform joins. When we execute the JOIN operation, Pig converts it into a Map Reduce job.

To perform optimizations while executing JOINS in pig, we can specify the join type as a replicated, skewed, or merge join. Let's understand what these joins mean.

Replicated Joins

Replicated joins are fragmented joins in which a piece of data is taken in memory and joined with other sets of data in a map function. These joins can only be performed on the Left Outer join. Having data in the memory helps reduce IO disk seeks and provides optimization during executions.



You can read more about Replicated Joins at `https://pig.apache.org/docs/r0.11.1/perf.html#replicated-joins`.

Skewed Joins

Parallel joins can be vulnerable to any skews in data. If the data is skewed enough, it might encounter some issues when balancing data parts during executions. To avoid this, skewed joins compute a histogram of key spaces and use this information to allocate reducers for a given key. Skewed joins only have certain conditions: first, currently, a skewed join can only be performed on two tables and not more than that at one go, second, it does not address uneven data balancing across reducers; it makes sure that tasks are executed slowly rather than failing them.



You can read more about skewed joins at `https://pig.apache.org/docs/r0.11.1/perf.html#skewed-joins`.

Merge Joins

Joins are beneficial when they are performed in the map phase of a Map Reduce program. Pig supports merge joins, which work with presorted data. Here, the idea is to gain performance benefits by performing joins in the map phase and avoid sending huge amounts of data through the sort and shuffle phase.



You can read more about merge joins at `https://pig.apache.org/docs/r0.11.1/perf.html#merge-joins`.

Writing a user-defined function in Pig

In this recipe, we will learn how to write user-defined functions (UDFs) in order to have our own custom filters.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Pig installed on it. We will also need an IDE, such as Eclipse, to write the Java class.

How to do it...

In this recipe, we are going to write user-defined functions for the dataset we have been considering in this chapter. Our dataset is an employee dataset, so let's assume that we want to convert all the names present in our dataset into uppercase. To do this, we will write a user-defined function to convert the lowercase letters into uppercase letters.

Writing a UDF is very simple: we need to write a class that extends the `EvalFunc` Pig class. In order to have this and other Hadoop classes in our class path, first of all, we need to create a maven project, and add the following dependencies to the `POM.xml` project:

```
<dependency>
    <groupId>org.apache.pig</groupId>
    <artifactId>pig</artifactId>
    <version>0.15.0</version>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-core</artifactId>
    <version>2.7.1</version>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>2.7.1</version>
</dependency>
```

Now, we write a class that extends the `EvalFunc` class and overwrites the `exec` method in it:

```
package com.demo.pig.udf;

import java.io.IOException;

import org.apache.pig.EvalFunc;
import org.apache.pig.data.DataByteArray;
import org.apache.pig.data.Tuple;


/**
 * Used Defined Function in Pig to covert given string argument into
Upper Case
 *
 */
public class ToUpper extends EvalFunc<String> {
```

```
        /**
         * Pig hands over the tuple instead of specific arguement, we need
    to get
         * the specific attribute from tuple by calling tuple.get() method
         *
         */
        @Override
        public String exec(Tuple input) throws IOException {
            if (null == input || input.size() == 0) {
                return null;
            }
            try {
                DataByteArray name = (DataByteArray) input.get(1);
                return name.toString().toUpperCase();

            } catch (Exception e) {
                throw new IOException("Caught exception while processing
    ", e);
            }

        }

    }
```

In the `exec` method, we get a complete tuple, and we need to convert the name column, which is at index `1` in our tuple. By default, if we don't specify any data type while loading data into the bag, the data type is assumed to be `DataByteArray`. So, we simply extract it and convert it into uppercase and return the converted string.

Now, we build our maven project to create a JAR of this implementation. Once the JAR is ready, we copy it to the Hadoop cluster and do the following things in order to use this UDF:

```
REGISTER ToUpperUDF-1.0.jar;
emps_upper = FOREACH emps GENERATE com.demo.pig.udf.ToUpper(*);
DUMP emps_upper;
```

In the first statement, we register the UDF jar with Pig. Next, we create a new bag, which would have converted the uppercase names from our previous `emps` bag. To start the execution, we DUMP the values in `emps_upper`.

Once the execution is complete, we will see the following results:

```
(TANMAY)
(SNEHA)
(SAKALYA)
```

```
(AVINASH)
(MANISHA)
(VINIT)
```

How it works...

Whenever we register a function in Pig, we include the implementation in the Pig class path. So, whenever the UDF class is mentioned, Pig looks for this class in its class path. If the proper name is provided at the time of execution, Pig will send the tuples to the respective implementation and get back the returned results.

There's more...

Similar to `EvalFunc`, we can also write `FilterFunc`, which can be used to filter out tuples based on certain values of the tuple. Here is an example function that filters out `ENGINEERING` department employees from the dataset:

```
package com.demo.pig.udf;

import java.io.IOException;

import org.apache.pig.FilterFunc;
import org.apache.pig.data.DataByteArray;
import org.apache.pig.data.Tuple;

public class EngFilter extends FilterFunc {

    @Override
    public Boolean exec(Tuple input) throws IOException {
        if (null == input || input.size() == 0) {
            return false;
        }
        try {
            DataByteArray name = (DataByteArray) input.get(2);
            if (name.toString().equalsIgnoreCase("ENGINEERING")) {
                return true;
            } else {
                return false;
            }

        } catch (Exception e) {
            throw new IOException("Caught exception while processing
", e);
        }
    }

}
```

Here, we check whether the department is engineering and only then return `true`, else we return `false`. Now, we build the jar and execute it using this filter in the following way:

```
REGISTER udfs-1.0.jar;
emps_eng = FILTER emps BY com.demo.pig.udf.EngFilter(*);
DUMP emps_eng;
```

At the end of the execution, you will see results like these:

```
(1,Tanmay,ENGINEERING,5000)
(3,Sakalya,ENGINEERING,7000)
```

Analyzing web log data using Pig

In this recipe, we will learn how to use Pig scripts to analyze web log data.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Pig installed on it.

How to do it...

In the previous chapter, we saw how to analyze web logs using the MapReduce program. In this recipe, we are going to take a look at how to use Pig scripts to analyze web log data. Let's consider two use cases:

Here is a sample of web log data:

```
106.208.17.105 - - [12/Nov/2015:21:20:32 -0800] "GET /tutorials/
mapreduce/advanced-map-reduce-examples-1.html HTTP/1.1" 200 0
"https://www.google.co.in/" "Mozilla/5.0 (Windows NT 6.3; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.86 Safari/537.36"
60.250.32.153 - - [12/Nov/2015:21:42:14 -0800] "GET /tutorials/
elasticsearch/install-elasticsearch-kibana-logstash-on-windows.html
HTTP/1.1" 304 0 - "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/46.0.2490.86 Safari/537.36"
49.49.250.23 - - [12/Nov/2015:21:40:56 -0800] "GET /tutorials/hadoop/
images/internals-of-hdfs-file-read-operations/HDFS_Read_Write.png
HTTP/1.1" 200 0 "http://hadooptutorials.co.in/tutorials/spark/install-
apache-spark-on-ubuntu.html" "Mozilla/5.0 (Windows NT 10.0; WOW64;
Trident/7.0; Touch; LCTE; rv:11.0) like Gecko"
```

```
60.250.32.153 - - [12/Nov/2015:21:36:01 -0800] "GET /tutorials/
elasticsearch/install-elasticsearch-kibana-logstash-on-windows.html
HTTP/1.1" 200 0 - "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/46.0.2490.86 Safari/537.36"

91.200.12.136 - - [12/Nov/2015:21:30:14 -0800] "GET /tutorials/hadoop/
hadoop-fundamentals.html HTTP/1.1" 200 0 "http://hadooptutorials.co.in/
tutorials/hadoop/hadoop-fundamentals.html" "Mozilla/5.0 (Windows NT 6.1)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.99 Safari/537.36"
```

These logs are formatted in the following manner:

- `%h` is the remote host (that is, the IP client)
- `%l` is the identity of the user determined by an identifier (this is not usually used since it's not reliable)
- `%u` is the username determined by the HTTP authentication
- `%t` is the time taken by the server to finish processing a request
- `%r` is the request line from a client ("GET / HTTP/1.0")
- `%>s` is the status code sent from a server to a client (200, 404, and so on)
- `%b` is the size of the response to a client (in bytes)
- The referrer is the page that is linked to the URL
- The user agent is a browser identification string

Now, let's load this data into the bag and start analyzing it:

```
hadoop fs –mkdir /data

hadoop fs –put web_log.tx /data
```

In grunt, we start loading this data into a bag:

```
raw_logs = LOAD '/data/web_log.txt' USING TextLoader AS (line:chararray);
```

This way, we can load each web load as a single line. Next, we write regex, which parses the data into respective attributes:

```
logs_base = FOREACH raw_logs GENERATE FLATTEN (REGEX_EXTRACT_
ALL(line,'^(\\S+) (\\S+) (\\S+) \\[([\\w:/]+\\s[+\\-]\\d{4})\\] "(.+?)"
(\\S+) (\\S+) "([^"]*)" "([^"]*)"') ) AS (ip: chararray, id: chararray,
user: chararray, time: chararray, request: chararray, status: int, bytes_
string: chararray, referrer: chararray, browser: chararray);
```

Now, we need to find the referrer sites from which we are getting the most traffic. To do this, we execute the following commands:

```
by_referrer = GROUP logs_base BY referrer

referrer_counts = FOREACH by_referrer GENERATE
group as referrer,
COUNT(logs_base);

DUMP referrer_counts;
```

On successful completion, we will see these results:

```
(https://www.google.co.in/,1)
(http://hadooptutorials.co.in/tutorials/hadoop/hadoop-fundamentals.
html,1)
(http://hadooptutorials.co.in/tutorials/spark/install-apache-spark-on-
ubuntu.html,1)
(,0)
```

Next, we can find the requests that have resulted into some errors. Here, an error could be any request whose status is 4XX or 5XX. Now, we write a filter function to execute this:

```
package com.demo.pig.udf;

import java.io.IOException;

import org.apache.pig.FilterFunc;
import org.apache.pig.data.Tuple;

public class ErrorFilter extends FilterFunc {

    @Override
    public Boolean exec(Tuple input) throws IOException {
        if (null == input || input.size() == 0) {
            return false;
        }
        try {
            String statusStr = (String) input.get(5);
            if (null == statusStr){
                return false;
            }
            int status = Integer.parseInt(statusStr);
            if (status <= 400 && status >= 505) {
                return true;
```

```
                } else {
                    return false;
                }

            } catch (Exception e) {
                throw new IOException("Caught exception while processing
    ", e);
            }
        }
    }
```

We create a jar of this class and copy it in the Hadoop cluster. We'll execute the following to get the web logs with errors:

```
REGISTER udfs-1.0.jar ;
log_base_error = FILTER logs_base BY com.demo.pig.udf.ErrorFilter(*);
DUMP log_base_error;
```

How it works...

Here, we use various things that we have already taken a look at in the earlier recipes. The important thing is to parse the logs using the correct regex. If you get this right, the next few things will be a cakewalk for you.

Performing the Hbase operation in CLI

Now that we have gone through various operations in Pig and Hive, it's time to learn about some operations in Hbase. Hbase is a NoSQL database and runs on top of HDFS. In this recipe, we are going to take a look at how to perform various Hbase command-line operations.

Getting ready

To perform this recipe, you should have a Hadoop cluster running as well as the latest version of Hbase installed on it. In case you don't know how to install Hbase, here is a link you can refer to `http://hbase.apache.org/book.html#quickstart`.

How to do it

Once the installation is complete, execute the following command to start daemons that are related to Hbase such as Zookeeper, Hbase Master, and Hbase region server:

```
/usr/local/hbase/bin/start-hbase.sh
```

Once all the daemons are up and running, you can start the Hbase shell by executing the following code:

```
hbase shell
```

You can also open the Hbase UI in a browser. You can hit the following URL to view the Hbase UI:

`http://<hostname>:16010`.

Let's start with a command in Hbase.

1. List:

   ```
   hbase(main):001:0> list
   TABLE
   test_table
   1 row(s) in 0.3730 seconds

   => ["test_table"]
   ```

   This lists the tables that are present in the key space.

2. Status:

   This returns the status of the system and the number of servers:

   ```
   hbase(main):002:0> status
   1 servers, 0 dead, 3.0000 average load
   ```

3. Version:

   This prints the running version of Hbase:

   ```
   hbase(main):003:0> version
   1.0.1.1, re1dbf4df30d214fca14908df71d038081577ea46, Sun May 17
   12:34:26 PDT 2015
   ```

4. `table_help`:

   This provides help for table-related operations:

   ```
   hbase(main):004:0> table_help
   ```

   Help for table-reference commands.

   You can either create a table via `create` and then manipulate the table via commands like `put`, `get`, and so on.

   See the standard help information for how to use each of these commands.

   However, as of 0.96, you can also get a reference to a table, on which you can invoke commands.

For instance, you can get create a table and keep around a reference to it via:

```
hbase> t = create 't', 'cf'
```

Or, if you have already created the table, you can get a reference to it:

```
hbase> t = get_table 't'
```

You can do things like call `put` on the table:

```
hbase> t.put 'r', 'cf:q', 'v'
```

which puts a row 'r' with column family 'cf', qualifier 'q' and value 'v' into table `t`. To read the data out, you can scan the table:

```
hbase> t.scan
```

which will read all the rows in table 't'.

Essentially, any command that takes a table name can also be done via table reference. Other commands include things like: `get`, `delete`, `deleteall`., `get_all_columns`, `get_counter`, `count`, `incr`. These functions, along with the standard `JRuby` object methods are also available via tab completion.

For more information on how to use each of these commands, you can also just type:

```
hbase> t.help 'scan'
```

which will output more information on how to use that command.

You can also perform general admin actions directly on `table`; `things` such as `enable`, `disable`, `flush`, and `drop` just by typing this:

```
hbase> t.enable
hbase> t.flush
hbase> t.disable
hbase> t.drop
```

Note that after dropping a table, your reference to it becomes useless and further usage is undefined (and not recommended).

5. Creating a table:

   Now, let's create a table called `employee`, which contains two column families: one to store `personal details` and the other to store `professional details`:

```
hbase(main):005:0> create 'emp', 'personal details', 'professional
details';
```

6. Describe the table:

   We can describe the created table to know about its details:

```
hbase(main):017:0> describe 'emp'
Table emp is ENABLED
emp
```

```
COLUMN FAMILIES DESCRIPTION
{NAME => 'personal details', DATA_BLOCK_ENCODING => 'NONE',
BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0', VERSIONS => '1',
COMPRESSION => 'NONE'
, MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS =>
'FALSE', BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE =>
'true'}
{NAME => 'professional details', DATA_BLOCK_ENCODING => 'NONE',
BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0', VERSIONS => '1',
COMPRESSION => 'N
ONE', MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS =>
'FALSE', BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE =>
'true'}
2 row(s) in 0.0170 seconds
```

7. Inserting data into the table

   We can insert data into the tables using the `put` command, as shown here:

```
hbase(main):018:0> put 'emp','1','personal details:name','Ram'
0 row(s) in 0.0820 seconds

hbase(main):019:0> put 'emp','1','personal
details:location','Pune'
0 row(s) in 0.0100 seconds

hbase(main):020:0>
hbase(main):021:0* put 'emp','1','professional
details:designation','Software Engineer'
0 row(s) in 0.0090 seconds

hbase(main):022:0> put 'emp','1','professional
details:salary','10000'
0 row(s) in 0.0060 seconds
```

8. Viewing records:

   We can see the inserted data by using the `scan` command, as shown here:

```
hbase(main):023:0> scan 'emp'
ROW                                   COLUMN+CELL
 1                                    column=personal
details:location, timestamp=1448777442105, value=Pune

 1                                    column=personal
details:name, timestamp=1448777442076, value=Ram
```

```
 1                                    column=professional
details:designation, timestamp=1448777442149, value=Software
Engineer
 1                                    column=professional
details:salary, timestamp=1448777442169, value=10000

1 row(s) in 0.0280 seconds
```

9. Updating the record:

   To update the record, we need to use the `put` command, as shown in the following code snippet. Here, we will be updating the salary to `12000`:

```
hbase(main):024:0> put 'emp','1','professional
details:salary','12000'
0 row(s) in 0.0070 seconds

hbase(main):025:0> scan 'emp'
ROW                                  COLUMN+CELL
 1                                    column=personal
details:location, timestamp=1448777442105, value=Pune
 1                                    column=personal
details:name, timestamp=1448777442076, value=Ram
 1                                    column=professional
details:designation, timestamp=1448777442149, value=Software
Engineer
 1                                    column=professional
details:salary, timestamp=1448777689830, value=12000

1 row(s) in 0.0110 seconds
```

10. Getting the record

    To get the record from Hbase, we can use the `get` command:

```
hbase(main):026:0> get 'emp', '1'
COLUMN                               CELL

 personal details:location           timestamp=1448777442105, val
ue=Pune
 personal details:name               timestamp=1448777442076, val
ue=Ram
 professional details:designation    timestamp=1448777442149,
value=Software Engineer
 professional details:salary         timestamp=1448777689830,
value=12000
4 row(s) in 0.0240 seconds
```

11. Getting a specific column from the record:

```
hbase(main):002:0> get 'emp', '1', { COLUMN => 'personal
details:name'}
COLUMN                                  CELL
 personal details:name                  timestamp=1448777442076,
value=Ram
1 row(s) in 0.2270 seconds
```

12. Deleting a specific column:

    We can delete a specific column in Hbase by executing the `delete` commanding in the following manner

```
hbase(main):004:0> delete 'emp', '1', 'personal details:location',
1448777442076
0 row(s) in 0.0430 seconds
```

13. Deleting a complete row:

    We can delete a complete row by using the `delete all` command.

```
hbase(main):006:0> deleteall 'emp','1'
0 row(s) in 0.0070 seconds

hbase(main):007:0> scan 'emp'
ROW                                     COLUMN+CELL
0 row(s) in 0.0060 seconds
```

14. Counting the number of rows:

    We can use the `count` command to count the number of rows in a specific table.

```
hbase(main):014:0> count 'emp'
1 row(s) in 0.0090 seconds
=> 1
```

15. Dropping a table.

    We can drop a table by first disabling it and then executing the dropped table:

```
hbase(main):016:0> disable 'emp'
0 row(s) in 1.2120 seconds

hbase(main):017:0> drop 'emp'
0 row(s) in 0.1670 seconds
```

How it works...

Hbase is a NoSQL database, which runs on top of HDFS. We know that HDFS is used for the batch access of data, but it cannot be used to randomly access data, whereas Hbase is used to randomly access data. It is a columnar database where we can have thousands of columns for a particular row. The following is the table structure in which we can save data:

Row Id	Personal Details		Professional Details	
EMP ID	Name	Location	Designation	Salary
1	Ram	Pune	Software Engineer	12000
2	Ramesh	Mumbai	Manager	15000
3	Radha	Delhi	QA	8000
4	Rahul	Mumbai	Test Manager	10000

In the preceding table, we have a table called `emp`, and we have two column families, personal details and professional details. Each row is identified by the row ID, which is unique for each record. Hbase consists of master and slave architecture. It has one master and multiple slaves called region servers. Each region server manages multiple regions. A table is partitioned into multiple regions across the region servers. Actual data is saved to HFile. This is a block indexed file format where data is stored into a sequence of blocks, and a separate index is maintained at the end of file. Hbase maintains the in-memory log file called HLog. This file contains the updates happening in tables. This cache is flushed periodically.

Performing Hbase operations in Java

Hbase provides a Java client with which we can perform operations similar to those performed through the command line.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hbase installed on it.

How to do it

To get started, we first need to create a maven project and add the Hbase client dependency:

```
<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-client</artifactId>
    <version>1.0.1.1</version>
</dependency>
```

Now, we write a Java class, which uses APIs from the Hbase client to perform various operations:

```
package com.demo.hbase.hbase.client;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.util.Bytes;

public class JavaHbase {

    public static void main(String args[]) throws IOException {

        // Create connection and admin client
        Configuration config = HBaseConfiguration.create();
        Connection connection = ConnectionFactory.
createConnection(config);
        Admin admin = connection.getAdmin();

        // Provide table description
        HTableDescriptor descriptor = new HTableDescriptor(TableName.
valueOf("emp"));
        descriptor.addFamily(new HColumnDescriptor("personal
details"));
        descriptor.addFamily(new HColumnDescriptor("professional
details"));
        // create table
        admin.createTable(descriptor);
        System.out.println("Table Created");

        // Insert data
        Put name = new Put(Bytes.toBytes("1"));
```

```
        name.addColumn(Bytes.toBytes("personal details"), Bytes.
toBytes("name"), Bytes.toBytes("Ram"));

        HTable table = new HTable(config, "emp");
        table.put(name);

        System.out.println("Added data into Table");

        // Get data
        Get g = new Get(Bytes.toBytes("1"));
        Result r = table.get(g);
        byte[] value = r.getValue(Bytes.toBytes("personal details"),
Bytes.toBytes("name"));
        System.out.println("Fetched Data:" + value.toString());

    }
}
```

Now, we create a runnable jar, and execute it in the following manner:

```
java -cp hbase-client-1.0-jar-with-dependencies.jar com.demo.hbase.
hbase.client.JavaHbase
```

How it works...

Hbase provides clients in various languages such as Java, Scala, and C/C++. You can choose a language of your choice to handle data in Hbase remotely. You don't need anything special to use those APIs.

Executing the MapReduce programming with an Hbase Table

In this recipe, we are going to see how to use Hbase data as input to the MapReduce program.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hbase installed on it. You will also need an IDE such as Eclipse.

How to do it

To get started, we first need to create a maven project, and add the following dependencies to it:

```
<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-client</artifactId>
    <version>1.0.1.1</version>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-core</artifactId>
    <version>2.7.1</version>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>2.7.1</version>
</dependency>
<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-server</artifactId>
    <version>1.0.1.1</version>
</dependency>
```

Next, we write a driver class, which uses `TableMapReduceUtils`. In this example, we will be reading data from one Hbase table and write it to another Hbase table:

```
public class HbaseExample {
    public static void main(String[] args) throws Exception {
        Configuration config = HBaseConfiguration.create();
        Job job = Job.getInstance(config, "Hbase Example");
        job.setJarByClass(HbaseExample.class);

        Scan scan = new Scan();
        scan.setCaching(500);
        scan.setCacheBlocks(false);

        TableMapReduceUtil.initTableMapperJob("emp", scan, MyMapper.
class, Text.class, IntWritable.class, job);
        TableMapReduceUtil.initTableReducerJob("new_emp",
MyTableReducer.class, job);
        job.setNumReduceTasks(1);
```

```
        boolean b = job.waitForCompletion(true);
        if (!b) {
            throw new IOException("error with job!");
        }
    }
}
```

We write a `Mapper` class, which reads data from the Hbase table:

```
public static class MyMapper extends TableMapper<Text, IntWritable> {

        private final IntWritable ONE = new IntWritable(1);
        private Text text = new Text();

        public void map(ImmutableBytesWritable row, Result value,
Context context)
                throws IOException, InterruptedException {
            String val = new String(value.getValue(Bytes.
toBytes("peronsonal details"), Bytes.toBytes("name")));
            text.set(val);

            context.write(text, ONE);
        }
    }
```

Now, it's time to read the `Reducer` class, which writes each record to the new table:

```
public static class MyTableReducer extends TableReducer<Text,
IntWritable, ImmutableBytesWritable> {

        public void reduce(Text key, Iterable<IntWritable> values,
Context context)
                throws IOException, InterruptedException {
            int i = 0;
            for (IntWritable val : values) {
                i += val.get();
            }
            Put put = new Put(Bytes.toBytes(key.toString()));
            put.add(Bytes.toBytes("details"), Bytes.toBytes("count"),
Bytes.toBytes(i));

            context.write(null, put);
        }
    }
```

We build a jar out of these classes and execute them on Hadoop using the following command:

```
hadoop jar hbase-mr-1.0-jar-with-dependencies.jar com.demo.hbase.hbase.
client.HbaseExample
```

How it works

As Hbase also uses HDFS as its base to store data, it's quite easy to read that into a Map Reduce job using clients provided by Hbase. All other functionalities of Map Reduce remain the same.

5
Advanced Data Analysis Using Hive

- Processing JSON data in Hive using JSON `SerDe`
- Processing XML data in Hive using XML `SerDe`
- Processing Hive data in the Avro format
- Writing a user-defined function in Hive
- Performing table joins in Hive
- Executing map side joins in Hive
- Performing context Ngram in Hive
- Analyzing a call data record using Hive
- Performing sentiment analysis using Hive on Twitter data
- Implementing **Change Data Capture** (**CDC**) using Hive
- Inserting data in multiple tables data using Hive

Introduction

In the previous chapter, we discussed various tasks that can be performed using Hive, Pig, and Hbase. In this chapter, we are going to take a look at how to perform some advanced tasks using Hive. We will see how to analyze data in various formats such as JSON, XML, and AVRO. We will also explore how to write **User-Defined Functions** (**UDFs**) in Hive, deploy them, and use them in Hive queries. Now let's get started.

Processing JSON data in Hive using JSON SerDe

These days, JSON is a very common data structure that's used for data communication and storage. Its key value-based structure gives great flexibility in handling data. In this recipe, we are going to take a look at how to process data stored in the JSON format in Hive. Hive does not have any built-in support to handle JSON, so we will be using JSON `SerDe`. `SerDe` is a program that consists of a serializer and deserializer, which tell Hive how to read and write data.

Getting ready

To perform this recipe, you should have a running Hadoop cluster with the latest version of Hive installed on it. Here, I am using Hive 1.2.1. Apart from Hive, we also need JSON `SerDe`.

There are various JSON `SerDe` binaries available from various developers. The most popular, though, can be found at `https://github.com/rcongiu/Hive-JSON-Serde`.

This project contains code for JSON `SerDe` and is compatible with the latest version of Hive. You can either download the code and build your own JAR, or you can download the precompiled jars from `https://github.com/sheetaldolas/Hive-JSON-Serde/tree/master/dist`.

I am using `https://github.com/sheetaldolas/Hive-JSON-Serde/blob/master/dist/json-serde-1.1.9.9-Hive1.2.jar`.

Download the jar, and copy it to the node where you have Hive installed.

How to do it...

Now that we have set up our environment, let's consider a use situation where we have the JSON data with us and want to process this data using Hive. Let's say we have an e-commerce website where we have data coming in the JSON format for various products, as shown here:

```
{
    "name": "iPhone 6",
    "price": 600,
    "category": "phone",
    "color": "gold",
    "stock": 10,
    "tags": ["phone", "iphone", "cell"]
}
```

To do this, we first need to put this data in HDFS. Here is some sample data:

```
{"name":"iPhone 6", "price": 600, "category" :"phone", "color":
"gold", "stock": 10, "tags" : ["phone", "iphone", "cell"] }
{"name":"iPhone 6 plus", "price": 660, "category" :"phone", "color":
"silver", "stock": 20, "tags" : ["phone", "iphone", "cell"] }
{"name":"Samsung S6", "price": 600, "category" :"phone", "color":
"white", "stock": 10, "tags" : ["phone", "samsung", "cell"] }
{"name":"Macbook Pro", "price": 800, "category" :"computer", "color":
"silver", "stock": 8, "tags" : ["laptop", "apple"] }
{"name":"Samsung Refrigerator 13 Ltr", "price": 400, "category"
:"fridge", "color": "red", "stock": 10, "tags" : ["fridge", "samsung"]
}
```

hadoop fs -mkdir /data/products

hadoop fs -copyFromLocalproducts.json /data/products

Now let's start the Hive prompt and add the jar we downloaded earlier.

```
hive> ADD JAR json-serde-1.1.9.9-Hive1.2.jar;
Added [json-serde-1.1.9.9-Hive1.2.jar] to class path
Added resources: [json-serde-1.1.9.9-Hive1.2.jar]
```

Now, it's time to create a Hive table using JSON `SerDe`:

```
CREATE TABLE products_json (
name STRING,
price DOUBLE,
category STRING,
color STRING,
stock INT,
tags ARRAY<STRING>)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
LOCATION '/data/products';
```

When using `SerDe`, we also have to use Row format `SerDe` and provide the main class for this JAR.

If everything is okay, you will get a confirmation that the Hive table has been created.

Now we can query and process this data like any other Hive query. To confirm that the data has been loaded properly, we can execute the `select` query:

```
hive> select * from products_json;
OK
iPhone 6  600.0  phone  gold   10     ["phone","iphone","cell"]
iPhone 6 plus    660.0  phone  silver 20     ["phone","iphone","cell"]
```

```
Samsung S6       600.0  phone  white  10     ["phone","samsung","cell"]
Macbook Pro      800.0  computer      silver 8      ["laptop","apple"]
Samsung Refrigerator 13 Ltr   400.0  fridge red    10
    ["fridge","samsung"]
Time taken: 0.763 seconds, Fetched: 5 row(s)
```

We can run a query to get product counts by their category, as shown here:

```
SELECT category, count(*) FROM products_json
GROUP BY category;
```

This will start a map reduce job, and on its completion, we will have results like this:

```
computer  1
fridge    1
phone     3
```

JSON `SerDe` supports nested JSON as well. Consider a situation where we have an attribute called a feature, as shown here:

```
{
    "name": "iPhone 6",
    "price": 600,
    "category": "phone",
    "color": "gold",
    "stock": 10,
    "tags": ["phone", "iphone", "cell"],
    "features": {
        "storage": "64GB",
        "battery": "15Hrs",
        "warranty": "1 Yr"
    }
}
```

To have this type of data, we need to update the `create` table statement:

```
CREATE TABLE products_json (
name STRING,
price DOUBLE,
category STRING,
color STRING,
stock INT,
tags ARRAY<STRING>,
features STRUCT<
storage:STRING,
battery:STRING,
```

```
warranty:STRING>)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
LOCATION '/data/products';
```

How it works...

- While reading the input, Hive's execution engine first reads the records using the default `InputFormat`. `RecordReader` keeps sending one record at a time. Here, we don't provide `InputFormat`, so the default is text. The engine then invokes `Serde.deserialize()` to deserialize the record. The Hive engine also gets hold of an object inspector by invoking the `Serde.getObjectInspector()` method. The engine then passes the deserialized object and object inspector to the operators for their use. Each operator uses this deserialized object and object inspector to get data from the data structure.



Hive also supports the writing of your own custom `SerDe`; the documentation for this can be found at—`https://cwiki.apache.org/confluence/display/Hive/DeveloperGuide#DeveloperGuide-HowtoWriteYourOwnSerDe`.

Processing XML data in Hive using XML SerDe

XML has been one of the most important data structures and has been used for quite a long time for data transfers and storage. Parsing XML data and then processing it is always a tricky task as parsing XML is one of the most costliest operations. Hive does not have any built-in support for XML data processing, but many organizations and individuals have made open source contributions to XML `SerDe`.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am using Hive 1.2.1. Apart from Hive, we also need XML `SerDe`.

There are various XML `SerDe` that have been made available by open source developers. Out of these, XML `SerDe` at `https://github.com/dvasilen/Hive-XML-SerDe` is well developed and quite useful. So, we can download the jar from `http://search.maven.org/remotecontent?filepath=com/ibm/spss/hive/serde2/xml/hivexmlserde/1.0.5.3/hivexmlserde-1.0.5.3.jar`.

How to do it...

Now, let's assume that in our e-commerce project, we have data that is stored in the XML format, as follows:

```
<products>
    <product id="1">
        <name>iPhone 6</name>
        <category>phone</category>
        <price>600</price>
        <stock>14</stock>
        <features>
            <storage>64GB</storage>
            <battery>20Hrs</battery>
        </features>
    </product>
    <product id="2">
        <name>iPhone 6 Plus</name>
        <category>phone</category>
        <price>800</price>
        <stock>15</stock>
        <features>
            <storage>64GB</storage>
            <battery>10Hrs</battery>
        </features>
    </product>
    <product id="3">
        <name>Samsung S6</name>
        <category>phone</category>
        <price>650</price>
        <stock>12</stock>
        <features>
            <storage>16GB</storage>
            <battery>10Hrs</battery>
        </features>
    </product>
</products>
```

We will be using the XML `SerDe` to analyze this data. First of all, we need to start a Hive prompt and add the downloaded jar:

```
hive> ADD JAR hivexmlserde-1.0.5.3.jar;
Added [hivexmlserde-1.0.5.3.jar] to class path
Added resources: [hivexmlserde-1.0.5.3.jar]
```

Next, we'll write create a table statement, as shown here:

```
CREATE TABLE product_xml(
id STRING,
name STRING,
category STRING,
price DOUBLE,
stock BIGINT,
features map<string,string>)
ROW FORMAT SERDE 'com.ibm.spss.hive.serde2.xml.XmlSerDe'
WITH SERDEPROPERTIES (
"column.xpath.id"="/product/@id",
"column.xpath.name"="/product/name/text()",
"column.xpath.category"="/product/category/text()",
"column.xpath.price"="/product/price/text()",
"column.xpath.stock"="/product/stock/text()",
"column.xpath.features"="/product/features/*"
)
STORED AS
INPUTFORMAT 'com.ibm.spss.hive.serde2.xml.XmlInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat'
TBLPROPERTIES (
"xmlinput.start"="<product id",
"xmlinput.end"="</product>"
);
```

Now that the table is created, we can load the data into this table and execute the `select` query to check whether the data is loaded properly:

```
LOAD DATA LOCAL INPATH 'products.xml' INTO TABLE product_xml;
hive> select * from product_xml;
OK
1   iPhone 6      phone  600.0  14     {"battery":"20Hrs","storage":"64G
B"}
2   iPhone 6 Plus phone  800.0  15     {"battery":"10Hrs","storage":"64G
B"}
3   Samsung S6    phone  650.0  12
    {"battery":"10Hrs","storage":"16GB"}
Time taken: 0.071 seconds, Fetched: 3 row(s)
```

We can execute various queries, such as other Hive queries, and the XML `SerDe` will automatically take care of serialization and deserialization:

```
hive> select * from product_xml
>where price > 620;
OK
```

```
2   iPhone 6 Plus phone  800.0  15     {"battery":"10Hrs","storage":"64G
B"}
3   Samsung S6    phone  650.0  12     {"battery":"10Hrs","storage":"16G
B"}
Time taken: 0.123 seconds, Fetched: 2 row(s)
```

How it works

All `SerDe` work in the manner that is explained in the *How it works* section of the previous recipe.

Processing Hive data in the Avro format

Avro is an evolvable schema-driven binary data format. It is hosted and maintained by the Apache Software Foundation (`http://avro.apache.org/`). It provides a rich data structure to store compact, fast binary data, and it relies on schemas. Avro files store data and schemas together; this helps faster reading of data as the files do not need to look for schema anywhere else. It can also be used in **Remote Procedure Calls** (**RPC**). There, the schema is transferred at the time of handshake between a client and server. In this recipe, we will take a look at how to process Avro files in Hive.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am using Hive 1.2.1. Hive has built-in support for the Avro file format, so we don't need to import any third-party JARs.

How to do it...

Using Avro `SerDe`, we can either read data that is already in the Avro format or write new data in the Avro file. We cannot write the Avro file using an editor, so we will be using a sample file that's provided along with the Hive installation, that is, `doctors.avro`. This contains a list of doctors. To read the Avro data, we need to create a table in the following manner:

```
CREATE TABLE doctors(
number INT,
first_name STRING,
last_name STRING)
STORED AS AVRO;
```

Now, let's load data into this table, as shown here:

```
LOAD DATA LOCAL INPATH '/usr/local/hive/examples/files/doctors.avro'
INTO TABLE doctors;
```

We can execute the SELECT query to check whether we are able read the records:

```
hive> select * from doctors;
OK
    6  Colin  Baker
    3  Jon    Pertwee
    4  Tom    Baker
    5  Peter  Davison
    11 Matt   Smith
    1  William      Hartnell
    7  Sylvester    McCoy
    8  Paul   McGann
    2  Patrick      Troughton
    9  Christopher  Eccleston
    10 David  Tennant
    Time taken: 0.051 seconds, Fetched: 11 row(s)
```

We can execute any query just like other Hive queries in this table.

We can also write the output of a Hive query in the Avro format by creating another table, as shown here:

```
CREATE TABLE doctors_selected(
number INT,
first_name STRING,
last_name STRING)
STORED AS AVRO;
```

We import the selected data from the previous table, as shown here:

```
INSERT INTO TABLE doctors_selected
SELECT * FROM doctors where number > 5;
```

Once the map reduce execution is complete, we can take a look at the imported data:

```
hive> select * from doctors_selected;
OK
    6  Colin  Baker
    11 Matt   Smith
    7  Sylvester    McCoy
    8  Paul   McGann
    9  Christopher  Eccleston
    10 David  Tennant
    Time taken: 0.061 seconds, Fetched: 6 row(s)
```

We can also take a look at the generated file:

```
hadoop fs -ls /user/hive/warehouse/doctors_selected
    Found 1 items
    -rwxr-xr-x   1 admin1 supergroup        420 2015-12-06 12:44
    /user/hive/warehouse/doctors_selected/000000_0
```

Here the `000000_0` file is in the Avro file format.

How it works...

All `SerDe` work in a similar manner as explained in the *How it works* sections of the recipe—*Processing JSON data using Hive JSON SerDe*.

Writing a user-defined function in Hive

In the previous chapter, we talked about how to write user-defined functions in Pig; in this recipe, we are going to do the same for Hive. Hive supports the adding of temporary functions, which can be used to process data. We will be writing UDF in Java and will also create functions that can be used in data processing.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am using Hive 1.2.1. We will also need the Eclipse IDE for development.

How to do it

There are various system functions that are supported by Hive, but sometimes, you will need to do something different that cannot be handled by system provided functions. To do this, we will need to write a custom function.

Take a situation where we have census data and a person's income, and we want to categorize them into three parts based on the person's income. The following is some sample data where we have the ID, name, and salary:

```
1|Tanmay|10000
2|Sneha|12000
3|Sakalya|14000
4|Ramesh|3000
5|Rahul|4000
```

```
6|Rajesh|18000
7|Ram|3000
```

Now, we will create a Hive table and load this data into the table:

```
CREATE TABLE census(
id INT,
name STRING,
salary INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH 'census.txt' INTO TABLE census;
```

We can check the loaded data, as shown here:

```
hive> select * from census;
OK
 1  Tanmay 10000
 2  Sneha  12000
 3  Sakalya      14000
 4  Ramesh 3000
 5  Rahul  4000
 6  Rajesh 18000
 7  Ram    3000
 Time taken: 0.052 seconds, Fetched: 7 row(s)
```

Let's write a UDF to segregate people based on their income group. To do so, we will need to create a maven project, and add the following dependency in it:

```
<dependency>
        <groupId>org.apache.hive</groupId>
        <artifactId>hive-exec</artifactId>
        <version>1.2.1</version>
</dependency>
```

Here is the Java class that provides logic for the income group function. To write a UDF, we need to extend the UDF class from `hive-exec` jar:

```
packagecom.demo.hive.udfs;

importorg.apache.hadoop.hive.ql.exec.UDF;
importorg.apache.hadoop.io.IntWritable;
```

```
importorg.apache.hadoop.io.Text;

public class IncomeClassifier extends UDF {

    public Text evaluate(IntWritable income) {
        Text incomeGroup = new Text();
        if (income.get() <= 5000) {
            incomeGroup.set("lower");
        } else if (income.get() >= 5001 &&income.get() <= 15000) {
            incomeGroup.set("middle");
        } else if (income.get() >= 15001) {
            incomeGroup.set("upper");
        }
        returnincomeGroup;
    }
}
```

We compile this code and create a jar out of it. Now, we copy this jar to the place where Hive is installed, and do the following things to create a function:

```
LOAD DATA LOCAL INPATH 'census.txt' INTO TABLE census;
ADD JAR hive-udfs.jar
CREATE TEMPORARY FUNCTION income_group as 'com.demo.hive.udf.
IncomeClassifier';
```

Use this UDF in the following manner:

```
SELECT id, name, income_group(salary)
FROM census;
hive> SELECT id, name, income_group(salary)
> FROM census;
OK
    1  Tanmay middle
    2  Sneha  middle
    3  Sakalya      middle
    4  Ramesh lower
    5  Rahul  lower
    6  Rajesh upper
    7  Ram    lower
    Time taken: 0.058 seconds, Fetched: 7 row(s)
```

How it works...

Whenever UDF is referenced during execution, the respective class is invoked and the `evaluate` method is executed. Based on the logic provided, a value is returned. UDFs are effective till the time your session is alive. When the terminal is closed, the same UDF cannot be referenced again. If we want to do that, we need add the jar again.

Performing table joins in Hive

In the previous chapter, we talked about how to perform joins in Pig. In this recipe, we are going to take a look at how to perform joins in Hive. Hive supports various types of joins such as inner, outer, and so on.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am using Hive 1.2.1.

How to do it...

To perform joins, we will need two types of datasets, which have something in common to join. Consider a situation where we have two employee tables and departments, and every employee table has a structure (ID, name, salary, and department ID) and every department table has an ID and a name. We will quickly create tables and load data into them:

```
CREATE TABLE emp(
id INT,
name STRING,
salary DOUBLE,
deptId INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH 'emp.txt' INTO TABLE emp;
```

```
hive> select * from emp;
OK
1  Tanmay 3500.0 101
2  Sneha  5599.0 101
3  Avinash       6700.0 102
4  Manisha       6400.0 103
```

```
5  Sakalya      3200.0 102
6  Vinit  3200.0 105
Time taken: 0.064 seconds, Fetched: 6 row(s)
```

```
CREATE TABLE dept(
id INT,
name STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;
LOAD DATA LOCAL INPATH 'dept.txt' INTO TABLE dept;
```

```
hive> select * from dept;
OK
101      Engineering
102      HR
103      Finance
104      Facilities
Time taken: 0.061 seconds, Fetched: 4 row(s)
```

Now, let's start performing various joins.

For the inner join, here is the code:

```
SELECT e.name, d.name FROM
emp e JOIN dept d
ON (e.deptId = d.id);
```

The result will look similar to the following code. The inner join only returns matching records from both tables:

```
Tanmay Engineering
Sneha  Engineering
Avinash      HR
Manisha      Finance
Sakalya      HR
```

Left outer join

The Left outer join gives all records from the left-hand side of a table and only matches records with the other table:

```
SELECT e.name, d.name FROM
emp e LEFT OUTER JOIN dept d
ON (e.deptId = d.id);
```

The results of this are as follows:

```
Tanmay    Engineering
Sneha     Engineering
Avinash   HR
Manisha   Finance
Sakalya   HR
Vinit     NULL
```

Right outer join

The right outer join gives all the records from the right-hand side of a table and only matches records from the other table:

```
SELECT e.name, d.name FROM
emp e RIGHT OUTER JOIN dept d
ON (e.deptId = d.id);
```

The results of this are as follows:

```
Tanmay     Engineering
Sneha      Engineering
Avinash    HR
Sakalya    HR
Manisha    Finance
NULL       Facilities
```

Full outer join

The full outer join gives a complete set of matching and non-matching records from both tables:

```
SELECT e.name, d.name FROM
emp e FULL OUTER JOIN dept d
ON (e.deptId = d.id);
```

The results of this are as follows:

```
Sneha     Engineering
Tanmay    Engineering
Sakalya   HR
Avinash   HR
Manisha   Finance
NULL      Facilities
Vinit     NULL
```

Left semi join

The left semi join can be used when you want columns from only one table. When compared to the inner join, it gives a similar output except that it does not return columns from the right-hand side of the table:

```
SELECT e.name, e. salary FROM
emp e LEFT SEMI JOIN dept d
ON (e.deptId = d.id);
```

The results of this are as follows:

```
Tanmay   3500.0
Sneha    5599.0
Avinash  6700.0
Manisha  6400.0
Sakalya  3200.0
```

How it works...

In every join operation, Hive will execute a Map Reduce job. In *Chapter 3*, *Mastering Map Reduce Programs*, we took a look at the code that gets implemented for various join operations. Refer to those recipes for more details.

Executing map side joins in Hive

Map side joins are special types of optimizations; Hive executes these automatically based on table sizes. In this recipe, we are going to explore map side joins in further detail.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am using Hive 1.2.1.

How to do it...

To perform map joins, we need two types of datasets that have something in common to join. One dataset also has to be big, and the other has to be small in comparison. Consider a situation where we have two tables for employees and departments; the employee table has a structure (ID, name, salary, and department ID) and the department table has an ID and a name.

We will quickly create tables and load data into them:

```
CREATE TABLE emp(
id INT,
name STRING,
salary DOUBLE,
deptId INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;
```

```
LOAD DATA LOCAL INPATH 'emp.txt' INTO TABLE emp;
hive> select * from emp;
OK
1   Tanmay 3500.0 101
2   Sneha  5599.0 101
3   Avinash       6700.0 102
4   Manisha       6400.0 103
5   Sakalya       3200.0 102
6   Vinit  3200.0 105
Time taken: 0.064 seconds, Fetched: 6 row(s)
```

```
CREATE TABLE dept(
id INT,
name STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;
```

```
LOAD DATA LOCAL INPATH 'dept.txt' INTO TABLE dept;
hive> select * from dept;
OK
101 Engineering
102 HR
103 Finance
104 Facilities
Time taken: 0.061 seconds, Fetched: 4 row(s)
```

To execute map side joins, we don't need to do anything special; Hive automatically decides when to execute the map side join and convert it into a map side join:

```
> SELECT * FROM
>emp e  JOIN dept d
> ON (e.deptId = d.id);
Query ID = admin1_20151213081130_ad7a6c17-0c98-4e82-bd51-b9a6a9d687b7
Total jobs = 1
15/12/13 08:11:32 WARN util.NativeCodeLoader: Unable to load native-
hadoop library for your platform... using builtin-java classes where
applicable
Execution log at: /tmp/admin1/admin1_20151213081130_ad7a6c17-0c98-
4e82-bd51-b9a6a9d687b7.log
2015-12-13 08:11:33    Starting to launch local task to process map
join;     maximum memory = 477364224
2015-12-13 08:11:35    Dump the side-table for tag: 1 with group
count: 4 into file: file:/tmp/admin1/4406a7b1-64cf-44e3-968f-
857c4bd17bb7/hive_2015-12-13_08-11-30_543_7059067307272955469-1/-
local-10003/HashTable-Stage-3/MapJoin-mapfile51--.hashtable
2015-12-13 08:11:35    Uploaded 1 File to: file:/tmp/
admin1/4406a7b1-64cf-44e3-968f-857c4bd17bb7/hive_2015-12-13_08-11-
30_543_7059067307272955469-1/-local-10003/HashTable-Stage-3/MapJoin-
mapfile51--.hashtable (370 bytes)
2015-12-13 08:11:35    End of local task; Time Taken: 1.255 sec.
Execution completed successfully
MapredLocal task succeeded
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_1449972302581_0007, Tracking URL = http://
admin1:8088/proxy/application_1449972302581_0007/
Kill Command = /usr/local/hadoop/bin/hadoopjob  -kill
job_1449972302581_0007
Hadoop job information for Stage-3: number of mappers: 1; number of
reducers: 0
2015-12-13 08:11:40,923 Stage-3 map = 0%,  reduce = 0%
2015-12-13 08:11:46,201 Stage-3 map = 100%,  reduce = 0%, Cumulative
CPU 1.7 sec
MapReduce Total cumulative CPU time: 1 seconds 700 msec
Ended Job = job_1449972302581_0007
MapReduce Jobs Launched:
Stage-Stage-3: Map: 1   Cumulative CPU: 1.7 sec   HDFS Read: 6787 HDFS
Write: 160 SUCCESS
```

```
Total MapReduce CPU Time Spent: 1 seconds 700 msec
OK
1   Tanmay 3500.0 101    101    Engineering
2   Sneha  5599.0 101    101    Engineering
3   Avinash       6700.0 102    102    HR
4   Manisha       6400.0 103    103    Finance
5   Sakalya       3200.0 102    102    HR
Time taken: 17.748 seconds, Fetched: 5 row(s)
```

The preceding execution shows that Hive informs you when it has converted this join into a map side join.

How it works...

Map side joins are performed when we have one small table and a very big table join. While executing, Hive reads the small table completely into memory and writes this table to a hash table. This hash table is referenced by each mapper to join the bigger table. By getting one table in its memory, Hive has fewer disk I/O operations, and this means less time to get records. There are limits that are set in the Hive configurations that decide whether the table is small or not. The `hive.auto.convert.join.noconditionaltask` and `hive.auto.convert.join.noconditionaltask.size` configurations decide the auto conversion of joins and the size of the table that can fit into the memory.



You can read more about map side joins at `https://www.facebook.com/notes/facebook-engineering/join-optimization-in-apache-hive/470667928919`.

Performing context Ngram in Hive

Ngrams are sequences that are collected from specific sets of words and are based on their occurrence in a given text. N-grams are generally used to find the occurrence of certain words in a sequence, which helps in the calculation of sentiment analysis. Hive provides built-in support for Ngram calculations by providing a function. In this recipe, we will take a look at how to use this function in order to analyze text data.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am using Hive 1.2.1.

How to do it...

N-gram can be used to find the most frequently used word after a sequence of words in a give text dataset. To do this, let's first create a Hive table and load data into it.

Take a situation where we have data from Twitter where people are writing about their sentiments about chocolate. Let's assume that we have text data, as follows:

```
Chocolate is good
Chocolate is bad
Chocolate is harmful
Chocolate is sweet
Chocolate is good
Chocolate is bad
Chocolate is good
Chocolate is bad
Chocolate is good
Chocolate is bad
Chocolate is harmful
Chocolate is sweet Chocolate is harmful
Chocolate is sweet Chocolate is harmful
Chocolate is sweet Chocolate is harmful
Chocolate is sweet
Chocolate is good
Chocolate is bad
```

Let's load this data into the Hive table:

```
CREATE TABLE textData(
data STRING)
ROW FORMAT DELIMITED;

LOAD DATA LOCAL INPATH 'twitter.txt' INTO TABLE textData;
```

Now, we will write a Hive query to find out what the most frequent word after `Chocolate is` is:

```
SELECT context_ngrams(sentences(data),
array("Chocolate","is",null),10) FROM
textData;
```

On complete execution, we will get the following output:

```
[{
    "ngram": ["bad"],
    "estfrequency": 5.0
}, {
    "ngram": ["good"],
    "estfrequency": 5.0
}, {
    "ngram": ["harmful"],
    "estfrequency": 5.0
}, {
    "ngram": ["sweet"],
    "estfrequency": 5.0
}]
```

This way, we can easily find out frequent words after a set of strings. This also helps in performing natural language processing on unstructured text data.

How it works...

Context Ngrams looks for a specific sequence of words and keep counting them using the map reduce program. On complete execution, they sort out the output by the given frequency of words in order to know the top word/s being used after a set of strings.



To know more about Ngrams, you can visit `https://en.wikipedia.org/wiki/N-gram`.

Call Data Record Analytics using Hive

Call Data Records (**CDR**) are special types of records that are used in the telecom domain to keep track of calls made by individuals. We can use Hive to analyze these records in order to give special offers to customers.



You can read more about CDR at `https://en.wikipedia.org/wiki/Call_detail_record`.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am using Hive 1.2.1.

How to do it...

First of all, let's consider a situation where we have the following type of dataset with us. To analyze it, we first need to create a Hive table and load data into it:

```
CALLER_PHONE_NO|RECEIVER_PHONE_NUMBER|START_TIME|END_TIME|CALL_TYPE
11111|22222|2015-01-12 01:20:00|2015-01-12 01:30:00|VOICE
11111|22222|2015-02-12 01:35:00|2015-02-12 01:35:30|VOICE
11111|22222|2015-02-12 02:20:00|2015-02-12 02:20:00|SMS
33333|44444|2015-01-12 01:20:00|2015-01-12 01:30:00|VOICE
11111|33333|2015-05-12 04:02:00|2015-05-12 05:12:00|VOICE
22222|44444|2015-07-12 06:12:00|2015-07-12 06:12:00|SMS
22222|33333|2015-08-12 08:45:00|2015-08-12 08:50:00|VOICE
```

Let's create this Hive table and load data into it:

```
CREATE TABLE CDR(
caller_phone_no string,
receiver_phone_no string,
start_time timestamp,
end_time timestamp,
call_type string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH 'cdr.txt' INTO TABLE CDR;
```

Now that the data is loaded, we can execute the query to find out the duration of voice calls made by each subscriber in the given data:

```
SELECT caller_phone_no,
SUM(UNIX_TIMESTAMP(end_time) - UNIX_TIMESTAMP(start_time))
FROM cdr
WHERE call_type = 'VOICE'
GROUP BY caller_phone_no;
```

This will use the caller phone number and calculate the number of seconds it made calls for:

```
11111     4830
22222     300
33333     600
```

Next, we can execute queries to find out what the most frequently used medium of communication is. To know this, we will execute the following query:

```
SELECT call_type, count(*) as call_count FROM
CDR
GROUP BY call_type
ORDER  BYcall_count DESC;
```

On execution, we will get the following results:

```
VOICE     5
SMS       2
```

How it works...

The preceding section explains regular Hive queries, and we already know how they work.

Twitter sentiment analysis using Hive

Twitter is one of the most important data sources that helps you to know the sentiments behind various things. In this recipe, we will take a look at how to perform sentiment analysis using Hive on Twitter data.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am using Hive 1.2.1.

How to do it...

First of all, we need a dataset to perform this recipe. We will be using a dataset that can be found at `http://s3.amazonaws.com/hw-sandbox/tutorial13/SentimentFiles.zip`.

Next, we will unzip this data and upload it on HDFS. The zip contains three folders: the first for raw Twitter data, the second for a dictionary, and the third for a time zone:

```
hadoop fs -mkdir /data
hadoop fs -put tweets_raw /data
hadoop fs -put time_zone_map /data
hadoop fs -put dictionary /data
```

We use Hive's JSON SerDe jar to read the tweeter data, as shown here:

```
ADD JAR json-serde-1.1.9.9-Hive1.2-jar-with-dependencies.jar;
CREATE EXTERNAL TABLE tweets_raw (
id BIGINT,
created_at STRING,
source STRING,
favorited BOOLEAN,
retweet_count INT,
retweeted_status STRUCT<
text:STRING,
user_rt:STRUCT<screen_name:STRING,name:STRING>>,
entities STRUCT<
urls:ARRAY<STRUCT<expanded_url:STRING>>,
user_mentions:ARRAY<STRUCT<screen_name:STRING,name:STRING>>,
hashtags:ARRAY<STRUCT<text:STRING>>>,
text STRING,
user_struct STRUCT<
screen_name:STRING,
name:STRING,
friends_count:INT,
followers_count:INT,
statuses_count:INT,
verified:BOOLEAN,
utc_offset:STRING,
time_zone:STRING>,
in_reply_to_screen_name STRING
)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
WITH SERDEPROPERTIES ( "mapping.user_rt" = "user", "mapping.user_
struct" = "user" )
LOCATION '/data/tweets_raw';
```

This will create a Hive table and link data to it. Hive 1.2.1 puts certain constraints on reserve keywords such as user. So, we have to explicitly map these words to some other words.

Next, we load the dictionary and time zone tables. The dictionary will be used to determine whether a word is positive or negative:

```
CREATE EXTERNAL TABLE dictionary (
type string,
lengthint,
word string,
pos string,
stemmed string,
```

```
polarity string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/data/dictionary';

CREATE EXTERNAL TABLE time_zone_map (
time_zone string,
country string,
notes string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/data/time_zone_map';
```

We only select text column from the tweets_raw table and then clean it up:

```
CREATE VIEW tweets_simple AS
SELECT
id,
  cast ( from_unixtime( unix_timestamp(concat( '2013 ',
substring(created_at,5,15)), 'yyyy MMM ddhh:mm:ss')) as timestamp) ts,
text,
user_struct.time_zone
FROM tweets_raw;

CREATE VIEW tweets_clean AS
SELECT
id,
ts,
text,
m.country
 FROM tweets_simple t LEFT OUTER JOIN time_zone_map m ON t.time_zone =
m.time_zone;
```

We execute some queries in order to add the tweet ID and calculated sentiment:

```
create view l1 as select id, words from tweets_raw lateral view
explode(sentences(lower(text))) dummy as words;

create view l2 as select id, word from l1 lateral view explode( words
) dummy as word ;

create view l3 as select
id,
    l2.word,
```

```
cased.polarity
when  'negative' then -1
when 'positive' then 1
else 0 end as polarity
from l2 left outer join dictionary d on l2.word = d.word;

 -- give rights to hive user on tmpdir
 !hadoop fs -chmod -R 777 /tmp;


create table tweets_sentiment stored as RCFILE as select
id,
case
when sum( polarity ) > 0 then 'positive'
when sum( polarity ) < 0 then 'negative'
else 'neutral' end as sentiment
from l3 group by id;


CREATE TABLE tweets_senti
STORED AS RCFILE
AS
SELECT
t.*,
cases.sentiment
when 'positive' then 2
when 'neutral' then 1
when 'negative' then 0
end as sentiment
FROM tweets_clean t LEFT OUTER JOIN tweets_sentiment s on t.id = s.id;
```

As a result, you have now reached the tweet and its sentiment.

How it works

To perform the sentiment analysis, we first get only the text column from the raw table. Next, we explode each word in the text using the lateral view. Then, we join these words with the dictionary to know how they're polarized. Now that we know how each word's polarized, we perform 'group by' on the tweet ID to sum up the polarization. Once we have the final polarization, we check whether it is positive, negative, or neutral and save it against each tweet.

Implementing Change Data Capture using Hive

Change Data Capture or CDC is one the most painful areas in Data Warehousing. CDC captures the changes that occur in a table. A change could be in the form of new records getting added, updated, or getting deleted. In this recipe, we are going to take a look at how to perform CDC in Hive.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am using Hive 1.2.1.

How to do it

First of all, we need a data sample. Consider a simple employee table that has columns, such as the employee ID, name, and salary. Let's say we import this table from a source table in week 1, and after a week, we want to know about the changes that have taken place in the same table. Let's say we have a table, employee1, which was imported in week 1, and we have another table, which was imported in week 2. Week 2 being the latest week, we want to know the changes that have taken place. Here is some sample data:

Table – employee1

```
1,A,1000
2,B,2000
3,C,3000
4,D,2000
5,E,1000
6,F,3000
7,G,1000
8,H,3000
9,I,1000
10,J,2000
11,K,1000
12,L,1000
13,M,1000
14,N,3000
15,O,3000
```

```
16,P,1000
17,Q,1000
18,R,1000
19,S,2000
20,T,3000
```

Table – employee2

```
1,A,1000
2,B,2000
3,C,5000
4,D,2000
5,EE,1000
6,F,3000
7,G,1000
8,H,3000
10,J,2000
11,K,1000
12,L,1000
13,MM,1000
14,N,3000
15,O,3000
16,PQ,1000
17,Q,1000
19,S,2000
20,T,3000
21,S,2000
22,T,3000
23,S,2000
24,T,3000
```

Now, we want to know which row is updated, which row is deleted, and which rows are newly added. In order to do know this, we can execute the following query:

Create the employee1 table and load data into it:

```
CREATE TABLE employee1 (
id INT,
name STRING,
```

```
salary BIGINT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

Create the employee2 table and load into data:

```
LOAD DATA LOCAL INPATH 'emp1.txt' INTO TABLE employee1;

CREATE TABLE employee2 (
id INT,
name STRING,
salary BIGINT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH 'emp2.txt' INTO TABLE employee2;
```

Now, we write a query in order to know the CDC:

```
select
case when b1.cdc_codes = 'Updates' then b1.employee1s
 when b1.cdc_codes = 'NoChange' then b1.employee2s
 when b1.cdc_codes = 'New' then b1.employee2s
 when b1.cdc_codes = 'Deletes' then b1.employee1s
else 'Error' end as fin_cols
from (select case when e1.id = e2.id and concat(e1.name,e1.salary) =
concat(e2.name,e2.salary)     then  'NoChange'
when e1.id = e2.id and  concat(e1.name,e1.salary) <> concat(e2.name,e2.
salary) then  'Updates'
when e1.id is null then 'New'
when e2.id is null then 'Deletes'
else 'Error' end as cdc_codes,
concat(e1.id,',',e1.name,',',e1.salary) as employee1s,
concat(e2.id,',',e2.name,',',e2.salary) as employee2s
from employee1 as e1 full outer join employee2 as e2
on e1.id = e2.id) as b1
```

Once we execute the preceding query, we get the following data, which is the updated data:

```
1,A,1000
2,B,2000
3,C,3000
4,D,2000
5,E,1000
6,F,3000
7,G,1000
8,H,3000
9,I,1000
10,J,2000
11,K,1000
12,L,1000
13,M,1000
14,N,3000
15,O,3000
16,P,1000
17,Q,1000
18,R,1000
19,S,2000
20,T,3000
21,S,2000
22,T,3000
23,S,2000
24,T,3000
```

How it works

In order to find the changed data, we perform a full outer join of two tables. This will include the entries from both tables that match with their IDs. Next, we check the data that is present and then decide whether the record is new, updated, or deleted. This way, we can easily capture the changes that have taken place. This technique is very useful in cases where you have built your data warehouse using Hive.

Multiple table inserting using Hive

Hive allows you to write data to multiple tables or directories at a time. This is an optimized solution as a source table needs to be read only once, which helps reduce the time. In this recipe, we are going to take a look at how write data to multiple tables/directories in a single query.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Hive installed on it. Here, I am using Hive 1.2.1.

How to do it

Let's say we have an employee table with columns such as ID, name, and salary:

Table – employee

```
1,A,1000
2,B,2000
3,C,3000
4,D,2000
5,E,1000
6,F,3000
7,G,1000
8,H,3000
9,I,1000
10,J,2000
11,K,1000
12,L,1000
13,M,1000
14,N,3000
15,O,3000
16,P,1000
17,Q,1000
18,R,1000
19,S,2000
20,T,3000
```

Let's create the table and load the data into it:

```
CREATE TABLE employee (
id INT,
name STRING,
salary BIGINT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH 'emp.txt' INTO TABLE employee;
```

First, we need to create tables, and data from the employee table is inserted into these table:

```
CREATE TABLE emp1 LIKE employee;
CREATE TABLE emp2 LIKE employee;
```

Now, in order to insert data into two tables at a time, we can execute the following query:

```
FROM employee
INSERT INTO TABLE emp1
SELECT * WHERE id <= 10
INSERT INTO TABLE emp2
SELECT * WHERE id > 10 ;
```

As a result, you should see something like this:

```
hive> select * from emp1;
OK
1       A       1000
2       B       2000
3       C       3000
4       D       2000
5       E       1000
6       F       3000
7       G       1000
8       H       3000
9       I       1000
10      J       2000
```

```
Time taken: 0.161 seconds, Fetched: 10 row(s)
hive> select * from emp2;
OK
11      K       1000
12      L       1000
13      M       1000
14      N       3000
15      O       3000
16      P       1000
17      Q       1000
18      R       1000
19      S       2000
20      T       3000
```

We can also write data to the HDFS directory and table at a time, as shown here:

```
FROM employee1
INSERT INTO TABLE emp1
SELECT * WHERE id <= 10
INSERT OVERWRITE DIRECTORY '/data/emp1'
SELECT * WHERE id > 10 ;
```

This will write data to both the table and directory at one go.

How it works

The preceding queries read the data from the source table only once and write it in multiple places using multiple jobs. The benefit of doing it in this way is that you only read the data once, which optimizes the data read time.

6

Data Import/Export Using Sqoop and Flume

The chapter covers the following topics:

- Importing data from RDMBS to HDFS using Sqoop
- Exporting data from HDFS to RDBMS
- Using query operator in Sqoop import
- Importing data using Sqoop in compressed format
- Performing Atomic export using Sqoop
- Importing data into Hive tables using Sqoop
- Importing data into HDFS from Mainframes
- Incremental import using Sqoop
- Creating and executing Sqoop job
- Importing data from RDBMS to Hbase using Sqoop
- Importing Twitter data into HDFS using Flume
- Importing data from Kafka into HDFS using Flume
- Importing web logs data into HDFS using Flume

Introduction

In the previous chapter, we talked about advanced analytics options using Apache Hive. In this chapter, we are going to talk about two very important tools, Sqoop and Flume, which do not directly help us do analytics but help us get the data in and out of Hadoop. So, let's try to understand more about these technologies.

Importing data from RDMBS to HDFS using Sqoop

Most organizations use RDBMS databases as their primary storage for their data. To analyze the data, we need to first import that data on HDFS. Sqoop is a tool that helps us achieve this with ease, and with just a single command, we can import data into HDFS as required. In this recipe, we are going to see how to import data from MySQL to Hadoop using Sqoop.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Sqoop installed on it. Here I am using Sqoop 1.4.6. We would also need MySQL database to be present in the network. Installing Sqoop is easy by downloading Sqoop tar ball and setting it in the system path. As we are going to import data from MySQL, we would also need to download MySQL connector from `https://dev.mysql.com/downloads/connector/`. Based on your MySQL version, download the right connector jar and copy it into the `lib` directory of Sqoop installation.

How to do it...

Perform the following steps:

1. First we will log into MySQL and create a database called company and create a table in it called `employee`.

   The schema for the same is as follows:

```
mysql> desc employee;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| id    | int(11)     | YES  |     | NULL    |       |
| name  | varchar(20) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

2. Next we will add some records into it, so the MySQL employee table would look like as shown in the following:

```
mysql> select * from employee;
+------+--------+
| id   | name   |
```

```
+------+--------+
|   16 | john   |
|   17 | robert |
|   18 | andrew |
|   19 | katty  |
|   21 | tom    |
|   22 | tim    |
|   23 | james  |
|   24 | paul   |
|   27 | edward |
|   29 | alan   |
|   31 | kerry  |
|   34 | terri  |
+------+--------+
12 rows in set (0.00 sec)
```

3. Now we are going to execute Sqoop command to import this data into the HDFS directory called `/data/employee`.

```
$sqoop import --connect jdbc:mysql://localhost:3306/company
--username root --password password --table employee --target-dir
/data/employee -m 1
```

Once you execute the preceding command, a Map Reduce job will start and import the records and write them to HDFS. Here we need to provide the host on which MySQL is running, its credentials, which table to import, and in which directory on HDFS you wish to store the data. I have also given `m = 1`, which means using a single mapper to do sequential import. This is a mandatory setting if your table does not have a primary ID. The console output would look like the following:

```
16/02/28 09:29:24 INFO impl.YarnClientImpl: Submitted application
application_1456628547885_0004
16/02/28 09:29:24 INFO mapreduce.Job: The url to track the job:
http://admin1:8088/proxy/application_1456628547885_0004/
16/02/28 09:29:24 INFO mapreduce.Job: Running job:
job_1456628547885_0004
16/02/28 09:29:37 INFO mapreduce.Job: Job job_1456628547885_0004
running in uber mode : false
16/02/28 09:29:37 INFO mapreduce.Job:  map 0% reduce 0%
16/02/28 09:29:49 INFO mapreduce.Job:  map 100% reduce 0%
16/02/28 09:29:49 INFO mapreduce.Job: Job job_1456628547885_0004
completed successfully
16/02/28 09:29:49 INFO mapreduce.Job: Counters: 30
```

```
        File System Counters
            FILE: Number of bytes read=0
            FILE: Number of bytes written=132328
            FILE: Number of read operations=0
            FILE: Number of large read operations=0
            FILE: Number of write operations=0
            HDFS: Number of bytes read=87
            HDFS: Number of bytes written=104
            HDFS: Number of read operations=4
            HDFS: Number of large read operations=0
            HDFS: Number of write operations=2
        Job Counters
            Launched map tasks=1
            Other local map tasks=1
            Total time spent by all maps in occupied slots (ms)=8607
            Total time spent by all reduces in occupied slots (ms)=0
            Total time spent by all map tasks (ms)=8607
            Total vcore-seconds taken by all map tasks=8607
            Total megabyte-seconds taken by all map tasks=8813568
        Map-Reduce Framework
            Map input records=12
            Map output records=12
            Input split bytes=87
            Spilled Records=0
            Failed Shuffles=0
            Merged Map outputs=0
            GC time elapsed (ms)=57
            CPU time spent (ms)=1960
            Physical memory (bytes) snapshot=142516224
            Virtual memory (bytes) snapshot=407183360
            Total committed heap usage (bytes)=85983232
        File Input Format Counters
            Bytes Read=0
        File Output Format Counters
            Bytes Written=104
    16/02/28 09:29:49 INFO mapreduce.ImportJobBase: Transferred 104
    bytes in 29.694 seconds (3.5024 bytes/sec)
    16/02/28 09:29:49 INFO mapreduce.ImportJobBase: Retrieved 12
    records.
```

4. Now we can look at the HDFS directory to see the records getting imported:

```
hadoop fs -cat /data/employee/part-m-00000
    16,john
    17,robert
```

```
18,andrew
19,katty
21,tom
22,tim
23,james
24,paul
27,edward
29,alan
31,kerry
34,terri
```

By default, Sqoop uses a comma (,) as a column delimiter. We can also specify our own delimited data, as shown in the following:

```
sqoop import --connect jdbc:mysql://localhost:3306/company
--username root --password password --table employee --target-dir
/data/employee_pipe --fields-terminated-by '|' -m 1
```

```
$ hadoop fs -cat /data/employee_pipe/part-m-00000
16|john
17|robert
18|andrew
19|katty
21|tom
22|tim
23|james
24|paul
27|edward
29|alan
31|kerry
34|terri
```

In case you feel that providing a password in plain text is not safe, you may also provide that interactively:

```
sqoop import --connect jdbc:mysql://localhost:3306/company --username
root --table employee --target-dir /data/employee_pipe_1 --fields-
terminated-by '|' -m 1 -P
```

It will ask you to enter your password.

You can also get the password from the password file. For that, first you need to upload that file to HDFS and then point that in the `sqoop` command:

```
echo "password" > mysql.password
hadoop fs -put mysql.password /user/admin
hadoop fs -chown 400 /user/admin1/mysql.password
```

Now we can use this file in password:

```
sqoop import --connect jdbc:mysql://localhost:3306/company --username
root --table employee --target-dir /data/employee_pipe_3 --fields-
terminated-by '|' -m 1 --password-file /user/admin1/mysql.password
```

How it works...

When `sqoop` command is executed, it generates a Map Reduce job, rather a map only job, to import the data from the given RDBMS. By default, it uses four mappers to import the data from a given table. If the table has a primary key, then each mapper takes care of importing an equal number of records based on their primary ID. If the table does not have a primary ID, we need to use only a single mapper that will import the data sequentially. Based on the delimiter provided, the columns would be separated by the given characters.

Exporting data from HDFS to RDBMS

In the previous recipe, we spoke about how to import data from MySQL to HDFS. Now it is time to see how to export data from HDFS to RDBMS using Sqoop. Generally, this is required when you want to keep processed data in RDBMS to be used by some reporting tools.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Sqoop installed on it. Here I am using Sqoop 1.4.6. We would also need the MySQL database to be present in the network. Installing Sqoop is easy; by downloading Sqoop tar ball and setting it in system path. As we are going to import data from MySQL, we would also need to download MySQL connector. Based on your MySQL version, download the right connector jar and copy it into the `lib` directory of Sqoop installation.

How to do it...

In the previous recipe, we imported data from MySQL into HDFS. Now we are going to export this data back to some different MySQL table.

1. To do this, first we need to create the table in MySQL:

```
mysql> create table employee_export like employee;
Query OK, 0 rows affected (0.02 sec)
```

2. Now we will execute the `sqoop export` command to fetch the data fromthe `/data/employee` directory and put it into the `employee_export` table we just created in the previous step:

```
sqoop export --connect jdbc:mysql://localhost/company --username
root --password password --table employee_export --export-dir /
data/employee
```

   The console output for this will look like the following:

```
16/02/28 09:34:23 INFO mapreduce.Job: Running job:
job_1456628547885_0005
16/02/28 09:34:36 INFO mapreduce.Job: Job job_1456628547885_0005
running in uber mode : false
16/02/28 09:34:36 INFO mapreduce.Job:  map 0% reduce 0%
16/02/28 09:35:04 INFO mapreduce.Job:  map 100% reduce 0%
16/02/28 09:35:09 INFO mapreduce.Job: Job job_1456628547885_0005
completed successfully
16/02/28 09:35:10 INFO mapreduce.Job: Counters: 30
    File System Counters
        FILE: Number of bytes read=0
        FILE: Number of bytes written=528692
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=788
        HDFS: Number of bytes written=0
        HDFS: Number of read operations=16
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=0
    Job Counters
        Launched map tasks=4
        Data-local map tasks=4
        Total time spent by all maps in occupied slots (ms)=109699
        Total time spent by all reduces in occupied slots (ms)=0
        Total time spent by all map tasks (ms)=109699
        Total vcore-seconds taken by all map tasks=109699
```

```
            Total megabyte-seconds taken by all map tasks=112331776
        Map-Reduce Framework
            Map input records=12
            Map output records=12
            Input split bytes=516
            Spilled Records=0
            Failed Shuffles=0
            Merged Map outputs=0
            GC time elapsed (ms)=838
            CPU time spent (ms)=8110
            Physical memory (bytes) snapshot=507035648
            Virtual memory (bytes) snapshot=1635950592
            Total committed heap usage (bytes)=317456384
        File Input Format Counters
            Bytes Read=0
        File Output Format Counters
            Bytes Written=0
16/02/28 09:35:10 INFO mapreduce.ExportJobBase: Transferred 788
bytes in 51.1218 seconds (15.4142 bytes/sec)
16/02/28 09:35:10 INFO mapreduce.ExportJobBase: Exported 12
records.
```

3. Once the command execution is completed, you can go to `mysql` prompt and check if the data import is proper:

```
mysql> select * from employee_export;
+------+--------+
| id   | name   |
+------+--------+
|   24 | paul   |
|   27 | edward |
|   29 | alan   |
|   19 | katty  |
|   21 | tom    |
|   22 | tim    |
|   23 | james  |
|   31 | kerry  |
|   34 | terri  |
|   16 | john   |
|   17 | robert |
|   18 | andrew |
+------+--------+
12 rows in set (0.01 sec)
```

If, in your HDFS files, the column delimiter is other than a comma (`,`) then we have to specify the delimiter as shown in the following:

```
sqoop export --connect jdbc:mysql://localhost/company --username
root --password password --table employee_export --export-dir /
data/employee  --fields-terminated-by '|'
```

This will read the `|` delimited data from the given HDFS directory and save it into the `employee_export` table.

Here you can protect the plain text password being mentioned in the command directly by using the following options.

First, we can choose the – + *P* option, which would ask for the password in interactive mode:

```
sqoop export --connect jdbc:mysql://localhost/company --username
root --table employee_export --export-dir /data/employee -P
```

Second, you can also create a password file as explained in the previous recipe:

```
sqoop export --connect jdbc:mysql://localhost/company --username
root --table employee_export --export-dir /data/employee
--password-file /user/admin1/mysql.password
```

How it works...

`sqoop export` also creates a Map Reduce job for exporting data to RDBMS. It uses the provided credentials to insert records into a given table. Here it is important to match the data type in files with the data type in table. We should also make sure that the destination table exists before starting the `sqoop export` job, otherwise it would fail. In case the column delimiter is other than comma, then make sure you mention that in the `sqoop export` command to avoid wrong entries.

Using query operator in Sqoop import

In previous examples, we saw how we import/export complete table. Sometimes you may want to only use selective data, so to achieve this, Sqoop provides a query operation in which we can write SQL query as we need to import data into HDFS.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Sqoop installed on it. Here I am using Sqoop 1.4.6. We would also need a MySQL database to be present in the network. Installing Sqoop is easy by downloading Sqoop tar ball and setting it in system path. As we are going to import data from MySQL, we would also need to download MySQL connector. Based on your MySQL version, download the right connector jar and copy it into the `lib` directory of the Sqoop installation.

How to do it...

1. Now we, again, log in to MySQL and take a look at the employee table. Let's say we want to import only those employees whose IDs are greater than 20. Then we can either use query or `use where` condition as shown in the following:

```
sqoop import --connect jdbc:mysql://localhost:3306/company
--username root --password password --target-dir /data/employee_
query_1 --query 'SELECT * FROM employee WHERE $CONDITIONS AND id>
20' -m 1
```

2. On successful execution, you should see only those records on HDFS whose IDs are greater than 20:

```
$hadoop fs -cat /data/employee_query_1/part-m-00000
    21,tom
    22,tim
    23,james
    24,paul
    27,edward
    29,alan
    31,kerry
    34,terri
```

You can also use query options to import data from multiple tables together. You can also import joined records using query options:

```
sqoop import --connect jdbc:mysql://localhost:3306/company --username
root --password password --target-dir /data/a_b_join --query SELECT
a.*, b.* FROM a JOIN b on (a.id == b.id) WHERE $CONDITIONS' -m 1
```

The preceding `sqoop` job would join two tables `a` and `b` on column ID and would be imported into the `/data/a_b_join` folder.

How it works...

With query option, `Sqoop import` works exactly the same the way it works for other cases. The only difference is that we can write a free-form query with it.

Importing data using Sqoop in compressed format

In this recipe, we are going to talk about a very important feature of Sqoop that allows us to compress the imported data. We can also choose the compression algorithm.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Sqoop installed on it. Here I am using Sqoop 1.4.6. We would also need a MySQL database to be present in the network. Installing Sqoop is easy; by downloading Sqoop tar ball and setting it in the system path. As we are going to import data from MySQL, we would also need to download MySQL connector. Based on your MySQL version, download the right connector jar and copy it into the `lib` directory of the Sqoop installation.

How to do it...

1. Sqoop import allows us to import the data in a compressed format. We don't need to install anything else apart from default installation. The following query will import the employee data into a compressed format and will store in a given directory:

```
sqoop import \
--connect jdbc:mysql://localhost/company \
--username root \
--password password \
--table employee \
--compress \
--target-dir /data/employee_compressed \
--m 1
```

2. Once the job execution is complete, we can take a look at the HDFS directory:

```
hadoop fs -ls /data/employee_compressed
    Found 2 items
    -rw-r--r--   1 admin1 supergroup          0 2015-12-29
    14:41 /data/employee_compressed/_SUCCESS
    -rw-r--r--   1 admin1 supergroup        106 2015-12-29
    14:41 /data/employee_compressed/part-m-00000.gz
```

   By default, the data is compressed in gun `zip` format. In case you want any other compression format, then you can specify the same as shown in the following:

```
sqoop import \
--connect jdbc:mysql://localhost/company \
```

```
--username root \
--password password \
--table employee \
--compress \
--target-dir /data/employee_compressed_bz2 \
--m 1 \
--compression-codec org.apache.hadoop.io.compress.BZip2Codec
```

3. On completion, you can take a look at the target directory:

```
hadoop fs -ls /data/employee_compressed_bz2

    Found 2 items

    -rw-r--r--   1 admin1 supergroup          0 2015-12-29
    14:45 /data/employee_compressed_bz2/_SUCCESS

    -rw-r--r--   1 admin1 supergroup        113 2015-12-29
    14:45 /data/employee_compressed_bz2/part-m-00000.bz2
```

Similarly, you can choose which compression algorithm you need and specify it in the `sqoop import` command.

How it works...

As Sqoop has come from the Hadoop family, it inherits support for various compression algorithms. It's very handy to import compressed data and save it on HDFS for efficient space utilization. We have to be very careful while choosing the compression algorithm as some algorithms do not allow splitting files. LZOP is one algorithm that allows file splitting; you can think about using it.

Performing Atomic export using Sqoop

We have learned in basic database concepts about atomicity, which means doing a complete job or doing nothing. Similarly, if you are exporting data using Sqoop to a table that is very important from the application's point of view and you want make sure that Sqoop should export all data present in HDFS or do nothing, this recipe will help. In this recipe, we are going to see how to ensure atomicity of data export.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Sqoop installed on it. Here I am using Sqoop 1.4.6. We would also need a MySQL database to be present in the network. Installing Sqoop is easy; by downloading Sqoop tar ball and setting it in the system path. As we are going to import data from MySQL, we would also need to download MySQL connector. Based on your MySQL version, download the right connector jar and copy it into the `lib` directory of the Sqoop installation.

How to do it...

1. To perform the atomic export, first we need to create a table that can be used as a staging table:

```
mysql> create table staging_employee like employee;
Query OK, 0 rows affected (0.01 sec)
```

2. Then you can execute the `sqoop export` command specifying the destination table as well as the staging table:

```
sqoop export \
--connect jdbc:mysql://localhost/company \
--username root \
--password password \
--table atomic_employee \
--export-dir /data/employee \
--staging-table staging_employee
```

3. Last couple of lines of the job log will inform you that Sqoop is migrating data from the staging table to the destination table:

```
15/12/29 15:11:04 INFO mapreduce.ExportJobBase: Transferred 788
bytes in 21.6322 seconds (36.4272 bytes/sec)
15/12/29 15:11:04 INFO mapreduce.ExportJobBase: Exported 12
records.
15/12/29 15:11:04 INFO mapreduce.ExportJobBase: Starting to
migrate data from staging table to destination.
15/12/29 15:11:04 INFO manager.SqlManager: Migrated 12 records
from `staging_employee` to `atomic_employee`
```

Sqoop makes sure that if any parallel executing jobs fail, it will not copy anything to the destination table.

How it works...

During atomic export, Sqoop first writes data to the staging table. If any of the parallel executing jobs fail, Sqoop will not copy data to the destination table. This way we make sure that no half writes would go to the destination table. The only disadvantage of using the staging table is we would need to have that much space available for us in our database. Generally, it is recommended to use the staging table while exporting data to production tables.

Importing data into Hive tables using Sqoop

Till now we have seen how to import data into HDFS folders. Now it's time to understand how to import data directly into Hive table.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Sqoop installed on it. Here I am using Sqoop 1.4.6. We would also need a MySQL database to be present in the network. Installing Sqoop is easy; by downloading Sqoop tar ball and setting it in the system path. As we are going to import data from MySQL, we would also need to download MySQL connector. Based on your MySQL version, download the right connector jar and copy it into the `lib` directory of Sqoop installation.

How to do it...

Sqoop provides us the facility to directly import data into Hive table. This saves our time in creating Hive tables, specifying matching schema, loading data into HDFS, and then creating external Hive table. We can do this in a couple of commands:

1. The following first command creates a table in Hive as per the schema of the source table, and the second command imports the actual data from the source RDBMS table and stores it into Hive table:

```
sqoop create-hive-table --connect jdbc:mysql://localhost/company \
--table employee --hive-table employee_sqoop \
--username root \
--password password \
```

2. The following creates a table in Hive as per the schema of the source table; we can go to Hive and check if the table is created:

```
hive> desc employee_sqoop;
OK
id                      int
name                    string
Time taken: 0.114 seconds, Fetched: 2 row(s)
```

3. Next, we execute the following command to import the data into the newly created Hive table:

```
sqoop import --hive-import --connect jdbc:mysql://localhost/
company \
--table employee --hive-table employee_sqoop \
```

```
--username root \
--password password \
--m 1
```

4. We can then go to the Hive prompt and see if the data is present:

```
hive> select * from employee_sqoop;
    OK
    16 john
    17 robert
    18 andrew
    19 katty
    21 tom
    22 tim
    23 james
    24 paul
    27 edward
    29 alan
    31 kerry
    34 terri
    Time taken: 1.297 seconds, Fetched: 12 row(s)
```

How it works...

Sqoop Hive import works similar to regular import; the only extra thing it does is copying data under the Hive warehouse directory. The `create-hive-table` command reads the source table schema first and then creates Hive table accordingly.

Importing data into HDFS from Mainframes

Mainframes is one of the most used datasets in financial domain for quite a long time. Sqoop supports importing datasets from Mainframes into HDFS. This is an important recipe for those who are looking to migrate from Mainframes to Hadoop base systems.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Sqoop installed on it. Here I am using Sqoop 1.4.6. We would also need a MySQL database to be present in the network. Installing Sqoop is easy by downloading Sqoop tar ball and setting it in the system path.

How to do it...

Sqoop provides a tool called `import-mainframe`, using which, we can connect to a certain mainframe host and select the dataset to be imported. The following command connects to a mainframe host with the provided credentials and then imports the mentioned dataset into the HDFS target directory:

```
 sqoop import-mainframe --connnect <mainframes-host> \
--dataset <dataset-to-be-imported> \
--target-dir /dest \
 --username foo\
 --password bar
```

You can also choose in which format you wish to import the data, if it needs to be compressed and if you wish to have different field delimiter than the default.

How it works...

Sqoop mainframes imports sequential data into a partitioned dataset. It has certain limitations like records in a dataset should be character data only. Once imported data will be saved as a single text record, and it would still be in its early stages. It would get more mature as time passes.

Incremental import using Sqoop

In an enterprise world, the data gets increased every single day, hour, minute, and second. It's important to import data in an incremental way to do our analysis on up-to-date data. In this recipe, we are going to learn how to import data incrementally.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Sqoop installed on it. Here I am using Sqoop 1.4.6. We would also need a MySQL database to be present in the network. Installing Sqoop is easy by downloading Sqoop tar ball and setting it in the system path. As we are going to import data from MySQL, we would also need to download MySQL connector. Based on your MySQL version, download the right connector jar and copy it into the `lib` directory of the Sqoop installation.

How to do it...

1. To learn about incremental `import`, we will create a new table in MySQL called `newemployee` and add some records in it:

```
create table newemployee(id INT NOT NULL AUTO_INCREMENT,
 name varchar(20) ,
 salary float,
 PRIMARY KEY ( id ) );
insert into newemployee (name, salary) values ("Tanmay", 2000.00);
insert into newemployee (name, salary) values ("Sneha", 5000.00);
insert into newemployee (name, salary) values ("Avinash",
8000.00);
insert into newemployee (name, salary) values ("Sakalya",
6000.00);
insert into newemployee (name, salary) values ("Manisha",
4000.00);
```

   So, the table would like the following:

```
mysql> select * from newemployee;
+----+---------+--------+
| id | name    | salary |
+----+---------+--------+
|  1 | Tanmay  |   2000 |
|  2 | Sneha   |   5000 |
|  3 | Avinash |   8000 |
|  4 | Sakalya |   6000 |
|  5 | Manisha |   4000 |
+----+---------+--------+
5 rows in set (0.00 sec)
```

2. First, we import data in a regular manner:

```
sqoop import \
--connect jdbc:mysql://localhost/company \
--username root \
--password password \
--table newemployee \
--target-dir /data/newemployee \
--m 1
```

Now let's assume that a few more rows got added into our `newemployee` table:

```
insert into newemployee (name, salary) values
("Kapila",4400.00);
insert into newemployee (name, salary) values ("Vinit",
6500.00);
```

```
mysql> select * from newemployee;
+----+---------+--------+
| id | name    | salary |
+----+---------+--------+
|  1 | Tanmay  |   2000 |
|  2 | Sneha   |   5000 |
|  3 | Avinash |   8000 |
|  4 | Sakalya |   6000 |
|  5 | Manisha |   4000 |
|  6 | Kapila  |   4400 |
|  7 | Vinit   |   6500 |
+----+---------+--------+
7 rows in set (0.00 sec)
```

3. Now to import only new rows, we can use the incremental option in `sqoop import`. Following is the command we need to use:

```
sqoop import \
--connect jdbc:mysql://localhost/company \
--username root \
--password password \
--table newemployee \
--target-dir /data/newemployee \
--incremental append \
--check-column id \
--last-value 5 \
--m 1
```

4. Here we need to specify which column Sqoop should look at and what its last value is. Append mode is helpful when you have incremental IDs and you wish to import only those rows.

   Once you execute the job, you will see that only two new rows get imported into HDFS.

5. In case you wish to import updated rows using Sqoop, you can use `--incremental lastmodified`. To use this, first we need to have a column where we can save the last modified date and make sure that on every update, you update the last modified data column as well. Then you can execute the following command to get only updated records:

```
sqoop import \
--connect jdbc:mysql://localhost/company \
--username root \
--password password \
--table newemployee \
--target-dir /data/newemployee \
--incremental lastmodified \
--check-column lastmod \
--last-value 01-01-1970 \
--m 1
```

Here `lastmod` is my column in MySQL table where I am storing the `lastmodified` date.

How it works...

Sqoop incremental import modifies the underlying Map Reduce job based on the conditions given in the command. It filters out values for the columns specified in the command. This is a very useful tool where the database gets updated periodically.

Creating and executing Sqoop job

Sqoop provides you the facility to store certain jobs that can be used easily. In this recipe, we are going to see how to create a job and execute the same.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Sqoop installed on it. Here I am using Sqoop 1.4.6. We would also need a MySQL database to be present in the network. Installing Sqoop is easy by downloading Sqoop tar ball and setting it in system path. As we are going to import data from MySQL, we would also need to download MySQL connector. Based on your MySQL version, download the right connector jar and copy it into the `lib` directory of Sqoop installation.

How to do it...

1. We will create a `sqoop job` for incremental import that we performed in the last recipe. Following is the command to create a `sqoop` job:

```
sqoop job \
--create incremental_import_job \
-- import \
--connect jdbc:mysql://localhost/company \
--username root \
--password password \
--table newemployee \
--target-dir /data/newemployee \
--incremental append \
--check-column id \
--last-value 7 \
--m 1
```

2. We can see the list of jobs as shown in the following:

```
$ sqoop job --list
15/12/30 12:00:38 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6
Available jobs:
  incremental_import_job
```

We can also view details of a particular job:

```
$ sqoop job --show incremental_import_job
15/12/30 12:05:51 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6
Job: incremental_import_job
Tool: import
Options:
----------------------------
verbose = false
incremental.last.value = 7
db.connect.string = jdbc:mysql://localhost/company
codegen.output.delimiters.escape = 0
codegen.output.delimiters.enclose.required = false
codegen.input.delimiters.field = 0
hbase.create.table = false
db.require.password = true
hdfs.append.dir = true
db.table = newemployee
codegen.input.delimiters.escape = 0
import.fetch.size = null
accumulo.create.table = false
```

```
codegen.input.delimiters.enclose.required = false
db.username = root
reset.onemapper = false
codegen.output.delimiters.record = 10
import.max.inline.lob.size = 16777216
hbase.bulk.load.enabled = false
hcatalog.create.table = false
db.clear.staging.table = false
incremental.col = id
codegen.input.delimiters.record = 0
enable.compression = false
hive.overwrite.table = false
hive.import = false
codegen.input.delimiters.enclose = 0
accumulo.batch.size = 10240000
hive.drop.delims = false
codegen.output.delimiters.enclose = 0
hdfs.delete-target.dir = false
codegen.output.dir = .
codegen.auto.compile.dir = true
relaxed.isolation = false
mapreduce.num.mappers = 1
accumulo.max.latency = 5000
import.direct.split.size = 0
codegen.output.delimiters.field = 44
export.new.update = UpdateOnly
incremental.mode = AppendRows
hdfs.file.format = TextFile
codegen.compile.dir = /tmp/sqoop-admin1/compile/
b125ff516f7a0143479b9582373230e7
direct.import = false
hdfs.target.dir = /data/newemployee
hive.fail.table.exists = false
db.batch = false
```

3. Next, we can execute the job using the following command:

```
sqoop job --exec incremental_import_job
```

This will execute the job we had specified. Now you can add more records into the MySQL table and execute this job. It will automatically keep on updating the lastvalue parameter.

How it works...

Sqoop saves the configuration provided in the job, and the next time you execute, it will remember the parameters you provided. It will also make sure to update the last-value attribute in case of incremental imports. It's a very handy tool for taking care of daily imports with ease.

Importing data from RDBMS to Hbase using Sqoop

These days, lots of people want to make use of the power of NoSQL databases. In order to do so, they need to migrate their existing application from RDBMS to NoSQL databases like Hbase. In this recipe, we are going to learn how to import data from MySQL to Hbase.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Sqoop installed on it. Here I am using Sqoop 1.4.6. We would also need a MySQL database to be present in the network. Installing Sqoop is easy; by downloading Sqoop tar ball and setting it in the system path. As we are going to import data from MySQL, we would also need to download MySQL connector. Based on your MySQL version, download the right connector jar and copy it into the `lib` directory of the Sqoop installation.

How to do it...

First of we need to create a table in MySQL and insert some records in it:

```
mysql> desc employee;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| id    | int(11)     | YES  |     | NULL    |       |
| name  | varchar(20) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
mysql> select * from employee;
+------+--------+
| id   | name   |
+------+--------+
```

```
|   16 | john   |
|   17 | robert |
|   18 | andrew |
|   19 | katty  |
|   21 | tom    |
|   22 | tim    |
|   23 | james  |
|   24 | paul   |
|   27 | edward |
|   29 | alan   |
|   31 | kerry  |
|   34 | terri  |
+------+--------+
12 rows in set (0.02 sec)
```

Next we need to create a table in Hbase:

```
hbase>create 'emp', 'personal data', 'professional data'
```

Now we can write a Sqoop job in order to fetch data from this MySQL table as shown in the following:

```
sqoop import \
    --connect jdbc:mysql://localhost:3306/company \
    --username root -P \
    --table employee \
    --columns "id,name" \
    --hbase-table emp \
    --column-family 'personal data' \
    --hbase-row-key id -m 1
```

In the preceding command, we are specifying the source table, destination table, columns to be imported, and which column should be used as the key.

Executing the preceding command will start a Map Reduce job and you will see records getting imported into Hbase:

```
16/02/28 08:44:53 INFO mapreduce.Job: The url to track the job: http://
admin1:8088/proxy/application_1456628547885_0002/
16/02/28 08:44:53 INFO mapreduce.Job: Running job: job_1456628547885_0002
16/02/28 08:45:08 INFO mapreduce.Job: Job job_1456628547885_0002 running
in uber mode : false
16/02/28 08:45:08 INFO mapreduce.Job:  map 0% reduce 0%
```

```
16/02/28 08:45:23 INFO mapreduce.Job:  map 100% reduce 0%
16/02/28 08:45:24 INFO mapreduce.Job: Job job_1456628547885_0002
completed successfully
16/02/28 08:45:24 INFO mapreduce.Job: Counters: 30
    File System Counters
        FILE: Number of bytes read=0
        FILE: Number of bytes written=159042
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=87
        HDFS: Number of bytes written=0
        HDFS: Number of read operations=1
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=0
    ob Counters
        Launched map tasks=1
        Other local map tasks=1
        Total time spent by all maps in occupied slots (ms)=11945
        Total time spent by all reduces in occupied slots (ms)=0
        Total time spent by all map tasks (ms)=11945
        Total vcore-seconds taken by all map tasks=11945
        Total megabyte-seconds taken by all map tasks=12231680
    Map-Reduce Framework
        Map input records=12
        Map output records=12
        Input split bytes=87
        Spilled Records=0
        Failed Shuffles=0
        Merged Map outputs=0
        GC time elapsed (ms)=194
        CPU time spent (ms)=4060
        Physical memory (bytes) snapshot=160456704
        Virtual memory (bytes) snapshot=418877440
        Total committed heap usage (bytes)=81002496
    File Input Format Counters
```

```
        Bytes Read=0
    File Output Format Counters
        Bytes Written=0
16/02/28 08:45:24 INFO mapreduce.ImportJobBase: Transferred 0 bytes in
38.7996 seconds (0 bytes/sec)
16/02/28 08:45:24 INFO mapreduce.ImportJobBase: Retrieved 12 records.
```

Now we can check in the Hbase table as shown in the following:

```
hbase(main):006:0> scan 'emp'
ROW                   COLUMN+CELL
 16                   column=personal data:name, timestamp=1456629321444,
value=
                      John
 17                   column=personal data:name, timestamp=1456629321444,
value=
                      Robert
 18                   column=personal data:name, timestamp=1456629321444,
value=
                      Andrew
 19                   column=personal data:name, timestamp=1456629321444,
value=
                      Katty
 21                   column=personal data:name, timestamp=1456629321444,
value=
                      Tom
 22                   column=personal data:name, timestamp=1456629321444,
value=
                      Tim
 23                   column=personal data:name, timestamp=1456629321444,
value=
                      James
 24                   column=personal data:name, timestamp=1456629321444,
value=
                      Paul
 27                   column=personal data:name, timestamp=1456629321444,
value=
                      Edward
 29                   column=personal data:name, timestamp=1456629321444,
value=
                      Alan
 31                   column=personal data:name, timestamp=1456629321444,
value=
                      Kerry
 34                   column=personal data:name, timestamp=1456629321444,
value=
                      Terri
12 row(s) in 0.2990 seconds
```

How it works...

Sqoop reads the command provided and makes a JDBC connection to MySQL. It verifies the existence of source and destination tables and then starts the import. Hbase being a columnar store database, means it writes row data and a column definition in each record.

Importing Twitter data into HDFS using Flume

Flume is another tool that helps us import data from various other sources into HDFS. In this recipe, we are going to see how to import Twitter data using Flume. Twitter data is a great source of information provided by individuals. This data can be used to do sentiment analytics of certain products, persons, companies, and so on.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Flume installed on it. Here I am using Flume 1.6.

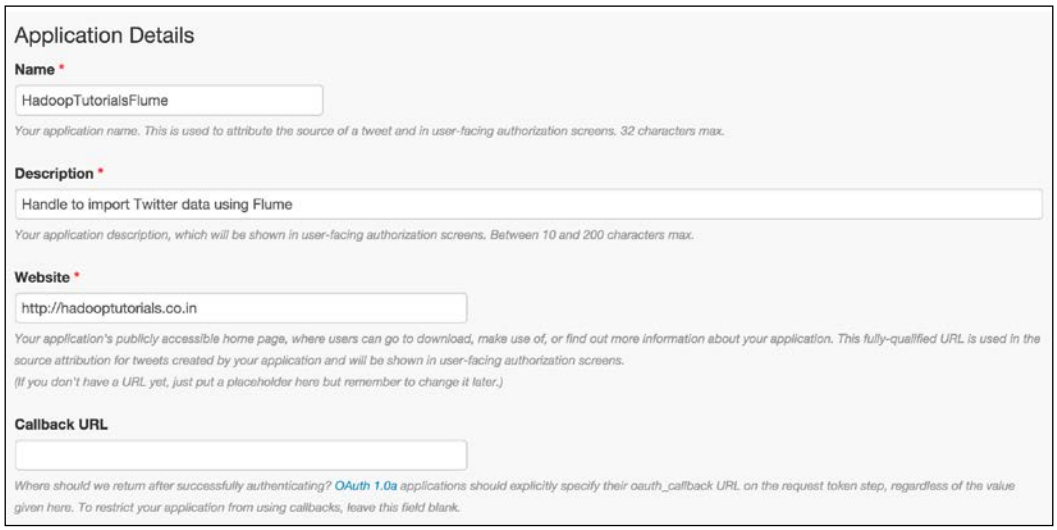
How to do it...

To import data using Flume, first of all we need to have a Twitter account and we need to generate credentials. These credentials would be used by Flume agent to import the data. Flume by default supports sources to import data from Twitter, so there is no need to do anything else other than having an account and generating credentials.

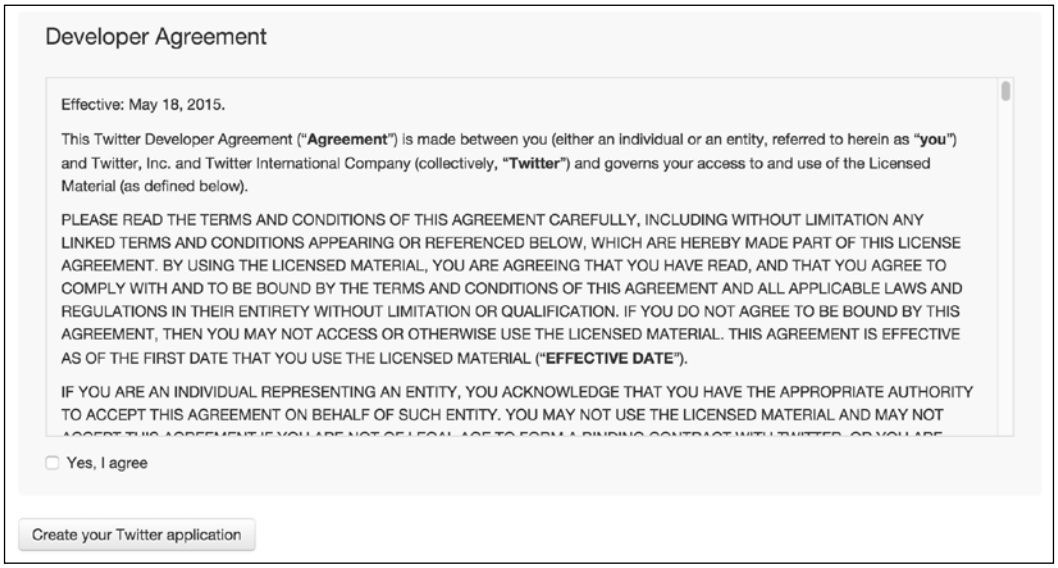
Following is the step-by-step process to generate Twitter authorization tokens:

1. Log in to `https://apps.twitter.com/`.

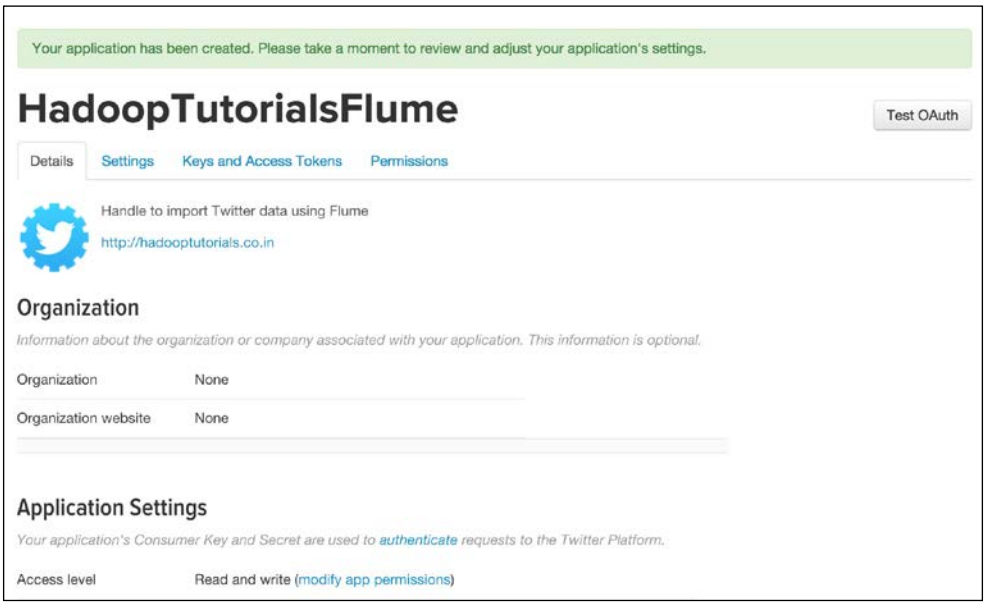
2. Click on **Create New Application**. Fill in the required form as shown in the following screenshot:



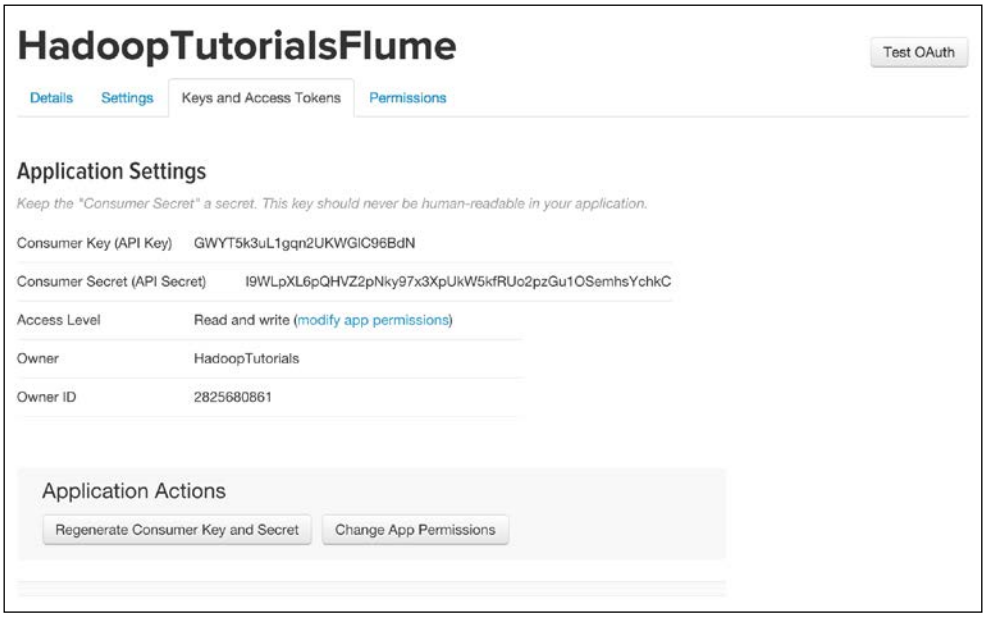
3. Read and accept the **Developer Agreement** if you think it's good for you:



4. Click on **Create your Twitter application** to save your application. Now we can use this handle to import data from Twitter:



5. Click on **Keys and Access Tokens**; there you will find the application specific **Consumer Key** and **Consumer Secret** Key. Keep a copy of it as we will need it in the next steps.



6. On scrolling down, you will also see an option to generate an access token, so do that and keep a copy of it for future use.
7. Now edit `$FLUME_HOME/conf/flume.conf` to copy the exact values of the access key, secret key, access token, and access token secret. Save the changes made to the file.

   The following is how your `flume.conf` should look like:

```
#### TwitterAgent for collecting Twitter data to Hadoop HDFS #####

TwitterAgent.sources = Twitter
TwitterAgent.channels = FileChannel
TwitterAgent.sinks = HDFS

TwitterAgent.sources.Twitter.type = org.apache.flume.source.
twitter.TwitterSource
TwitterAgent.sources.Twitter.channels = FileChannel
TwitterAgent.sources.Twitter.consumerKey = <consumer-key>
TwitterAgent.sources.Twitter.consumerSecret = <consume-secret>
TwitterAgent.sources.Twitter.accessToken = <access-token>
TwitterAgent.sources.Twitter.accessTokenSecret = <access-token-
secret>
TwitterAgent.sources.Twitter.maxBatchSize = 50000
TwitterAgent.sources.Twitter.maxBatchDurationMillis = 100000

#TwitterAgent.sources.Twitter.keywords = Apache, Hadoop,
Mapreduce, hadooptutorial, Hive, Hbase, MySql

TwitterAgent.sinks.HDFS.channel = FileChannel
TwitterAgent.sinks.HDFS.type = hdfs
TwitterAgent.sinks.HDFS.hdfs.path = hdfs://localhost:9000/user/
flume/tweets/
TwitterAgent.sinks.HDFS.hdfs.fileType = DataStream
TwitterAgent.sinks.HDFS.hdfs.writeFormat = Text
TwitterAgent.sinks.HDFS.hdfs.batchSize = 200000
TwitterAgent.sinks.HDFS.hdfs.rollSize = 0
TwitterAgent.sinks.HDFS.hdfs.rollCount = 2000000

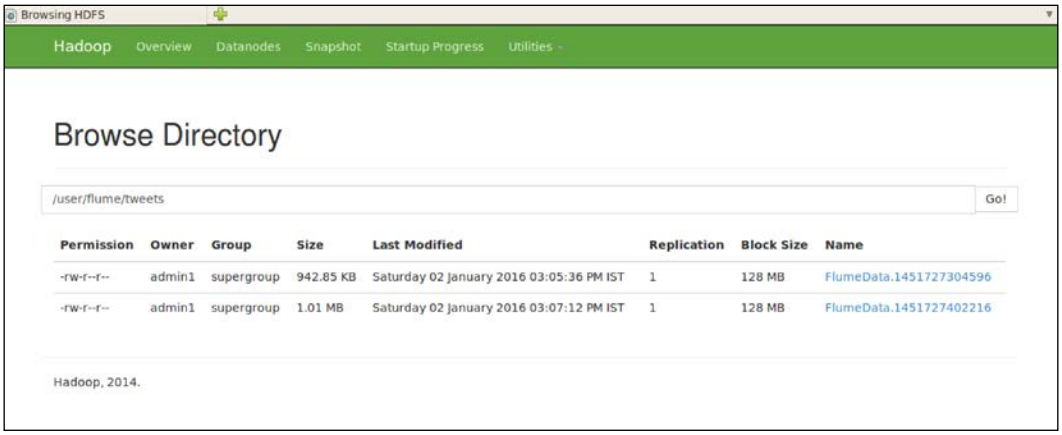
TwitterAgent.channels.FileChannel.type = file
TwitterAgent.channels.FileChannel.checkpointDir = /var/log/flume/
checkpoint/
TwitterAgent.channels.FileChannel.dataDirs = /var/log/flume/data/
```

8. Now it's time to execute Flume agent to start fetching data from Twitter and save it on HDFS:

```
/usr/local/flume/bin/flume-ng agent -n TwitterAgent -c /usr/local/
flume/conf -f /usr/local/flume/conf/flume.conf
```

   Execute the preceding command to fetch data from Twitter and save it on HDFS.

9. Now you can take a look at the HDFS path `/user/flume/tweets` to see the actual Twitter data.



How it works

Flume has built-in support for the Twitter source. It uses Twitter exposed APIs to download the data and store it on HDFS. It internally browses through the tweets available for your user and downloads them. It keeps on creating a new file based on time. You can also download only those tweets with certain keywords by uncommenting the following line from `flume.conf` and rerunning the Flume agent:

```
TwitterAgent.sources.Twitter.keywords = Apache, Hadoop, Mapreduce,
hadooptutorial, Hive, Hbase, MySql
```



Please make a note that the application you have created has all of the same rights as a user. So don't share the keys with anyone else.

Importing data from Kafka into HDFS using Flume

Kafka is one the most popular message queue systems being used these days. We can listen to Kafka topics and put the message data directly into HDFS using Flume. The latest Flume version supports importing data from Kafka easily. In this recipe, we are going to learn how to import Kafka messages to HDFS.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Flume installed on it. Here I am using Flume 1.6. We also need Kafka installed and running on one of the machines. I am using kafka_2.10-0.9.0.0.

How to do it...

1. To import the data from Kafka, first you need to have Kafka running on your machine. The following command starts Kafka and Zookeeper:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
bin/kafka-server-start.sh config/server.properties
```

2. Next I create a topic called `weblogs` which we will be listening to:

```
bin/kafka-topics.sh --create --zookeeper localhost:2181
--replication-factor 1 --partitions 1 --topic weblogs
```

3. Next we will start a producer which we will write to weblogs topic:

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
weblogs
```

4. Now that all Kafka-related infra is set up, we need to also create Flume configuration, which will be listening to the weblogs topic and writing it to HDFS:

```
# Flume config to listen to Kakfa topic and write to HDFS.
flume1.sources  = kafka-source-1
flume1.channels = hdfs-channel-1
flume1.sinks    = hdfs-sink-1

# For each source, channel, and sink, set
# standard properties.
flume1.sources.kafka-source-1.type = org.apache.flume.source.
kafka.KafkaSource
flume1.sources.kafka-source-1.zookeeperConnect = localhost:2181
flume1.sources.kafka-source-1.topic = weblogs
flume1.sources.kafka-source-1.batchSize = 100
```

```
flume1.sources.kafka-source-1.channels = hdfs-channel-1

flume1.channels.hdfs-channel-1.type   = memory
flume1.sinks.hdfs-sink-1.channel = hdfs-channel-1
flume1.sinks.hdfs-sink-1.type = hdfs
flume1.sinks.hdfs-sink-1.hdfs.writeFormat = Text
flume1.sinks.hdfs-sink-1.hdfs.fileType = DataStream
flume1.sinks.hdfs-sink-1.hdfs.filePrefix = test-events
flume1.sinks.hdfs-sink-1.hdfs.useLocalTimeStamp = true
flume1.sinks.hdfs-sink-1.hdfs.path = /tmp/kafka/%{topic}/%y-%m-%d
flume1.sinks.hdfs-sink-1.hdfs.rollCount=100
flume1.sinks.hdfs-sink-1.hdfs.rollSize=0

# Other properties are specific to each type of
# source, channel, or sink. In this case, we
# specify the capacity of the memory channel.
flume1.channels.hdfs-channel-1.capacity = 10000
```

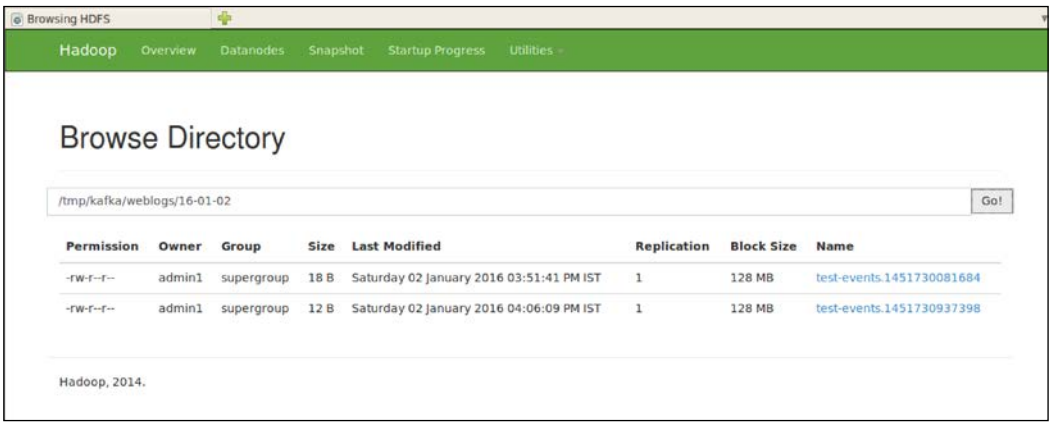
5. I am saving this configuration as `kafka.conf`. Now to execute this Flume agent, we use the following command:

```
/usr/local/flume/bin/flume-ng agent -n flume1 -c /usr/local/
flume/conf -f /usr/local/flume/conf/kafka.conf -Dflume.root.
logger=INFO,console
```

This will start a process that will be listening to the weblogs topic in Kafka. Now to test this setup, go to the screen where we are running the Kafka producers and send some messages to the weblogs topic. As soon as you send a message to the topic, you will see logs on Flume agent conveying that it is writing to HDFS.

You can now check out the HDFS path to see if the messages are being written there:

```
hadoop fs -cat /tmp/kafka/weblogs/16-01-02/test-events.1451730937398
Hello World
```



How it works

Flume agent listens to a given topic. As soon as the message is received, it uses the defined HDFS sinks to write the data to the HDFS path. First it creates a `.tmp file`, and after a certain message is received, it renames it to a regular file. Flume, by default, uses the group ID as Flume to avoid any message loss from the actual application. We can also run multiple Flume agents to distribute the load.

Importing web logs data into HDFS using Flume

One of the most important use cases of Flume is importing logs data into HDFS as and when it is produced. In this recipe, we will be executing a Flume agent which will be listening to the logs file.

Getting ready

To perform this recipe, you should have a Hadoop cluster running with you as well as the latest version of Flume installed on it.

How to do it...

1. To import data into HDFS from web servers, we have to install Flume agent on each web server instance.Following is the configuration we have to use for Flume agent configuration:

```
flume1.sources  = weblogs-source-1
flume1.channels = hdfs-channel-1
flume1.sinks    = hdfs-sink-1

# For each source, channel, and sink, set
# standard properties.
flume1.sources.weblogs-source-1.type = exec
flume1.sources.weblogs-source-1.command = tail -f /path/to/log/
file.log
flume1.sources.weblogs-source-1.batchSize = 100
flume1.sources.weblogs-source-1.channels = hdfs-channel-1

flume1.channels.hdfs-channel-1.type   = memory
flume1.sinks.hdfs-sink-1.channel = hdfs-channel-1
flume1.sinks.hdfs-sink-1.type = hdfs
flume1.sinks.hdfs-sink-1.hdfs.writeFormat = Text
```

```
flume1.sinks.hdfs-sink-1.hdfs.fileType = DataStream
flume1.sinks.hdfs-sink-1.hdfs.filePrefix = test-events
flume1.sinks.hdfs-sink-1.hdfs.useLocalTimeStamp = true
flume1.sinks.hdfs-sink-1.hdfs.path = /logs/web/%y-%m-%d
flume1.sinks.hdfs-sink-1.hdfs.rollCount=100
flume1.sinks.hdfs-sink-1.hdfs.rollSize=0

# Other properties are specific to each type of
# source, channel, or sink. In this case, we
# specify the capacity of the memory channel.
flume1.channels.hdfs-channel-1.capacity = 10000
```

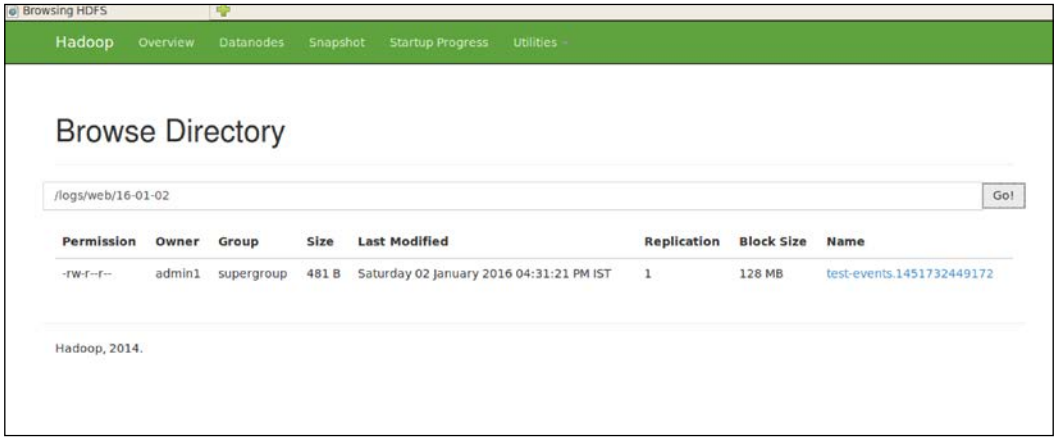
2. This will be listening to the given log file path. Here we are using `exec` source, which executes given command after a certain time interval.

   Here the source is an output of the `exec` command we have provided. The channel transfers the data to Sink, which writes it to HDFS.

3. To execute the preceding configuration, we have to run the following command:

   ```
   /usr/local/flume/bin/flume-ng agent -n flume1 -c /usr/local/
   flume/conf -f /usr/local/flume/conf/weblogs.conf -Dflume.root.
   logger=INFO,console
   ```

Now, as and when the log file is updated, the data will be written to HDFS as well:



How it works...

`Exec` source executes the command on a given instance and redirects the output to a channel. The Channel then makes sure that it is given to the sink. Here we are using HDFS sink, which writes the data to HDFS in the given format.

7
Automation of Hadoop Tasks Using Oozie

In this chapter, we'll take a look at the following recipes:

- Implementing a Sqoop action job using Oozie
- Implementing a Map Reduce action job using Oozie
- Implementing a Java action job using Oozie
- Implementing a Hive action job using Oozie
- Implementing a Pig action job using Oozie
- Implementing an e-mail action job using Oozie
- Executing parallel jobs using Oozie (fork)
- Scheduling a job in Oozie

Introduction

In the previous chapter, we talked about two very important tools, Sqoop and Flume, which help us seamlessly import and export data in and out of Hadoop. Now that we have talked about most of the Hadoop ecosystem tools and their advanced usage, it's time to understand how to automate these tasks using another interesting tool called Oozie. Oozie is a job scheduler, which helps us execute a series of Hadoop tasks in a workflow.

Implementing a Sqoop action job using Oozie

In the previous chapter, we took a look at how to use Sqoop to import and export data from RDBMS to HDFS. In this recipe, you are going to learn how to automate this Sqoop import and export using Oozie.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Sqoop and Oozie installed on it.

How to do it...

Any Oozie job execution consists of two important things, a `workflow.xml` and a properties file. The `workflow.xml` file is where we need to specify the flow of an execution. The following is an example of `workflow.xml`, which uses the Sqoop action:

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="sqoop-wf">
<start to="sqoop-node"/>

<action name="sqoop-node">
<sqoop xmlns="uri:oozie:sqoop-action:0.2">
<job-tracker>${jobTracker}</job-tracker>
<name-node>${nameNode}</name-node>

<configuration>
<property>
<name>mapred.job.queue.name</name>
<value>${queueName}</value>
</property>
</configuration>
<command>import --connect jdbc:mysql://localhost:3306/company
--username root --password password --table employee --target-dir /
data/employee-oozie -m 1</command>
</sqoop>
<ok to="end"/>
<error to="fail"/>
</action>
```

```
<kill name="fail">
<message>Sqoop failed, error message[${wf:errorMessage(wf:lastErrorNo
de())}]</message>
</kill>
<end name="end"/>
</workflow-app>
```

This is the command section where we need to provide the actual command that is to be executed from Sqoop.

Now, we have to upload `workflow.xml` to HDFS. After this, we need to create a `job.properties` file, which would invoke this workflow. Here is a sample code snippet for this purpose:

```
nameNode=hdfs://localhost:9000
jobTracker=localhost:8032
queueName=default
examplesRoot=examples
oozie.libpath=${nameNode}/user/admin1/share/lib
oozie.use.system.libpath=true

oozie.wf.application.path=${nameNode}/user/admin1/share/examples/apps/
sqoop
```

We have a property called `jobTracker` for which we have to give `ResourceManager` a host and a port.

Now, we have to submit this job using the following command:

```
oozie job -oozie http://localhost:11000/oozie -config job.properties
–run
```

This will start the workflow which will fetch data from MySQL and store it in the given HDFS directory. From Oozie's UI, you can track the progress of this job.

How it works

Oozie internally calls Sqoop libraries in order to import and export data as given in the command section. You can also use the `<prepare>` statements in order to make sure that directory where you export the data has been cleared before executing the job.

Implementing a Map Reduce action job using Oozie

In the previous recipe, we talked about how to use a Sqoop action to import data to HDFS. In this recipe, we are going to take a look at how to execute Map Reduce jobs using Oozie.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Oozie installed on it.

How to do it...

Any Oozie job execution consists of two important things, `workflow.xml` and a properties file. The `Workflow.xml` file is where we need to specify the flow of execution. The following is an example of `workflow.xml`, which uses the MR action. Here, we also need to provide the jar file that contains the the map reduce code:

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="map-reduce-wf">
<start to="mr-node"/>
<action name="mr-node">
<map-reduce>
<job-tracker>${jobTracker}</job-tracker>
<name-node>${nameNode}</name-node>
<prepare>
<delete path="${nameNode}/user/${wf:user()}/${examplesRoot}/output-
data/${outputDir}"/>
</prepare>
<configuration>
<property>
<name>mapred.job.queue.name</name>
<value>${queueName}</value>
</property>
<property>
<name>mapred.mapper.class</name>
<value>com.demo.WordMapper</value>
</property>
<property>
<name>mapred.reducer.class</name>
<value>com.demo.SumReducer</value>
</property>
<property>
```

```
<name>mapred.input.dir</name>
<value>/in</value>
</property>
<property>
<name>mapred.output.dir</name>
<value>${outputDir}</value>
</property>
</configuration>
</map-reduce>
<ok to="end"/>
<error to="fail"/>
</action>
<kill name="fail">
<message>Map/Reduce failed, error message[${wf:errorMessage(wf:lastErr
orNode())}]</message>
</kill>
<end name="end"/>
</workflow-app>
```

In the preceding `workflow.xml`, it is important to provide the mapper and reducer classes, respectively.

Now, we have to upload `workflow.xml` and the jar file to HDFS. After this, we need to create the `job.properties` file, which will invoke this workflow. Here is a sample code snippet for this purpose:

```
nameNode=hdfs://localhost:9000
jobTracker=localhost:8032
queueName=default

oozie.wf.application.path=${nameNode}/user/admin1/share/examples/apps/
map-reduce/workflow.xml
outputDir=/newout
```

For the property called `jobTracker`, we can pass the `ResourceManager` host and port number.

We have to submit this job using the following command:

```
oozie job -oozie http://localhost:11000/oozie -config job.properties
–run
```

This will start the workflow, which will start reading data from the `/in` directory, calculate the word count, and write the output to the `/newout` directory.

How it works...

Oozie internally submits the map reduce job to the Resource Managers and works similar to our regular map reduce jobs. You can also use the `<prepare>` statements in order to make sure that the directory where you export the data has been cleared before executing the job.

Implementing a Java action job using Oozie

In the previous recipe, we talked about how to use Oozie to execute the Map Reduce job. In this recipe, we are going to take a look at how to execute any Java class using Oozie.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Oozie installed on it.

How to do it

Any Oozie job execution consists of two important things, `workflow.xml` and a properties file. The `workflow.xml` is where we need to specify the flow of execution. The following is an example of `workflow.xml`, which uses a Java action. Here, we need to provide the jar file in which the code is present:

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="java-main-wf">
<start to="java-node"/>
<action name="java-node">
<java>
<job-tracker>${jobTracker}</job-tracker>
<name-node>${nameNode}</name-node>
<configuration>
<property>
<name>mapred.job.queue.name</name>
<value>${queueName}</value>
</property>
</configuration>
<main-class>com.demo.JavaMain</main-class>
<arg>Hello</arg>
<arg>Oozie!</arg>
</java>
<ok to="end"/>
<error to="fail"/>
</action>
```

```
<kill name="fail">
<message>Java failed, error message[${wf:errorMessage(wf:lastErrorNo
de())}]</message>
</kill>
<end name="end"/>
</workflow-app>
```

In the preceding e `workflow.xml`, we need to provide the main class that should be the entry point of our Java program. We can also provide arguments that can be passed to the Java class.

Now, we have to upload `workflow.xml` and the jar file to HDFS. After this, we need to create the `job.properties` file, which will invoke this workflow. Here is a sample code snippet for this purpose:

```
nameNode=hdfs://localhost:8020
jobTracker=localhost:8032
queueName=default
examplesRoot=examples

oozie.wf.application.path=${nameNode}/user/admin1/${examplesRoot}/
apps/java-main
```

Here, we have a property called `jobTracker` for which we have to provide `ResourceManager` a host and a port.

Now, we have to submit this job using the following command:

```
oozie job -oozie http://localhost:11000/oozie -config job.properties
-run
```

This will start the workflow, which will execute the code given in the `Main` class using the arguments that are passed.

How it works

Oozie executes the `Main` class provided along with the arguments in `workflow.xml`. It is important to note that while using Oozie, if you are trying to access any resources, make sure that a user has access to them.

Implementing a Hive action job using Oozie

In this recipe, we are going to take a look at how to use a Hive action in order to automate Hive query executions.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Oozie and Hive installed on it.

How to do it...

Any Oozie job execution consists of two important things, `workflow.xml` and a properties file. The `workflow.xml` file is where we need to specify the flow of execution. The following is an example of `workflow.xml`, which uses the Hive action:

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="hive-wf">
<start to="hive-node"/>

<action name="hive-node">
<hive xmlns="uri:oozie:hive-action:0.2">
<job-tracker>${jobTracker}</job-tracker>
<name-node>${nameNode}</name-node>
<configuration>
<property>
<name>mapred.job.queue.name</name>
<value>${queueName}</value>
</property>
</configuration>
<script>script.q</script>
</hive>
<ok to="end"/>
<error to="fail"/>
</action>

<kill name="fail">
<message>Hive failed, error message[${wf:errorMessage(wf:lastErrorNo
de())}]</message>
</kill>
<end name="end"/>
</workflow-app>
```

Here, we can provide the Hive commands that are to be executed in the `script.q` file. The following is an example of the contents of `script.q`:

```
CREATE EXTERNAL TABLE test (a INT) STORED AS TEXTFILE LOCATION '/data/
input/test';
CREATE TABLE test_new (a INT) STORED AS TEXTFILE;
INSERT OVERWRITE TABLE test_new SELECT * FROM test;
```

In this script, you can write as many Hive queries as you wish to execute using Oozie.

Now, we have to upload `workflow.xml` and `script.q` to HDFS. After this, we need to create the `job.properties` file, which will invoke this workflow. Here is a sample code snippet for this purpose:

```
nameNode=hdfs://localhost:9000
jobTracker=localhost:8032
queueName=default
examplesRoot=examples

oozie.use.system.libpath=true

oozie.wf.application.path=${nameNode}/user/admin1/share/examples/apps/
hive
```

Here, we have a property called `jobTracker` for which we have to give `ResourceManager` a host and a port.

Now, we have to submit this job using the following command:

```
oozie job -oozie http://localhost:11000/oozie -config job.properties
–run
```

This will start the workflow, which will execute the given Hive queries.

How it works...

Oozie internally calls Hive to execute the given Hive query. It is important to provide fully qualified paths for input directories and not relative directories so there are no errors because of invalid paths. Each and everytime this can be performed from the Hive action as well.

Implementing a Pig action job using Oozie

In this recipe, we are going to take a look at how to use a Pig action in order to automate the `Pigscripts` executions.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Oozie and Pig installed on it.

How to do it...

A Oozie job execution consists of two important things, `workflow.xml` and a properties file. The `workflow.xml` file is where we need to specify the flow of execution. The following is an example of `workflow.xml`, which uses the Pig action:

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="pig-wf">
<start to="pig-node"/>
<action name="pig-node">
<pig>
<job-tracker>${jobTracker}</job-tracker>
<name-node>${nameNode}</name-node>
<prepare>
<delete path="${nameNode}/user/${wf:user()}/${examplesRoot}/output-
data/pig"/>
</prepare>
<configuration>
<property>
<name>mapred.job.queue.name</name>
<value>${queueName}</value>
</property>
</configuration>
<script>script.pig</script>
<param>INPUT=/user/${wf:user()}/${examplesRoot}/input-data/text</
param>
<param>OUTPUT=/user/${wf:user()}/${examplesRoot}/output-data/pig</
param>
</pig>
<ok to="end"/>
<error to="fail"/>
</action>
<kill name="fail">
```

```
<message>Pig failed, error message[${wf:errorMessage(wf:lastErrorNo
de())}]</message>
</kill>
<end name="end"/>
</workflow-app>
```

Here, we can provide the Pig commands to be executed in the `script.pig` file. The following is an example of the contents of `script.pig`:

```
A = load '$INPUT' using PigStorage(':');
B = foreach A generate $0 as id;
store B into '$OUTPUT' USING PigStorage();
```

In this script, you can write any Pig command that you wish to execute using Oozie.

Now, we have to upload `workflow.xml` and `script.pig` to HDFS. After this, we need to create the `job.properties` file, which will invoke this workflow. Here is a sample code snippet for this purpose:

```
nameNode=hdfs://localhost:9000
jobTracker=localhost:8032
queueName=default
examplesRoot=examples

oozie.use.system.libpath=true

oozie.wf.application.path=${nameNode}/user/admin1/share/examples/apps/
pig
```

Here, we have a property, called `jobTracker`, for which we have to give `ResourceManager` a host and a port.

Now, we have to submit this job using the following command:

```
oozie job -oozie http://localhost:11000/oozie -config job.properties
–run
```

This will start the workflow, which will execute the given `Pigcommands`.

How it works

Oozie internally calls Pig to execute the given Pig script. It is important to provide fully qualified paths for input directories and not relative directories. Each and everytime, this can be performed from the Pig action as well.

Implementing an e-mail action job using Oozie

In this recipe, we are going to take a look at how to use an e-mail action in order to notify users about job executions in Oozie.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Oozie installed on it.

How to do it...

An Oozie job execution consists of two important things, `workflow.xml` and a properties file. The `workflow.xml` file is where we need to specify the flow of execution. The following is an example of `workflow.xml`, which uses the e-mail action:

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="pig-wf">
<start to="notify"/>
<action name="notify">
<email xmlns="uri:oozie:email-action:0.1">
<to>a@.b.com</to>
<cc>b@b.com</cc>
<subject>Email notifications for ${wf:id()}</subject>
<body>The wf ${wf:id()} successfully completed.</body>
</email>
<error to="fail"/>
</action>
<kill name="fail">
<message>Email, error message[${wf:errorMessage(wf:lastErrorNo
de())}]</message>
</kill>
<end name="end"/>
</workflow-app>
```

To use the e-mail action, we first need to configure SMTP settings in `oozie-site.xml`, and the following properties need to be set:

- `oozie.email.smtp.host`: The host where the e-mail action may find the SMTP server (the localhost by default)
- `oozie.email.smtp.port`: The port to connect to for the SMTP server (25 by default)

- `oozie.email.from.address`: The `from` address to be used for the purpose of mailing all e-mails (`oozie@localhost` by default)
- `oozie.email.smtp.auth`: The Boolean property that toggles when an authentication is to be done or not (false by default)
- `oozie.email.smtp.username`: If an authentication is enabled, this shows the username to login as (empty by default)
- `oozie.email.smtp.password`: If the authentication is enabled, this shows the username's password (empty by default)

Now, we have to upload `workflow.xml` to HDFS. After this, we need to create the `job.properties` file, which will invoke this workflow. Here is a sample code snippet for this purpose:

```
nameNode=hdfs://localhost:9000
jobTracker=localhost:8032
queueName=default
examplesRoot=examples

oozie.use.system.libpath=true

oozie.wf.application.path=${nameNode}/user/admin1/share/examples/apps/
email
```

Here, we have a property, called `jobTracker`, for which we have to give `ResourceManager` a host and a port.

Now, we have to submit this job using the following command:

```
oozie job -oozie http://localhost:11000/oozie -config job.properties
–run
```

This will start the workflow, which will send the e-mail with given the details.

How it works...

The e-mail action is very important as it helps us notify users about the progress. My recommendation is that you should use this action at the start of the workflow, in case of any errors, and on successful completion.

Executing parallel jobs using Oozie (fork)

In this recipe, we are going to take a look at how to execute parallel jobs using the Oozie fork node. Here, we will be executing one Hive and one Pig job in parallel.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Oozie, Hive, and Pig installed on it.

How to do it...

For parallel execution, we need to use the fork node given by Oozie. The following is a sample workflow that executes Hive and Pig jobs in parallel:

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="demo-wf">

<start to="fork-node"/>



<fork name="fork-node">
<path start="pig-node"/>
<path start="hive-node"/>
</fork>

<action name="pig-node">
<pig>
<job-tracker>${jobTracker}</job-tracker>
<name-node>${nameNode}</name-node>
<prepare>
<delete path="${nameNode}/user/${wf:user()}/${examplesRoot}/output-
data/pig"/>
</prepare>
<configuration>
<property>
<name>mapred.job.queue.name</name>
<value>${queueName}</value>
</property>
</configuration>
<script>script.pig</script>
<param>INPUT=/user/${wf:user()}/${examplesRoot}/input-data/text</
param>
```

```
<param>OUTPUT=/user/${wf:user()}/${examplesRoot}/output-data/pig</
param>
</pig>
<ok to="join-node"/>
<error to="fail"/>
</action>


<action name="hive-node">
<hive xmlns="uri:oozie:hive-action:0.2">
<job-tracker>${jobTracker}</job-tracker>
<name-node>${nameNode}</name-node>
<configuration>
<property>
<name>mapred.job.queue.name</name>
<value>${queueName}</value>
</property>
</configuration>
<script>script.q</script>
</hive>
<ok to="join-node"/>
<error to="fail"/>
</action>



<join name="join-node" to="end"/>

<kill name="fail">
<message>Demo workflow failed, error message[${wf:errorMessage(wf:last
ErrorNode())}]</message>
</kill>

<end name="end"/>

</workflow-app>
```

We can provide any number of jobs in the fork node to execute it in parallel. Each forked node should be joined back in a join node.

Now, we have to submit this job using the following command:

```
oozie job -oozie http://localhost:11000/oozie -config job.properties
-run
```

This will start the workflow that will execute Hive and Pig scripts in parallel.

How it works...

The fork node is helpful when we have two more processes which are independent of each other. This helps reduce the time for execution by consuming available resources.

Scheduling a job in Oozie

In this recipe, we are going to take a look at a schedule that has recurring jobs using the Oozie coordinator.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Oozie installed on it.

How to do it...

Oozie provides one more type of job called a coordinator job. This type of job is used to schedule application jobs. With the help of a coordinator job, we can execute an application job. The following is an example of a coordinator job that runs daily:

```
<coordinator-app name="sample-coordinator"
  frequency="${coord:days(1)}"
  start="2016-01-01T18:56Z" end="2017-01-01T18:56Z" timezone="UTC"
  xmlns="uri:oozie:coordinator:0.2">

<controls>

<concurrency>1</concurrency>
<execution>FIFO</execution>
<throttle>5</throttle>
</controls>


<action>
<workflow>
<app-path>${applicationPath}</app-path>
<configuration>
       ...
</configuration>
</workflow>
</action>
</coordinator-app>
```

Coordinator jobs contain one important attribute called frequency, which decides the frequency of job executions within a given start and end time.

Here is sample `job.properties` to execute the preceding workflow:

```
nameNode=hdfs://localhost:9000
jobTracker=localhost:8032
queueName=default
examplesRoot=examples

oozie.use.system.libpath=true

oozie.coord.application.path=${nameNode}/user/admin1/share/examples/
apps/coordinator
```

Now, we have to submit this job using the following command:

```
oozie job -oozie http://localhost:11000/oozie -config job.properties
–run
```

This will start the coordinator workflow, which will keep on running as per the schedule.

How it works...

Coordinator jobs work like `crontab`, where you set the frequency of certain task executions. Internally, it then invokes the application workflow provided in the coordinator workflow.

8

Machine Learning and Predictive Analytics Using Mahout and R

In this chapter, we'll cover the following recipes:

- Setting up the Mahout development environment
- Creating an item-based recommendation engine using Mahout
- Creating a user-based recommendation engine using Mahout
- Using predictive analytics for the marketing data of a bank
- Clustering text data using K-Means
- Performing population data analytics using R
- Performing Twitter Sentiment Analytics using R
- Performing Predictive Analytics using R

Introduction

In the previous chapter, we talked about how to automate Hadoop and its ecosystem tasks using Oozie. In this chapter, we will go deeper into the concepts of machine learning using Mahout and R. Mahout is a machine learning library, which allows us to solve machine learning problems with ease, whereas R is a statistical tool, which helps us build models. So, let's get started.

Setting up the Mahout development environment

In this recipe, we are going to take a look at how to set up the Mahout development environment.

Getting ready

To perform this recipe, you should have a running Hadoop cluster.

How to do it...

Setting up the Mahout environment is very easy:

1. To start with, we first need to download the latest version of Mahout from `http://www.apache.org/dyn/closer.cgi/mahout/`.
2. I am going to use version 0.11.1 ,which can be found at `http://www.eu.apache.org/dist/mahout/0.11.1/apache-mahout-distribution-0.11.1.tar.gz`.
3. Next, unzip the tar and rename the folder as Mahout for simplicity's sake:

   ```
   sudo tar  -xzf  apache-mahout-distribution-0.11.1.tar.gz
   sudo mv apache-mahout-distribution-0.11.1 mahout
   ```

4. To use the Mahout commands from everywhere, we add the distribution path to `PATH`.

   Edit `~/.bashrc` and add the following commands to it:

   ```
   export MAHOUT_HOME=/usr/local/mahout
   export PATH=$PATH:$MAHOUT_HOME/bin
   ```

5. Execute the following command to take a look at whether the changes are effective:

   ```
   source ~/.bashrc
   ```

6. Now you are all set to use Mahout:

   ```
   $ mahout
   Running on hadoop, using /usr/local/hadoop/bin/hadoop and HADOOP_
   CONF_DIR=
   MAHOUT-JOB: /usr/local/mahout/mahout-examples-0.11.1-job.jar
   An example program must be given as the first argument.
   Valid program names are:
     arff.vector: : Generate Vectors from an ARFF file or directory
     baumwelch: : Baum-Welch algorithm for unsupervised HMM training
     canopy: : Canopy clustering
     cat: : Print a file or resource as the logistic regression
   models would see it
   ```

```
  cleansvd: : Cleanup and verification of SVD output
  clusterdump: : Dump cluster output to text
  clusterpp: : Groups Clustering Output In Clusters
  cmdump: : Dump confusion matrix in HTML or text formats
  cvb: : LDA via Collapsed Variation Bayes (0th deriv. approx)
  cvb0_local: : LDA via Collapsed Variation Bayes, in memory
locally.
  describe: : Describe the fields and target variable in a data
set
  evaluateFactorization: : compute RMSE and MAE of a rating matrix
factorization against probes
  fkmeans: : Fuzzy K-means clustering
  hmmpredict: : Generate random sequence of observations by given
HMM
  itemsimilarity: : Compute the item-item-similarities for item-
based collaborative filtering
  kmeans: : K-means clustering
  lucene.vector: : Generate Vectors from a Lucene index
  matrixdump: : Dump matrix in CSV format
  matrixmult: : Take the product of two matrices
  parallelALS: : ALS-WR factorization of a rating matrix
  qualcluster: : Runs clustering experiments and summarizes
results in a CSV
  recommendfactorized: : Compute recommendations using the
factorization of a rating matrix
  recommenditembased: : Compute recommendations using item-based
collaborative filtering
  regexconverter: : Convert text files on a per line basis based
on regular expressions
  resplit: : Splits a set of SequenceFiles into a number of equal
splits
  rowid: : Map SequenceFile<Text,VectorWritable> to {SequenceFile<
IntWritable,VectorWritable>, SequenceFile<IntWritable,Text>}
  rowsimilarity: : Compute the pairwise similarities of the rows
of a matrix
  runAdaptiveLogistic: : Score new production data using a
probably trained and validated AdaptivelogisticRegression model
  runlogistic: : Run a logistic regression model against CSV data
  seq2encoded: : Encoded Sparse Vector generation from Text
sequence files
  seq2sparse: : Sparse Vector generation from Text sequence files
  seqdirectory: : Generate sequence files (of Text) from a
directory
  seqdumper: : Generic Sequence File dumper
  seqmailarchives: : Creates SequenceFile from a directory
containing gzipped mail archives
```

```
  seqwiki: : Wikipedia xml dump to sequence file
  spectralkmeans: : Spectral k-means clustering
  split: : Split Input data into test and train sets
  splitDataset: : split a rating dataset into training and probe
parts
  ssvd: : Stochastic SVD
  streamingkmeans: : Streaming k-means clustering
  svd: : Lanczos Singular Value Decomposition
  testnb: : Test the Vector-based Bayes classifier
  trainAdaptiveLogistic: : Train an AdaptivelogisticRegression
model
  trainlogistic: : Train a logistic regression using stochastic
gradient descent
  trainnb: : Train the Vector-based Bayes classifier
  transpose: : Take the transpose of a matrix
  validateAdaptiveLogistic: : Validate an
AdaptivelogisticRegression model against hold-out data set
  vecdist: : Compute the distances between a set of Vectors (or
Cluster or Canopy, they must fit in memory) and a list of Vectors
  vectordump: : Dump vectors from a sequence file to text
viterbi: : Viterbi decoding of hidden states from given output
states sequence
```

How it works...

We have just set the Mahout home, and it shows all the algorithms that are supported along with their purpose. Not all Mahout algorithms are able to use Hadoop/Map Reduce in order to understand that in a better manner.



I suggest that you take a look at `https://mahout.apache.org/users/basics/algorithms.html`.

Creating an item-based recommendation engine using Mahout

In this recipe, we are going to take a look at how to use Mahout to generate item-based recommendations. Recommendation engine is one of the most seen use cases. A recommendation engine generates recommendations based on the input data provided to it. In this recipe, we are going to take a look at how to generate recommendations based on user preferences for certain items.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Mahout installed on it.

How to do it...

Mahout provides built-in support for item-based recommendations. In order to execute a program using Mahout, we first need to prepare the input data and store it in a certain folder. The input data needs to be in a specified format (`userId`, `itemId`, and preference). Here, `userId` is the unique user identifier, `itemId`, is the unique item identifier, while the preference can be a rating given by a user to a specific item, number of items that are ordered, and so on.

First, we capture and prepare our data and store it in HDFS:

```
hadoop fs -mkdir /recommender-in
hadoop fs -put user-item.txt /recommender-in
```

Here is some sample input:

```
1,10,1.0
1,11,2.0
1,12,5.0
1,13,5.0
1,14,5.0
1,15,4.0
1,16,5.0
1,17,1.0
1,18,5.0
2,10,1.0
2,11,2.0
2,15,5.0
2,16,4.5
2,17,1.0
2,18,5.0
3,11,2.5
3,12,4.5
3,13,4.0
3,14,3.0
3,15,3.5
```

```
3,16,4.5
3,17,4.0
3,18,5.0
4,10,5.0
4,11,5.0
4,12,5.0
4,13,0.0
4,14,2.0
4,15,3.0
4,16,1.0
4,17,4.0
4,18,1.0
```

To start the job, we have to execute the following command:

```
mahout recommenditembased -s SIMILARITY_LOGLIKELIHOOD -i /recommender-in
-o /recommender-out --numRecommendations 5
```

The preceding command starts a series of Map Reduce programs, which run over the input data. `SIMILARITY_LOGLIKELIHOOD` is the name of a class that determines the similarity between items. You can either use this default, or you can choose from a variety of similar algorithms that are available.

Once the execution is complete, the output that's generated is as follows:

```
$ hadoop fs -ls /recommender-out
    Found 2 items
    -rw-r--r--   1 admin1 supergroup          0 2016-01-28 11:46
    /recommender-out/_SUCCESS
    -rw-r--r--   1 admin1 supergroup         28 2016-01-28 11:46
    /recommender-out/part-r-00000
$ hadoop fs -cat /recommender-out/part-r-00000
    3  [10:3.8597424]
    4  [13:4.0]
```

This output indicates that the recommender has generated two recommendations. For user 3, it recommends item 10, and for user 4, it recommends item 13.

This way, you can keep on putting your data in HDFS and keep running the recommendation jobs every time that the data is updated. You can write custom code to use this output and integrate it into your application.

How it works...

Mahout uses various algorithms to generate the recommendations. It first puts the data that's been given as the input in the form of model data. It uses a similarity calculation algorithm given by the user in order to do this. These two things are used as input in order to calculate the similarity between the items. Once this is calculated, based on the items that are not used by users, it recommends them.

This implementation can used as is without any modifications if it can get your data in the given format. You can use this recommendation engine to generate recommendations on e-commerce websites for production recommendations, or it can also be used to track web user activity and recommend web pages accordingly.

Creating a user-based recommendation engine using Mahout

In this recipe, we are going to take a look at how to use Mahout to generate user-based recommendations. The user-based recommendation engine is not available directly to be used as a Map Reduce job. We have to run it in a sequential manner, as described in the next section.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Mahout installed on it. We will also need an Eclipse-like IDE or any other IDE of your choice for code development.

How to do it...

User-based recommendations work on the simple principle of user similarities and their likelihood toward the same set of items. To implement them, we first need to create a Maven project, and add the following dependency to it:

```
<dependency>
    <groupId>org.apache.mahout</groupId>
    <artifactId>mahout-mr</artifactId>
    <version>0.10.0</version>
</dependency>
```

Next, we create a Java class, as shown here:

```
package com.demo.recommender;

import java.io.File;
import java.io.IOException;
import java.util.List;

import org.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import org.apache.mahout.cf.taste.impl.neighborhood.
ThresholdUserNeighborhood;
import org.apache.mahout.cf.taste.impl.recommender.
GenericUserBasedRecommender;
import org.apache.mahout.cf.taste.impl.similarity.
PearsonCorrelationSimilarity;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.neighborhood.UserNeighborhood;
import org.apache.mahout.cf.taste.recommender.RecommendedItem;
import org.apache.mahout.cf.taste.recommender.UserBasedRecommender;
import org.apache.mahout.cf.taste.similarity.UserSimilarity;

public class MyUserBasedRecommender
{
    public static void main( String[] args ) throws IOException,
TasteException
    {
    ClassLoader classLoader = MyUserBasedRecommender.class.
getClassLoader();
    DataModel model = new FileDataModel(new File(classLoader.
getResource("dataset.csv").getFile()));
    UserSimilarity similarity = new PearsonCorrelationSimilarity(mod
el);
    UserNeighborhood neighborhood = new ThresholdUserNeighborhood(0.1,
similarity, model);
        UserBasedRecommender recommender = new GenericUserBasedRecomme
nder(model, neighborhood, similarity);
    List<RecommendedItem> recommendations = recommender.recommend(2,
3);
    for (RecommendedItem recommendation : recommendations) {
      System.out.println(recommendation);
    }
    }
}
```

In the preceding code, we first need data from a resource folder. We have copied the `dataset.csv` file into the `src/main/resources` folder. The following code shows you the contents of this folder. You should read the data in the file in the specified format such as `UserId`, `ItemId`, and preference:

```
1,10,1.0
1,11,2.0
1,12,5.0
1,13,5.0
1,14,5.0
1,15,4.0
1,16,5.0
1,17,1.0
1,18,5.0
2,10,1.0
2,11,2.0
2,15,5.0
2,16,4.5
2,17,1.0
2,18,5.0
3,11,2.5
3,12,4.5
3,13,4.0
3,14,3.0
3,15,3.5
3,16,4.5
3,17,4.0
3,18,5.0
4,10,5.0
4,11,5.0
4,12,5.0
4,13,0.0
4,14,2.0
4,15,3.0
4,16,1.0
4,17,4.0
4,18,1.0
```

In order to create a user-based recommendation, we need to first calculate the similarity between these users. Here, we choose the Pearson Similarity Algorithm to calculate the similarity. Next, we pass the similarity, model, and the threshold to decide the neighborhood. Then, we create a user-based recommender by providing the preceding calculated values.

In the preceding code, on execution, we want the top three item recommendations for user 2.

On the execution of the code, we will get the following output. This is the list of items that are to be recommended along with the recommendation score:

```
RecommendedItem[item:12, value:4.8328104]
RecommendedItem[item:13, value:4.6656213]
RecommendedItem[item:14, value:4.331242]
```

How it works...

Mahout's user-based recommendation first tries to calculate the similarity between the users, and then it recommends other similar users from their kitty. There are various similarity calculations available, and you can choose any one of them; the preceding example uses the Pearson Similarity Algorithm. More on this can be found at `https://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient`.

Make a note that this recommendation can only be run in a sequential manner, so there is no Map Reduce implementation for this.

Using Predictive analytics on Bank Data using Mahout

In this recipe, we are going to take a look at how to use Mahout to generate a predictive model and validate how good this model is against some sample data. Here, we will be using the sample data collected by a bank during their marketing operations.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Mahout installed on it.

How to do it...

In this recipe, we are going to use Logistic Regression in order to predict the occurrence of an event. It uses predictors from the given data in order to calculate the probability. The Mahout implementation uses the **Stochastic Gradient Descent** (**SGD**) algorithm for logistic regression. You can learn more about SGD for logistic regression at `http://blog.trifork.com/2014/02/04/an-introduction-to-mahouts-logistic-regression-sgd-classifier/`.

SGD is, by default, a sequential algorithm so we cannot run any parallel activities on it. Even though it is sequential, it runs blazing fast on large datasets for which we don't need to worry about the size of data that we are handling.

Now, we are going to take a look at how to execute this algorithm in our data. The following is sample data from a bank's marketing team, which has the last column labelled as 'out'. This is the output that is shown when a customer who's given their details has said yes to the term deposit or not:

```
age,job,marital,education,out
30,unemployed,married,primary,no
33,services,married,secondary,yes
35,management,single,tertiary,no
30,management,married,tertiary,no
59,blue-collar,married,secondary,no
35,management,single,tertiary,no,
36,self-employed,married,tertiary,yes
39,technician,married,secondary,no
41,entrepreneur,married,tertiary,no
43,services,married,primary,no
39,services,married,secondary,no
43,admin.,married,secondary,no
36,technician,married,tertiary,yes
20,student,single,secondary,no
31,blue-collar,married,secondary,yes
```

We save this data in a file called `bank.csv`. Now, let's execute the Mahout program to train a model for this data using this command:

```
mahout  trainlogistic --input bank.csv --output ./model --target out
--categories 2  --predictors age job marital education --types word
--features 4
```

Here is the `input` file in the CSV format, which is used to train the model. We want the output to be created in a file called model. We expect a type of output called `categories` (yes/no), `predictors` called columns are used to generate the mode, and `types` can be words/numerical. And no. of `features` to be used for generating model.

On execution, you will get the following output:

```
Running on hadoop, using /usr/local/hadoop/bin/hadoop and HADOOP_CONF_
DIR=
MAHOUT-JOB: /usr/local/mahout/mahout-examples-0.11.1-job.jar
4
out ~
-0.003*Intercept Term + -0.007*age=20 + -0.006*age=30 +
-0.005*age=31 + -0.002*age=33 + -0.005*age=35 + -0.006*age=36 +
-0.005*age=39 + -0.006*age=41 + -0.006*age=43 + -0.004*age=59
+ -0.007*education=primary + -0.002*education=secondary +
-0.004*education=tertiary + -0.002*job=admin. + -0.005*job=blue-
collar + -0.005*job=entrepreneur + -0.004*job=management +
-0.007*job=self-employed + -0.006*job=services + -0.005*job=student +
-0.006*job=technician + -0.006*job=unemployed + -0.006*marital=married
+ -0.007*marital=single
```

```
         Intercept Term -0.00298
                 age=20 -0.00696
                 age=30 -0.00645
                 age=31 -0.00457
                 age=33 -0.00159
                 age=35 -0.00545
                 age=36 -0.00645
                 age=39 -0.00545
                 age=41 -0.00557
                 age=43 -0.00645
                 age=59 -0.00398
     education=primary -0.00696
   education=secondary -0.00159
    education=tertiary -0.00406
           job=admin. -0.00159
      job=blue-collar -0.00457
     job=entrepreneur -0.00457
        job=management -0.00398
     job=self-employed -0.00696
          job=services -0.00557
           job=student -0.00457
        job=technician -0.00557
        job=unemployed -0.00645
       marital=married -0.00645
        marital=single -0.00696
    -0.003981019    -0.002976461    -0.001590416    -0.002469779
16/01/28 12:44:01 INFO MahoutDriver: Program took 644 ms (Minutes:
0.010733333333333333)
```

Now that we have generated the model to predict the outcome, we can validate it against the data shown here:

```
mahout runlogistic --input "bank.csv" --model ./model --auc –confusion
Running on hadoop, using /usr/local/hadoop/bin/hadoop and HADOOP_CONF_
DIR=
MAHOUT-JOB: /usr/local/mahout/mahout-examples-0.11.1-job.jar
AUC = 0.73
confusion: [[11.0, 4.0], [0.0, 0.0]]
entropy: [[-0.7, -0.4], [-0.7, -0.4]]
```

This will run the logistic regression on `bank.csv` using the model generated in the previous step in order to calculate AUC and a confusion matrix.

AUC stands for the Area Under a Curve. You can read more about AUC at `http://www.mathwords.com/a/area_under_a_curve.htm`.

The AUC should lie between 0.5 and 1.0 in scenarios where you're using a good model, while 1.0 indicates a perfect model.

A Confusion Matrix is a matrix that shows the performance of an algorithm by comparing its actual results with its predicted results. You can read more about the confusion matrix at `https://en.wikipedia.org/wiki/Confusion_matrix`.

How it works...

Mahout's implementation of SGD calculates the probability of the occurrence of a certain event based on the values considered as predictors. When creating a model, it considers the values of given predictors and these are used for predictions as well. In order to understand Logistic Regression and how it works internally.



I would suggest that you read `http://blog.trifork.com/2014/02/04/an-introduction-to-mahouts-logistic-regression-sgd-classifier/`.

Clustering text data using K-Means

In this recipe, we are going to take a look at how to use Mahout to cluster text data using Mahout's implementation of the K-Means algorithm. K-Means is very popular clustering algorithm; you can read more about it at `https://en.wikipedia.org/wiki/K-means_clustering`.

Getting ready

To perform this recipe, you should have a running Hadoop cluster as well as the latest version of Mahout installed on it.

How to do it...

In this recipe, we are going to use Mahout's K Means algorithm to cluster the text data that is available. To do this, we first need to get some text data and copy it to HDFS:

```
hadoop fs –mkdir /kmeans
hadoop fs –put mydata.txt /kmeans/input
```

In order to execute the K-Means job on the given data, we first need to convert it into sequential files and from these sequential files to TF-IDF vectors. Mahout provides built-in utilities to perform these actions. The following are the commands to do this.

To convert text data into a sequential file, here is the code:

```
mahout seqdirectory -i /kmeans/input/ -o /kmeans/sequencefiles
```

To convert a sequential file to a vector, here is the code:

```
mahout seq2sparse -i /kmeans/sequencefiles -o /kmeans/sparse
```

This command will take some time to work as it will run a series of Map Reduce programs to transform the data.

Next, we can run a clustering job on vectors that have been generated in the previous steps:

```
mahout kmeans -i hdfs://localhost:9000/kmeans/sparse/tfidf-vectors/ -c
hdfs://localhost:9000/kmeans/cl/ -o  hdfs://localhost:9000/kmeans/out -dm
org.apache.mahout.common.distance.CosineDistanceMeasure -x 10 –clustering
```

This will start a Map Reduce program that performs the actual clustering.

The output of the clustering will be available in the `/kmeans/out` folder. We can also use the `clusterdump` command to view the output:

```
mahout clusterdump -dt sequencefile -d /kmeans/sparse/dictionary. file-0
-i /kmeans/out/clusters-1-final
```

In this way, we can use Mahout to perform the clustering of text data.

How it works...

The K-Means algorithm plots a graph of points. It then calculates the distance between nodes in order to classify them. In order to optimize the clustering, iterations are performed, and in each new iteration, the centroid of each cluster becomes the mean of the next set of calculations.

Apart from K-Means, Mahout supports various other clustering algorithms such as Canopy, Fuzzy K-means, Streaming K-Means, and so on.

Performing population Data Analytics using R

So far, we talked about how to use Mahout to solve various machine learning problems. Now, we are going to explain another tool/language called R, which has built-in support for various mathematical and statistical operations.

Getting ready

To perform this recipe, you should have R installed on your machine. You can download the installer from `https://cran.r-project.org/bin/windows/base/`.

How to do it...

In this recipe, we are going to learn some basic operations that one can perform using R. To start with, we will have one dataset that has information about Australia's population in various states. This is what the dataset looks like:

```
Year NSW Vic. Qld SA WA Tas. NT ACT Aust.
1917 1904 1409 683 440 306 193 5 3 4941
1927 2402 1727 873 565 392 211 4 8 6182
1937 2693 1853 993 589 457 233 6 11 6836
1947 2985 2055 1106 646 502 257 11 17 7579
1957 3625 2656 1413 873 688 326 21 38 9640
1967 4295 3274 1700 1110 879 375 62 103 11799
1977 5002 3837 2130 1286 1204 415 104 214 14192
1987 5617 4210 2675 1393 1496 449 158 265 16264
1997 6274 4605 3401 1480 1798 474 187 310 18532
```

I am saving this data in a file called `auspop.txt`. Now, we start R and load this file in an R object by executing the following command:

```
>austpop <- read.table("H:/workspace/austpop.txt", header=TRUE)
```

We can check whether the data is loaded properly by printing it:

```
>print(austpop)
```

```
  Year  NSW Vic.  Qld   SA   WA Tas.  NT ACT Aust.
1 1917 1904 1409  683  440  306  193   5   3  4941
2 1927 2402 1727  873  565  392  211   4   8  6182
3 1937 2693 1853  993  589  457  233   6  11  6836
4 1947 2985 2055 1106  646  502  257  11  17  7579
5 1957 3625 2656 1413  873  688  326  21  38  9640
6 1967 4295 3274 1700 1110  879  375  62 103 11799
7 1977 5002 3837 2130 1286 1204  415 104 214 14192
8 1987 5617 4210 2675 1393 1496  449 158 265 16264
9 1997 6274 4605 3401 1480 1798  474 187 310 18532
```

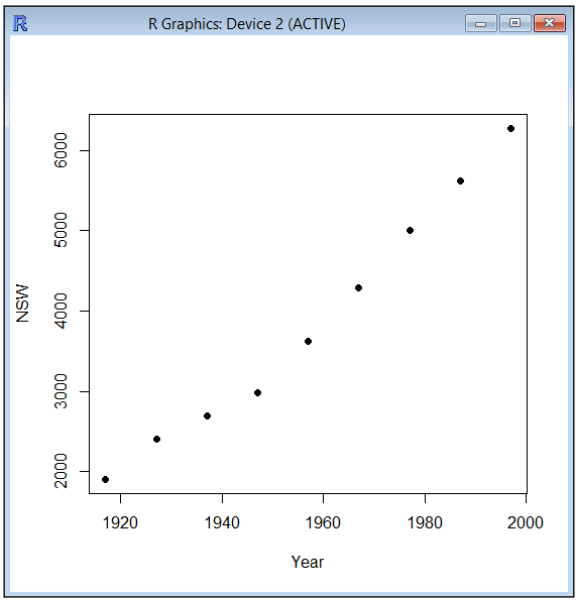
We can print the column names using the following command:

```
> names(austpop)
 [1] "Year"  "NSW"   "Vic."  "Qld"   "SA"    "WA"    "Tas."  "NT"
"ACT"   "Aust."
```

Next, we can plot a graph. Let's plot a graph of the population in the state of NSW for the past few years. In order to do so, we have to execute the following command. The input to this will appear in the form of the `x` axis, `y` axis, and the size of the graph:

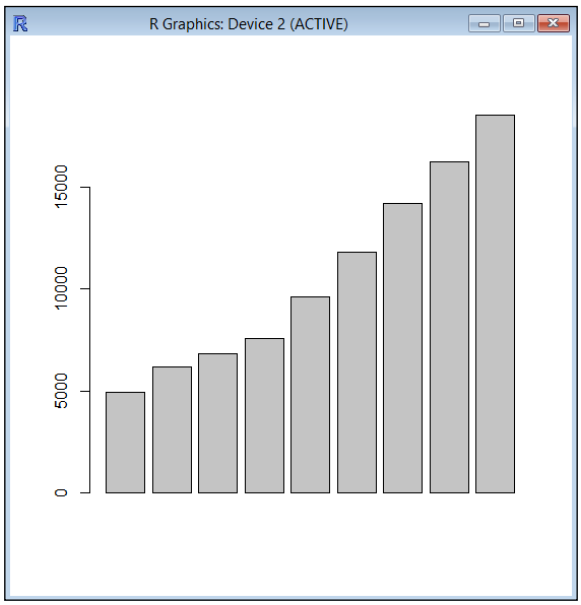
```
>plot(NSW ~ Year, data=austpop, pch=16)
```

The graph will look like this:



There are various options that are available for different types of graphs. This is how you can plot a bar graph for the total population of this state over the last few years:

```
> barplot(austpop$Aust.)
```



You can also easily get the summary of the values for each state by executing the following command:

```
> summary(austpop)
      Year           NSW            Vic.           Qld             SA
WA
 Min.   :1917   Min.   :1904   Min.   :1409   Min.   : 683   Min.   :
440.0   Min.   : 306
 1st Qu.:1937   1st Qu.:2693   1st Qu.:1853   1st Qu.: 993   1st Qu.:
589.0   1st Qu.: 457
 Median :1957   Median :3625   Median :2656   Median :1413   Median :
873.0   Median : 688
 Mean   :1957   Mean   :3866   Mean   :2847   Mean   :1664   Mean   :
931.3   Mean   : 858
 3rd Qu.:1977   3rd Qu.:5002   3rd Qu.:3837   3rd Qu.:2130   3rd
Qu.:1286.0   3rd Qu.:1204
 Max.   :1997   Max.   :6274   Max.   :4605   Max.   :3401   Max.
:1480.0   Max.   :1798
      Tas.             NT            ACT             Aust.
 Min.   :193.0   Min.   :  4   Min.   :  3.0   Min.   : 4941
 1st Qu.:233.0   1st Qu.:  6   1st Qu.: 11.0   1st Qu.: 6836
 Median :326.0   Median : 21   Median : 38.0   Median : 9640
 Mean   :325.9   Mean   : 62   Mean   :107.7   Mean   :10663
 3rd Qu.:415.0   3rd Qu.:104   3rd Qu.:214.0   3rd Qu.:14192
 Max.   :474.0   Max.   :187   Max.   :310.0   Max.   :18532
```

In case you want the summary of only one column, you can get it by executing the following command:

```
> summary(austpop$Aust.)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   4941    6836    9640   10660   14190   18530
```

You can also perform operations such as sorting the given data, as shown here:

```
> sort_pop <- sort(austpop$ACT)
> sort_pop
[1]   3   8  11  17  38 103 214 265 310
```

How it works...

As mentioned earlier, R has been built keeping statisticians in mind, so whatever they require on a daily basis is provided by R. There are various packages also available in case you wish to do something different. The working of the preceding commands is self-explanatory.

Performing Twitter Sentiment Analytics using R

In an earlier chapter, we saw how to perform Twitter sentiment analytics using Hive and Hadoop. In this recipe, we are going to take a look at how to do this using R.

Getting ready

To perform this recipe, you should have R installed on your machine. You should also have a Twitter account and an application that has an API key, API secret, Access Token, and an Access Secret with you so that you can receive tweets in real time.

How to do it...

To get started, first of all, we need to install certain R packages, which will be required in this recipe. The following are the commands:

```
>install.packages("twitteR")
>install.packages("plyr")
>install.packages("stringr")
>install.packages(c("devtools", "rjson", "bit64", "httr"))
```

Once the installation is complete, load the following packages:

```
>library(devtools)
>library(twitteR)
```

Next, we need to provide the keys provided that are by Twitter on its application page, as follows:

```
>api_key <-  "XXXXXX"
>api_secret <- "XXXXXX"
>access_token <- "XXXXXX"
>access_secret <- "XXXXXXX"
>ssetup_twitter_oauth(api_key, api_secret, access_token, access_
secret)
```

Here, instead of `XXXX`, add your own keys.

If the keys that are provided by you are correct, you will not see any errors. If there are any missing keys, you will see an appropriate error message.

Next, we need some dictionary words, which are segregated as positive or negative, in order to determine the sentiment of each tweet. So, we download a list of words from `http://www.cs.uic.edu/~liub/FBS/opinion-lexicon-English.rar` and store them in a certain place. Download, extract, and save these files.

Then, we load these lists of words into objects, as shown here:

```
>pos.from.file = scan('H:/opinion-lexicon-English/positive-words.
txt',what='character', comment.char=';')
>neg.from.file = scan('H:/opinion-lexicon-English/negative-words.
txt',what='character', comment.char=';')
```

We also remove the comments that are labelled as headers in the text files.

We can also add some more words if they are not already part of the given files, as shown here:

```
>pos.words = c(pos.from.file,'upgrade', 'love','achievement','impress
ed')
>neg.words = c( neg.from.file, 'wtf', 'wait', 'waiting','hate', 'die')
```

Now, we are going to download the tweets using the `searchTwitter` method of the `TwitterR` package:

```
hadoop.tweets = searchTwitter('#hadoop', n=1000)
```

The preceding command will download 1,000 tweets with the `hadoop` hash tag and save it as the `hadoop.tweets` object.

You can check whether the data is imported properly:

```
> hadoop.tweets
[[1]]
[1] "bviallet: RT @IBMAnalytics: See how the video game industry is
using analytics to create happy customers: https://t.co/LcyTEFqUrT
#Hadoop https://t.c…"

[[2]]
[1] "sheltonmkagande: RT @SAS_Cares: Learn five ways #SAS gets to data
in #hadoop https://t.co/yMsXTVw7Dd #SASusers"

[[3]]
[1] "thinkittraining: Secondary Indexing for MapR-DB using
Elasticsearch @https://www.mapr.com/blog #Hadoop #Mapr #training"

[[4]]
[1] "BigDataTweetBot: RT @andnegr: https://t.co/Yd87vicDlz @SASitaly
#BigData #Hadoop @ClouderaITA"

[[5]]
[1] "BigDataTweetBot: RT @techjunkiejh: Is Apache #Hadoop the only
option to implement #BigData ? https://t.co/b4z7ZSHTiJ https://t.co/
mGRFXqS6Dx"
```

```
[[6]]
[1] "shakamunyi: RT @andnegr: https://t.co/Yd87vicDlz @SASitaly
#BigData #Hadoop @ClouderaITA"

[[7]]
[1] "shakamunyi: RT @techjunkiejh: Is Apache #Hadoop the only option
to implement #BigData ? https://t.co/b4z7ZSHTiJ https://t.co/
mGRFXqS6Dx"

[[8]]
[1] "andnegr: https://t.co/Yd87vicDlz @SASitaly #BigData #Hadoop @
ClouderaITA"

[[9]]
[1] "techjunkiejh: Is Apache #Hadoop the only option to implement
#BigData ? https://t.co/b4z7ZSHTiJ https://t.co/mGRFXqS6Dx"

[[10]]
[1] "gitsacademy: #fact of the day\n#Hadoop was created by Doug
Cutting (while at #Yahoo) and named it after his son's elephant
https://t.co/RUnDJzSpml"
```

Now, we want only the tweet text from the preceding data, so we execute the following command:

```
>hadoop.text = laply(hadoop.tweets, function(t) t$getText() )
```

We will be writing one algorithm that takes each tweet and then computes how many positive and negative words it has. It will give a score based on the sentiment of the overall tweet:

```
score.sentiment = function(sentences, pos.words, neg.words,
.progress='none')
{
require(plyr)
require(stringr)

#Get vectors from the list
scores = laply(sentences, function(sentence, pos.words, neg.words) {

# clean up sentences, remove punctuations etc.
sentence = gsub('[[:punct:]]', '', sentence)
sentence = gsub('[[:cntrl:]]', '', sentence)
sentence = gsub('\\d+', '', sentence)
# and convert to lower case:
sentence = tolower(sentence)

# split into words.
```

```
word.list = str_split(sentence, '\\s+')
words = unlist(word.list)

# compare our words to the dictionaries of positive & negative terms
pos.matches = match(words, pos.words)
neg.matches = match(words, neg.words)

# match() returns the position of the matched term or NA
# we just want a TRUE/FALSE so we remove all NA
pos.matches = !is.na(pos.matches)
neg.matches = !is.na(neg.matches)

# and conveniently enough, TRUE/FALSE will be treated as 1/0 by sum():
score = sum(pos.matches) - sum(neg.matches)

return(score)
}, pos.words, neg.words, .progress=.progress )

scores.df = data.frame(score=scores, text=sentences)
return(scores.df)
}
```

We can test this algorithm by using some sample text, as shown here:

```
> sample = c("You're awesome and I love you",
"I hate and hate and hate. So angry. Die!",
"Impressed and amazed: you are peerless in your
achievement of unparalleled mediocrity.")

> result = score.sentiment(sample, pos.words, neg.words)
```

Now, we can see the results:

```
> result
  score                                                    text
1     2
You're awesome and I love you
2    -5                                                I hate and hate
and hate. So angry. Die!
3     3 Impressed and amazed: you are peerless in your\nachievement of
unparalleled mediocrity.
```

This means that our algorithm works, so we will test on real Twitter data now:

```
>hadoop.scores = score.sentiment(hadoop.text, pos.words, neg.words,
.progress='text')
```

Here are the results:

```
>  hadoop.scores
   score
text
1      1 RT @IBMAnalytics: See how the video game industry is using
analytics to create happy customers: https://t.co/LcyTEFqUrT #Hadoop
https://t.c…
2      0                                                    RT @SAS_Cares:
Learn five ways #SAS gets to data in #hadoop https://t.co/yMsXTVw7Dd
#SASusers
3      0                                          Secondary Indexing for
MapR-DB using Elasticsearch @https://www.mapr.com/blog #Hadoop #Mapr
#training
4      0
RT @andnegr: https://t.co/Yd87vicDlz @SASitaly #BigData #Hadoop @
ClouderaITA
5      0                  RT @techjunkiejh: Is Apache #Hadoop the only
option to implement #BigData ? https://t.co/b4z7ZSHTiJ https://t.co/
mGRFXqS6Dx
6      0
RT @andnegr: https://t.co/Yd87vicDlz @SASitaly #BigData #Hadoop @
ClouderaITA
7      0                  RT @techjunkiejh: Is Apache #Hadoop the only
option to implement #BigData ? https://t.co/b4z7ZSHTiJ https://t.co/
mGRFXqS6Dx
8      0
https://t.co/Yd87vicDlz @SASitaly #BigData #Hadoop @ClouderaITA
9      0                                       Is Apache #Hadoop the only
option to implement #BigData ? https://t.co/b4z7ZSHTiJ https://t.co/
mGRFXqS6Dx
10     0        #fact of the day\n#Hadoop was created by Doug Cutting
(while at #Yahoo) and named it after his son's elephant https://t.co/
RUnDJzSpml
```

This way, you can perform sentiment analytics for any kind of Twitter data.

How it works...

Our sentiment analysis algorithm is very simple. We first get the sentence, break it into words, remove any punctuation, and so on. Later, we compare each word with a predefined list of positive and negative words. Next, we subtract the number of negative words from the number of positive words and get the score. If the score is a positive number, then it means that the sentiment of the tweet is positive, and if the score is negative, it means that the tweet is negative. If the score is zero, this means that it is a neutral tweet.

This algorithm can be extended further to understand sarcasm.

Performing Predictive Analytics using R

In the previous recipe, we talked about how to perform sentiment analytics using R. In this recipe, we are going to take a look at how to perform predictive analytics using R. Here, we will be using the IRIS flower classification data in order to predict its species based on the features. You can learn more about this at `https://en.wikipedia.org/wiki/Iris_flower_data_set`.

Getting ready

To perform this recipe, you should have R installed on your machine.

How to do it...

To get started, we need to install an R package called `e1071`:

```
>install.packages("e1071")
```

This package contains the IRIS flower dataset. So, we load the library and then load the data into it:

```
>library(e1071)
>data(iris)
```

You can check whether the data is loaded properly or not by executing the following command:

```
>iris
```

In this example, we are going to use the Naive Bayes algorithm to classify the data into specifies. So, now we have to train the model using Naive Bayes, as shown here:

```
>model<- naiveBayes(Species~., data=iris)
```

You can print the model in order to take a look at how it has calculated probabilities:

```
>model

   Naive Bayes Classifier for Discrete Predictors

   Call:
   naiveBayes.default(x = X, y = Y, laplace = laplace)

   A-priori probabilities:
   Y
       setosa versicolor  virginica
    0.3333333  0.3333333  0.3333333
```

```
Conditional probabilities:
            Sepal.Length
Y             [,1]      [,2]
  setosa     5.006 0.3524897
  versicolor 5.936 0.5161711
  virginica  6.588 0.6358796

            Sepal.Width
Y             [,1]      [,2]
  setosa     3.428 0.3790644
  versicolor 2.770 0.3137983
  virginica  2.974 0.3224966

            Petal.Length
Y             [,1]      [,2]
  setosa     1.462 0.1736640
  versicolor 4.260 0.4699110
  virginica  5.552 0.5518947

            Petal.Width
Y             [,1]      [,2]
  setosa     0.246 0.1053856
  versicolor 1.326 0.1977527
  virginica  2.026 0.2746501

> summary(model)
        Length Class  Mode
apriori 3      table  numeric
tables  4      -none- list
levels  3      -none- character
call    4      -none- call
```

Now, we test this model against actual data:

```
>predictions <- predict(model, iris[,1:4])
```

This will give us the model and data to be used for predictions. On execution, the predictions object will have these predicted values:

```
> predictions
```

```
  [1] setosa     setosa     setosa     setosa     setosa     setosa
setosa
  [8] setosa     setosa     setosa     setosa     setosa     setosa
setosa
 [15] setosa     setosa     setosa     setosa     setosa     setosa
setosa
```

```
 [22] setosa     setosa     setosa     setosa     setosa     setosa
setosa
 [29] setosa     setosa     setosa     setosa     setosa     setosa
setosa
 [36] setosa     setosa     setosa     setosa     setosa     setosa
setosa
 [43] setosa     setosa     setosa     setosa     setosa     setosa
setosa
 [50] setosa     versicolor versicolor virginica  versicolor
versicolor versicolor
 [57] versicolor versicolor versicolor versicolor versicolor
versicolor versicolor
 [64] versicolor versicolor versicolor versicolor versicolor
versicolor versicolor
 [71] virginica  versicolor versicolor versicolor versicolor
versicolor versicolor
 [78] virginica  versicolor versicolor versicolor versicolor
versicolor versicolor
 [85] versicolor versicolor versicolor versicolor versicolor
versicolor versicolor
 [92] versicolor versicolor versicolor versicolor versicolor
versicolor versicolor
 [99] versicolor versicolor virginica  virginica  virginica  virginica
virginica
[106] virginica  versicolor virginica  virginica  virginica  virginica
virginica
[113] virginica  virginica  virginica  virginica  virginica  virginica
virginica
[120] versicolor virginica  virginica  virginica  virginica  virginica
virginica
[127] virginica  virginica  virginica  virginica  virginica  virginica
virginica
[134] versicolor virginica  virginica  virginica  virginica  virginica
virginica
[141] virginica  virginica  virginica  virginica  virginica  virginica
virginica
[148] virginica  virginica  virginica
Levels: setosa versicolor virginica
```

Next, we can build the confusion matrix in order to show how good our model is:

```
> table(predictions, iris$Species)
```

```
predictions  setosa versicolor virginica
  setosa         50          0         0
  versicolor      0         47         3
  virginica       0          3        47
```

This way, you can build a model using various algorithms and validate them.

How it works...

R is a platform that allows us to perform various operations. The package we installed contains implementations of various well-know algorithms; here, we are just making use of those that are useful for real-world problem solving.



You can do read more about the Naive Bayes algorithm at `https://en.wikipedia.org/wiki/Naive_Bayes_classifier`.

9
Integration with Apache Spark

In this chapter, we'll take a look at the following recipes:

- Running Spark standalone
- Running Spark on YARN
- Performing Olympics Athletes analytics using the Spark Shell
- Creating Twitter trending topics using Spark Streaming
- Analyzing Parquet files using Spark
- Analyzing JSON data using Spark
- Processing graphs using Graph X
- Conducting predictive analytics using Spark `MLib`

Introduction

In the previous chapter, we talked about how to use Mahout and R to solve machine learning problems. In this chapter, we are going to talk about the latest sensation in the Big Data industry called Apache Spark. By now, everyone is aware, and has acknowledged the power of Apache Spark. This is a general and fast engine that processes large-scale data. It provides high-level APIs in Java, Scala, Python, and R. Spark can perform batch processing as well as stream processing. In this chapter, we are going to explore certain important topics related to Apache Spark such as batch processing, Spark SQL, streaming processing, machine learning with `MLib`, and graph processing using Spark's `GraphX` library. So, let's get started.

Running Spark standalone

Spark can be executed in various modes. To get started, we are going to take a look at how to install Apache Spark on a standalone machine.

Getting ready

To perform this recipe, you should download the latest version of Spark. For this recipe, I am using Apache Spark 1.6.0. You can visit the download page at `http://spark.apache.org/downloads.html`.

How to do it...

Apache Spark is a computation engine. It has a built-in cluster manager. It can also use other cluster managers such as YARN/Mesos and so on. In this recipe, we are going to use the built-in resource manager that's provided by Spark:

1. Copy the downloaded Spark binary to a desired location.
2. Extract the tar ball:

   ```
   $sudo tar –xzfspark-1.6.0-bin-hadoop2.6.tgz
   ```

3. Rename the `spark` folder for ease of use:

   ```
   $sudo mv spark-1.6.0-bin-hadoop2.6 spark
   ```

4. Add environment variables in `~/.bashrc`:

   ```
   export SPARK_HOME=/usr/local/spark

   export PATH=$PATH:$SPARK_HOME/bin
   ```

5. Source `~/.bashrc` to make the changes effective:

   ```
   $source ~/.bashrc
   ```

6. In case you want to run Spark in cluster mode, you need to repeat the preceding commands on each node. Now, we are all set to use Apache Spark.

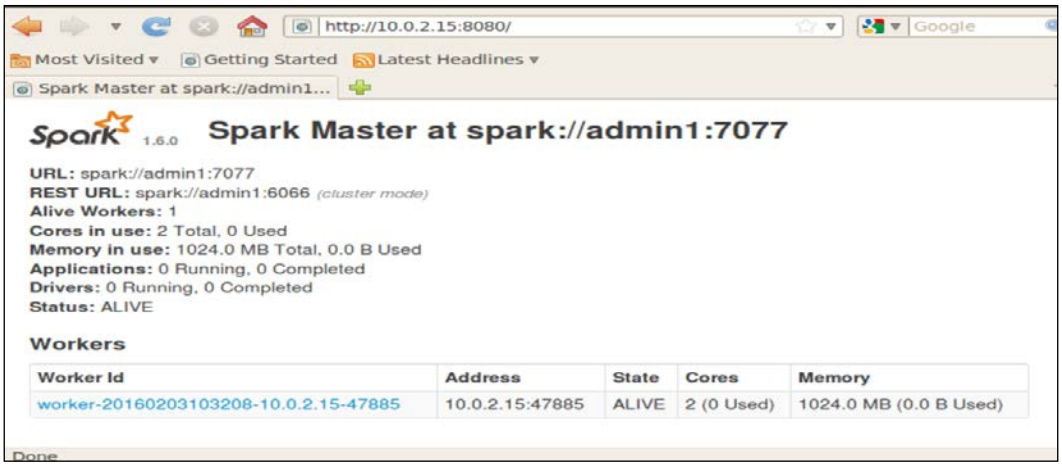
   To start the master, use this code:

   ```
   $/usr/local/spark/sbin/start-master.sh
   ```

   To start the slaves, use this code:

   ```
   $/usr/local/spark/sbin/start-slave.sh spark://<masterhost>:<port>
   ```

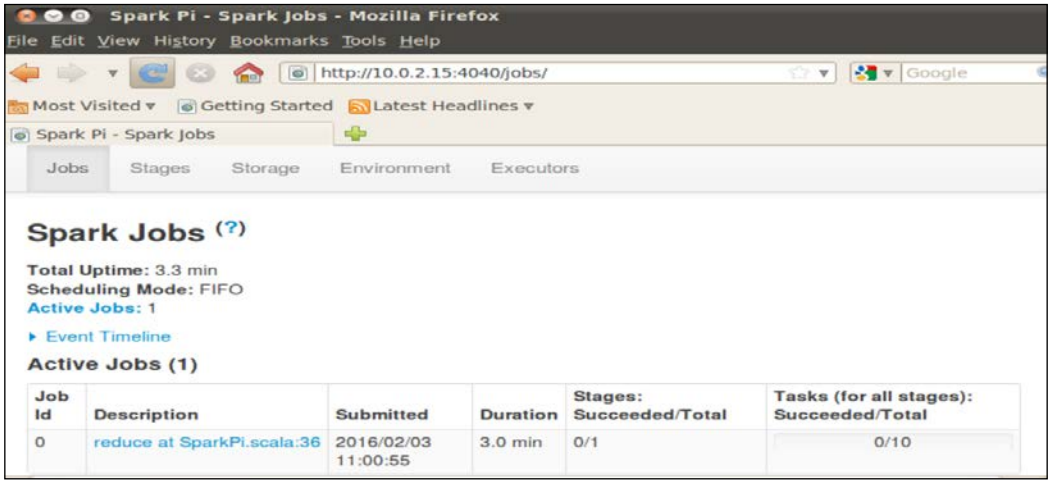
7. Now, visit `http://localhost:8080/` to take a look at Spark Master's UI and the number of workers that are registered with it.



8. We can now try running one sample Spark application in order to check whether everything is working. The following command will launch a Spark application, which will calculate the value for Pi:

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master spark://<master>:7077 \
--deploy-mode cluster \
lib/spark-examples*.jar \
    10
```

9. While the application is being executed, you should be able to take a look at all the stages that are involved as well as other things in the application web UI. To do this, visit `http://localhost:4040`.



How it works...

Spark works in the master-slave architecture. Here, we have started Spark Master and a worker. Spark master acts like a coordinator between the slaves. Once you submit the application, Spark creates a directed acyclic graph of operations. It decides which stages can be executed in parallel and which need to be executed sequentially. Each task is executed by workers, and the status is reported back to the master.

Running Spark on YARN

In the previous recipe, we took a look at how to use Spark's built-in cluster manager; in this recipe, we are going to explore how to use YARN as a cluster manager to execute the Spark application.

Getting ready

To perform this recipe, you should have a running Hadoop cluster. You should also have performed the previous recipe.

How to do it...

As mentioned in the previous recipe, we can either use Spark's built-in cluster manager, or we can use an external cluster manager such as YARN. In order to execute the Spark application on YARN, we need to edit `SPARK_HOME/conf/spark-env.sh`, and add the following properties to it:

```
export HADOOP_CONF_DIR=/usr/local/hadoop/etc/hadoop
export YARN_CONF_DIR=/usr/local/hadoop/etc/Hadoop
```

Here, `/usr/local/hadoop/etc/hadoop` is the directory where we have our Hadoop and YARN configuration files.

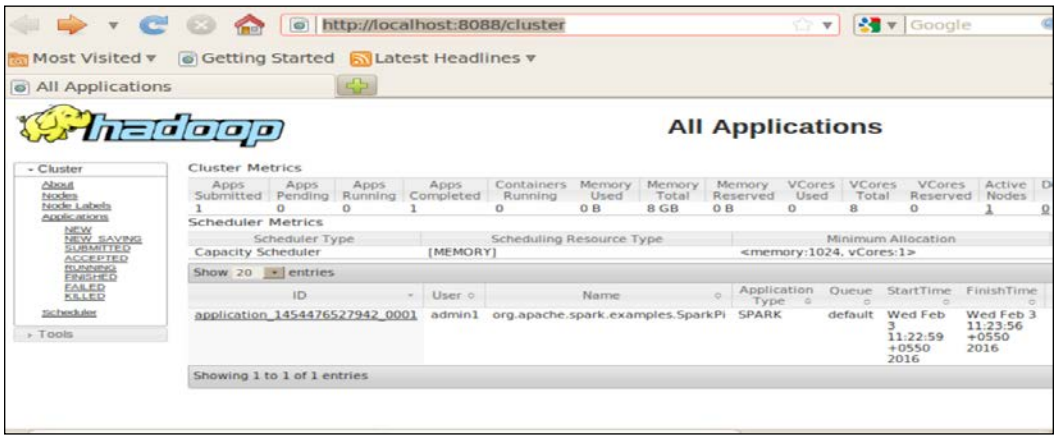
Now, let's execute the same Spark application on YARN using the following command:

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master yarn \
    --deploy-mode cluster \
lib/spark-examples*.jar \
    10
```

Unlike the previous recipe, we don't need to give YARN IP—PORT, but spark will take it from YARN's configuration directory. We need to give YARN the keyword for the master property.

Now, we can use the YARN application UI in order to check the progress of the job so far.

Go to `http://localhost:8088/cluster` for the following output:



How it works...

Working on the Spark application is pretty similar to what was explained in the previous recipe; the only difference is the executors, Resource Manager, and the Node Managers. Spark submits the application to the YARN application master, and the application master then asks the Resource Managers and Node Managers to execute the tasks.

Performing Olympics Athletes analytics using the Spark Shell

Spark supports an interactive Scala-based shell, which can be used to process data as and when we receive actionable commands. In this recipe, we are going to analyze one sample dataset, which contains information about the athletes that have participated in the Olympics.

Getting ready

To perform this recipe, you should have Hadoop and Spark installed. You also need to install Scala. I am using Scala 2.11.0.

How to do it...

First of all, you need to download data from `https://github.com/deshpandetanmay/hadoop-real-world-cookbook/blob/master/data/OlympicAthletes.csv`, and store it in HDFS using the following commands:

```
$hadoop fs –mkdir /athletes
$hadoop fs –put OlympicAthletes.csv /athletes
```

The following is some sample data from the file for your reference. The data comma-separated file contains various columns in a sequence such as the athlete name, country, year, gold, silver, bronze, and the total number of medals won by each athlete:

```
Yang Yilin,China,2008,Gymnastics,1,0,2,3
Leisel Jones,Australia,2000,Swimming,0,2,0,2
Go Gi-Hyeon,South Korea,2002,Short-Track Speed Skating,1,1,0,2
Chen Ruolin,China,2008,Diving,2,0,0,2
Katie Ledecky,United States,2012,Swimming,1,0,0,1
Ruta Meilutyte,Lithuania,2012,Swimming,1,0,0,1
Dániel Gyurta,Hungary,2004,Swimming,0,1,0,1
Arianna Fontana,Italy,2006,Short-Track Speed Skating,0,0,1,1
Olga Glatskikh,Russia,2004,Rhythmic Gymnastics,1,0,0,1
Kharikleia Pantazi,Greece,2000,Rhythmic Gymnastics,0,0,1,1
Kim Martin,Sweden,2002,Ice Hockey,0,0,1,1
Kyla Ross,United States,2012,Gymnastics,1,0,0,1
Gabriela Dragoi,Romania,2008,Gymnastics,0,0,1,1
Tasha Schwikert-Warren,United States,2000,Gymnastics,0,0,1,1
Yang Yun,China,2000,Gymnastics,0,0,1,1
Sophie Lamon,Switzerland,2000,Fencing,0,1,0,1
Alejandra Orozco,Mexico,2012,Diving,0,1,0,1
Yuliya Koltunova,Russia,2004,Diving,0,1,0,1
Shawn Johnson,United States,2008,Gymnastics,1,3,0,4
Carly Patterson,United States,2004,Gymnastics,1,2,0,3
```

Next, we start the Apache Spark shell using the following command:

```
$cd /usr/local/spark
$./bin/spark-shell
```

This will start the Spark shell, and it will prompt Scala. Now, to get started, we will first initialize the spark context and start with loading files into it from HDFS:

```
scala>valatheletes =
sc.textFile("hdfs://localhost:9000/athletes/OlympicAthletes.csv")
```

Let's check whether the data is loaded properly by executing the following command:

```
scala>atheletes.take(10).foreach(println)
~
    Yang Yilin,China,2008,Gymnastics,1,0,2,3
    Leisel Jones,Australia,2000,Swimming,0,2,0,2
    Go Gi-Hyeon,South Korea,2002,Short-Track Speed Skating,1,1,0,2
    Chen Ruolin,China,2008,Diving,2,0,0,2
    Katie Ledecky,United States,2012,Swimming,1,0,0,1
    Ruta Meilutyte,Lithuania,2012,Swimming,1,0,0,1
    Dániel Gyurta,Hungary,2004,Swimming,0,1,0,1
    Arianna Fontana,Italy,2006,Short-Track Speed Skating,0,0,1,1
    Olga Glatskikh,Russia,2004,Rhythmic Gymnastics,1,0,0,1
    Kharikleia Pantazi,Greece,2000,Rhythmic Gymnastics,0,0,1,1
    ~
```

It will print the first 10 athlete records. Next, let's count the total number of records in the dataset. That can be done by the following command:

```
scala>atheletes.count
~
    16/02/03 13:05:28 INFO DAGScheduler: Job 2 finished: count at
    <console>:30, took 0.188182 s
    res2: Long = 8613
    ~
```

Now, let's filter out athletes from India using the following commands:

```
scala>valindianAthletes = atheletes.filter(_.split(",")(1) == "India").
cache
```

We can print the records using this command:

```
scala>indianAthletes.foreach(println)
    Vijender Singh,India,2008,Boxing,0,0,1,1
    Saina Nehwal,India,2012,Badminton,0,0,1,1
    Sushil Kumar,India,2008,Wrestling,0,0,1,1
    Karnam Malleswari,India,2000,Weightlifting,0,0,1,1
    Abhinav Bindra,India,2008,Shooting,1,0,0,1
    Vijay Kumar,India,2012,Shooting,0,1,0,1
    Yogeshwar Dutt,India,2012,Wrestling,0,0,1,1
```

```
Sushil Kumar,India,2012,Wrestling,0,1,0,1
Gagan Narang,India,2012,Shooting,0,0,1,1
M. C. Mary Kom,India,2012,Boxing,0,0,1,1
Rajyavardhan Rathore,India,2004,Shooting,0,1,0,1
```

Next, we would like to find out the number of athlete records on the basis of their countries; thus, we will need to execute the following command:

```
scala>valatheletesTuples = atheletes.map(line =>line.split(","))

scala>valatheletesKeyValuePairs = atheletesTuples.map(line =>
(line(1), 1))

scala>valgroupByCountry = atheletesKeyValuePairs.reduceByKey(_+_,
1).collect
```

In the preceding commands, we first split each record by a comma and then emit the country as the key and 1 as the value. Next, we execute reduce by operation to get the final results. The output is shown here:

```
groupByCountry: Array[(String, Int)] = Array((Australia,524),
(Brazil,217), (Mexico,38), (Uzbekistan,19), (France,287),
(South Korea,274), (Finland,112), (Germany,552), (Macedonia,1),
(Montenegro,14), (Uruguay,1), (Cuba,188), (Bahrain,1), (North
Korea,21), (Sweden,167), (Vietnam,2), (Serbia,31), (Iran,24),
(Slovakia,33), (Venezuela,4), (Denmark,89), (Chinese Taipei,20),
(Saudi Arabia,6), (Paraguay,17), (Serbia and Montenegro,38),
(Sudan,1), (Botswana,1), (Greece,59), (Italy,307), (Slovenia,24),
(Iceland,15), (Netherlands,286), (Spain,195), (Kuwait,2),
(Hong Kong,3), (Mongolia,10), (Malaysia,3), (Kazakhstan,42),
(Ukraine,137), (Romania,97), (Egypt,8), (Indonesia,22), (Latvia,17),
(Eritrea,1), (Armenia,10), (Norway,158), (Thailand,18), (Poland,73),
(Tajikistan,3), (Afghanistan,2)
```

Similarly, we can also execute commands to get the number of athletes by the sport they play:

```
scala>valsportsWiseKeyValuePairs = atheletesTuples.map(line =>
(line(3), 1))

scala>valgroupBySport = sportsWiseKeyValuePairs.reduceByKey(_+_,
1).collect
```

The output for this is as follows:

```
groupBySport: Array[(String, Int)] = Array((Basketball,287),
(Judo,224), (Football,407), (Modern Pentathlon,24), (Luge,36),
(Table Tennis,67), (Swimming,487), (Synchronized Swimming,109),
(Cycling,261), (Athletics,687), (Baseball,216), (Diving,113), (Cross
Country Skiing,128), (Waterpolo,306), (Equestrian,157), (Skeleton,18),
(Hockey,386), (Alpine Skiing,61), (Rhythmic Gymnastics,84),
(Handball,351), (Softball,134), (Gymnastics,194), (Nordic Combined,39),
(Canoeing,295), (Weightlifting,180), (Wrestling,245), (Sailing,210),
(Taekwondo,112), (Ski Jumping,40), (Trampoline,24), (Short-Track Speed
Skating,96), (Boxing,188), (Biathlon,94), (Tennis,71), (Triathlon,24),
(Figure Skating,54), (Badminton,91), (Curling,82), (Rowing,567),
(Snowboarding,48), (Beach Volleyball,48), (Ice Hockey,384)
```

This way, you can perform various operations to analyze data interactively.

How it works...

The Spark shell gives us interactive ways to analyze data. Internally, it's the same Spark application that gets executed, as explained in the previous recipes.

Creating Twitter trending topics using Spark Streaming

Spark supports various modules. In this recipe, we are going to take a look at its SQL module, which allows the execution of SQL queries through a Spark application. We are going to explore how to access Hive from Spark and perform analytics.

Getting ready

To perform this recipe, you should have Hadoop and Spark installed. You also need to install Scala. I am using Scala 2.11.0. You should also have Hive installed.

How to do it...

To use Hive from Spark, we are going to write one sample spark application in Scala. You can choose an IDE of your choice. Since we are going to write the application in Scala, you will need Scala and SBT installed on your machine.

First of all, I am going to create a folder called `HiveFromSpark`, and add the following files to it:

```
HiveFromSpark\src\main\scala\com\demo\HiveFromSpark.scala
HiveFromSpark\ project\assembly.sbt
HiveFromSpark\build.sbt
HiveFromSpark\src\main\resources\emp.txt
```

Then, we set `build.sbt` to add dependencies, as shown here:

```
name := "HiveFromSpark"

version := "1.0"

scalaVersion := "2.11.7"

libraryDependencies ++= Seq(

 "com.google.guava" % "guava" % "14.0",
 "org.apache.spark" %% "spark-core" % "1.6.0" % "provided",
 "org.apache.spark" %% "spark-sql"  % "1.6.0" % "provided",
 "org.apache.spark" %% "spark-hive" % "1.6.0" % "provided"
)
resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

Next, we create the `assembly.sbt` file so that it is able to generate the fat jar:

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.12.0")
```

We then add `emp.txt`, which is a simple file that we are going to use during execution:

```
16,john
17,robert
18,andrew
19,katty
21,tom
22,tim
23,james
24,paul
27,edward
29,alan
31,kerry
34,terri
```

Now, we will write a Scala class, which will create a Hive table and load `emp.txt` into the table. Later on, we will perform various operations in this table. The following code is from `HiveFromSpark.scala`:

```
packagecom.demo

importcom.google.common.io.ByteStreams
importcom.google.common.io.Files


importjava.io.File

importorg.apache.spark._
importorg.apache.spark.sql._
importorg.apache.spark.sql.hive.HiveContext

objectHiveFromSpark {
case class Record(id: Int, name: String)

  // Copy emp.txt file from classpath to temporary directory
valempStream = HiveFromSpark.getClass.getResourceAsStream("/emp.txt")
valempFile = File.createTempFile("emp", "txt")
empFile.deleteOnExit()
ByteStreams.copy(empStream, Files.newOutputStreamSupplier(empFile))

def main(args: Array[String]) {
valsparkConf = new SparkConf().setAppName("HiveFromSpark")
valsc = new SparkContext(sparkConf)


valhiveContext = new HiveContext(sc)
importhiveContext.implicits._
importhiveContext.sql

sql("CREATE TABLE IF NOT EXISTS empSpark (id INT, name STRING) ROW
FORMAT DELIMITED FIELDS TERMINATED BY ','")
sql(s"LOAD DATA LOCAL INPATH '${empFile.getAbsolutePath}' INTO TABLE
empSpark")

    // Queries are expressed in HiveQL
println("Result of 'SELECT *': ")
sql("SELECT * FROM empSpark").collect().foreach(println)

    // Aggregation queries are also supported.
```

```
val count = sql("SELECT COUNT(*) FROM empSpark").collect().head.
getLong(0)
println(s"COUNT(*): $count")

    // The results of SQL queries are themselves RDDs and support all
normal RDD functions.  The
    // items in the RDD are of type Row, which allows you to access
each column by ordinal.
valrddFromSql = sql("SELECT id, name FROM empSpark WHERE id < 20 ORDER
BY id")

println("Result of RDD.map:")
valrddAsStrings = rddFromSql.map {
case Row(id: Int, name: String) =>s"Key: $id, Value: $name"
    }
    // You can also register RDDs as temporary tables within a
HiveContext.
valrdd = sc.parallelize((1 to 100).map(i => Record(i, s"val_$i")))
rdd.toDF().registerTempTable("records")

    // Queries can then join RDD data with data stored in Hive.
println("Result of SELECT *:")
sql("SELECT * FROM records r JOIN src s ON r.key = s.key").collect().
foreach(println)

sc.stop()
  }
}
```

In order to execute the preceding code, we run the following commands to create the assembled fat jar of this application:

```
sbt compile
sbt package
sbt assembly
```

This will create a fat jar, called `HiveFromSpark-assembly-1.0.jar`, in the `target\scala-2.11` folder.

Now, we take this jar and copy it in a Spark cluster. Execute the following command to start the Spark application:

```
/usr/local/spark/bin/spark-submit \
  --class "com.demo.HiveFromSpark" \
  --master yarn \
  HiveFromSpark-assembly-1.0.jar
```

If you want to run this on the Spark default master slave, then you will have to execute the following command:

```
/usr/local/spark/bin/spark-submit \
  --class "com.demo.HiveFromSpark" \
  --master spark://admin1:7077 \
  HiveFromSpark-assembly-1.0.jar
```

This will start the Spark application in YARN. This will create a table in Hive, called `empspark`, where data from `emp.txt` will be loaded into it.

You can go to the Hive prompt and check whether this table, called `empspark`, exists:

```
hive>descempspark;
hive>select * from empspark;
```

How it works...

Spark SQL is an interpreter between the underlying technology and Spark. It sends queries and accepts results. All the features of Spark RDDs are available here.

Twitter trending topics using Spark streaming

In the previous recipe, we took a look at the SQL integrations of Spark. In this recipe, we are going to explore yet another powerful module called Spark Streaming. As the name suggests, Spark Streaming can listen to a stream of events and process data as and when it arrives.

Getting ready

To perform this recipe, you should have Hadoop and Spark installed. You also need to install Scala. I am using Scala 2.11.0. You should also have a Twitter account and some keys and tokens.

How to do it...

Spark streaming supports input from various sources such as Flume, HDFS, Kafka, Twitter, and so on. In this recipe, we are going to use Spark Streaming's Twitter source where we will be listening to streaming tweets and compute the top trending topics on Twitter.

To perform this recipe, we are going to write one Spark Streaming application in Scala.

In order to create the application, I am creating a folder called `TwitterSpark`, and it will have the following files in it:

```
TwitterSpark\build.sbt
TwitterSpark\src\main\scala\com\demo\TwitterPopularTags.scala
TwitterSpark\project\assembly.sbt
```

The contents of `build.sbt` are as follows:

```
name := "TwitterSpark"

version := "1.0"

scalaVersion := "2.11.7"

mergeStrategy in assembly <<= (mergeStrategy in assembly) { (old) =>
   {
casePathList("META-INF", xs @ _*) =>MergeStrategy.discard
case x =>MergeStrategy.first
   }
}

libraryDependencies ++= Seq(

 "com.google.guava" % "guava" % "14.0",
 "org.apache.spark" %% "spark-core" % "1.6.0" % "provided",
 "org.apache.spark" %% "spark-streaming" % "1.6.0" % "provided",
 "org.apache.spark" %% "spark-streaming-twitter" % "1.6.0"
)

resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

Here, we need to add dependencies for Spark Streaming and Spark Streaming Twitter.

Next, `assembly.sbt` will show you the following:

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.12.0")
```

We need to add a Scala class, which fetches Twitter streams and performs computations on them:

```
packagecom.demo


importorg.apache.spark.streaming.{Seconds, StreamingContext}
importorg.apache.spark.SparkContext._
```

```
importorg.apache.spark.streaming.twitter._
importorg.apache.spark.SparkConf

objectTwitterPopularTags {
def main(args: Array[String]) {


valconsumerKey = "XXXX"
valconsumerSecret = "XXXX"
valaccessToken = "XXXX"
valaccessTokenSecret = "XXXX"
val filters = args.takeRight(args.length - 4)
    // Set the system properties so that Twitter4j library used by
twitter stream
    // can use them to generat OAuth credentials
System.setProperty("twitter4j.oauth.consumerKey", consumerKey)
System.setProperty("twitter4j.oauth.consumerSecret", consumerSecret)
System.setProperty("twitter4j.oauth.accessToken", accessToken)
System.setProperty("twitter4j.oauth.accessTokenSecret",
accessTokenSecret)

valsparkConf = new SparkConf().setAppName("TwitterPopularTags")
valssc = new StreamingContext(sparkConf, Seconds(2))
val stream = TwitterUtils.createStream(ssc, None, filters)

valhashTags = stream.flatMap(status =>status.getText.split("
").filter(_.startsWith("#")))

valtopCountsInOneMinute = hashTags.map((_, 1)).reduceByKeyAndWindow(_
+ _, Seconds(60))
                    .map{case (topic, count) => (count, topic)}
                    .transform(_.sortByKey(false))



    // Print popular hashtags
topCountsInOneMinute.foreachRDD(rdd => {
valtopList = rdd.take(10)
println("\nPopular topics in last 1 minute (%s total):".format(rdd.
count()))
topList.foreach{case (count, tag) =>println("%s (%s tweets)".
format(tag, count))}
    })
```

```
    ssc.start()
    ssc.awaitTermination()
      }
    }
```

Now, we build an assembly jar for this project by executing the following commands:

```
sbt compile
sbt assembly
```

This will create a fat jar with the name `TwitterSpark-assembly-1.0.jar`. To execute it, we copy it into a Spark cluster and execute the following commands.

Here is the code to execute it:

```
/usr/local/spark/bin/spark-submit \
  --class "com.demo.TwitterPopularTags" \
  --master local[2] \
  TwitterSpark-assembly-1.0.jar
```

Here is the code to execute it on the Spark default master:

```
/usr/local/spark/bin/spark-submit \
  --class "com.demo.TwitterPopularTags" \
  --master spark://host:port \
  TwitterSpark-assembly-1.0.jar
```

Here is the code to execute it on the YARN cluster:

```
/usr/local/spark/bin/spark-submit \
  --class "com.demo.TwitterPopularTags" \
  --master yarn \
  TwitterSpark-assembly-1.0.jar
```

As a result you should see the top trending hash tags of every minute:

```
#style (1 tweets)
#followmejp (1 tweets)
#amateur (1 tweets)
#VotaSebastianVillalobos (1 tweets)
#poem (1 tweets)
#followme (1 tweets)
#thoughtoftheday (1 tweets)
#followcircle (1 tweets)
```

How it works...

Spark Streaming internally receives live data streams from the source. It then divides data into batches and processes these batches one by one. These streams are called `DStreams`. Along with the streaming context, we also need to provide the time slot that Spark needs in order to wait for a batch. In the preceding code, the time period needed is 2 seconds. Once the batch has been received by the spark engine, the rest of the processing is similar to what has already been explained in the previous recipes. You can read more about this at `http://spark.apache.org/docs/latest/streaming-programming-guide.html`.

Analyzing Parquet files using Spark

Parquet is columnar data file format, which is being used extensively. In this recipe, we are going to take a look at how to access this data from Spark and then process it.

Getting ready

To perform this recipe, you should have Hadoop and Spark installed. You also need to install Scala. I am using Scala 2.11.0.

How to do it...

Spark supports the accessing of Parquet files from the SQL context. You can read and write Parquet files using this SQL context. In this recipe, we are going to take a look at how to read a Parquet file from HDFS and process it.

First of all, download the sample parquet file, `users.parquet`, and store it in the HDFS `/parquet` path `https://github.com/deshpandetanmay/hadoop-real-world-cookbook/blob/master/data/users.parquet`.

We will create a Scala project with the following files:

```
SparkParquet\build.sbt
SparkParquet\project\assembly.sbt
SparkParquet\src\main\scala\com\demo\SparkParquet.scala
```

The contents of `build.sbt` are as follows:

```
name := "SparkParquet"

version := "1.0"

scalaVersion := "2.11.7"
```

```
libraryDependencies ++= Seq(

 "com.google.guava" % "guava" % "14.0",
 "org.apache.spark" %% "spark-core" % "1.6.0" % "provided",
 "org.apache.spark" %% "spark-sql"  % "1.6.0" % "provided"
)

resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

The contents of `assembly.sbt` are as follows:

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.12.0")
```

Then, we write a Scala class, which reads the parquet file stored in the given HDFS location and creates another file from it, `.SparkParquet.scala`, as follows:

```
packagecom.demo


importorg.apache.spark._
importorg.apache.spark.sql._

objectSparkParquet {


def main(args: Array[String]) {
valsparkConf = new SparkConf().setAppName("SparkParquet")
valsc = new SparkContext(sparkConf)
valsqlContext = new org.apache.spark.sql.SQLContext(sc)
val users = sqlContext.read.load("/parquet/users.parquet")

users.select("name", "favorite_color").write.save("/parquet/
namesAndFavColors.parquet")

users.registerTempTable("userParquetFile")
users.map(t => "Name: " + t(0)).collect().foreach(println)
sc.stop()
  }
}
```

We build an assembly jar for this project using the following commands:

```
sbt compile
sbt assembly
```

Then, we execute the application using the following commands:

We use this code for local threads:

```
 /usr/local/spark/bin/spark-submit \
 --class "com.demo.SparkParquet" \
 --master local[2] \
SparkParquet-assembly-1.0.jar
```

We use this code for YARN:

```
 /usr/local/spark/bin/spark-submit \
 --class "com.demo.SparkParquet" \
 --master yarn \
SparkParquet-assembly-1.0.jar
```

This will start the application and print something like this:

```
~
Name: Alyssa
Name: Ben
~
```

I will also create another parquet file called `namesAndFavColors.parquet` in the `/parquet` HDFS path.

How it works...

The Spark SQL context has features to serialize and deserialize Parquet files. The preceding code makes use of this feature to read the data and print it. We can also register a temporary table and use SQL to analyze the parquet file data using its schema.

Analyzing JSON data using Spark

JSON is one of most frequently used data storage and exchange formats in use these days. In this recipe, we are going to take a look at how to access the JSON file data from Spark and process it.

Getting ready

To perform this recipe, you should have Hadoop and Spark installed. You also need to install Scala. I am using Scala 2.11.0 here.

How to do it...

Spark supports the accessing of JSON files from the SQL context. You can read and write JSON files using the SQL context. In this recipe, we are going to take a look at how to read a JSON file from HDFS and process it.

First of all, download the `people.json` sample JSON file and store it in the `/json` HDFS path using this link:

`https://github.com/deshpandetanmay/hadoop-real-world-cookbook/blob/master/data/people.json`.

We will create a Scala project using the following files:

```
SparkJSON\build.sbt
SparkJSON\project\assembly.sbt
SparkJSON\src\main\scala\com\demo\SparkJSON.scala
```

Here is the content of `build.sbt`:

```
name := "SparkJSON"

version := "1.0"

scalaVersion := "2.11.7"

libraryDependencies ++= Seq(

 "com.google.guava" % "guava" % "14.0",
 "org.apache.spark" %% "spark-core" % "1.6.0" % "provided",
 "org.apache.spark" %% "spark-sql"  % "1.6.0" % "provided"
)

resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

The contents of `assembly.sbt` are as follows:

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.12.0")
```

Then, we write a Scala class, which reads the JSON file stored in the given HDFS location and process it. `SparkJSON.scala` is as follows:

```
packagecom.demo

importorg.apache.spark._
importorg.apache.spark.sql._
```

```
objectSparkJSON {


def main(args: Array[String]) {
valsparkConf = new SparkConf().setAppName("SPARKJSON")
valsc = new SparkContext(sparkConf)
valsqlContext = new org.apache.spark.sql.SQLContext(sc)
val users = sqlContext.read.json("/json/people.json")

users.map(t => "Name: " + t(0)).collect().foreach(println)
users.show()

        // Print the schema in a tree format
        users.printSchema()

        // Select only the "name" column
        users.select("name").show()

        // Select everybody, but increment the age by 1
        users.select(users("name"), users("age") + 1).show()

        // Select people older than 21
        users.filter(users("age") > 21).show()

        // Count people by age
        users.groupBy("age").count().show()

sc.stop()
  }
}
```

We build an assembly jar for this project using the following commands:

```
sbt compile
sbt assembly
```

Then, we execute the application using the following command:

First on local threads

```
 /usr/local/spark/bin/spark-submit \
 --class "com.demo.SparkJSON" \
 --master local[2] \
SparkJSON-assembly-1.0.jar
```

For YARN, we use the following code:

```
/usr/local/spark/bin/spark-submit \
--class "com.demo.SparkJSON" \
--master yarn \
SparkJSON-assembly-1.0.jar
```

This will start the application and print something like this:

```
+----+-----+
| age|count|
+----+-----+
|50|    1|
|  19|    1|
|  30|    1|
+----+-----+
```

How it works...

The Spark SQL context has features that are used to serialize and deserialize JSON files. The preceding code makes use of these features to read the data from the files and then print it. We can also register a temporary table and use SQL to analyze the JSON file data using its schema.

Processing graphs using Graph X

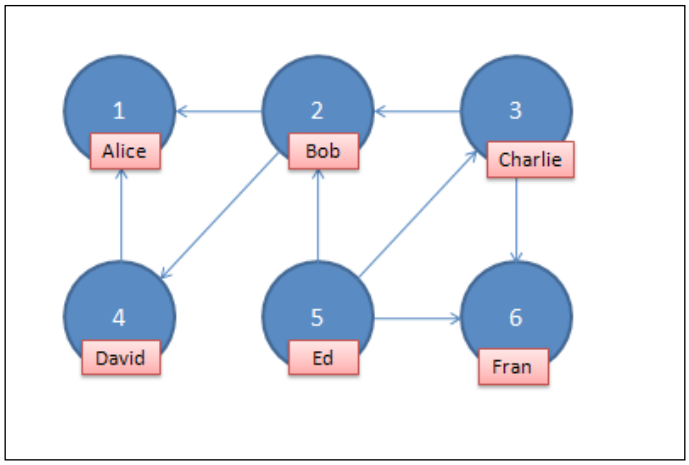
A graph is combination of vertices and edges. Spark provides a module to define a graph and then process these graphs in real time. In this recipe, we are going to look at a social graph example and process data using Spark.

Getting ready

To perform this recipe, you should have Hadoop and Spark installed. You also need to install Scala. I am using Scala 2.11.0 here.

How to do it...

Take a social networking site that has users and other users such as the activities of a user. Based on the likes, we can conclude who is connected to whom. Consider the following data:



Now, we have to analyze this graph based on the likes that are provided. In order to do so, we start the Spark Shell, and run the following commands:

Import graph libraries like this:

```
scala>import org.apache.spark.graphx._
scala>import org.apache.spark.rdd.RDD
```

Next, we define vertices and edges:

```
scala>valvertexArray = Array(

  (1L, ("Alice", 28)),
  (2L, ("Bob", 27)),
  (3L, ("Charlie", 65)),
  (4L, ("David", 42)),
  (5L, ("Ed", 55)),
  (6L, ("Fran", 50))
  )
```

Here, we have a username and age as the properties of each vertex:

```
scala>valedgeArray = Array(
Edge(2L, 1L, 7),
Edge(2L, 4L, 2),
Edge(3L, 2L, 4),
Edge(3L, 6L, 3),
Edge(4L, 1L, 1),
Edge(5L, 2L, 2),
Edge(5L, 3L, 8),
Edge(5L, 6L, 3)
  )
```

Every edge displays the link between the vertices and the number of likes that are given to various activities.

Now, we create RDDs of given arrays and orders in order to distribute data in parallel:

```
scala>valvertexRDD: RDD[(Long, (String, Int))] =
sc.parallelize(vertexArray)
scala>valedgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)
```

Next, we need to build a graph out of these vertices and edges, as shown here:

```
scala>val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)
```

Now, we are ready to process this data. First of all, let's try to print all the users who are more than 30 years of age:

```
scala>graph.vertices.filter { case (id, (name, age)) => age > 30
}.collect.foreach {
case (id, (name, age)) =>println(s"$name is $age")
}
```

The output for this is as follows:

```
David is 42
Fran is 50
Charlie is 65
Ed is 55
```

We want to print the relationship between the users. We can do this as follows:

```
scala> for (triplet <- graph.triplets.collect) {
println(s"${triplet.srcAttr._1} likes ${triplet.dstAttr._1}")
      }
```

The output for this is as follows:

```
Bob likes Alice
Bob likes David
Charlie likes Bob
Charlie likes Fran
David likes Alice
Ed likes Bob
Ed likes Charlie
Ed likes Fran
```

We can conclude that if a user likes another user's activities more than five times, then they are in love. This can be represented through the following command:

```
scala>for (triplet <- graph.triplets.filter(t =>t.attr> 5).collect) {
println(s"${triplet.srcAttr._1} loves ${triplet.dstAttr._1}")
    }
```

The output for this is as follows:

```
Bob loves Alice
Ed loves Charlie
```

This way, you can analyze various datasets that are available in the graph format.

How it works...

Spark GraphX is used for the parallel computation of graph datasets. `GraphX` extends the Spark RDD by introducing a directed multi-attribute graph with vertices and edges. To support graph processing, it exposes sets of APIs such as subgraphs, join vertices, and so on. It also includes a set of algorithms, which will be helpful in graph computations such as the Page Rank Algorithm.



You can read more about Graphx at `http://spark.apache.org/docs/latest/graphx-programming-guide.html`.

Conducting predictive analytics using Spark MLib

Spark has a very rich machine learning library called `MLib`. This is a collection of various algorithms that are used for classification, clustering, recommendations, and so on. In this recipe, we are going to take a look at how to build a predictive model using `MLib`.

Getting ready

To perform this recipe, you should have Hadoop and Spark installed. You also need to install Scala. Here, I am using Scala 2.11.0.

How to do it...

For this recipe, we are going use the classic example dataset of iris flowers; you can find out more about this at `https://en.wikipedia.org/wiki/Iris_flower_data_set`.

Here, based on the petal length and width and the sepal length and width, we need to classify the flowers into species. First, we build a model, and then run tests on it to predict the output.

To start with, we first download `iris.txt` from `https://github.com/deshpandetanmay/hadoop-real-world-cookbook/blob/master/data/iris.txt`.

Next, save it in HDFS.

We start the Spark Shell, and run the following commands:

```
scala>import org.apache.spark.mllib.classification.NaiveBayes
scala>import org.apache.spark.mllib.linalg.Vectors
scala>import org.apache.spark.mllib.regression.LabeledPoint
```

This will import the required libraries. We are going to use Naïve Bayes classifiers in this case.

First, we create a class to define the file data model:

```
scala>case class Iris(
id:java.util.UUID,
sepal_l:Double,
sepal_w:Double,
petal_l:Double,
petal_w:Double,
species:String
)
```

Next, we load data from HDFS and parse it:

```
scala>val data = sc.textFile("iris.txt")

scala>val parsed = data.filter(!_.isEmpty).map {row =>
valsplitted = row.split(",")
val Array(sl, sw, pl, pw) = splitted.slice(0,4).map(_.toDouble)
    Iris (java.util.UUID.randomUUID(), sl, sw, pl, pw, splitted(4))
}
```

We can check whether the data is loaded properly by executing the following command:

```
parsed.take(10).foreach(println)
```

It would print something like this:

```
Iris(006f1567-e7ad-4e2a-8519-45272a66ce9b,5.1,3.5,1.4,0.2,Iris-setosa)
Iris(9ddd68aa-46f4-4d92-8de7-3c783bb5fd2b,4.9,3.0,1.4,0.2,Iris-setosa)
```

Here, we are going to use `LabledPoint`, which contains labels and features. The labels need to be in the Double format, and features need to be in the Vector format. Our label is a string in this case, so in order to convert it to the Double format, we create a map of unique values and then reverse them:

```
scala>val class2id  = parsed.map(_.species).distinct.collect.
zipWithIndex.map{case (k,v)=>(k, v.toDouble)}.toMap
scala>val id2class = class2id.map(_.swap)

scala>valparsedData = parsed.map { i =>LabeledPoint(class2id(i.
species), Vectors.dense(i.petal_l,i.petal_w,i.sepal_l,i.sepal_w)) }
```

Next, we train the model using the sample data we have:

```
scala>val model = NaiveBayes.train(parsedData)
```

Now, the model is ready to predict the species of the given flower on the basis of its features. Here is an example of this:

```
scala> id2class(model.predict(Vectors.dense(5, 1.5, 6.4, 3.2)))
res19: String = Iris-versicolor
```

This way, you can perform predictive analytics using Spark MLib.

How it works...

Spark `MLib` provides a huge list of supported algorithms. In the preceding example that uses a training dataset, Spark builds a model that has probability data. So, once the model is built, we can start predicting the output for certain given features. There are various classification algorithms available, so you can choose any of them.

10
Hadoop Use Cases

In this chapter, we'll take a look at the following recipes:

- Call Data Record analytics
- Web log analytics
- Sensitive data masking and encryption using Hadoop

Introduction

Throughout this book, we have been discussing Hadoop and its real-world use cases. In this final chapter, we are going to discuss the end-to-end implementation of a few such use cases. The motivation for this chapter is to apply the learning you've gathered from the earlier chapters. We will discuss use cases related the telecom, finance, and e-commerce domains. So, let's get started.

Call Data Record analytics

Call data records is data that is gathered by telecom operators that are specific to individual customers. We are going to take a look at telecom domain-specific use cases in this recipe.

Getting ready

To perform this recipe, you should have an up and running Hadoop cluster. We need some sample data for these use cases; I have written a data generator, which can used for your reference. You can find it at `https://github.com/deshpandetanmay/cdr-data-generator`.

How to do it...

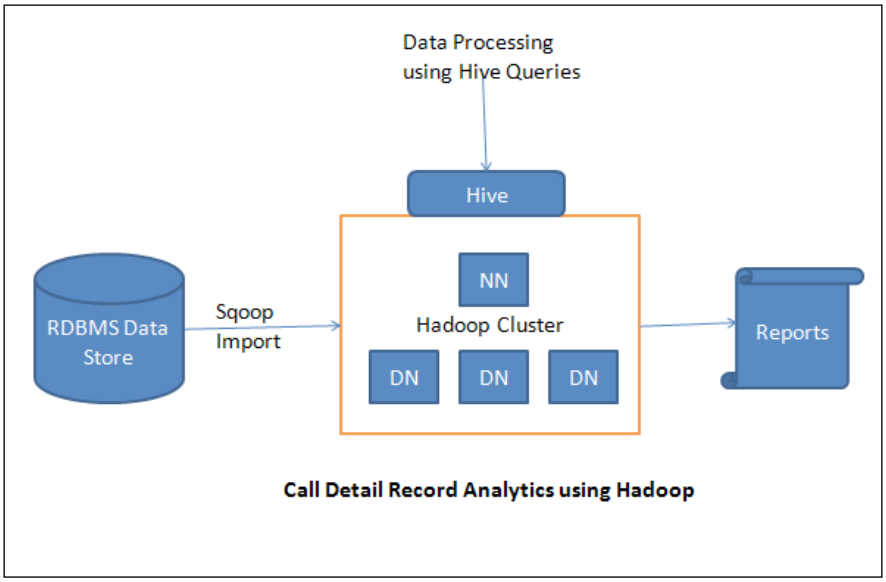
Before jumping into the solution let's first try to understand a problem statement.

Problem Statement

Telecom companies keep records of each and every call made by their subscribers. They also keep information such as when a call was made, who it was made to, the start time, end time, and so on. Detailed information, such as SMSes, data sessions, and so on, is also stored by these companies. Here, the problem statement is how we can use this data to make company operations run smoother and help them generate valuable information about their customers.

Solution

Take a situation where a telecom company stores its data in RDMBS such as MySQL or Postgres. So, our solution flow diagram will look like this:



Now, let's assume that we have our data already present in the MySQL database in the `abctelecom.cdr` table:

```
mysql> desc abctelecom.cdr;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | varchar(50) | NO   | PRI |         |       |
```

```
| caller     | varchar(20) | YES  |     | NULL    |       |
| calling    | varchar(20) | YES  |     | NULL    |       |
| start_time | varchar(20) | YES  |     | NULL    |       |
| end_time   | varchar(20) | YES  |     | NULL    |       |
| type       | varchar(20) | YES  |     | NULL    |       |
| charge     | double      | YES  |     | NULL    |       |
| result     | varchar(20) | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
8 rows in set (0.00 sec)
```

Now, let's execute the Sqoop command to import this data into HDFS.

First of all, make sure that the data directory is created, and then execute the Sqoop `import` command:

```
$ hadoop fs -mkdir /data
$sqoop import --connect jdbc:mysql://localhost:3306/abctelecom
--username root --password password --table cdr --target-dir /data/cdr
--fields-terminated-by '|'
```

This will import the data from the MySQL table to the HDFS `/data/cdr` directory.

Next, we start the Hive prompt and create an external Hive table pointing to the preceding directory:

```
CREATE EXTERNAL TABLE cdr (
id STRING,
caller_num STRING,
called_num STRING,
start_time STRING,
end_time STRING,
call_type STRING,
call_charge DOUBLE,
call_result STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
    LOCATION '/data/cdr';
```

We can confirm whether the data load is as expected by executing the following commands:

```
hive> SELECT * FROM cdr LIMIT 2;
OK
02240095-bc62-4973-a143-710a5fb3509a 0989256849    1238082266
    2016-01-19T11:01:14.      2016-01-19T11:01:14.       SMS
0.35525125
    ANSWERED
30d73a30-6e93-472a-82cb-b1e53c94d1cb 5564259141    0537330705
    2016-02-15T00:56:17.      2016-02-15T00:58:53.       VOICE
    0.29876852   ANSWERED
Time taken: 1.387 seconds, Fetched: 2 row(s)
```

Now, let's start analyzing this data. Let's say we want to know how many calls and SMSes subscribers are making. To know this information, we can execute the following query:

```
SELECT caller_num, call_type, COUNT(*)
FROM cdr
        GROUP BY caller_num, call_type;
```

The result for this is as follows:

```
0537330705       SMS 53
0537330705       VOICE 59
0989256849       SMS 59
0989256849       VOICE 53
1238082266       SMS 55
1238082266       VOICE 56
5314043825       SMS 56
5314043825       VOICE 55
5564259141       SMS 55
5564259141       VOICE 56
```

Next, we want to find out the total duration of time for which a subscriber is making voice calls:

```
SELECT caller_num, SUM(UNIX_TIMESTAMP(end_time , "yyyy-MM-
dd'T'HH:mm:ss.SSSX") - UNIX_TIMESTAMP(start_time, "yyyy-MM-
dd'T'HH:mm:ss.SSSX"))  as duration
FROM cdr
group by caller_num, call_type
HAVING  call_type = 'VOICE';
```

This will give us information on how much time each subscriber has spent on voice calls. Now, we can filter out the subscribers who make voice calls for more than a specific amount of time and offer them a voice call-specific package.

The result of the preceding query is as follows:

```
0537330705      8698
0989256849      7929
1238082266      8554
5314043825      7988
5564259141      4007
```

Next, we can find out the customers who spend the most amount of money and offer them better packages. In order to find the top five billed customers, execute the following query:

```
SELECT caller_num, sum(call_charge) AS bill
FROM cdr
GROUP BY caller_num
ORDER BY bill DESC LIMIT 5;
```

The result is as follows:

```
5314043825      58.67936635200001
0537330705      56.971656405699996
1238082266      56.0622671385
0989256849      53.69393956600001
5564259141      42.46016877900002
```

We can round off the bill as follows:

```
SELECT caller_num, round(sum(call_charge)) AS bill
FROM cdr
GROUP BY caller_num
ORDER BY bill DESC LIMIT 5;
```

The result of this is as follows:

```
5314043825      59.0
0537330705      57.0
1238082266      56.0
0989256849      54.0
5564259141      42.0
```

We can also find out how many times a call was placed but the called number was busy:

```
SELECT call_result, count(*)
FROM cdr
GROUP BY call_result;
```

The result of this is as follows:

```
ANSWERED  527
BUSY      29
```

This way, the more data you have, the more analysis you can do with it.

How it works...

We have already taken a look at how Hadoop, Sqoop, and Hive work in the previous chapters, so in case you need more details on any of these, refer to the respective chapters.

Web log analytics

Web logs is data generated by web servers running a website. This use case is applicable to domains where companies have their websites hosted and want to know more about their website performance and customer behavior on the website.

Getting ready

To perform this recipe, you should have an up and running Hadoop cluster. I have uploaded the data of some sample web logs from

`https://github.com/deshpandetanmay/hadoop-real-world-cookbook/blob/master/data/mylog.txt`.

How to do it...

Before jumping into the solution, let's first try to understand the problem statement:

Problem statement

Many companies run businesses on their websites. Their website performance decides the sales or profitability. Web servers generally log information about the user, browser, IP address, and so on. We can use this information in order to make the website browsing experience smoother for users, which would help increase profitability.

Solution

Here, we assume that a company hosting its website on an Apache server and we will be listening to the logs generated by the server. We will use Flume to copy the logs to the HDFS folder, and then use Apache Hive to process the data.

To import data into HDFS from web servers, we have to install the Flume agent on each web server instance. The following is the configuration that we have to use for the Flume agent configuration:

```
flume1.sources  = weblogs-source-1
flume1.channels = hdfs-channel-1
flume1.sinks    = hdfs-sink-1

# For each source, channel, and sink, set
# standard properties.
flume1.sources.weblogs-source-1.type = exec
flume1.sources.weblogs-source-1.command = tail -f /path/to/log/file.
log
flume1.sources.weblogs-source-1.batchSize = 100
flume1.sources.weblogs-source-1.channels = hdfs-channel-1

flume1.channels.hdfs-channel-1.type   = memory
flume1.sinks.hdfs-sink-1.channel = hdfs-channel-1
flume1.sinks.hdfs-sink-1.type = hdfs
flume1.sinks.hdfs-sink-1.hdfs.writeFormat = Text
flume1.sinks.hdfs-sink-1.hdfs.fileType = DataStream
flume1.sinks.hdfs-sink-1.hdfs.filePrefix = test-events
flume1.sinks.hdfs-sink-1.hdfs.useLocalTimeStamp = true
flume1.sinks.hdfs-sink-1.hdfs.path = /logs/web/%y-%m-%d
flume1.sinks.hdfs-sink-1.hdfs.rollCount=100
flume1.sinks.hdfs-sink-1.hdfs.rollSize=0

# Other properties are specific to each type of
# source, channel, or sink. In this case, we
# specify the capacity of the memory channel.
flume1.channels.hdfs-channel-1.capacity = 10000
```

This would be listening to the given log file path. Here, we are using the `exec` source, which executes the given command after a certain time interval.

The source is the output of the `exec` command that we have provided. The channel transfers the data to Sink, which writes it to HDFS.

To execute the preceding configuration, we have to run the following command:

```
/usr/local/flume/bin/flume-ng agent -n flume1 -c /usr/local/flume/conf
-f /usr/local/flume/conf/weblogs.conf -Dflume.root.logger=INFO,console
```

Now, as and when the log file is updated, the data will be written to HDFS as well.

The data we get is as follows:

```
103.22.239.216 - - [10/Feb/2016:23:05:30 -0800] "GET / HTTP/1.1" 304 0
"http://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=5&cad=
rja&uact=8&ved=0ahUKEw
js6N3jku_KAhXFkI4KHbRKBOwQFghGMAQ&url=http%3A%2F%2Fhadooptutorials.co.
in%2F&usg=AFQjCNEnec258W4qoaZBxqTY1SxCekMBtw&sig2=GUz56esk1WQ9ga
H6IoKBFA" "Mozilla/5.0 (Windows NT 6.1; Trident/7.0; rv:11.0) like
Gecko"
202.183.129.168 - - [10/Feb/2016:23:03:24 -0800] "GET /js/freelancer.
js HTTP/1.1" 304 0 - "Mozilla/4.0 (compatible;)"
202.168.90.179 - - [10/Feb/2016:23:03:06 -0800] "GET /tutorials/
hadoop/images/understanding-map-reduce-programming/map-reduce-flow.PNG
HTTP/1.1" 304 0 - "Mozilla/4.0 (compatible;)"
115.249.142.9 - - [10/Feb/2016:23:02:54 -0800] "GET /tutorials/
hadoop/images/big-data-analytics-what-is-that/3Vs_of_big_data.png
HTTP/1.1" 200 0 "http://hadooptutorials.co.in/tutorials/elasticsearch/
install-elasticsearch-kibana-logstash-on-windows.html" "Mozilla/5.0
(Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/48.0.2564.97 Safari/537.36"
115.249.142.9 - - [10/Feb/2016:23:02:54 -0800] "GET /tutorials/
sqoop/images/apache-sqoop-advanced-features/apache-sqoop-advanced-
features.png HTTP/1.1" 200 0 "http://hadooptutorials.co.in/tutorials/
elasticsearch/install-elasticsearch-kibana-logstash-on-windows.html"
"Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/48.0.2564.97 Safari/537.36"
```

Next, we start the Hive prompt and create a table in Hive using `RegexSerde`:

```
CREATE TABLE apache_combined_log (
  host STRING,
  identity STRING,
  userid STRING,
  time STRING,
  request STRING,
  status STRING,
  size STRING,
  referer STRING,
  agent STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  "input.regex" = "([^ ]*) ([^ ]*) ([^ ]*) (-|\\[[^\\]]*\\]) ([^
\"]*|\"[^\"]*\") (-|[0-9]*) (-|[0-9]*)(?: ([^ \"]*|\"[^\"]*\") ([^
\"]*|\"[^\"]*\"))?",
  "output.format.string" = "%1$s %2$s %3$s %4$s %5$s %6$s %7$s %8$s
%9$s"
)
STORED AS TEXTFILE;
```

Load the data to this table from HDFS.

Now we are all set to start our analytics.

First of all, let's try to find the most popular pages on the website by executing the following query:

```
SELECT request, COUNT(*) AS page_view_count FROM apache_combined_log
GROUP BY request
ORDER BY page_view_count DESC
LIMIT 5;
```

The output will be shown as follows:

```
"GET /font-awesome-4.1.0/css/font-awesome.min.css HTTP/1.1"    409
"GET /css/freelancer.css HTTP/1.1"   405
"GET /js/classie.js HTTP/1.1" 402
"GET /js/cbpAnimatedHeader.js HTTP/1.1"    400
"GET /js/freelancer.js HTTP/1.1"    396
```

Next, we want to know the most number of hits from the referral page, and for this, we need to execute the following command:

```
SELECT referer, COUNT(*) AS referer_count FROM apache_combined_log
GROUP BY referer
ORDER BY referer_count DESC
LIMIT 5;
```

The output for this is as shown here:

```
-  1374
"http://hadooptutorials.co.in/"     1096
"http://hadooptutorials.co.in/tutorials/spark/install-apache-spark-on-
ubuntu.html"    1029
"http://hadooptutorials.co.in/tutorials/elasticsearch/install-
elasticsearch-kibana-logstash-on-windows.html"    848
"http://hadooptutorials.co.in/tutorials/hive/hive-best-practices.html"
573
```

Next, we want to know which server threw an error for which pages during browsing. We can work on such pages in order to improve the site performance. Here, an error occurs when the HTTP status is not 200:

```
SELECT request, status FROM apache_combined_log
WHERE status != 200;
```

The output for this is as follows:

```
"GET /js/contact_me.js HTTP/1.1"     304
"GET /js/jqBootstrapValidation.js HTTP/1.1" 304
"GET /js/freelancer.js HTTP/1.1"     304
"GET /tutorials/spark/images/install-apache-spark-on-ubuntu/WordCount_
Demo.PNG HTTP/1.1"      304
```

Next, we want to do analysis based on location. But in our data, we don't have any such information; however, we can find this information using an IP address.

To do this, we are going to use a UDF, as shown here:

```
package in.co.hadooptutorials.ip.location.hive.udf;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public class LocationFinder extends UDF {

    public Text evaluate( Text inputIp) throws IOException{
        URL url = new URL("http://freegeoip.net/json/"+inputIp.
toString());
        HttpURLConnection conn = (HttpURLConnection) url.
openConnection();
        conn.setRequestMethod("GET");
        conn.setRequestProperty("Accept", "application/json");

        if (conn.getResponseCode() != 200) {
            throw new RuntimeException("Failed : HTTP error code : "
                    + conn.getResponseCode());
        }

        BufferedReader br = new BufferedReader(new InputStreamReader(
             (conn.getInputStream())));

        String output = br.readLine();
        return new Text(output);
    }


}
```

First, we take out unique IPs:

```
CREATE TABLE unique_ips as
SELECT DISTINCT(host) FROM apache_combined_log;
```

Then, we use the UDF to find details about each IP:

```
ADD JAR ip-location-hive-udf-1.0.jar;
CREATE TEMPORARY FUNCTION get_details AS 'in.co.hadooptutorials.
ip.location.hive.udf.LocationFinder' ;

CREATE TABLE location_details ( data string) ;
INSERT INTO location_details
SELECT get_details(host) FROM unique_ips;

Next we create table with JSON SerDe.
ADD JAR json-serde-1.1.9.9-Hive1.2-jar-with-dependencies.jar;

CREATE TABLE ip_location_details (
ip STRING,
country_code STRING,
country_name STRING,
region_code STRING,
region_name STRING,
latitude DOUBLE,
longitude DOUBLE,
zip_code STRING,
time_zone STRING,
city STRING,
metro_code INT
)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
LOCATION '/user/hive/warehouse/location_details';
```

We are all set to use the location details by joining the tables. For example, to get the number of sessions per country, we need to execute the following command:

```
SELECT ild.country_name, COUNT(*) FROM
ip_location_details ild JOIN apache_combined_log acl
ON (ild.ip = acl.host)
GROUP BY ild.country_name;
For City,
SELECT ild.city, COUNT(*) FROM
ip_location_details ild JOIN apache_combined_log acl
ON (ild.ip = acl.host)
GROUP BY ild.city;
```

We can also create reports in order to visualize the results of these queries.

How it works...

We have already seen how Hadoop, Flume, and Hive work in previous chapters, so in case you need further details on any of these, refer to the respective chapters.

Sensitive data masking and encryption using Hadoop

A lot of companies handle sensitive information such as SSN numbers, names, credit card numbers, and so on. In this recipe, we are going to take a look at how to use Hadoop to mask or encrypt this data in order to secure it. This recipe can be referred to by various domains, such as finance, retail, telecom, and those people who handle critical information.

Getting ready

To perform this recipe, you should have an up and running Hadoop cluster.

How to do it...

Before jumping into the solution, let's first try to understand the problem statement.

Problem statement

Handling sensitive information is a critical part of today's data operations. Here, the problem statement is to transform critical information into masked data or completely encrypted data.

Solution

Here, we assume that we already have data with us in flat files and it has been loaded into HDFS.

Let's say we have some sample data, as shown here, which has the name and credit card number of a person:

```
Ryan Levine|4716840526341330
Erika Smith|4539326321106479
Brooklyn Sloan|4916571896673675
Karen Mayer|5428124448681073
Eddie O'neill|5374594066409623
Nancy Stevens|5390742852412406
Chasity Conway|378238168241362
```

```
Aaron McCray|374762693530829
Kendra Allen|374606063397725
Jenna Burns|5335126551184557
Jane Mayo|343637926486924
George Suarez|347956127969751
Janet Abbott|6011345775765400
Steven Marshall|6011414499816624
Shane Heath|6011322907488797
Unborn Vega|6011417989731237
Kaitlyn Wilder|6011055063404451
Donald Morgan|5140126646868581
```

First of all, we will create a Hive table, and then load data into it:

```
CREATE TABLE personal_details(
name STRING,
ccnum STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|';

LOAD DATA LOCAL INPATH 'personal_details.txt' INTO TABLE personal_
details;
```

Now, we can write Hive UDFs to mask the name as well as the credit card number. Credit card numbers are not random numbers, but they are generated through an algorithm called Luhn's Algorithm, which can be found at `https://en.wikipedia.org/wiki/Luhn_algorithm`.

We can write algorithms based on our complexity but for the purposes of demonstration, I am using a simple algorithm, which subtracts each number from 9 and returns it. Before starting anything, make sure that you add the required dependencies in `pom.xml`:

```
<dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-exec</artifactId>
    <version>1.2.0</version>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>2.7.1</version>
</dependency>
```

Our CC number masking UDF code is as follows:

```
package in.co.hadooptutorials.hive.udf.masking;

import java.io.IOException;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public class CCMasker extends UDF {
    public Text evaluate(Text ccNumber) throws IOException {
        String[] num = ccNumber.toString().split("");
        StringBuilder sb = new StringBuilder();
        for (String c : num) {
            if(!("".equals(c))){

                int i = 9 - Integer.parseInt(c);
                sb.append(i);
            }
        }
        return new Text(sb.toString());
    }

}
```

Our name masking UDF code is as follows:

```
package in.co.hadooptutorials.hive.udf.masking;

import java.io.IOException;
import java.util.Random;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public class NameMasker extends UDF {
    public Text evaluate(Text name) throws IOException {

        return new Text(mask(name.toString(), 0));
    }

    char randomChar(Random r, String cs, boolean uppercase) {
        char c = cs.charAt(r.nextInt(cs.length()));
        return uppercase ? Character.toUpperCase(c) : c;
    }
```

```
        String mask(String str, int seed) {

            final String cons = "bcdfghjklmnpqrstvwxz";
            final String vowel = "aeiouy";
            final String digit = "0123456789";

            Random r = new Random(seed);
            char data[] = str.toCharArray();

            for (int n = 0; n < data.length; ++n) {
                char ln = Character.toLowerCase(data[n]);
                if (cons.indexOf(ln) >= 0)
                    data[n] = randomChar(r, cons, ln != data[n]);
                else if (vowel.indexOf(ln) >= 0)
                    data[n] = randomChar(r, vowel, ln != data[n]);
                else if (digit.indexOf(ln) >= 0)
                    data[n] = randomChar(r, digit, ln != data[n]);
            }

            return new String(data);

        }
    }
```

We build the project and copy the JAR file to the Hadoop cluster:

```
ADD JAR masking-udf-1.0.jar;

CREATE TEMPORARY FUNCTION ccmask AS 'in.co.hadooptutorials.hive.udf.
masking.CCMasker';

CREATE TEMPORARY FUNCTION namemask AS 'in.co.hadooptutorials.hive.udf.
masking.NameMasker';

And then we execute the query to start masking name and cc number.
CREATE TABLE personal_details_masked AS
SELECT namemask(name), ccmask(ccnum) FROM
personal_details;
```

Once the execution is complete, you can see these results:

```
hive> select * from personal_details_masked;
    Buek Typozi     5283159473658669
    Aleky Rpozs     5460673678893520
    Bleytryc Zsyyr  5083428103326324
    Bumyt Ryoos     4571875551318926
```

```
Almyy Y'poosw   4625405933590376
Bumky Rpoziww   4609257147587593
Blekyry Coswyy  621761831758637
Aumyt RpCziy    625237306469170
Bumkty Ycziw    625393936602274
Bumky Ryczs     4664873448815442
Bumy Tyyo 656362073513075
Buekty Poosyw   652043872030248
Bumyt Ypcosw    3988654224234599
Blekyr Pozswyrd 3988585500183375
Bleky Ryozs     3988677092511202
Almytr Pozi     3988582010268762
Buektyp Coswyr  3988944936595548
Bumytr Pozsyw   4859873353131418
```

This way, you can use Hadoop to mask or encrypt sensitive information.

How it works...

We have already taken a look at how Hadoop and Hive work in previous chapters, so in case you need further details, refer to the respective chapters.

Module 3

Mastering Hadoop

Go beyond the basics and master the next generation of Hadoop data processing platforms

1
Hadoop 2.X

"There's nothing that cannot be found through some search engine or on the Internet somewhere."

-Eric Schmidt, Executive Chairman, Google

Hadoop is the de facto open source framework used in the industry for large scale, massively parallel, and distributed data processing. It provides a computation layer for parallel and distributed computation processing. Closely associated with the computation layer is a highly fault-tolerant data storage layer, the **Hadoop Distributed File System** (**HDFS**). Both the computation and data layers run on commodity hardware, which is inexpensive, easily available, and compatible with other similar hardware.

In this chapter, we will look at the journey of Hadoop, with a focus on the features that make it enterprise-ready. Hadoop, with 6 years of development and deployment under its belt, has moved from a framework that supports the MapReduce paradigm exclusively to a more generic cluster-computing framework. This chapter covers the following topics:

- An outline of Hadoop's code evolution, with major milestones highlighted
- An introduction to the changes that Hadoop has undergone as it has moved from 1.X releases to 2.X releases, and how it is evolving into a generic cluster-computing framework
- An introduction to the options available for enterprise-grade Hadoop, and the parameters for their evaluation
- An overview of a few popular enterprise-ready Hadoop distributions

The inception of Hadoop

The birth and evolution of the Internet led to **World Wide Web** (**WWW**), a huge set of documents written in the markup language, HTML, and linked with one another via hyperlinks. Clients, known as browsers, became the user's window to WWW. Ease of creation, editing, and publishing of these web documents meant an explosion of document volume on the Web.

In the latter half of the 90s, the huge volume of web documents led to discoverability problems. Users found it hard to discover and locate the right document for their information needs, leading to a gold rush among web companies in the space of web discovery and search. Search engines and directory services for the Web, such as Lycos, Altavista, Yahoo!, and Ask Jeeves, became commonplace.

These search engines started ingesting and summarizing the Web. The process of traversing the Web and ingesting the documents is known as **crawling**. Smart crawlers, those that can download documents quickly, avoid link cycles, and detect document updates, have been developed.

In the early part of this century, Google emerged as the torchbearer of the search technology. Its success was attributed not only to the introduction of robust, spam-defiant relevance technology, but also its minimalistic approach, speed, and quick data processing. It achieved the former goals by developing novel concepts such as **PageRank**, and the latter goals by innovative tweaking and applying existing techniques, such as MapReduce, for large-scale parallel and distributed data processing.



PageRank is an algorithm named after Google's founder Larry Page. It is one of the algorithms used to rank web search results for a user. Search engines use keyword matching on websites to determine relevance corresponding to a search query. This prompts spammers to include many keywords, relevant or irrelevant, on websites to trick these search engines and appear in almost all queries. For example, a car dealer can include keywords related to shopping or movies and appear in a wider range of search queries. The user experience suffers because of irrelevant results.

PageRank thwarted this kind of fraud by analyzing the quality and quantity of links to a particular web page. The intention was that important pages have more inbound links.

In Circa 2004, Google published and disclosed its MapReduce technique and implementation to the world. It introduced **Google File System** (**GFS**) that accompanies the MapReduce engine. Since then, the MapReduce paradigm has become the most popular technique to process massive datasets in parallel and distributed settings across many other companies. Hadoop is an open source implementation of the MapReduce framework, and Hadoop and its associated filesystem, HDFS, are inspired by Google's MapReduce and GFS, respectively.

Since its inception, Hadoop and other MapReduce-based systems run a diverse set of workloads from different verticals, web search being one of them. As an example, Hadoop is extensively used in `http://www.last.fm/` to generate charts and track usage statistics. It is used for log processing in the cloud provider, Rackspace. Yahoo!, one of the biggest proponents of Hadoop, uses Hadoop clusters not only to build web indexes for search, but also to run sophisticated advertisement placement and content optimization algorithms.

The evolution of Hadoop

Around the year 2003, Doug Cutting and Mike Cafarella started work on a project called **Nutch**, a highly extensible, feature-rich, and open source crawler and indexer project. The goal was to provide an off-the-shelf crawler to meet the demands of document discovery. Nutch can work in a distributed fashion on a handful of machines and be polite by respecting the `robots.txt` file on websites. It is highly extensible by providing the plugin architecture for developers to add custom components, for example, third-party plugins, to read different media types from the Web.



Robot Exclusion Standard or the **robots.txt** protocol is an advisory protocol that suggests crawling behavior. It is a file placed on website roots that suggest the public pages and directories that can or cannot be crawled. One characteristic of a polite crawler is its respect for the advisory instructions placed within the `robots.txt` file.

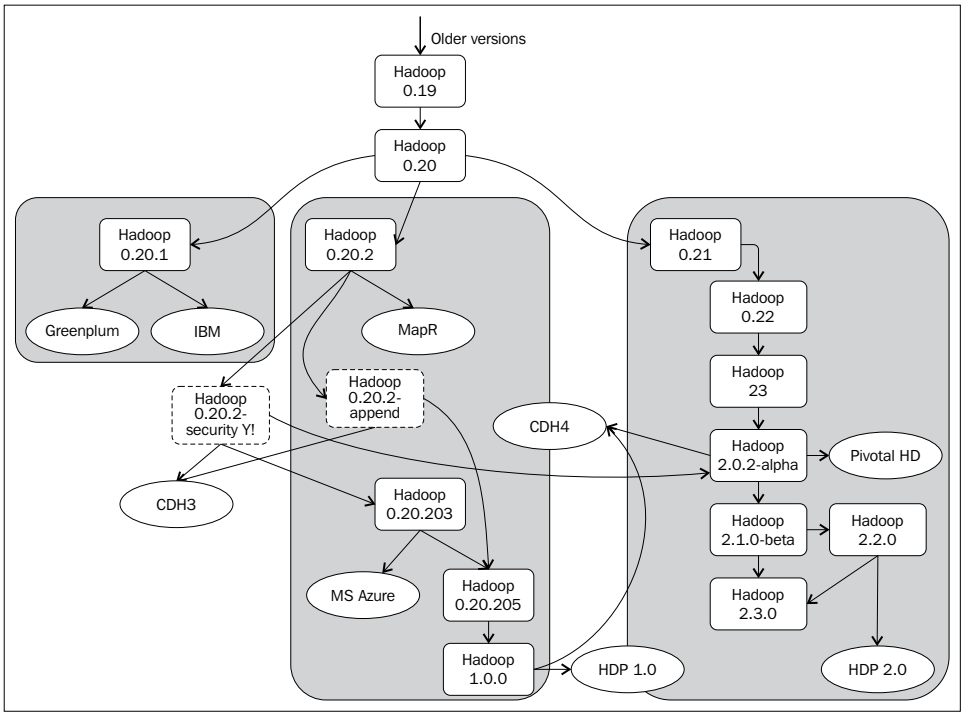
Nutch, together with indexing technologies such as Lucene and Solr, provided the necessary components to build search engines, but this project was not at web scale. The initial demonstration of Nutch involved crawling 100 million web pages using four machines. Moreover, debugging and maintaining it was tedious. In 2004, concepts from the seminal MapReduce and GFS publications from Google addressed some of Nutch's scaling issues. The Nutch contributors started integrating distributed filesystem features and the MapReduce programming model into the project. The scalability of Nutch improved by 2006, but it was not yet web scale. A few 100 million web documents could be crawled and indexed using 20 machines. Programming, debugging, and maintaining these search engines became easier.

In 2006, Yahoo hired Doug Cutting, and Hadoop was born. The Hadoop project was part of **Apache Software Foundation** (**ASF**), but was factored out of the existing Nutch project and allowed to evolve independently. A number of minor releases were done between 2006 and 2008, at the end of which Hadoop became a stable and web-scale data-processing MapReduce framework. In 2008, Hadoop won the terabyte sort benchmark competition, announcing its suitability for large-scale, reliable cluster-computing using MapReduce.

Hadoop's genealogy

The Hadoop project has a long genealogy, starting from the early releases in 2007 and 2008. This project that is part of Apache Software Foundation (ASF) will be termed **Apache Hadoop** throughout this book. The Apache Hadoop project is the parent project for subsequent releases of Hadoop and its distribution. It is analogous to the main stem of a river, while branches or distributions can be compared to the distributaries of a river.

The following figure shows the Hadoop lineage with respect to Apache Hadoop. In the figure, the black squares represent the major Apache Hadoop releases, and the ovals represent the distributions of Hadoop. Other releases of Hadoop are represented by dotted black squares.



Apache Hadoop has three important branches that are very relevant. They are:

- The 0.20.1 branch
- The 0.20.2 branch
- The 0.21 branch

The Apache Hadoop releases followed a straight line till 0.20. It always had a single major release, and there was no forking of the code into other branches. At release 0.20, there was a fan out of the project into three major branches. The 0.20.2 branch is often termed MapReduce v1.0, MRv1, or simply Hadoop 1.0.0. The 0.21 branch is termed MapReduce v2.0, MRv2, or Hadoop 2.0. A few older distributions are derived from 0.20.1. The year 2011 marked a record number of releases across different branches.

There are two other releases of significance, though they are not considered major releases. They are the Hadoop-0.20-append and Hadoop-0.20-Security releases. These releases introduced the HDFS append and security-related features into Hadoop, respectively. With these enhancements, Apache Hadoop came closer to becoming enterprise-ready.

Hadoop-0.20-append

Append is the primary feature of the Hadoop-0.20-append release. It allows users to run HBase without the risk of data loss. HBase is a popular column-family store that runs on HDFS, providing an online data store in a batch-oriented Hadoop environment. Specifically, the append feature helps write durability of HBase logs, ensuring data safety. Traditionally, HDFS supported input-output for MapReduce batch jobs. The requirement for these jobs was to open a file once, write a lot of data into it, and close the file. The closed file was immutable and read many times. The semantics supported were write-once-read-many-times. No one could read the file when a write was in progress.

Any process that failed or crashed during a write had to rewrite the file. In MapReduce, a user always reran tasks to generate the file. However, this is not true for transaction logs for online systems such as HBase. If the log-writing process fails, it can lead to data loss as the transaction cannot be reproduced. Reproducibility of a transaction, and in turn data safety, comes from log writing. The append feature in HDFS mitigates this risk by enabling HBase and other transactional operations on HDFS.

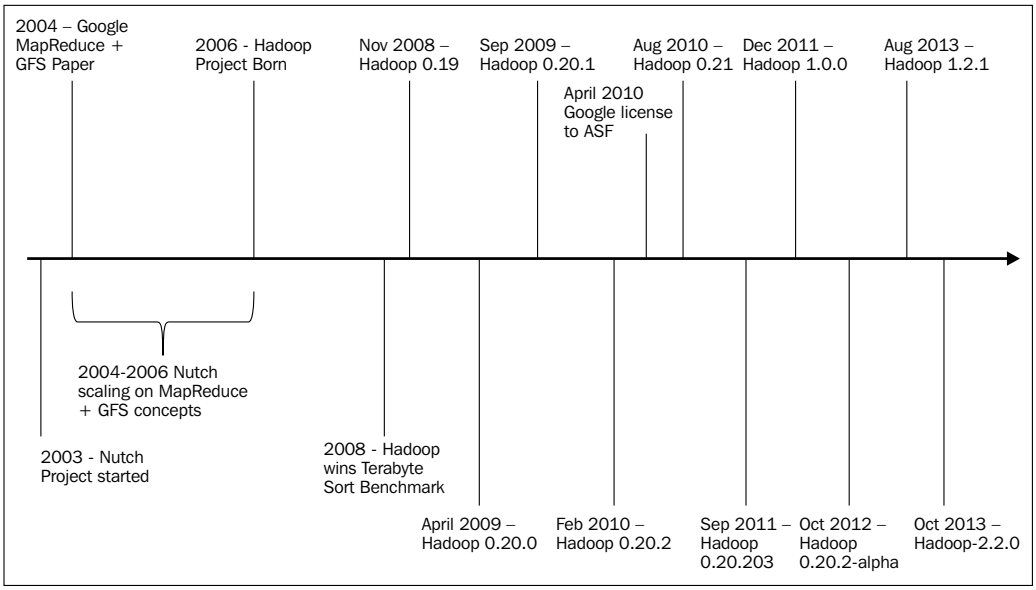
Hadoop-0.20-security

The Hadoop team at Yahoo took the initiative to add security-related features in the Hadoop-0.20-Security release. Enterprises have different teams, with each team working on different kinds of data. For compliance, client privacy, and security, isolation, authentication, and authorization of Hadoop jobs and data is important. The security release is feature-rich to provide these three pillars of enterprise security.

The full Kerberos authentication system is integrated with Hadoop in this release. **Access Control Lists** (**ACLs**) were introduced on MapReduce jobs to ensure proper authority in exercising jobs and using resources. Authentication and authorization put together provided the isolation necessary between both jobs and data of the different users of the system.

Hadoop's timeline

The following figure gives a timeline view of the major releases and milestones of Apache Hadoop. The project has been there for 8 years, but the last 4 years has seen Hadoop make giant strides in big data processing. In January 2010, Google was awarded a patent for the MapReduce technology. This technology was licensed to the Apache Software Foundation 4 months later, a shot in the arm for Hadoop. With legal complications out of the way, enterprises—small, medium, and large—were ready to embrace Hadoop. Since then, Hadoop has come up with a number of major enhancements and releases. It has given rise to businesses selling Hadoop distributions, support, training, and other services.



Hadoop 1.0 releases, referred to as 1.X in this book, saw the inception and evolution of Hadoop as a pure MapReduce job-processing framework. It has exceeded its expectations with a wide adoption of massive data processing. The stable 1.X release at this point of time is 1.2.1, which includes features such as append and security. Hadoop 1.X tried to stay flexible by making changes, such as HDFS append, to support online systems such as HBase. Meanwhile, big data applications evolved in range beyond MapReduce computation models. The flexibility of Hadoop 1.X releases had been stretched; it was no longer possible to widen its net to cater to the variety of applications without architectural changes.

Hadoop 2.0 releases, referred to as 2.X in this book, came into existence in 2013. This release family has major changes to widen the range of applications Hadoop can solve. These releases can even increase efficiencies and mileage derived from existing Hadoop clusters in enterprises. Clearly, Hadoop is moving fast beyond MapReduce to stay as the leader in massive scale data processing with the challenge of being backward compatible. It is becoming a generic cluster-computing and storage platform from being only a MapReduce-specific framework.

Hadoop 2.X

The extensive success of Hadoop 1.X in organizations also led to the understanding of its limitations, which are as follows:

- Hadoop gives unprecedented access to cluster computational resources to every individual in an organization. The MapReduce programming model is simple and supports a *develop once deploy at any scale* paradigm. This leads to users exploiting Hadoop for data processing jobs where MapReduce is not a good fit, for example, web servers being deployed in long-running *map* jobs. MapReduce is not known to be affable for iterative algorithms. Hacks were developed to make Hadoop run iterative algorithms. These hacks posed severe challenges to cluster resource utilization and capacity planning.
- Hadoop 1.X has a centralized job flow control. Centralized systems are hard to scale as they are the single point of load lifting. JobTracker failure means that all the jobs in the system have to be restarted, exerting extreme pressure on a centralized component. Integration of Hadoop with other kinds of clusters is difficult with this model.

- The early releases in Hadoop 1.X had a single NameNode that stored all the metadata about the HDFS directories and files. The data on the entire cluster hinged on this single point of failure. Subsequent releases had a cold standby in the form of a secondary NameNode. The secondary NameNode merged the edit logs and NameNode image files, periodically bringing in two benefits. One, the primary NameNode startup time was reduced as the NameNode did not have to do the entire merge on startup. Two, the secondary NameNode acted as a replica that could minimize data loss on NameNode disasters. However, the secondary NameNode (secondary NameNode is not a backup node for NameNode) was still not a hot standby, leading to high failover and recovery times and affecting cluster availability.
- Hadoop 1.X is mainly a Unix-based massive data processing framework. Native support on machines running Microsoft Windows Server is not possible. With Microsoft entering cloud computing and big data analytics in a big way, coupled with existing heavy Windows Server investments in the industry, it's very important for Hadoop to enter the Microsoft Windows landscape as well.
- Hadoop's success comes mainly from enterprise play. Adoption of Hadoop mainly comes from the availability of enterprise features. Though Hadoop 1.X tries to support some of them, such as security, there is a list of other features that are badly needed by the enterprise.

Yet Another Resource Negotiator (YARN)

In Hadoop 1.X, resource allocation and job execution were the responsibilities of JobTracker. Since the computing model was closely tied to the resources in the cluster, MapReduce was the only supported model. This tight coupling led to developers force-fitting other paradigms, leading to unintended use of MapReduce.

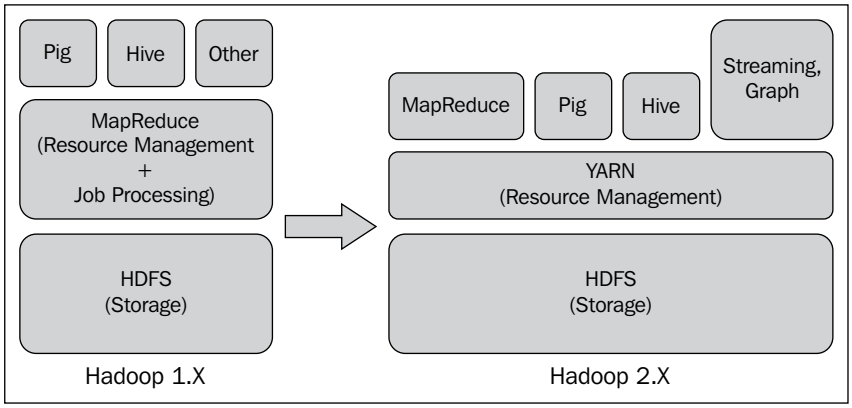
The primary goal of YARN is to separate concerns relating to resource management and application execution. By separating these functions, other application paradigms can be added onboard a Hadoop computing cluster. Improvements in interoperability and support for diverse applications lead to efficient and effective utilization of resources. It integrates well with the existing infrastructure in an enterprise.

Achieving loose coupling between resource management and job management should not be at the cost of loss in backward compatibility. For almost 6 years, Hadoop has been the leading software to crunch massive datasets in a parallel and distributed fashion. This means huge investments in development; testing and deployment were already in place.

YARN maintains backward compatibility with Hadoop 1.X (hadoop-0.20.205+) APIs. An older MapReduce program can continue execution in YARN with no code changes. However, recompiling the older code is mandatory.

Architecture overview

The following figure lays out the architecture of YARN. YARN abstracts out resource management functions to a platform layer called **ResourceManager** (**RM**). There is a per-cluster RM that primarily keeps track of cluster resource usage and activity. It is also responsible for allocation of resources and resolving contentions among resource seekers in the cluster. RM uses a generalized resource model and is agnostic to application-specific resource needs. For example, RM need not know the resources corresponding to a single Map or Reduce slot.



Planning and executing a single job is the responsibility of **Application Master** (**AM**). There is an AM instance per running application. For example, there is an AM for each MapReduce job. It has to request for resources from the RM, use them to execute the job, and work around failures, if any.

The general cluster layout has RM running as a daemon on a dedicated machine with a global view of the cluster and its resources. Being a global entity, RM can ensure fairness depending on the resource utilization of the cluster resources. When requested for resources, RM allocates them dynamically as a node-specific bundle called a **container**. For example, 2 CPUs and 4 GB of RAM on a particular node can be specified as a container.

Every node in the cluster runs a daemon called **NodeManager** (**NM**). RM uses NM as its node local assistant. NMs are used for container management functions, such as starting and releasing containers, tracking local resource usage, and fault reporting. NMs send heartbeats to RM. The RM view of the system is the aggregate of the views reported by each NM.

Jobs are submitted directly to RMs. Based on resource availability, jobs are scheduled to run by RMs. The metadata of the jobs are stored in persistent storage to recover from RM crashes. When a job is scheduled, RM allocates a container for the AM of the job on a node in the cluster.

AM then takes over orchestrating the specifics of the job. These specifics include requesting resources, managing task execution, optimizations, and handling tasks or job failures. AM can be written in any language, and different versions of AM can execute independently on a cluster.

An AM resource request contains specifications about the locality and the kind of resource expected by it. RM puts in its best effort to satisfy AM's needs based on policies and availability of resources. When a container is available for use by AM, it can launch application-specific code in this container. The container is free to communicate with its AM. RM is agnostic to this communication.

Storage layer enhancements

A number of storage layer enhancements were undertaken in the Hadoop 2.X releases. The number one goal of the enhancements was to make Hadoop enterprise ready.

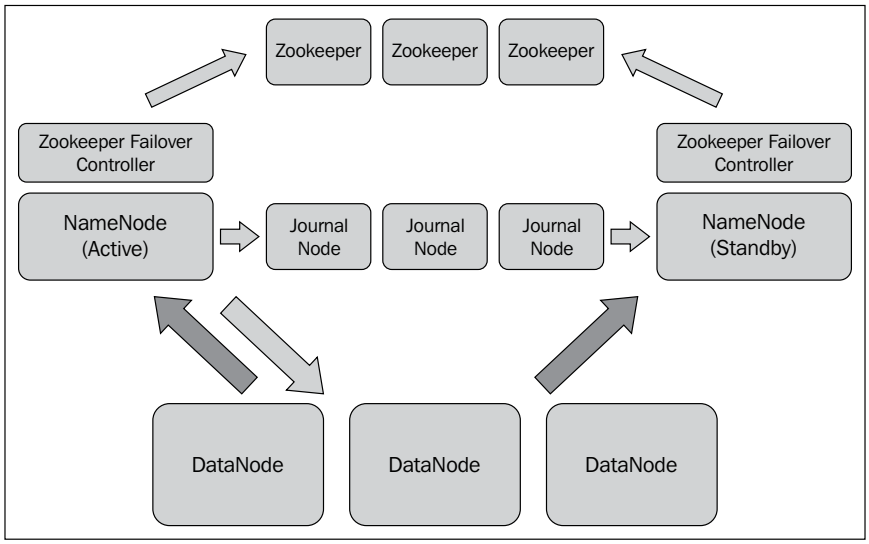
High availability

NameNode is a directory service for Hadoop and contains metadata pertaining to the files within cluster storage. Hadoop 1.X had a secondary Namenode, a cold standby that needed minutes to come up. Hadoop 2.X provides features to have a hot standby of NameNode. On the failure of an active NameNode, the standby can become the active Namenode in a matter of minutes. There is no data loss or loss of NameNode service availability. With hot standbys, automated failover becomes easier too.

The key to keep the standby in a hot state is to keep its data as current as possible with respect to the active Namenode. This is achieved by reading the edit logs of the active NameNode and applying it onto itself with very low latency. The sharing of edit logs can be done using the following two methods:

- A shared NFS storage directory between the active and standby NameNodes: the active writes the logs to the shared location. The standby monitors the shared directory and pulls in the changes.
- A quorum of Journal Nodes: the active NameNode presents its edits to a subset of journal daemons that record this information. The standby node constantly monitors these journal daemons for updates and syncs the state with itself.

The following figure shows the high availability architecture using a quorum of Journal Nodes. The data nodes themselves send block reports directly to both the active and standby NameNodes:



Zookeeper or any other High Availability monitoring service can be used to track NameNode failures. With the assistance of Zookeeper, failover procedures to promote the hot standby as the active NameNode can be triggered.

HDFS Federation

Similar to what YARN did to Hadoop's computation layer, a more generalized storage model has been implemented in Hadoop 2.X. The block storage layer has been generalized and separated out from the filesystem layer. This separation has given an opening for other storage services to be integrated into a Hadoop cluster. Previously, HDFS and the block storage layer were tightly coupled.

One use case that has come forth from this generalized storage model is **HDFS Federation**. Federation allows multiple HDFS namespaces to use the same underlying storage. Federated NameNodes provide isolation at the filesystem level. In *Chapter 10*, *HDFS Federation*, we will delve into the details of this feature.

HDFS snapshots

Snapshots are point-in-time, read-only images of the entire or a particular subset of a filesystem. Snapshots are taken for three general reasons:

- Protection against user errors
- Backup
- Disaster recovery

Snapshotting is implemented only on NameNode. It does not involve copying data from the data nodes. It is a persistent copy of the block list and file size. The process of taking a snapshot is almost instantaneous and does not affect the performance of NameNode.

Other enhancements

There are a number of other enhancements in Hadoop 2.X, which are as follows:

- The wire protocol for RPCs within Hadoop is now based on **Protocol Buffers**. Previously, Java serialization via Writables was used. This improvement not only eases maintaining backward compatibility, but also aids in *rolling the upgrades* of different cluster components. RPCs allow for client-side retries as well.
- HDFS in Hadoop 1.X was agnostic about the type of storage being used. Mechanical or SSD drives were treated uniformly. The user did not have any control on data placement. Hadoop 2.X releases in 2014 are aware of the type of storage and expose this information to applications as well. Applications can use this to optimize their data fetch and placement strategies.
- HDFS append support has been brought into Hadoop 2.X.

- HDFS access in Hadoop 1.X releases has been through HDFS clients. In Hadoop 2.X, support for NFSv3 has been brought into the NFS gateway component. Clients can now mount HDFS onto their compatible local filesystem, allowing them to download and upload files directly to and from HDFS. Appends to files are allowed, but random writes are not.
- A number of I/O improvements have been brought into Hadoop. For example, in Hadoop 1.X, clients collocated with data nodes had to read data via TCP sockets. However, with *short-circuit local reads*, clients can directly read off the data nodes. This particular interface also supports zero-copy reads. The CRC checksum that is calculated for reads and writes of data has been optimized using the Intel SSE4.2 CRC32 instruction.

Support enhancements

Hadoop is also widening its application net by supporting other platforms and frameworks. One dimension we saw was onboarding of other computational models with YARN or other storage systems with the Block Storage layer.
The other enhancements are as follows:

- Hadoop 2.X supports Microsoft Windows natively. This translates to a huge opportunity to penetrate the Microsoft Windows server land for massive data processing. This was partially possible because of the use of the highly portable Java programming language for Hadoop development. The other critical enhancement was the generalization of compute and storage management to include Microsoft Windows.
- As part of Platform-as-a-Service offerings, cloud vendors give out on-demand Hadoop as a service. OpenStack support in Hadoop 2.X makes it conducive for deployment in elastic and virtualized cloud environments.

Hadoop distributions

In the present day, Hadoop and its individual ecosystem components are complex projects. As we saw earlier in this chapter, Hadoop has a number of different forks or code branches over a large number of releases. There are also a lot of different distributions of Hadoop. The distribution with the most activity and community involvement is the one that resides as part of Apache Software Foundation. This distribution is free and has a very large community behind it. The community contributions to the Apache Hadoop distribution shape the general direction taken by Hadoop. Support in the Apache Hadoop distribution is via online forums, where questions are addressed to the community and answered by its members.

Deployment and management of the Apache Hadoop distribution within an enterprise is tedious and nontrivial. Apache Hadoop is written in Java and optimized to run on Linux filesystems. This can lead to impedance mismatch between Hadoop and existing enterprise applications and infrastructures. Integration between the Hadoop ecosystem components is buggy and not straightforward.

To bridge these issues, a few companies came up with distribution models for Hadoop. There are three primary kinds of Hadoop distribution flavors. One flavor is to provide commercial or paid support and training for the Apache Hadoop distribution. Secondly, there are companies that provide a set of supporting tools for deployment and management of Apache Hadoop as an alternative flavor. These companies also provide robust integration layers between the different Hadoop ecosystem components. The third model is for companies to supplement Apache Hadoop with proprietary features and code. These features are paid enhancements, many of which solve certain use cases.

The parent of all these distributions is Apache Software Foundation's Hadoop sources. Users of these other distributions, particularly from companies following the third distribution model, might integrate proprietary code into Apache Hadoop. However, these distributions will always stay in touching distance with Apache Hadoop and follow its trends. Distributions are generally well tested and supported in a deep and timely manner, saving administration and management costs for an organization. The downside of using a distribution other than Apache Hadoop is vendor lock-in. The tools and proprietary features provided by one vendor might not be available in another distribution or be noncompatible with other third-party tools, bringing in a cost of migration. The cost of migration is not limited to technology shifts alone. It also involves training, capacity planning, and rearchitecting costs for the organization.

Which Hadoop distribution?

There are a number of Hadoop distributions offered by companies since 2008. Distributions excel in some or the other attribute. Decisions on the right distribution for an enterprise or organization should be made on a case-by-case basis. There are different criteria to evaluate distributions. We will inspect a few important ones.

Performance

The ability of the Hadoop distribution running on a cluster to process data quickly is obviously a desired feature. Traditionally, this has been the cornerstone for all performance benchmarks. This particular performance measure is termed as "throughput". A wide range of analysis workloads that are being processed on Hadoop, coupled with the diversity of use cases supported by analytics, brings in "latency" as an important performance criterion as well. The ability of the cluster to ingest input data and emit output data at a quick rate becomes very important for low-latency analytics. This input-output cost forms an integral part of the data processing workflow.



Latency is the time required to perform an action. It is measured in time units such as milliseconds, seconds, minutes, or hours.

Throughput is the number of actions that can be performed in unit time. It gives a sense of the amount of work done for every time unit.

Scaling up hardware is one way to achieve low latency independent of the Hadoop distribution. However, this approach will be expensive and saturate out quickly. Architecturally, low I/O latency can be achieved in different ways; one will be able to reduce the number of intermediate data-staging layers between the data source or the data sink and Hadoop cluster. Some distributions provide streaming writes into the Hadoop cluster in an attempt to reduce intermediate staging layers. Operators used for filtering, compressing, and lightweight data processing can be plugged into the streaming layer to preprocess the data before it flows into storage.

The Apache Hadoop distribution is written in Java, a language that runs in its own virtual machine. Though this increases application portability, it comes with overheads such as an extra layer of indirection during execution by means of byte-code interpretation and background garbage collection. It is not as fast as an application directly compiled for target hardware. Some vendors optimize their distributions for particular hardware, increasing job performance per node. Features such as compression and decompression can also be optimized for certain hardware types.

Scalability

Over time, data outgrows the physical capacity of the compute and storage resources provisioned by an organization. This will require expansion of resources in both the compute and storage dimensions. Scaling can be done vertically or horizontally. Vertical scaling or scaling up is expensive and tightly bound to hardware advancements. Lack of elasticity is another downside with vertical scaling. Horizontal scaling or scaling out is a preferred mode of scaling compute and storage.

Ideally, scaling out should be limited to addition of more nodes and disks to the cluster network, with minimal configuration changes. However, distributions might impose different degrees of difficulty, both in terms of effort and cost on scaling a Hadoop cluster. Scaling out might mean heavy administrative and deployment costs, rewriting a lot of the application's code, or a combination of both. Scaling costs will depend on the existing architecture and how it complements and complies with the Hadoop distribution that is being evaluated.



Vertical scaling or **scaling up** is the process of adding more resources to a single node in a system. For example, adding additional CPUs, memory, or storage to a single computer comes under this bucket of scaling. Vertical scaling increases capacity, but does not decrease system load.

Horizontal scaling or **scaling out** is the process of adding additional nodes to a system. For example, adding another computer to a distributed system by connecting it to the network comes under this category of scaling. Horizontal scaling decreases the load on a system as the new node takes a part of the load. The capacity of individual nodes does not increase.

Reliability

Any distributed system is subject to partial failures. Failures can stem from hardware, software, or network issues, and have a smaller mean time when running on commodity hardware. Dealing with these failures without disrupting services or compromising data integrity is the primary goal of any highly available and consistent system.

A distribution that treats reliability seriously provides high availability of its components out of the box. Eliminating **Single Point of Failures** (**SPOF**) ensures availability. The means of eliminating SPOFs is to increase the redundancy of components. For a long time, Apache Hadoop had a single NameNode. Any failure to the NameNode's hardware meant the entire cluster becoming unusable. Now, there is the concept of a secondary NameNode and hot standbys that can be used to restore the name node in the event of NameNode failure.

Distributions that reduce manual tasks for cluster administrators are more reliable. Human intervention is directly correlated to higher error rates. An example of this is handling failovers. Failovers are critical periods for systems as they operate with lower degrees of redundancy. Any error during these periods can be disastrous for the application. Also, automated failover handling means the system can recover and run in a short amount of time. Lower the recovery time from failure better is the availability of the system.

The integrity of data needs to be maintained during normal operations and when failures are encountered. Data checksums for error detection and possible recovery, data replication, data mirroring, and snapshots are some ways to ensure data safety. Replication follows the redundancy theme to ensure data availability. Rack-aware smart placement of data and handling under or over replication are parameters to watch out for. Mirroring helps recovery from site failures by asynchronous replication across the Internet. Snapshotting is a desirable feature in any distribution; not only do they aid disaster recovery but also facilitate offline access to data. Data analytics involves experimentation and evaluation of rich data. Snapshots can be a way to facilitate this to a data scientist without disrupting production.

Manageability

Deploying and managing the Apache Hadoop open source distribution requires internal understanding of the source code and configuration. This is not a widely available skill in IT administration. Also, administrators in enterprises are caretakers of a wide range of systems, Hadoop being one of them.

Versions of Hadoop and its ecosystem components that are supported by a distribution might need to be evaluated for suitability. Newer versions of Hadoop support paradigms other than MapReduce within clusters. Depending on the plans of the enterprise, newer versions can increase the efficiency of enterprise-provisioned hardware.

Capabilities of Hadoop management tools are key differentiators when choosing an appropriate distribution for an enterprise. Management tools need to provide centralized cluster administration, resource management, configuration management, and user management. Job scheduling, automatic software upgrades, user quotas, and centralized troubleshooting are other desirable features.

Monitoring cluster health is also a key feature in the manageability function. Dashboards for visualization of cluster health and integration points for other tools are good features to have in distribution. Ease of data access is another parameter that needs to be evaluated; for example, support for POSIX filesystems on Hadoop will make browsing and accessing data convenient for engineers and scientists within any enterprise. On the flip side, this makes mutability of data possible, which can prove to be risky in certain situations.

Evaluation of options for data security of a distribution is extremely important as well. Data security entails authentication of a Hadoop user and authorization to datasets and data confidentiality. Every organization or enterprise might have its authentication systems such as Kerberos or LDAP already in place. Hadoop distribution, with capabilities to integrate with existing authentication systems, is a big plus in terms of lower costs and higher compliance. Fine-grained authorization might help control access to datasets and jobs at different levels. When data is moving in and out of an organization, encryption of the bits travelling over the wire becomes important to protect against data snooping.

Distributions offer integration with development and debugging tools. Developers and scientists in an enterprise will already be using a set of tools. The more overlap between the toolset used by the organization and distribution, the better it is. The advantage of overlap not only comes in the form of licensing costs, but also in a lesser need for training and orientation. It might also increase productivity within the organization as people are already accustomed to certain tools.

Available distributions

There are a number of distributions of Hadoop. A comprehensive list can be found at `http://wiki.apache.org/hadoop/Distributions%20and%20Commercial%20Support`. We will be examining four of them:

- Cloudera Distribution of Hadoop (CDH)
- Hortonworks Data Platform (HDP)
- MapR
- Pivotal HD

Cloudera Distribution of Hadoop (CDH)

Cloudera was formed in March 2009 with a primary objective of providing Apache Hadoop software, support, services, and training for enterprise-class deployment of Hadoop and its ecosystem components. The software suite is branded as **Cloudera Distribution of Hadoop** (**CDH**). The company being one of the Apache Software Foundation sponsors, pushes most enhancements it makes during support and servicing of Hadoop deployments upstream back into Apache Hadoop.

CDH is in its fifth major version right now and is considered a mature Hadoop distribution. The paid version of CDH comes with a proprietary management software, Cloudera Manager.

Hortonworks Data Platform (HDP)

The Yahoo Hadoop team spurned off to form Hortonworks in June, 2011, a company with objectives similar to Cloudera. Their distribution is branded as **Hortonworks Data Platform** (**HDP**). The HDP suite's Hadoop and other software are completely free, with paid support and training. Hortonworks also pushes enhancements upstream, back to Apache Hadoop.

HDP is in its second major version currently and is considered the rising star in Hadoop distributions. It comes with a free and open source management software called Ambari.

MapR

MapR was founded in 2009 with a mission to bring enterprise-grade Hadoop. The Hadoop distribution they provide has significant proprietary code when compared to Apache Hadoop. There are a handful of components where they guarantee compatibility with existing Apache Hadoop projects. Key proprietary code for the MapR distribution is the replacement of HDFS with a POSIX-compatible NFS. Another key feature is the capability of taking snapshots.

MapR comes with its own management console. The different grades of the product are named as M3, M5, and M7. M5 is a standard commercial distribution from the company, M3 is a free version without high availability, and M7 is a paid version with a rewritten HBase API.

Pivotal HD

Greenplum is a marquee parallel data store from EMC. EMC integrated Greenplum within Hadoop, giving way to an advanced Hadoop distribution called Pivotal HD. This move alleviated the need to import and export data between stores such as Greenplum and HDFS, bringing down both costs and latency.

The HAWQ technology provided by Pivotal HD allows efficient and low-latency query execution on data stored in HDFS. The HAWQ technology has been found to give 100 times more improvement on certain MapReduce workloads when compared to Apache Hadoop. HAWQ also provides SQL processing in Hadoop, increasing its popularity among users who are familiar with SQL.

Summary

In this chapter, we saw the evolution of Hadoop and some of its milestones and releases. We went into depth on Hadoop 2.X and the changes it brings into Hadoop. The key takeaways from this chapter are:

- MapReduce was born out of the necessity to gather, process, and index data at web scale. Apache Hadoop is an open source distribution of the MapReduce computational model.
- In over 6 years of its existence, Hadoop has become the number one choice as a framework for massively parallel and distributed computing. The community has been shaping Hadoop to gear up for enterprise use. In 1.X releases, HDFS append and security, were the key features that made Hadoop enterprise-friendly.
- MapReduce supports a limited set of use cases. Onboarding other paradigms into Hadoop enables support for a wider range of analytics and can also increase cluster resource utilization. In Hadoop 2.X, the JobTracker functions are separated and YARN handles cluster resource management and scheduling. MapReduce is one of the applications that can run on YARN.
- Hadoop's storage layer was enhanced in 2.X to separate the filesystem from the block storage service. This enables features such as supporting multiple namespaces and integration with other filesystems. 2.X shows improvements in Hadoop storage availability and snapshotting.
- Distributions of Hadoop provide enterprise-grade management software, tools, support, training, and services. Most distributions shadow Apache Hadoop in their capabilities.

MapReduce is still an integral part of Hadoop's DNA. In the next chapter, we will explore MapReduce optimizations and best practices.

2
Advanced MapReduce

MapReduce is a programming model for parallel and distributed processing of data. It consists of two steps: **Map** and **Reduce**. These steps are inspired from functional programming, a branch of computer science that deals with mathematical functions as computational units. Properties of functions such as immutability and statelessness are attractive for parallel and distributed processing. They provide a high degree of parallelism and fault tolerance at lower costs and semantic complexity.

In this chapter, we will look at advanced optimizations when running MapReduce jobs on Hadoop clusters. Every MapReduce job has input data and a Map task per split of this data. The Map task calls a `map` function repeatedly on every record, represented as a key-value pair. The `map` is a function that transforms data from one domain to another. The intermediate output records of each Map task are shuffled and sorted before transferring it downstream to the Reduce tasks. Intermediate data with the same keys go to the same Reduce task. The Reduce task calls the `reduce` function for a key and all its associated values. Outputs are then collected and stored.

The Map step has the greatest degree of parallelism. It is used to implement operations such as filtering, sorting, and transformations on data. The Reduce step is used to implement summarization operations on data. Hadoop also provides features such as **DistributedCache** as a side channel to distribute data and **Counters** to collect job-related global statistics. We will be looking at their utility in processing MapReduce jobs.

The advanced features and optimizations will be explained with the help of examples of code. Hadoop 2.2.0 will be used throughout this chapter. It is assumed that you have access to the Java development environment and a Hadoop cluster, either in your organization, the cloud, or as a standalone/pseudo-distributed mode installation on your personal computers. You need to have knowledge on how to compile Java programs and run Hadoop jobs to try out the examples.

In this chapter, we will look at the following topics:

- The different phases of a MapReduce job and the optimizations that can be applied at each phase. The input, Map, Shuffle/Sort, Reduce, and the output phases will be covered in depth with relevant examples.
- The application of useful Hadoop features such as DistributedCache and Counters.
- The types of data joins that can be achieved in a MapReduce job and the patterns to achieve them.

MapReduce input

The Map step of a MapReduce job hinges on the nature of the input provided to the job. The Map step provides maximum parallelism gains, and crafting this step smartly is important for job speedup. Data is split into chunks, and Map tasks operate on each of these chunks of data. Each chunk is called `InputSplit`. A Map task is asked to operate on each `InputSplit` class. There are two other classes, `InputFormat` and `RecordReader`, which are significant in handling inputs to Hadoop jobs.

The InputFormat class

The input data specification for a MapReduce Hadoop job is given via the `InputFormat` hierarchy of classes. The `InputFormat` class family has the following main functions:

- Validating the input data. For example, checking for the presence of the file in the given path.
- Splitting the input data into logical chunks (`InputSplit`) and assigning each of the splits to a Map task.
- Instantiating a `RecordReader` object that can work on each `InputSplit` class and producing records to the Map task as key-value pairs.

The `FileInputFormat` subclass and associated classes are commonly used for jobs that take inputs from HFDS. The `DBInputFormat` subclass is a specialized class that can be used to read data from a SQL database. `CombineFileInputFormat` is the direct abstract subclass of the `FileInputFormat` class, which can combine multiple files into a single split.

The InputSplit class

The abstract `InputSplit` class and its associated concrete classes represent a byte view of the input data. An `InputSplit` class is characterized by the following main attributes:

- The input filename
- The byte offset in the file where the split starts
- The length of the split in bytes
- The node locations where the split resides

In HDFS, an `InputSplit` class is created per file if the file size is less than the HDFS block size. For example, if the HDFS block size is 128 MB, any file with a size less than 128 MB resides in its own `InputSplit` class. For files that are broken up into blocks (size of the file is greater than the HDFS block size), a more complex formula is used to calculate `InputSplit`. The `InputSplit` class has an upper bound of the HDFS block size, unless the minimum size of a split is greater than the block size. Such cases are rare and could lead to locality problems.

Based on the locations of the splits and the availability of resources, the scheduler makes a decision on which node should execute the Map task for that split. The split is then communicated to the node that executes the task.

```
InputSplitSize = Maximum(minSplitSize,
Minimum(blocksize, maxSplitSize))

minSplitSize :
mapreduce.input.fileinputformat.split.minsize

blocksize: dfs.blocksize

maxSplitSize -
mapreduce.input.fileinputformat.split.maxsize
```



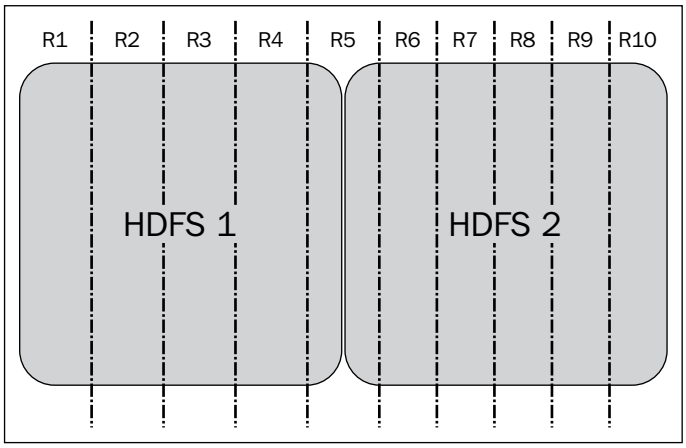
In previous releases of Hadoop, the minimum split size property was `mapred.min.split.size` and the maximum split size was given by the value of the property `mapred.max.split.size`. These are deprecated now.

The RecordReader class

Unlike `InputSplit`, the `RecordReader` class presents a record view of the data to the Map task. `RecordReader` works within each `InputSplit` class and generates records from the data in the form of key-value pairs. The `InputSplit` boundary is a guideline for `RecordReader` and is not enforced. On one extreme, a custom `RecordReader` class can be written to read an entire file (though this is not encouraged). Most often, a `RecordReader` class will have to read from a subsequent `InputSplit` class to present the complete record to the Map task. This happens when records overlap InputSplit classes.

The reading of bytes from a subsequent `InputSplit` class happens via the `FSDataInputStream` objects. Though this reading does not respect locality in itself, generally, it gathers only a few bytes from the next split and there is not a significant performance overhead. But in some cases where record sizes are huge, this can have a bearing on the performance due to significant byte transfers across nodes.

In the following diagram, a file with two HDFS blocks has the record **R5** spanning both blocks. It is assumed that the minimum split size is less than the block size. In this case, `RecordReader` is going to gather the complete record by reading bytes off the next block of data.



File with two blocks and record R5 spanning blocks

Hadoop's "small files" problem

Hadoop's problem with small files—files that are significantly smaller than the HDFS block size—is well known. When dealing with small files as input, a Map task is created for each of these files introducing bookkeeping overheads. The same Map task is able to finish processing in a matter of a few seconds, a processing time much smaller than the time taken to spawn and cleanup the task. Each object in the NameNode occupies about 150 bytes of memory. Many small files will proliferate in the presence of these objects and adversely affect NameNode's performance and scalability. Reading a set of smaller files is also very inefficient because of the large number of disk seeks and hops across DataNodes to fetch them.

Unfortunately, small files are a reality, but there are the following strategies to handle small files:

- Combining smaller files into a bigger file as a preprocessing step before storing it in HDFS and running the job. `SequenceFile` and `TFile` formats are popular ways of combining smaller files into a bigger file. Using **Hadoop archive files** (**HAR**) is another way of alleviating NameNode memory pressures. HAR is a meta-filesystem that resides on top of HFDS.
- Using `CombineFileInputFormat` to combine multiple smaller files into `InputSplit`. This also takes into consideration node and rack locality for better performance. It may not relieve the memory requirements of the NameNode though, as the number of files that need to be tracked still remains the same.

To illustrate the working of `CombineFileInputFormat`, we have a public NSF grant proposal abstracts dataset from the years 1990 to 2003 at `https://archive.ics.uci.edu/ml/datasets/NSF+Research+Award+Abstracts+1990-2003`. Though the dataset has 130,000 grant proposals, we will consider a subset of 441 grants. The standard output for a MapReduce Hadoop job that reads each line from the proposals spawns 441 input splits, as shown in following snippet. In this sample job, the number of reduce tasks has been set to zero:

```
14/04/10 07:50:03 INFO input.FileInputFormat: Total input paths to
process : 441
14/04/10 07:50:03 INFO mapreduce.JobSubmitter: number of splits:441
```

As we saw previously, inputs to a Hadoop MapReduce job are specified using the `InputFormat`, `InputSplit`, and `RecordReader` classes. In this program, we will combine all 441 proposals into a single split.

`CombineFileInputFormat` is an abstract class that facilitates input specifications to combine files. The only override that it expects the developer to fill is the `createRecordReader()` method. This is a method that instantiates a custom `RecordReader` class to read records. The `CombineFileInputFormat` class returns the `CombineFileSplit` object in the `getSplits()` method. Each split might be a combination of blocks from different files. If the `setMaxSplitSize()` method is used to set a maximum split size, local node files are combined in a split. Residue blocks are combined with other blocks from the same rack. However, if this value is not set, combining is not attempted at the node level; it is only attempted at the rack level. If the `setMaxSplitSize()` method is used to set the maximum split size to the block size in HDFS, default behavior is seen, that is, each block is a split.

The following code shows the concrete class based on this abstract class:

```
package MasteringHadoop;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.util.LineReader;
import java.io.IOException;
public class MasteringHadoopCombineFileInputFormat extends
    CombineFileInputFormat<LongWritable, Text>{
    @Override
    public RecordReader<LongWritable, Text>
        createRecordReader(InputSplit inputSplit, TaskAttemptContext
            taskAttemptContext) throws IOException {
        return new CombineFileRecordReader<LongWritable,
          Text>((CombineFileSplit) inputSplit, taskAttemptContext,
            MasteringHadoopCombineFileRecordReader.class);
    }
}
```



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.



The `CombineFileFormat` class has an `isSplitable()` method. The default setting is true, but it can be made false to ensure that a file is processed by a single Map task in its entirety.

The following code shows the custom `RecordReader` class that is created to return records from `CombineFileSplit`. The difference between `CombineFileSplit` and `FileSplit` is the presence of multiple paths implying multiple offsets and lengths. The custom `RecordReader` class will be called for every file in the split. Therefore, it is mandatory for the constructor of the custom `RecordReader` class to have an `Integer` index that specifies the file that is being considered for record generation.

The second important method is `nextKeyValue()`, which generates the next key-value pair. The `getCurrentKey()` and `getCurrentValue()` methods return this generated key-value pair. In the following example, keys are byte offsets in the file and values are lines of text. A `LineReader` object is used to read each line:

```
public static class MasteringHadoopCombineFileRecordReader extends
    RecordReader<LongWritable, Text>{
        private LongWritable key;
        private Text value;
        private Path path;
        private FileSystem fileSystem;
        private LineReader lineReader;
        private FSDataInputStream fsDataInputStream;
        private Configuration configuration;
        private int fileIndex;
        private CombineFileSplit combineFileSplit;
        private long start;
        private long end;

        public MasteringHadoopCombineFileRecordReader
            (CombineFileSplit inputSplit, TaskAttemptContext
                context, Integer index) throws IOException{
            this.fileIndex = index;
            this.combineFileSplit = inputSplit;
            this.configuration = context.getConfiguration();
```

```
            this.path = inputSplit.getPath(index);
            this.fileSystem =
                this.path.getFileSystem(configuration);
            this.fsDataInputStream = fileSystem.open(this.path);
            this.lineReader = new
                LineReader(this.fsDataInputStream,
                    this.configuration);
            this.start = inputSplit.getOffset(index);
            this.end = this.start + inputSplit.getLength(index);
            this.key = new LongWritable(0);
            this.value = new Text("");

        }
      @Override
        public void initialize(InputSplit inputSplit,
            TaskAttemptContext taskAttemptContext) throws
                IOException, InterruptedException {
            //Overloaded in the constructor.
}
     @Override
        public boolean nextKeyValue() throws IOException,
            InterruptedException {
            int offset = 0;
            boolean isKeyValueAvailable = true;
            if(this.start < this.end){
                offset = this.lineReader.readLine(this.value);
                this.key.set(this.start);
                this.start += offset;
            }

            if(offset == 0){
                this.key.set(0);
                this.value.set("");
                isKeyValueAvailable = false;
            }

            return isKeyValueAvailable;

        }

        @Override
        public LongWritable getCurrentKey() throws IOException,
            InterruptedException {
```

```
            return key;
        }

        @Override
        public Text getCurrentValue() throws IOException,
            InterruptedException {
            return value;
        }

        @Override
        public float getProgress() throws IOException,
            InterruptedException {
  long splitStart = this.combineFileSplit.getOffset(fileIndex);
  f(this.start < this.end){
                return Math.min(1.0f, (this.start -  splitStart)/
                    (float) (this.end - splitStart));
            }

            return 0;
        }

        @Override
        public void close() throws IOException {
            if(lineReader != null){
                lineReader.close();
            }
        }
    }
```

The `Mapper` class and the driver program are given in the following snippet. The most important line in the driver is that which sets `InputFormat` as `job.setInputFormatClass(MasteringHadoop.MasteringHadoopCombineFileInputFormat.class)`. When the program is executed, the standard output obtained is also given after the snippet. The number of splits comes up as one. The size of the corpus in this case is 5 MB while the HDFS block size is 128 MB.

```
package MasteringHadoop;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
```

```
import java.io.IOException;

public class CombineFilesMasteringHadoop {
    public static class CombineFilesMapper extends
        Mapper<LongWritable, Text, LongWritable, Text>{

        @Override
        protected void map(LongWritable key, Text value, Context
            context) throws IOException, InterruptedException {
            context.write(key, value);
        }
    }
public static void main(String args[]) throws IOException,
    InterruptedException, ClassNotFoundException{
        GenericOptionsParser parser = new
            GenericOptionsParser(args);
        Configuration config = parser.getConfiguration();
        String[] remainingArgs = parser.getRemainingArgs();

        Job job = Job.getInstance(config, "MasteringHadoop-
            CombineDemo");
        job.setOutputKeyClass(LongWritable.class);
        job.setOutputValueClass(Text.class);
        job.setMapperClass(CombineFilesMapper.class);
       job.setNumReduceTasks(0);
job.setInputFormatClass(MasteringHadoop.MasteringHadoopCombineFile
    InputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
       FileInputFormat.addInputPath(job, new
           Path(remainingArgs[0]));
        TextOutputFormat.setOutputPath(job, new
            Path(remainingArgs[1]));
       job.waitForCompletion(true);
    }
}
```

The output is as shown as follows:

```
14/04/10 16:32:05 INFO input.FileInputFormat: Total input paths to
  process : 441
14/04/10 16:32:06 INFO mapreduce.JobSubmitter: number of splits:1
```

Filtering inputs

Filtering inputs to a job based on certain attributes is often required. Data-level filtering can be done within the Maps, but it is more efficient to filter at the file level before the Map task is spawned. Filtering enables only interesting files to be processed by Map tasks and can have a positive effect on the runtime of the Map by eliminating unnecessary file fetch. For example, files generated only within a certain time period might be required for analysis.

Let's use the 441-grant proposal file corpus subset to illustrate filtering. Let's process those files whose names match a particular regular expression and have a minimum file size. Both of these are specified as job parameters—`filter.name` and `filter.min.size`, respectively. Implementation entails extending the `Configured` class and implementing the `PathFilter` interface as shown in the following snippet. The `Configured` class is the base class for things that can be configured using `Configuration`. The `PathFilter` interface is the interface that contains an `accept()` method. The `accept()` method implementation takes in a `Path` parameter and returns `true` or `false` depending on whether the file has to be included in the input or not. The outline of the class is shown in the following snippet:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.PathFilter;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import java.io.IOException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public static class MasteringHadoopPathAndSizeFilter extends
    Configured implements PathFilter {
        private Configuration configuration;
        private Pattern filePattern;
        private long filterSize;
```

```
        private FileSystem fileSystem;

        @Override
        public boolean accept(Path path){
                //Your accept override implementation goes here
        }

        @Override
        public void setConf(Configuration conf){
                //Your setConf override implementation goes here
        }
    }
```

An important change is to override the `setConf()` method. This method is used to set the private `Configuration` variable and read off any properties from it. In the driver class, the job has to be informed about the presence of a filter using the following line:

```
FileInputFormat.setInputPathFilter(job,
  MasteringHadoopPathAndSizeFilter.class);
```

The implementation of the `setConf()` method is as follows:

```
    @Override
        public void setConf(Configuration conf){
            this.configuration = conf;

            if(this.configuration != null){
                String filterRegex =
                    this.configuration.get("filter.name");

                if(filterRegex != null){
                    this.filePattern =
                    Pattern.compile(filterRegex);
                }

                String filterSizeString =
                    this.configuration.get("filter.min.size");

                if(filterSizeString != null){
                    this.filterSize =
                        Long.parseLong(filterSizeString);
                }

                try{
```

```
                this.fileSystem =
                    FileSystem.get(this.configuration);
            }
            catch(IOException ioException){
                //Error handling
            }

        }
    }
```

In the following code, the `accept()` method returns `true` for all directories. The path of the current directory is one of the paths that will be provided to the `accept()` method. It uses the Java regular expression classes such as `Pattern` and `Matches` to determine whether any of the file paths match the expression and sets a Boolean variable appropriately. A second check is done to determine the file size and compare it with the file size filter. The `FileSystem` object exposes a `getFileStatus()` method that returns a `FileStatus` object, which can be examined for its file attributes via getters.

```
@Override
    public boolean accept(Path path){
      boolean isFileAcceptable = true;
      try{
            if(fileSystem.isDirectory(path)){
                  return true;
            }

            if(filePattern != null){
                Matcher m =
                    filePattern.matcher(path.toString());
                isFileAcceptable = m.matches();
            }

            if(filterSize > 0){
                long actualFileSize =
                    fileSystem.getFileStatus(path).getLen();
                if(actualFileSize > this.filterSize){
                    isFileAcceptable &= true;
                }
                else{
                    isFileAcceptable = false;
                }
            }

        }
```

```
            catch(IOException ioException){
                //Error handling goes here.

            }

            return isFileAcceptable;
        }
```

The following command line accepts files that have `a999645` in their names and have sizes greater than 2,500 bytes. If either parameter is omitted, no filter is applied for that attribute.

```
hadoop jar MasteringHadoop-1.0-SNAPSHOT-jar-with-dependencies.jar
-D filter.name=.*a999645.* -D filter.min.size=2500 grant-subset
grant-subset-filter
```

Three files pass the test and the output is shown as follows. The filtering happens before the splits are decided.

```
14/04/10 21:34:38 INFO input.FileInputFormat: Total input paths to
  process : 3
14/04/10 21:34:39 INFO mapreduce.JobSubmitter: number of splits:3
```

The Map task

The efficiency of the Map phase is decided by the specifications of the job inputs. We saw that having too many small files leads to proliferation of Map tasks because of a large number of splits. Another important statistic to note is the average runtime of a Map task. Too many or too few Map tasks are both detrimental for job performance. Striking a balance between the two is important, much of which depends on the nature of the application and data.



A rule of thumb is to have the runtime of a single Map task to be around a minute to three minutes, based on empirical evidence.

The dfs.blocksize attribute

The default block size of files in a cluster is overridden in the cluster configuration file, `hdfs-site.xml`, generally present in the `etc/hadoop` folder of the Hadoop installation. In some cases, a Map task might take only a few seconds to process a block. Giving a bigger block to the Map tasks in such cases is better. This can be done in the following ways:

- Increasing the `fileinputformat.split.minsize` parameter to be greater than the block size
- Increasing the block size of the input file stored in HDFS

The former leads to locality problems as `InputSplit` might have to import data from blocks residing in other nodes. The latter method preserves locality, but might require you to reload the file in HDFS. It can be done using the following command. A file `tiny.dat.txt` is being uploaded into HDFS with a block size of 512 MB. The default block size was 128 MB (in previous versions, it is 64 MB).

```
hadoop fs -D dfs.blocksize=536870912 -put tiny.dat.txt
tiny.dat.newblock.txt
```



The number of Map tasks should not exceed 60,000 or 70,000 for any application.

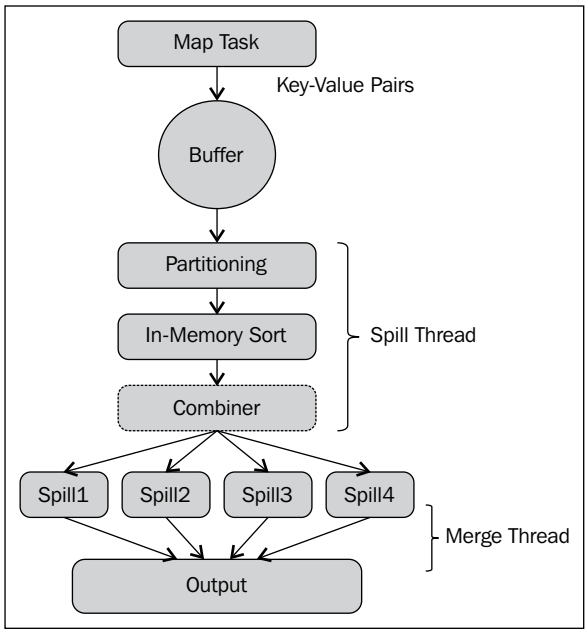
There could be situations where Map tasks are CPU bound, that is, I/O is an insignificant part of the Map task runtime. In such cases, it is better to utilize all available computing resources within the cluster. Decreasing the `fileinputformat.split.maxsize` property to be less than the HDFS block size can help increase cluster resource utilization.



Fewer Map tasks that exploit data locality are good for job performance. But in the face of failures, they might increase the job latency. A single Map task processes a significant chunk of the data and failure might hold up the entire job.

Sort and spill of intermediate outputs

The process of sending intermediate outputs from the Map tasks to the Reduce tasks involves complexity on both the Map side (as shown in the following diagram) and the Reduce side. Not only does the output of the Map tasks have to be partitioned based on the key to send it to the right Reduce task, but the keys within each partition have to be sorted as well. The partitioned data is then distributed to the appropriate reducers.



The Map task output workflow

The intermediate output records emitted by the Map task are not directly written on the disk. They are buffered in the local memory using a circular buffer before spilling them onto the disk. The size of this circular buffer is configured by the `mapreduce.task.io.sort.mb` property. The default value of this parameter is 100, that is, the circular buffer has a capacity of 100 MB. This property is overridden in the `mapred-default.xml` or `mapred-site.xml` file that is placed in the `etc/hadoop` directory of the Hadoop installation. All the properties discussed in this section go in the same config file. The buffered key-value records are serialized but not sorted.

Each key-value record is augmented with some accounting information. This accounting information has a constant value of 16 bytes per record regardless of the size of the actual key or value payload.

The soft threshold for the percentage of buffer that is allocated for the actual output record is given by the `mapreduce.task.io.sort.spill.percent` property. The default for this parameter is 0.8, that is, the output records from the buffer will be flushed to disk when the buffer becomes 80 percent full.



Before MAPREDUCE-64, the `io.record.sort.percent` property was a soft threshold for the percentage of the buffer that is allocated for accounting information. It had a default value of 0.05. Spills used to be triggered if the accounting information reached this threshold. This used to cause more spills and underutilization of the buffer, particularly for smaller records.

After this patch, the `io.record.sort.percent` property gets auto-tuned based on the record size instead of being set manually.

The spilling happens on a background thread after reaching the soft threshold for buffer occupancy. The Map task is not blocked to write onto the circular buffer when the spilling is going on. However, if the circular buffer reaches a hard limit, the Map task is blocked until the spill is complete. The spilling thread does a partition of the records based on the key, sorts the keys in memory within each partition, and writes them to a file. For every spill, there is a separate file that is written.



The method `map.sort.class` determines the sorting algorithm used for sorting keys. The default is QuickSort, implemented in `org.apache.hadoop.util.QuickSort`.

The partitioner class is determined by the `mapreduce.partitioner.class` property. The spill thread uses an instance of this class to determine which Reduce task partition the record has to be assigned to.

Once the Map task is complete, the spill files are merged, with keys sorted in each partition and written to a single output file. The `mapreduce.cluster.local.dir` parameter contains the directories where the output files are placed. The number of streams to merge simultaneously while writing the output is determined by the value of the `mapreduce.task.io.sort.factor` parameter. The default number is 10, that is, 10 open file handles will be present at a time during this step.

Each time a spill happens onto disk, the I/O required is three times a normal I/O operation. Once the spill file is written and the Map task has ended, it is read, merged to form a single output file, and rewritten to disk. It is best to spill only once at the very end of the Map task.



If the merge step at the end of a Map is taking very less time, don't bother optimizing the sort and spill step.

The sort and spill step can be made more efficient in the following ways:

- Increasing the size of the circular buffer by setting the `mapreduce.task.io.mb` property is one way to avoid or reduce the number of spills. When tweaking this parameter, it is good practice to monitor the Map task JVM heap size as well and increase it if necessary.
- Increasing the `mapreduce.task.io.sort.factor` property by a factor of 100 or so. This will make the merge process faster and reduce disk access.
- Writing efficient custom serializers for both keys and value types. The less the space taken up by serialized data, the more efficient the buffer usage.
- Writing **Combiners** to efficiently aggregate Map task outputs. This not only reduces the data transferred over the network to the Reduce task, but also helps in writing faster to the disk and lesser storage of the Map task spills and output files. The subsequent subsection gives more details about Combiners.
- Writing efficient key comparators and value grouping comparators can make a difference to the runtime of the sorting process.



MapReduce-4039

Sorting of keys within a single partition might not be necessary in many kinds of MapReduce applications. Termed as **Sort Avoidance**, it may lead to significant performance gains. Reducers need not wait for the all Map tasks to complete before starting off.

This enhancement is currently marked as open and could be coming in future releases.

Node-local Reducers or Combiners

Combiners are node-local reducers used to aggregate intermediate map output locally on individual mapper outputs. Combiners can help to reduce the amount of data that needs to be transferred across to reducers. The base class used to derive and implement a combiner is the same as that in the case of a reducer. However, depending on the application, the developer may choose to have different logic in the combiner and the reducer. The combiner is specified for a job using the call `setCombinerClass()`.



The JVM heap size for the Map task can be set using the `mapreduce.map.java.opts` parameter. The default value is `-Xmx1024m`.

If specified, the Combiner can be possibly called in two places:

- When the spills are being flushed onto disk by the spill thread
- When the spill files are being merged into a single output file to be consumed by the Reduce tasks

The former is called whenever a Combiner class is set for the job. The latter happens only if the number of spills exceeds the configuration value `mapreduce.map.combine.minspills`. The default value of this limit is three, that is, the Combiner is called during a merge only if there are three or more spills.



The intermediate files from a Map task matching a regular expression pattern can be preserved even after the job exits. This is done by specifying the pattern in the `mapreduce.task.files.preserve.filepattern` property.

Fetching intermediate outputs – Map-side

The Reducer needs to fetch Map task output files over the network to execute the Reduce task. The network being a bottleneck in a distributed system, the following Map-side optimizations can alleviate this:

- The intermediate outputs of the Map tasks can be compressed using a suitable Compression codec. The configuration property `mapreduce.map.output.compress` can be set to `true` to enable compression. The type of compression codec to be used can be specified by the property `mapreduce.map.output.compress.codec`. There are many choices available for compression, which are detailed in later chapters.
- The Reduce tasks fetch the output partitions from the Map task using the HTTP protocol. The `mapreduce.tasktracker.http.threads` property is used to configure the number threads that can service Reduce task fetch HTTP requests. Each fetch would need a single thread to service the request. Setting this property to a very low value would increase latency in servicing requests due of request queuing. The default value of this property is 40, indicating 40 threads.

The Reduce task

The Reduce task is an aggregation step. If the number of Reduce tasks is not specified, the default number is one. The risk of running one Reduce task would mean overloading that particular node. Having too many Reduce tasks would mean shuffle complexity and proliferation of output files that puts pressure on the NameNode. It is important to understand the data distribution and the partitioning function to decide the optimal number of Reduce tasks.



The ideal setting for each Reduce task to process is a range of 1 GB to 5 GB.

The number of Reduce tasks can be set using the `mapreduce.job.reduces` parameter. It can be programmatically set by calling the `setNumReduceTasks()` method on the `Job` object. There is a cap on the number of Reduce tasks that can be executed by a single node. It is given by the `mapreduce.tasktracker.reduce.maximum` property.



The heuristic to determine the right number of reducers is as follows:

0.95 * (nodes * `mapreduce.tasktracker.reduce.maximum`)

Alternatively, you can use the following:

1.75 * (nodes * `mapreduce.tasktracker.reduce.maximum`)

At 0.95, each of the reducers can launch immediately after the Map tasks are completed, and at 1.75, the faster nodes will finish their first Reduce task and move onto the second one. This is a better setting for load balancing.

Fetching intermediate outputs – Reduce-side

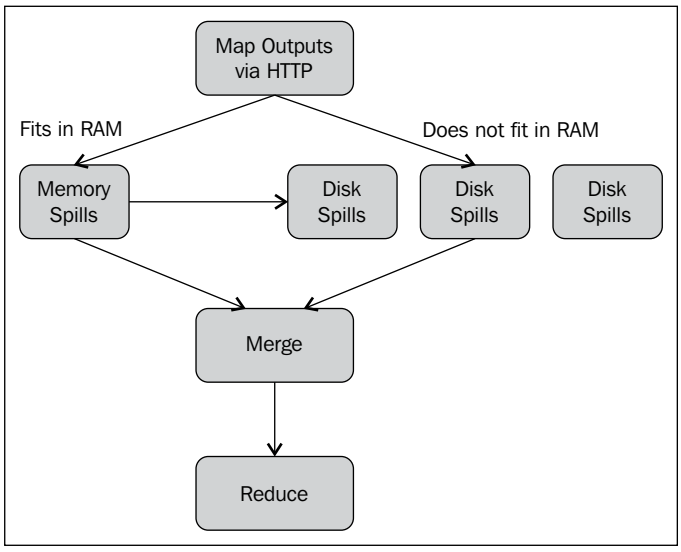
The Reduce task fetches relevant partitions from a Map task as and when they finish. This is called the **Copy phase**. The number of Map tasks from whom a Reduce task can fetch data in parallel is determined by the value of the `mapreduce.shuffle.reduce.parallelcopies` parameter. The lower this value, the more the queuing on the Reduce side. The Reduce task might have to wait for an available slot to fetch data from a Map task.

In situations where a Reduce task cannot reach the output data of the Map task due to network connectivity issues, it retries the fetch in an exponential backoff fashion. The retries continue until the time value specified by the `mapred.reduce.copy.backoff` property is reached. After that, the Reduce task is marked as failed.

Merge and spill of intermediate outputs

Similar to the Map task's sort and spill, the Reduce task also needs to merge and invoke the Reduce on files on multiple Map task outputs. The next diagram illustrates this process. Depending on the size of the Map task output, they are either copied to a memory buffer or to the disk. The `mapreduce.reduce.shuffle.input.buffer.percent` property configures the size of this buffer as a percentage of the heap size allocated to the task.

The value of the `mapreduce.reduce.shuffle.merge.percent` property determines the threshold beyond which this buffer has to be spilt to disk. The default value of this setting is 0.66. The `mapreduce.reduce.merge.inmem.threshold` property sets the threshold for the number of map outputs that can reside in memory before a disk spill happens. The default value of this property is 1000. When either threshold is reached, the map outputs are written onto the disk.



A background thread continuously merges the disk files. After all the outputs are received, the Reduce task moves into the **Merge** or **Sort** phase. Again, like the Map task merge, the number of file streams that are merged simultaneously is determined by the value of the `mapreduce.task.io.sort.factor` attribute. The tuning of these parameters can be done in a fashion similar to the Map-side spill and merge parameters. The key is to process as much as possible in the memory.

In later versions of Hadoop, two parameters, `mapreduce.reduce.merge.memtomem.enabled` and `mapreduce.reduce.merge.memtomem.threshold`, enable merging within the memory.

Any compression used for Map task outputs gets reversed in the memory during merging.

MapReduce output

The output is dependent on the number of Reduce tasks present in the job. Some guidelines to optimize outputs are as follows:

- Compress outputs to save on storage. Compression also helps in increasing HDFS write throughput.
- Avoid writing out-of-band side files as outputs in the Reduce task. If statistical data needs to be collected, the use of Counters is better. Collecting statistics in side files would require an additional step of aggregation.
- Depending on the consumer of the output files of a job, a splittable compression technique could be appropriate.
- Writing large HDFS files with larger block sizes can help subsequent consumers of the data reduce their Map tasks. This is particularly useful when we cascade MapReduce jobs. In such situations, the outputs of a job become the inputs to the next job. Writing large files with large block sizes eliminates the need for specialized processing of Map inputs in subsequent jobs.

Speculative execution of tasks

Stagglers are slow-running tasks that eventually complete successfully. A staggler Map task might not allow a Reduce task to start, thus delaying the completion of the job. Stagglers could be present because of hardware performance degradation or possible software misconfiguration.

Hadoop cannot automatically correct a staggler task but has the capability of identifying tasks that are running slower than normal. As a backup, it can spawn another equivalent task and use the results from the task that finishes first. The backup tasks can then be asked to terminate. This is termed **speculative execution**.

By default, Hadoop enables speculative execution. It can be turned off for Map tasks by setting `mapreduce.map.speculative` to `false` and for Reduce tasks by setting `mapreduce.reduce.speculative` to `false`.

The `mapreduce.job.speculative.speculativecap` is a property with values between 0 and 1, indicating the percentage of running tasks that can be speculatively executed. The default value of this property is 0.1. The `mapreduce.job.speculative.slowtaskthreshold` and `mapreduce.job.speculative.slownodethreshold` are two other configurable parameters whose values default to 1. They indicate how much slower the tasks should be executing than the average. They are measured in terms of standard deviation with respect to the average task progress rates.

MapReduce job counters

Counters are entities that can collect statistics at a job level. They can help in quality control, performance monitoring, and problem identification in Hadoop MapReduce jobs. Since they are global in nature, unlike logs, they need not be aggregated to be analyzed. Counters are grouped into logical groups using the `CounterGroup` class. There are sets of built-in counters for each MapReduce job.

The following example illustrates the creation of simple custom counters to categorize lines into lines having zero words, lines with less than or equal to five words, and lines with more than five words. The program when run on the grant proposal subset files gives the following output:

```
14/04/13 23:27:00 INFO mapreduce.Job: Counters: 23
    File System Counters
        FILE: Number of bytes read=446021466
        FILE: Number of bytes written=114627807
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=535015319
        HDFS: Number of bytes written=52267476
        HDFS: Number of read operations=391608
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=195363
    Map-Reduce Framework
        Map input records=27862
        Map output records=27862
        Input split bytes=56007
        Spilled Records=0
        Failed Shuffles=0
        Merged Map outputs=0
        GC time elapsed (ms)=66
        Total committed heap usage (bytes)=62037426176
    MasteringHadoop.MasteringHadoopCounters$WORDS_IN_LINE_COUNTER
        LESS_THAN_FIVE_WORDS=8449
```

```
        MORE_THAN_FIVE_WORDS=19413
        ZERO_WORDS=6766
    File Input Format Counters
        Bytes Read=1817707
    File Output Format Counters
        Bytes Written=189102
```

The first step in creating a counter is to define a Java enum with the names of the counters. The `enum` type name is the counter group as shown in the following snippet:

```
public static enum WORDS_IN_LINE_COUNTER{
        ZERO_WORDS,
        LESS_THAN_FIVE_WORDS,
        MORE_THAN_FIVE_WORDS
    };
```

When a condition is encountered to increment the counter, it can be retrieved by passing the name of the counter to the `getCounter()` call in the context object of the task. Counters support an `increment()` method call to globally increment the value of the counter.



An application should not use more than 15 to 20 custom counters.

The `getCounter()` method in the context has a couple of other overloads. It can be used to create a **dynamic counter** by specifying a group and counter name at runtime.

The `Mapper` class, as given in the following code snippet, illustrates incrementing the `WORDS_IN_LINE_COUNTER` group counters based on the number of words in each sentence of a grant proposal:

```
public static class MasteringHadoopCountersMap extends
    Mapper<LongWritable, Text, LongWritable, IntWritable> {
private IntWritable countOfWords = new IntWritable(0);
            @Override
            protected void map(LongWritable key, Text value,
                Context context) throws IOException,
                    InterruptedException {

                StringTokenizer tokenizer = new
                    StringTokenizer(value.toString());
                int words = tokenizer.countTokens();
if(words == 0)
    context.getCounter(WORDS_IN_LINE_COUNTER.ZERO_WORDS).increment(1);
                if(words > 0 && words <= 5)
```

```
context.getCounter(WORDS_IN_LINE_COUNTER.LESS_THAN_FIVE_WORDS)
    .increment(1);
                else
context.getCounter(WORDS_IN_LINE_COUNTER.MORE_THAN_FIVE_WORDS)
    .increment(1);
countOfWords.set(words);
                context.write(key, countOfWords);
            }
        }
```

Counters are global variables in a distributed setting and have to be used prudently. The higher the number of counters, the more are the overheads on the framework that keeps track of them. Counters should not be used to aggregate very fine-grained statistics of an application.

Handling data joins

Joins are commonplace in Big Data processing. They occur on the value of a join key and on a data type in the datasets that participate in a join. In this book, we will refrain from explaining the different join semantics such as inner joins, outer joins, and cross joins, and focus on inner join processing using MapReduce and the optimizations involved in it.

In MapReduce, joins can be done in either the Map task or the Reduce task. The former is called a **Map-side join** and the latter is called a **Reduce-side join**.

Reduce-side joins

Reduce-side joins are meant for more general purposes and do not impose too many conditions on the datasets that participate in the join. However, the shuffle step is very heavy on resources.

The basic idea involves tagging each record with a data source tag and extracting the join key in the Map tasks. The Reduce task receives all the records with the same join key and does the actual join. If one of the datasets participating in the join is very small, it can be distributed via a side channel such as the DistributedCache to every Reduce task.

For the Reduce-side joins to work, there are the following requirements:

- There needs to be a way of specifying the `InputFormat` and `Mapper` classes for the different datasets participating in the join. The `MultipleInputs` class is designed for this purpose. For a smaller file, the DistributedCache API can be used. A Map-side join, which will be explained later, shows how to use this side-file distribution channel.
- Secondary sorting capability needs to be there for optimal Reduce-side joins. The sorting of the join keys will happen, but it is important that the source is also sorted for each matching join key. By secondary sorting, one source occurs after the other, eliminating the need to hold all records for a particular key in the memory.

The following example illustrates Reduce-side joins. The dataset contains world cities and some information about the cities, the country code being one of them. It is available at `http://dev.maxmind.com/geoip/legacy/geolite/` in a CSV format. Countries have a two-letter ISO code. The `countrycodes.txt` file was taken from `http://www.spoonfork.org/isocodes.html`.

In this example and in subsequent examples of joins, the ISO code for the country is the join key. This key is used to get the country name and the total population of that country calculated by summing up the population of its individual cities. The join can be done by the following steps:

1. A custom `Writable` data type needs to be implemented to have the dataset tag information within the key. The following code shows the implementation of such a composite key:

```
package MasteringHadoop;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparable;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

public class CompositeJoinKeyWritable implements WritableComparabl
e<CompositeJoinKeyWritable> {

  private Text key = new Text();
```

```
private IntWritable source = new IntWritable();

public CompositeJoinKeyWritable(){

}

public CompositeJoinKeyWritable(String key, int source){

  this.key.set(key);
  this.source.set(source);

}

public IntWritable getSource(){
  return this.source;
}

public Text getKey(){
  return this.key;
}

public void setSource(int source){
  this.source.set(source);
}

public void setKey(String key){
  this.key.set(key);

}

@Override
public void write(DataOutput dataOutput) throws IOException {

  this.key.write(dataOutput);
  this.source.write(dataOutput);
}

@Override
public void readFields(DataInput dataInput) throws IOException {

  this.key.readFields(dataInput);
```

```
        this.source.readFields(dataInput);

      }


      @Override
      public int compareTo(CompositeJoinKeyWritable o) {

        int result = this.key.compareTo(o.key);

        if(result == 0){
          return this.source.compareTo(o.source);
        }


        return result;
      }

      @Override
      public boolean equals(Object obj){

        if(obj instanceof CompositeJoinKeyWritable){

          CompositeJoinKeyWritable joinKeyWritable =
    (CompositeJoinKeyWritable)obj;

          return (key.equals(joinKeyWritable.key) && source.
    equals(joinKeyWritable.source));
        }

        return false;

      }
    }
```

2. A custom `Partitioner` class needs to be implemented. `Partitioner` must partition the data based on the natural join key only; in this case, it is the ISO country code. This ensures that all the cities with the same country code are processed by the same Reduce task. The following code gives an implementation of a custom `Partitioner` class:

```
public static class CompositeJoinKeyPartitioner extends
    Partitioner<CompositeJoinKeyWritable, Text>{


        @Override

        public int getPartition(CompositeJoinKeyWritable
            key, Text value, int i) {

            return (key.getKey().hashCode() % i);

        }
    }
```

3. A custom grouping comparator needs to be written. Again, like the partitioner, the grouping has to be done on the natural key alone. The following code shows the grouping comparator for the composite key:

```
public static class CompositeJoinKeyComparator extends
    WritableComparator{

        protected CompositeJoinKeyComparator(){
             super(CompositeJoinKeyWritable.class, true);

        }

        @Override
        public int compare(Object a, Object b) {

            CompositeJoinKeyWritable compositeKey1 =
                (CompositeJoinKeyWritable) a;
            CompositeJoinKeyWritable compositeKey2 =
                (CompositeJoinKeyWritable) b;

            return compositeKey1.getKey()
               .compareTo(compositeKey2.getKey());

        }
    }
```

4. The `Mapper` classes have to be written for each kind of input datasets. In the following example, two `Mapper` classes are present: one for the city dataset and the other for the country dataset. The country dataset has a number less than the city dataset. This is done for efficiency. When a secondary sort is done on the dataset keys, the country dataset record appears before the city records at the Reducer:

```
public static class MasteringHadoopReduceSideJoinCountryMap
    extends Mapper<LongWritable, Text,
        CompositeJoinKeyWritable, Text>{

        private static short COUNTRY_CODE_INDEX = 0;
        private static short COUNTRY_NAME_INDEX = 1;

        private static CompositeJoinKeyWritable
            joinKeyWritable = new
                CompositeJoinKeyWritable("", 1);
        private static Text recordValue = new Text("");

        @Override
        protected void map(LongWritable key, Text value,
            Context context) throws IOException,
                InterruptedException {

            String[] tokens = value.toString().split(",", -
                1);

            if(tokens != null){
                joinKeyWritable.setKey(tokens[COUNTRY_CODE_
INDEX]);
                recordValue.set(tokens[COUNTRY_NAME_INDEX]);
                context.write(joinKeyWritable,
                    recordValue);
            }


        }
    }

    public static class
    MasteringHadoopReduceSideJoinCityMap extends
        Mapper<LongWritable, Text,
            CompositeJoinKeyWritable, Text>{

        private static short COUNTRY_CODE_INDEX = 0;


        private static CompositeJoinKeyWritable
```

```
            joinKeyWritable = new
                CompositeJoinKeyWritable("", 2);
        private static Text record = new Text("");

        @Override
        protected void map(LongWritable key, Text value,
            Context context) throws IOException,
                InterruptedException {

            String[] tokens = value.toString().split(",", -
                1);

            if(tokens != null){

joinKeyWritable.setKey(tokens[COUNTRY_CODE_INDEX]);
                record.set(value.toString());
                context.write(joinKeyWritable, record);
            }


        }
    }
```

5. The `Reducer` class takes advantage of secondary sorting to emit the joined records. The first value of the iterator is the country record. The name of the country is stored away and the population is calculated based on other records.

```
public static class MasteringHadoopReduceSideJoinReduce
    extends
            Reducer<CompositeJoinKeyWritable, Text, Text,
                LongWritable>{

        private static LongWritable populationValue = new
            LongWritable(0);
        private static Text countryValue = new Text("");
        private static short POPULATION_INDEX = 4;

        @Override
        protected void reduce(CompositeJoinKeyWritable key,
            Iterable<Text> values, Context context) throws
                IOException, InterruptedException {

            long populationTotal = 0;
            boolean firstRecord = true;
            String country = null;
```

```
            for(Text record : values){

                String[] tokens =
                    record.toString().split(",", -1);
                if(firstRecord){
                    firstRecord = false;
                    if(tokens.length > 1)
                        break;
                    else
                      country = tokens[0];
                }
                else{
                    String populationString =
                        tokens[POPULATION_INDEX];

                    if(populationString != null &&
                        populationString.isEmpty() ==
                            false){
                        populationTotal +=
                            Long.parseLong(populationString);
                    }

                }
            }

            if(country != null){
                populationValue.set(populationTotal);
                countryValue.set(country);
                context.write(countryValue,
                    populationValue);

            }


        }
    }
```

6. The driver program specifies all the custom data types that are required to do the Reduce-side join:

```
public static void main(String args[]) throws
IOException, InterruptedException, ClassNotFoundException{

        GenericOptionsParser parser = new
            GenericOptionsParser(args);
        Configuration config = parser.getConfiguration();
        String[] remainingArgs = parser.getRemainingArgs();

        Job job = Job.getInstance(config, "MasteringHadoop-
            ReduceSideJoin");

        job.setMapOutputKeyClass(CompositeJoinKeyWritable.class);
        job.setMapOutputValueClass(Text.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);

        job.setReducerClass(MasteringHadoopReduceSideJoinReduce
            .class);
        job.setPartitionerClass(CompositeJoinKeyPartitioner.
class);
        job.setGroupingComparatorClass(CompositeJoinKeyComparator
            .class);
        job.setNumReduceTasks(3);


        MultipleInputs.addInputPath(job, new
            Path(remainingArgs[0]), TextInputFormat.class,
                MasteringHadoopReduceSideJoinCountryMap.class);
        MultipleInputs.addInputPath(job, new
            Path(remainingArgs[1]), TextInputFormat.class,
                MasteringHadoopReduceSideJoinCityMap.class);

        job.setOutputFormatClass(TextOutputFormat.class);
        TextOutputFormat.setOutputPath(job, new
            Path(remainingArgs[2]));

        job.waitForCompletion(true);

    }
```

Map-side joins

Map-side joins, on the contrary, require either of two conditions satisfied in the datasets they join. These conditions are as follows:

- In addition to the presence of join keys, all inputs must be sorted using the join keys. The input datasets must have the same number of partitions. All records with the same key must reside in the same partition. Map-side joins are particularly attractive when operated on outputs of other MapReduce jobs. Such conditions are automatically satisfied in these cases. The `CompositeInputFormat` class can be used to run Map-side joins on such datasets. The configurations for inputs and join types can be specified using properties.
- If one of the datasets is small enough, side file distribution channels such as the DistributedCache can be used to do a Map-side join.

In the following example, the countries file is distributed across all nodes. During Map task setup, it is loaded into the memory onto a `TreeMap` data structure. The `setup()` method of the `Mapper` class is overridden to load the smaller data set in memory:

```
package MasteringHadoop;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.LineReader;

import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.TreeMap;

public class MasteringHadoopMapSideJoin {

    public static class MasteringHadoopMapSideJoinMap extends
        Mapper<LongWritable, Text, Text, LongWritable> {

        private static short COUNTRY_CODE_INDEX = 0;
```

```
private static short COUNTRY_NAME_INDEX = 1;
private static short POPULATION_INDEX = 4;


private TreeMap<String, String> countryCodesTreeMap = new
    TreeMap<String, String>();
private Text countryKey = new Text("");
private LongWritable populationValue = new
    LongWritable(0);


@Override
protected void setup(Context context) throws IOException,
    InterruptedException {

    URI[] localFiles = context.getCacheFiles();

    String path = null;
    for(URI uri : localFiles){
        path = uri.getPath();
        if(path.trim().equals("countrycodes.txt")){
             break;
        }

    }

    if(path != null){
       getCountryCodes(path, context);
    }

}
```

The `getCountryCodes()` private method, given as follows, is used to read the side file from the DistributedCache. Each line is processed and stored in the `TreeMap` instance. This method is a part of the `Mapper` class as well:

```
private void getCountryCodes(String path, Context context)
    throws IOException{

    Configuration configuration =
        context.getConfiguration();
    FileSystem fileSystem = FileSystem.get(configuration);
    FSDataInputStream in = fileSystem.open(new
        Path(path));
    Text line = new Text("");
```

```
            LineReader lineReader = new LineReader(in,
                configuration);

            int offset = 0;
            do{
                offset = lineReader.readLine(line);

                if(offset > 0){
                    String[] tokens = line.toString().split(",", -
                        1);
                    countryCodesTreeMap.put(tokens[COUNTRY_CODE_
INDEX],
                        tokens[COUNTRY_NAME_INDEX]);
                }

            }while(offset != 0);

        }
```

The map override method of the Mapper is where the join takes place. Each key is checked against the `TreeMap` data structure for a match. If a match exists, a joined record is emitted:

```
        @Override
        protected void map(LongWritable key, Text value, Context
            context) throws IOException, InterruptedException {

            String cityRecord = value.toString();
            String[] tokens = cityRecord.split(",", -1);

            String country = tokens[COUNTRY_CODE_INDEX];
            String populationString = tokens[POPULATION_INDEX];

            if(country != null && country.isEmpty() == false){

                if(populationString != null &&
                    populationString.isEmpty() == false){

                    long population =
                        Long.parseLong(populationString);
```

```
                    String countryName =
                        countryCodesTreeMap.get(country);

                    if(countryName == null) countryName = country;

                    countryKey.set(countryName);
                    populationValue.set(population);
                    context.write(countryKey, populationValue);

                }

            }
        }
    }
```

The Reduce task is a simple task that reduces on the join key and calculates the total population in a country. The code is given as follows:

```
    public static class MasteringHadoopMapSideJoinReduce extends
        Reducer<Text, LongWritable, Text, LongWritable>{

        private static LongWritable populationValue = new
            LongWritable(0);
        @Override
        protected void reduce(Text key, Iterable<LongWritable>
            values, Context context) throws IOException,
                InterruptedException {

             long populationTotal = 0;

             for(LongWritable population : values){
                populationTotal += population.get();
             }
            populationValue.set(populationTotal);
            context.write(key, populationValue);
        }
    }
```

Summary

In this chapter, you saw optimizations at different stages of the Hadoop MapReduce pipeline. With the join example, we saw a few other advanced features available for MapReduce jobs. Some key takeaways from this chapter are as follows:

- Too many Map tasks that are I/O bound should be avoided. Inputs dictate the number of Map tasks.
- Map tasks are primary contributors for job speedup due to parallelism.
- Combiners increase efficiency not only in data transfers between Map tasks and Reduce tasks, but also reduce disk I/O on the Map side.
- The default setting is a single Reduce task.
- Custom partitioners can be used for load balancing among Reducers.
- DistributedCache is useful for side file distribution of small files. Too many and too large files in the cache should be avoided.
- Custom counters should be used to track global job level statistics. But too many counters are bad.
- Compression should be used more often. Different compression techniques have different tradeoffs and the right technique is application-dependent.
- Hadoop has many tunable configuration knobs to optimize job execution.
- Premature optimizations should be avoided. Built-in counters are your friends.
- Higher-level abstractions such as Pig or Hive are recommended instead of bare metal Hadoop jobs.

In the next chapter, we will look at Pig, a framework to script MapReduce jobs on Hadoop. Pig provides higher-level relational operators that a user can employ to do data transformations, eliminating the need to write low-level MapReduce Java code.

3
Advanced Pig

Running Java MapReduce jobs on Hadoop provides the most flexibility with the least abstraction. However, abstractions are necessary to infer patterns, accomplish common data manipulation tasks, reduce complexity, and flatten the learning curve. Pig is a platform that provides a framework and high-level abstractions to build MapReduce programs for Hadoop. It has a scripting language called **Pig Latin**. Pig Latin can be compared to SQL in terms of operator capabilities.

Developed at Yahoo! around the year 2006, Pig was used as a framework to specify ad hoc MapReduce workflows. In the following year, it was moved to Apache Software Foundation. The latest release of Pig is 0.12.1.



The official release of Pig is currently incompatible with Hadoop 2.2.0. It expects libraries from Hadoop 1.2.1. Running any Pig script fails, with the following exception:

```
Unexpected System Error Occured: java.lang.
IncompatibleClassChangeError: Found interface org.
apache.hadoop.mapreduce.JobContext, but class was
expected.
```

Fixing this requires a recompile of the Pig binaries. Run the following command and replace the newly generated `pig.jar` and `pig-withouthadoop.jar` files:

```
ant clean jar-all -Dhadoopversion=23
```

In this chapter, we will look at the advanced features of Pig by:

- Looking at how Pig is different when compared to SQL
- Analyzing how Pig Latin scripts are translated to MapReduce programs
- Delving into the advanced relational operators that Pig supports; we will delve deep into these relational operators and look at their applications with examples
- Studying ways to extend Pig beyond its off-the-shelf capabilities using User-defined Functions or UDFs that can implement a variety of interfaces; we will examine some of these interfaces

Pig versus SQL

SQL is a very popular query and data processing language. Any high-level language for data processing deserves comparison with SQL. In this section, we will compare Pig Latin with SQL. The comparison is as follows:

- Pig Latin is primarily a procedural language. SQL, on the other hand, is declarative in nature. The data pipeline in SQL is not expressed as the data transformations happen. However, in Pig Latin, each step of the data transformation in the pipeline is specified in order. It is possible to mimic this behavior in SQL with the use of intermediate temporary tables, but creating, managing, and cleaning up these intermediate tables can be cumbersome and error-prone. Though Pig Latin scripts are specified procedurally, the statements are executed lazily, that is, they are not executed until the value is absolutely required.
- Developers writing data flows in a declarative language such as SQL overly depend on the query optimizer to choose the right implementation for the data transformation step. SQL engines do provide hints, but the flexibility of choosing or plugging in an implementation of choice is not present. Pig Latin naturally comes with this flexibility.
- SQL is ideal for linear data flows—transformations that yield a single result set. However, data flows are often **Directed Acyclic Graphs** (**DAGs**), where splitting data into streams, applying different transformation functions on each stream, and joining these streams are common operations. Implementing such DAGs in SQL require either repeating operations or materializing intermediate results. Pig handles data flow DAGs efficiently by reducing the number of disk reads and writes due to intermediate result materialization.

- SQL is not an **Extract-Transform-Load** (**ETL**) tool. It acts on data that is already present in a database. Pig facilitates UDFs, where users can specify Java code in the data flow. It allows for streaming, that is, insertion of an arbitrary executable in the data flow. Streaming can aid in the reuse of existing tools and code within the data-flow pipeline. These features of Pig make it a multiutility platform without the need for separate tools for ETL and processing.
- The procedural nature of Pig allows it to store data at any point during the data pipeline process. This can aid in introducing checkpoints manually and prevent the re-execution of the entire query from scratch on failures. This is particularly important when processing huge volumes of data, where data load and processing times are significantly large. SQL does not have this facility under the developers control, and it potentially requires the re-execution of significant portions of the query.

Different modes of execution

Pig has the following three modes of execution:

- **Interactive mode**: In this mode, a grunt shell is provided to the user. Users can type in Pig commands in an interactive session with Pig and the Hadoop cluster.
- **Batch mode**: In this mode, the user can write a series of Pig statements into a script file. The file can then be submitted for execution.
- **Embedded mode**: In this mode, any Java program can invoke Pig commands by importing the Pig libraries.

Apart from these modes of execution, Pig can either be executed locally, in the **local mode** using the local execution environment, or on a Hadoop cluster execution environment in the **mapreduce mode**. In the former mode, all commands are executed on a single system using the local filesystem. If the `-x` switch is not specified, Pig defaults to running in the **mapreduce** mode. Specifying the `-x` switch gives the user an option to run in the local or **mapreduce** mode and use the appropriate execution environment. The `HADOOP_CONF_DIR` environment variable is used to determine the Hadoop cluster for Pig to run MapReduce jobs.

The following code shows how to run Pig scripts in the `local` or `mapreduce` modes:

```
pig -x local …
pig  -x mapreduce …   OR pig …
```

Complex data types in Pig

Pig has primitive data types such as `int`, `long`, `float`, `double`, `chararray`, and `bytearray`. In addition, Pig also supports complex data types. Inputs and outputs to Pig's relational operators are specified using these complex data types. In some cases, the behavior of the operators depends on the complex data type used. These complex data types are as follows:

- `Map`: This data type should not be confused with the `map` function of MapReduce. The Map data type is an associative array data type that stores a `chararray` key and its associated value. There is no restriction on the data type of the value in a map. It can be a complex type too. If the type of the value cannot be determined, Pig defaults to the `bytearray` data type. The key and value association is syntactically done via the `#` symbol. The key values within a map have to be unique:

  ```
  [key#value, key1#value1…]
  ```

- `Tuple`: A Tuple data type is a collection of data values. They are of fixed length and are ordered. They can be compared to a record in a SQL table, without restrictions on the column types. Each data value is called a **field**. The ordering of values offers the capability to randomly access a value within a tuple:

  ```
  (value1, value2, value3…)
  ```

- `Bag`: A Bag data type is a container for tuples and other bags. They are unordered, that is, a tuple or a bag within a bag cannot be accessed randomly. There are no constraints on the structure of the tuples contained in a bag. Duplicate tuples or bags are allowed within a bag:

  ```
  {(tuple1), (tuple2)…}
  ```



The Bag data type in Pig can be spilt into disks, making it possible to hold a large number of tuples that collectively might not fit in memory. This is not the case with Map or Tuple data types.

Compiling Pig scripts

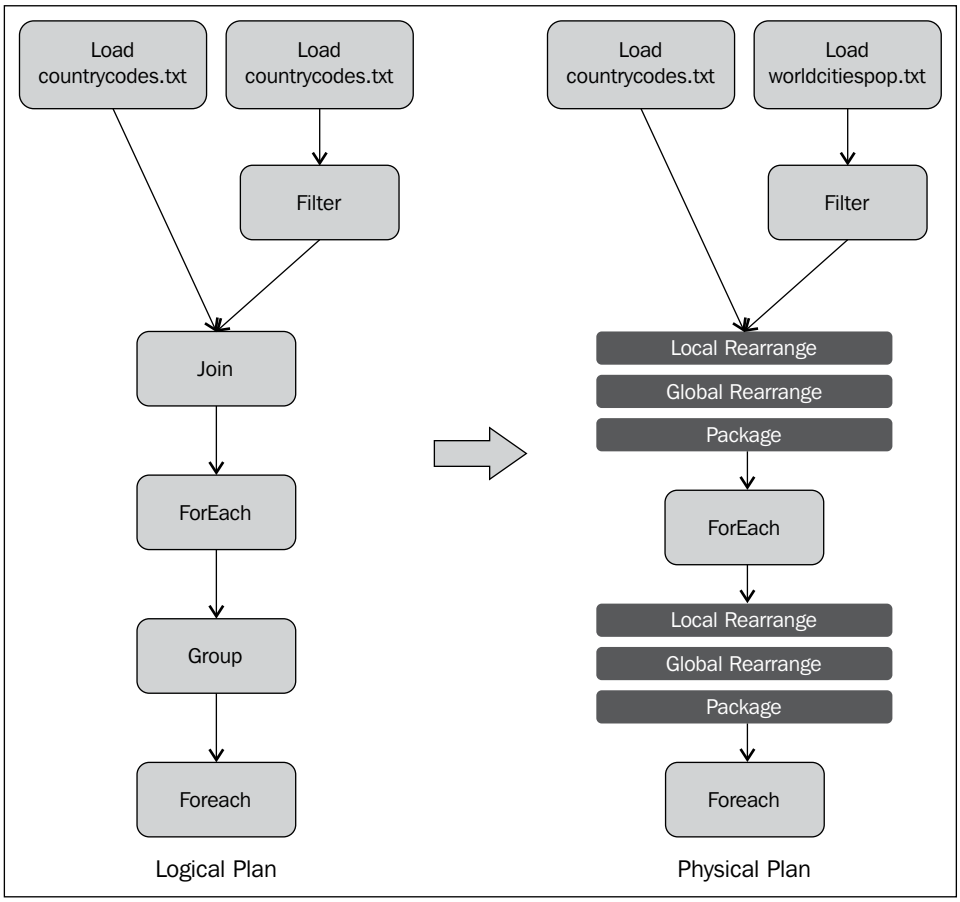
The Pig architecture is layered to facilitate pluggable execution engines. Hadoop's MapReduce is an execution platform that is plugged into Pig. There are three main phases when compiling and executing a Pig script: preparing the logical plan, transforming it into a physical plan, and finally, compiling the physical plan into a MapReduce plan that can be executed in the appropriate execution environment.

The logical plan

The Pig statements are first parsed for syntax errors. Validation of the input files and input data structures happens during parsing. Type checking in the presence of a schema is done during this phase. A logical plan, a DAG of operators as nodes, and data flow as edges are then prepared. The logical plan cannot be executed and is agnostic of the execution layer. Optimizations based on in-built rules happen at this stage. Some of these rules are discussed later in the chapter. The logical plan has a one-to-one correspondence with the operators available. In the following script, two text files are loaded as data inputs and stored in Pig variables, also called relations. The inputs then go through transformations such as filtering out null values and a join operation of the two data inputs. Finally, the transformed data is grouped on a join key and aggregated over each group:

```
cc = load 'countrycodes.txt' using PigStorage(',') as
    (ccode:chararray, cname:chararray);
ccity = load 'worldcitiespop.txt' using PigStorage(',') as
    (ccode:chararray, cityName:chararray, cityFullName:chararray,
        region:int, population:long, lat:double, long:double);
filteredCcity = filter ccity by population is not null;
joinCountry = join cc by ccode, ccity by ccode;
generateRecords = foreach joinCountry generate cc::cname,
    ccity::cityName, ccity::population;
groupByCountry = group generateRecords by cname;
populationByCountry = foreach groupByCountry generate group,
    SUM(generateRecords.population);
```

A logical plan block diagram for the script is as shown in the following figure:



Logical and physical plans for Pig scripts

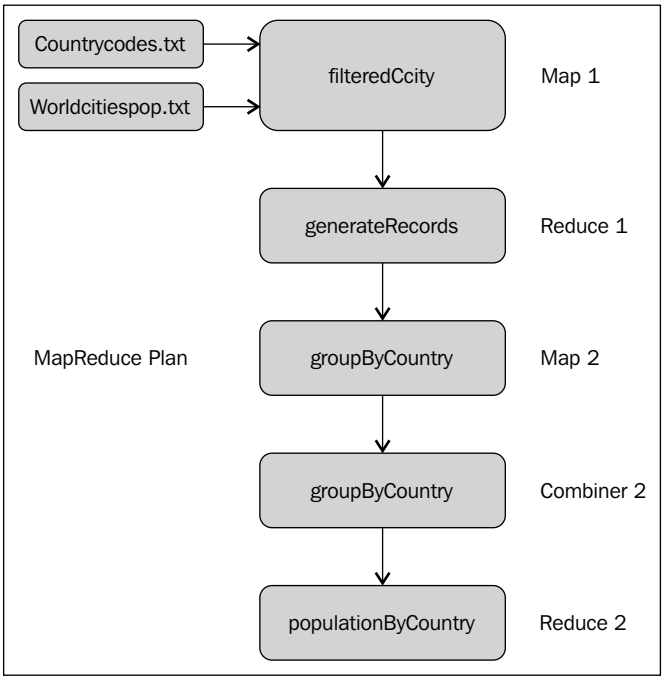
The physical plan

The physical plan is where the Pig compilation becomes aware of the execution platform. A translation of each operator into the physical form of execution happens during this stage. For the MapReduce framework, most operators have a one-to-one correspondence with the physical plan, except for a few of them. In addition to the logical operators, there are a few physical operators, which are the **Local Rearrange** (**LR**), **Global Rearrange** (**GR**), and **Package** (**P**) operators.

Logical operators such as `GROUP`, `COGROUP`, or `JOIN` are translated into a sequence of LR, GR, and P operators, as shown in the physical plan in the previous image. The LR operator corresponds to the shuffle preparation stage, where partitioning takes place based on the key. The GR operator corresponds to the actual shuffle between the Map and Reduce tasks. The P operator is the partitioning operator on the Reduce side.

The MapReduce plan

The final stage of Pig compilation is to compile the physical plan to actual MapReduce jobs. A Reduce task is required wherever an LR, a GR, and a P sequence is present in the physical plan. The compiler also looks for opportunities to put in Combiners wherever possible. The MapReduce plan for the physical plan in the previous image has two MapReduce jobs, one corresponding to the `JOIN` operator and the other to the `GROUP` operator in the logical plan. The following image shows the MapReduce plan for the query. The MapReduce task corresponding to the `GROUP` operator has a Combiner as well. It must be noted that the `GROUP` operation occurs in the Map task. This is possible because the output of **Reduce1** in the following image is sorted on the key:



Development and debugging aids

There are three important commands that can help develop, debug, and optimize Pig scripts.

The DESCRIBE command

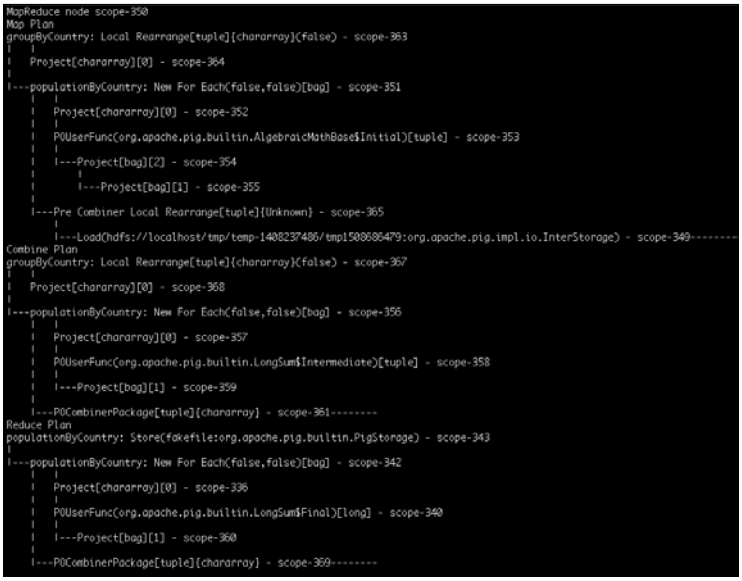
The `DESCRIBE` command gives the schema of a relation. This command is useful when you are a Pig Latin beginner and want to understand how operators transform the data. The output corresponding to the `groupByCountry` relation in the previous script code to find the population of the country is given as follows:

```
groupByCountry: {group: chararray,generateRecords: {(cc::cname:
  chararray,ccity::cityName: chararray,ccity::population: long)}}
```

The `DESCRIBE` output has the Pig syntax. In the preceding example, `groupByCountry` is a Bag data type that contains a group element and another bag, `generateRecords`.

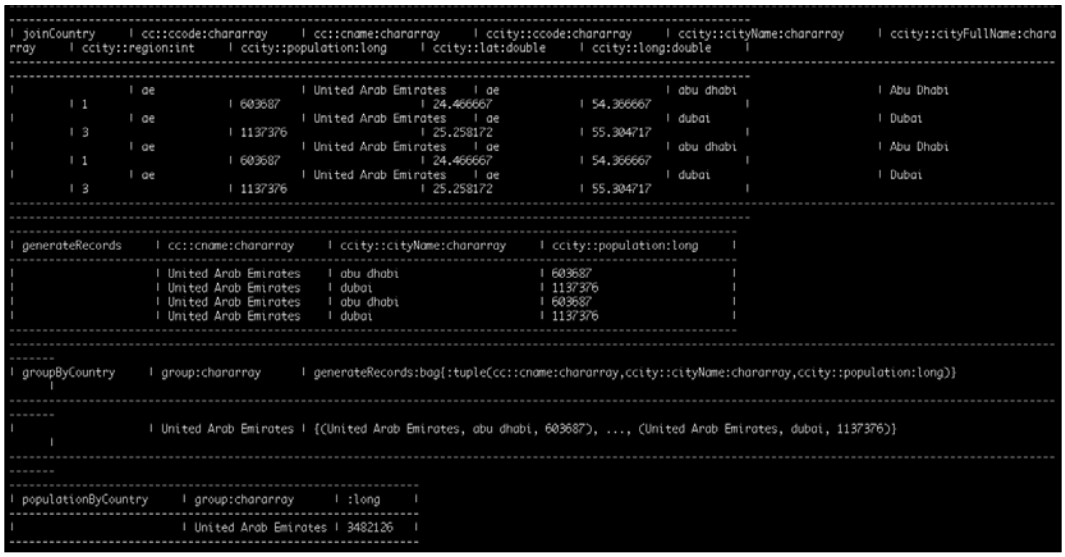
The EXPLAIN command

`EXPLAIN`, on a relation, shows how the Pig script will be executed. It is useful when trying to optimize Pig scripts or debug errors. It shows the logical, physical, and MapReduce plans of the relation. The following screenshot shows the MapReduce plan for the second MapReduce job (corresponding to the `GROUP` operator) when the `EXPLAIN` command was executed on `populationByCountry`. You can use the `EXPLAIN` command to study the optimizations that went into the plans.



The ILLUSTRATE command

`ILLUSTRATE` is perhaps the most important development aid. `ILLUSTRATE`, on a relation, samples the data and applies the query on it. This can save a lot of time when debugging. The sample is significantly smaller than the data. This makes the code-test-debug cycle very fast. In many situations, the `JOIN` or `FILTER` operators might not yield any output on a sample of data. In such cases, `ILLUSTRATE` manufactures records that pass through these operators and inserts them into the sample dataset. The partial result of executing `ILLUSTRATE` on the `populationByCountry` relation is given in the following screenshot:



The advanced Pig operators

In this section, we will examine some of the advanced features and hints available in Pig operators.

The advanced FOREACH operator

The `FOREACH` operator is primarily used to transform every record of the input relation into a transformed record. A list of expressions is used to make this transformation. There are situations where the `FOREACH` operator can increase the number of output records. They are discussed in the following sections.

The FLATTEN operator

The `FLATTEN` keyword is an operator, though it looks like a UDF in syntax. It is used to *un-nest* nested tuples and bags. However, the semantics of the elimination of nesting is different when it is used on tuples when compared to bags.

`FLATTEN` on a nested tuple yields a single tuple, as shown in the following snippet. All the nested tuples are elevated to the topmost level.

Consider data of the following nature:

```
(1, (2, 3, 4))
X = FOREACH A GENERATE $0, FLATTEN($1);
```

This will yield `(1,2,3,4)` as the resulting tuple.

For bags, the situation becomes more complicated. When we un-nest a bag, we create new tuples. If we have a relation that is made up of tuples of the `({(b,c),(d,e)})` form, and we apply `GENERATE FLATTEN($0)`, we end up with the two tuples `(b,c)` and `(d,e)`. In other words, `FLATTEN` does a cross-product, producing a row for every element in the bag.

Let's consider `worldcitiespop.txt` as an example:

```
cCity = load 'worldcitiespop.txt' using PigStorage(',') as
    (ccode:chararray, cityName:chararray, cityFullName:chararray,
        region:int, population:long, lat:double, long:double);
groupCcityByCcode = group cCity by ccode;
```

The `groupCcityByCcode` variable yields tuples with a nested bag, as shown in the following snippet. The number of such tuples is equal to the number of groups, in this case, the number of countries:

```
(ae, {(ae,ae,sharjah,Sharjah,6,543942,25.35731,55.403304),
(ae,ae,dubai,Dubai,3,1137376,25.258172,55.304717)})
```

The `FLATTEN` operator can be used on these tuples to ungroup them, yielding a record per city for this particular country, using the following code:

```
unGroupCcityByCcode = foreach groupCcityByCcode generate group,
    FLATTEN(cCity);
```

The result is a cross-product of the country code with each element of the nested bag, as follows:

```
(ae,ae,sharjah,Sharjah,6,543942,25.35731,55.403304)
(ae,ae,dubai,Dubai,3,1137376,25.258172,55.304717)
```



FLATTEN called on nested empty bags and empty tuples will yield no output records. This is because, mathematically, a cross-product between a nonempty and an empty set is an empty set. If this is not desired, it is good practice to replace empty bags and tuples with a constant bag.

The nested FOREACH operator

Relational operators can be applied within each record of a FOREACH operator. This is called a nested FOREACH or an inner FOREACH operator. Let's examine it with an example on the worldcitiespop.txt file. We want to find the details of the most populous city in each country. There can be many ways of solving this, but we will use the nested FOREACH operator in this case. The code is as follows:

```
ccity = load 'worldcitiespop.txt' using PigStorage(',') as
    (ccode:chararray, cityName:chararray, cityFullName:chararray,
        region:int, population:long, lat:double, long:double);
groupCcityByCcode = group cCity by ccode;
cityWithHighestPopulation = foreach groupCcityByCcode {
    citiesWithPopulation = filter cCity by (population is
        not null AND population > 0);
    orderCitiesWithPopulation = order citiesWithPopulation by
        population desc;
    topPopulousCity = limit orderCitiesWithPopulation 1;
    generate flatten(topPopulousCity); };
```

The steps performed are as follows:

1. The first step is to load worldcitiespop.txt based on a schema.
2. Then, group the data by country code. The nested FOREACH operator is the next statement. Flower braces ({) syntactically signify a nested FOREACH operator.
3. In the nested section, a variety of relational operators are applied to the group bag:
    1. A FILTER operator is applied on all the cities of the country to eliminate missing population values. A more efficient way will be to do the filtering before the FOREACH operator (refer to the *Best Practices* section). In this example, we will do the filtering in the nested FOREACH operator for illustration.
    2. The filtered city bag is sorted in descending order on the population of the city using the ORDER operators. Now, the first record is the most populous city.

3. We use the `LIMIT` operator to select the first record.
4. The last statement of a nested `FOREACH` operator is always the `GENERATE` method. Here, we generate the record for the most populous city. The `FLATTEN` operator is used to remove the bag nesting.

The tail of the final output is shown as follows:

```
(uz,tashkent,Tashkent,13,1978078,41.3166667,69.25)
(vc,kingstown,Kingstown,4,17995,13.1333333,-61.2166667)
(ve,maracaibo,Maracaibo,23,1948269,10.6316667,-71.6405556)
(vg,road town,Road Town,0,8449,18.4166667,-64.6166667)
(vn,ho chi minh city,Ho Chi Minh City,20,3467426,10.75,106.666667)
(vu,vila,Vila,8,35903,-17.7333333,168.3166667)
(wf,alele,Alele,0,901,-13.2333333,-176.15)
(ws,apia,Apia,0,40407,-13.8333333,-171.7333333)
(ye,aden,Aden,2,550744,12.7794444,45.0366667)
(yt,mamoudzou,Mamoudzou,0,54837,-12.7794444,45.2272222)
(za,cape town,Cape Town,11,3433504,-33.925839,18.423218)
(zm,lusaka,Lusaka,9,1267458,-15.4166667,28.2833333)
(zw,harare,Harare,4,2213701,-17.8177778,31.0447222)
```



Currently, `LIMIT`, `ORDER`, `DISTINCT`, `CROSS`, `FOREACH`, and `FILTER` are the relational operators supported within a nested `FOREACH` operator in Pig.

The COGROUP operator

This operator is similar to a `GROUP` operation. Instead of collecting records of one input based on a key, it collects records of *n* inputs based on the key. The `GROUP` operator works on a single input relation, but a `COGROUP` operator can work on many input relations. `COGROUP` can be thought of as the first phase of a join—a `COGROUP` operator followed by a `FOREACH` operator that flattens the bags is an inner join. The following code shows a join between the `worldcitiespop.txt` and `countrycodes.txt` files using `COGROUP`:

```
cc = load 'countrycodes.txt' using PigStorage(',') as
    (ccode:chararray, cname:chararray);
ccity = load 'worldcitiespop.txt' using PigStorage(',') as
    (ccode:chararray, cityName:chararray, cityFullName:chararray,
        region:int, population:long, lat:double, long:double);
groupedCity = cogroup cc by ccode, ccity by ccode;
flattendGroupedCity = foreach groupedCity generate flatten(cc),
    flatten (ccity);
filteredGroup = filter flattendGroupedCity by cc::ccode ==
    ccity::ccode;
```



The maximum number of relations a `COGROUP` operator can group together is 127.

Operations such as semi-joins can be performed using the `COGROUP` operator.



Semi-joins between two relations are the records in the first relation that have one or more matches on the join key in the second relation.

The UNION operator

The `UNION` operator is used to concatenate two or more datasets. Unlike SQL, the `UNION` operator in Pig does not impose any restrictions on the schema of the two datasets. If they are the same, the result will have the same schema. If one schema can be forced into the other schema by means of casts, the result will have this schema. Otherwise, the result will have no schema.

The `UNION` operator does not preserve the ordering of the tuples, and it does not eliminate any duplicate tuples. The `UNION` operator has an `ONSCHEMA` qualifier that is used to give a schema to the result. This schema is a set union of all the named fields between the datasets. The `ONSCHEMA` qualifier requires all the input relations to have a schema.

In our `countrycodes.txt` and `worldcitiespop.txt` files, the schemas do not match, and the result is without any schema. However, when we use the `ONSCHEMA` keyword along with the `UNION` operator, we see a schema. This schema is the union of the schemas of both relations. The following code illustrates this:

```
cc = load 'countrycodes.txt' using PigStorage(',') as
    (ccode:chararray, cname:chararray);
ccity = load 'worldcitiespop.txt' using PigStorage(',') as
    (ccode:chararray, cityName:chararray, cityFullName:chararray,
        region:int, population:long, lat:double, long:double);

unionCountryCity = union cc, ccity;
unionOnSchemaCountryCity = union onschema cc, ccity;
describe unionCountryCity;
describe unionOnSchemaCountryCity;
```



The schema comparison includes the names of the fields. The result of applying a `UNION` operator on datasets whose fields have different names will not have a schema. The workaround in such cases will be to introduce a `FOREACH` operator that renames fields to common names before the `UNION` statements.

The result of the `Describe` statements is as follows:

```
Schema for unionCountryCity unknown.
unionOnSchemaCountryCity: {ccode: chararray,cname:
    chararray,cityName: chararray,cityFullName: chararray,region:
        int,population: long,lat: double,long: double}
```

The CROSS operator

The `CROSS` operator performs the cross-set operations on two relations. The `CROSS` operator in Pig is implemented in parallel by constructing an artificial join key and then replicating records. This makes the `CROSS` operator very expensive, particularly in the shuffle and sort phases, as the records are replicated in each of the relations participating in the join for each artificial join key that was created.

However, there are situations where a `CROSS` operator is necessary. One such situation is a theta-join. Inner joins are based on equality of keys, that is, the equals-to operation is used to join the records across relations. However, there can be a need for inequality to determine record joins. This can be done by a `CROSS` operator followed by a `FILTER` operation on the join keys. The following hypothetical example illustrates using the `CROSS` operator for a theta-join. The join happens only if the value of `a1` is less than the value of `b1`:

```
A = LOAD 'inputA.txt' AS (a0:chararray, a1:int);
B = LOAD 'inputB.txt' AS (b0:chararray, b1:int);
ACrossB = CROSS A, B;
thetaJoin = FILTER ACrossB BY a1 < b1;
```

Fuzzy joins are another variant of the join operation that can be done using the `CROSS` operator. An example is to self-join `worldcitiespop.txt` on cities within the same region code.

Specialized joins in Pig

Pig supports join optimizations that can be used out of the box depending on the dataset and nature of the join. These join optimizations increase the performance of Pig scripts and are highly recommended.

The Replicated join

In *Chapter 2*, *Advanced MapReduce*, we implemented Map-side joins and Reduce-side joins as MapReduce jobs. When an input dataset for a join can be made to fit in-memory, the join can take place Map-side by replicating the smallest dataset across all the Map tasks. This is called a **Fragment-Replicate** join in Pig. It is Pig's implementation of the Map-side join. Some of the key points to remember in a Fragment-Replicate join are as follows:

- If the smaller file cannot be fit into memory, Pig throws an error and fails to execute. Pig also throws an error if the smaller file is greater than the value *I* for the `pig.join.replicated.max.bytes` property.
- The Fragment-Replicate join can be used to join more than two datasets. However, all datasets, except the first, will be loaded into memory.
- This replication of the input files across the different Map tasks happens via the DistributedCache, very similar to the implementation we discussed in *Chapter 2*, *Advanced MapReduce*.
- The Fragment-Replicate join can only be used to do an inner join and a left-outer join. Right-outer and full-outer joins are not possible. This is because the left relation is fragmented and the right relation is replicated entirely. When the join processor gets a record from the right relation, it has only a local view of the left relation and does not know if a matching key exists in some other split of the left relation.

Using the `replicated` keyword specifies a Fragment-Replicate join. In the following example, `countrycodes.txt` and `worldcitiespop.txt` are joined using this join. It must be noted that `countrycodes.txt` is the smaller relation and comes later in the join specification:

```
cc = load 'countrycodes.txt' using PigStorage(',') as
    (ccode:chararray, cname:chararray);
ccity = load 'worldcitiespop.txt' using PigStorage(',') as
    (ccode:chararray, cityName:chararray, cityFullName:chararray,
        region:int, population:long, lat:double, long:double);
joinCountryCity = join ccity by ccode, cc by ccode using
    'replicated';
```

Skewed joins

The presence of skews in data can hurt join performances by overloading a single Reduce task. Skews are statistical quirks, where a single or small number of keys have a significantly large number of records. Pig helps alleviate such a situation by providing **Skewed joins**. The idea here is to sample one of the input relations to a join and plot a histogram of the records for each key.

The histogram is then analyzed, and keys that have a large number of records are split. Each split is sent to a different Reduce task. In this way, load balancing of the records on the Reduce end is achieved. However, this also requires replicating the other input relation so that each of these load-balanced Reduce tasks has all the relevant records for a successful join.

Some of the key points to remember when executing a Skewed join are as follows:

- Skewed joins work only with two datasets. If there are more than two datasets that need to be joined, the responsibility is on the developer to break it down into multiple two-way joins.
- Sampling and constructing a histogram adds a performance overhead when using this join type. The average overhead observed is around 5 percent.
- The second relation of the join is the dataset that will be sampled.
- Pig is influenced by the value of the `pig.skewedjoin.reduce.memusage` Java parameter to decide how many additional Reduce tasks need to spawned to process a skewed key. The default value of this attribute is 0.5, that is, 50 percent of the JVM heap is available for the Reduce task to perform the join.

The `skewed` keyword is used to indicate a Skewed join. The following example shows its usage:

```
cc = load 'countrycodes.txt' using PigStorage(',') as
    (ccode:chararray, cname:chararray);
ccity = load 'worldcitiespop.txt' using PigStorage(',') as
    (ccode:chararray, cityName:chararray, cityFullName:chararray,
        region:int, population:long, lat:double, long:double);
joinCountryCity = join cc by ccode, ccity by ccode using 'skewed';
```

The Merge join

Again, as we saw in *Chapter 2*, *Advanced MapReduce*, it is possible to do a Map-side join if both the inputs are sorted on the join key. Pig has an implementation of this kind of join, which is known as a **Sort join** or **Merge join**.

The join algorithm works by making the second relation a side file and the first relation the input to Map tasks. The side file is sampled for the keys, and an index is built using a MapReduce job. The index is a mapping between the key and offset, where the key records the start in the file. Once this index is built, another MapReduce job is started with the first relation as the input file. Each record is read and looked up in the index for the corresponding offset in the second file. The records are read from the second relation, and the join is done.

The following example shows a Merge join using `countrycodes.txt` and `worldcitiespop.txt`. Again, note that `countrycodes.txt` is the file for which the index will be built. The `merge` keyword is used to indicate a Merge join:

```
cc = load 'countrycodes.txt' using PigStorage(',') as
    (ccode:chararray, cname:chararray);
ccity = load 'worldcitiespop.txt' using PigStorage(',') as
    (ccode:chararray, cityName:chararray, cityFullName:chararray,
        region:int, population:long, lat:double, long:double);
joinCountryCity = join cc by ccode, ccity by ccode using 'merge';
```

A variant of a Merge join is a **Merge-sparse join**. This is used when one of the relations is very sparse, that is, only a small number of records are matched during a join. This join type is still under experimentation. Currently, only inner joins are supported using the Merge-sparse join algorithm.

User-defined functions

User-defined functions or **UDFs**, are functions that can be implemented by the developer to extend the functionality of Pig and add custom processing. These functions can be called in almost all Pig operators. UDFs are written in Java. From Pig 0.8 onwards, Python UDFs are supported. In the latest version of Pig, in addition to Python and Java, UDFs can be written in Jython, JavaScript, Ruby, and Groovy.

Other than Java, the rest of the language bindings do not support all interfaces of Pig. For example, the load and store interfaces are not supported by the other language bindings. In this book, we will use Java to build and illustrate the power of UDFs.

There is a repository of Java UDFs called **piggy bank**. This is a public repository where you can take advantage of UDFs written by others and contribute your own UDFs to the community.

Before using a UDF in Pig, it is necessary to register the JAR file in the Pig script. The registration is done using the `REGISTER` command. In addition to an instance of the UDF class per Map or Reduce task, Pig creates an instance where the logical and physical plans for the script are created. This is done mainly for validation.



Each Map and Reduce task gets its own copy of UDF. States cannot be shared across Map and Reduce tasks; however, it can be shared within the same Map or Reduce task.

Pig UDFs can be broadly classified into the following types:

- Evaluation functions
- Load functions
- Store functions

Let's examine each one of them in detail.

The evaluation functions

As the name suggests, these are functions used for evaluation. The following example shows a custom upper UDF and its usage within a Pig script. All evaluation functions are extended from the `org.apache.pig.EvalFunc` base class. The most important method to override is the `exec` method. The `EvalFunc` class takes a generic type that signifies the return type of UDFs. The input to the `exec` method is a `Tuple` type. This `Tuple` has to be unwrapped using the `get()` method, and the resulting data item has to be processed by the `exec` method. The simplest of UDFs only has to override the `exec` method:

```
package MasteringHadoop;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import java.io.IOException;

public class UPPER extends EvalFunc<String>{


    @Override
    public String exec(Tuple objects) throws IOException {

       if(objects == null || objects.size() == 0){
           return null;
       }
       try{
```

```
            String inputString = (String) objects.get(0);
            return inputString.toUpperCase();
        }
        catch(Exception ex){

            throw new IOException("Error processing input ", ex);
        }

    }
}
register MasteringHadoop-1.0-SNAPSHOT-jar-with-dependencies.jar;
cc = load 'countrycodes.txt' using PigStorage(',') as
    (ccode:chararray, cname:chararray);
ccCapitalized = foreach cc generate
    MasteringHadoop.UPPER(cc.cname);
```

The aggregate functions

These UDFs are evaluation functions that are applied on groups. The in-built functions such as `SUM` and `COUNT` are functions of this kind. Aggregate UDFs take in a bag of values and return a scalar.



The entire record can be passed into UDFs by using `*`. When the entire record is passed, it is wrapped within another tuple. For example, to get the second element of a record, `input.get(0).get(1)` has to be executed. The first `get()` call unwraps the record from the tuple.

The Algebraic interface

Aggregate functions that implement the `Algebraic` interface can be used for local aggregation via Combiners. In *Chapter 2*, *Advanced MapReduce*, we studied how Combiners can help reduce data flow from Map tasks to Reduce tasks and also speed up the query by reducing the amount of IO.

An **algebraic** function is any function that can be divided into three functions: the initial function, the intermediate function, and the final function. If these three functions are applied in a cascading fashion, it is marked as an algebraic function. In other words, the data is divided into fragments, and the initial function is applied on it, followed by the intermediate function on its results, and finally, the final function on the results of the intermediate function. The `COUNT` function is an example of an algebraic function, where the initial function is count; the intermediate and final functions are both sums of the results.

A **distributive** function is a special case of an algebraic function where all three subfunctions do the same computation. `SUM` is an example of a distributive function.

Pig provides the `org.apache.pig.Algebraic` interface, which can be implemented to make the UDF algebraic. The following example shows a `COUNT` aggregate UDF implementing an algebraic interface.

Using an algebraic function translates to using Combiners on the Map task to execute the `Initial` and `Intermediate` static class `exec` functions. The Reduce task will execute the `Final` class' `exec` function:

```
package MasteringHadoop;

import org.apache.pig.Algebraic;
import org.apache.pig.EvalFunc;
import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.Tuple;

import java.io.IOException;
import java.util.Iterator;

public class COUNT extends EvalFunc<Long> implements Algebraic {

    protected static Long count(Tuple input) throws
        ExecException{
        DataBag dataBag = (DataBag) input.get(0);
        return dataBag.size();
    }

    protected static Long sum(Tuple input) throws ExecException{

        long returnSum = 0;
        DataBag dataBag = (DataBag) input.get(0);
        for(Iterator<Tuple> it = dataBag.iterator();
            it.hasNext();){
            Tuple tuple = it.next();
            returnSum += (long)tuple.get(0);
        }
        return returnSum;

    }

    static class Initial extends EvalFunc<Long>{

        @Override
```

```
        public Long exec(Tuple objects) throws IOException {
            return count(objects);
        }
    }

    static class Intermediate extends EvalFunc<Long>{

        @Override
        public Long exec(Tuple objects) throws IOException {
            return sum(objects);
        }
    }

    static class Final extends EvalFunc<Long>{

        @Override
        public Long exec(Tuple objects) throws IOException {
            return sum(objects);
        }
    }


    @Override
    public Long exec(Tuple objects) throws IOException {
        return count(objects);
    }


    @Override
    public String getInitial() {
        return Initial.class.getName();
    }

    @Override
    public String getIntermed() {
        return Intermediate.class.getName();
    }

    @Override
    public String getFinal() {
        return Final.class.getName();
    }
}
```

The Accumulator interface

In many cases, when a `GROUP` or `COGROUP` operator is used, all the bags of tuples for a particular key might not fit in memory. Also, the UDF might not need to see all the tuples at one go. Pig allows UDFs to implement the `Accumulator` interface to handle these situations. Instead of passing the entire record set at once, Pig incrementally passes subsets of the records for a given key through this interface.

Though the `Algebraic` interface alleviates the memory problem by aggregating early, there are many functions that are not algebraic. These functions might still be able to aggregate by accumulation and might not need to see the entire dataset.

Let's implement the UDF `LongMax`, which finds the biggest value in a bag using the `Accumulator` interface. As shown in the following snippet, the three methods that need to be implemented are `accumulate`, `getValue`, and `cleanup`. The `accumulate` method is called whenever an intermediate set of records is passed to UDFs. The `cleanup` method is called after each key is processed:

```
package MasteringHadoop;

import org.apache.pig.Accumulator;
import org.apache.pig.EvalFunc;
import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.Tuple;

import java.io.IOException;
import java.util.Iterator;

public class LONGMAX extends EvalFunc<Long> implements
    Accumulator<Long> {

    private Long intermediateMax = null;

    @Override
    public Long exec(Tuple objects) throws IOException {
        return max(objects);
    }

@Override
    public void accumulate(Tuple objects) throws IOException {
        Long newIntermediateMax = max(objects);

        if(newIntermediateMax == null){
            return;
```

```
            }

            if(intermediateMax == null){
                intermediateMax = Long.MIN_VALUE;
            }

            intermediateMax = Math.max(intermediateMax,
                newIntermediateMax);

        }

        @Override
        public Long getValue() {
            return intermediateMax;
        }

        @Override
        public void cleanup() {
            intermediateMax = null;
        }

        protected static Long max(Tuple input) throws ExecException{
            long returnMax = Long.MIN_VALUE;
            DataBag dataBag = (DataBag) input.get(0);
            for(Iterator<Tuple> it = dataBag.iterator();
                it.hasNext();){
                Tuple tuple = it.next();
                Long currentValue = (Long)tuple.get(0);
                if(currentValue > returnMax){
                    returnMax = currentValue;
                }
            }
            return returnMax;

        }

    }
```

The filter functions

Filter functions are also evaluation functions, but return a `Boolean` value. They can be used anywhere a `Boolean` expression is being evaluated. They are most commonly used as part of the `FILTER` operator. They implement the `FilterFunc` interface.

The load functions

These are functions that do input handling in Pig scripts. They implement the `LoadFunc` abstract class and are used along with the `LOAD` statement. The following example is a simple CSV file loader UDF. The methods that need to be overridden are `setLocation`, `getInputFormat`, `prepareToRead`, and `getNext`.

The `setLocation` function is used to inform the load location. The loader should in turn communicate this to `InputFormat`. The `setLocation` method might be called many times by Pig.

The `prepareToRead` method gets the `RecordReader` object of the `InputFormat` class. This `RecordReader` can then be used to read and parse records in the `getNext` function. The `getNext` method does the actual record parsing into Pig complex types. In the following example, it takes each line and parses record tuples.

The `getInputFormat` method gives Pig the `InputFormat` class used by the loader. Pig calls `InputFormat` in the same manner as a MapReduce Hadoop job. The following code snippet illustrates the CSV file loader UDF:



If files have to be read recursively from directories in HDFS, `PigFileInputFormat` and `PigTextInputFormat` can be used. These Pig-specific `InputFormat` classes are found in the `org.apache.pig.backend.hadoop.executionengine.mapReduceLayer` package. `TextInputFormat` and `FileInputFormat`, present natively in Hadoop, can only read files one level deep in the directory structure.

```
package MasteringHadoop;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.pig.LoadFunc;
import org.apache.pig.backend.hadoop.executionengine
    .mapReduceLayer.PigSplit;
import org.apache.pig.data.DataByteArray;
import org.apache.pig.data.Tuple;
import org.apache.pig.data.TupleFactory;

import java.io.IOException;
```

```
import java.util.ArrayList;


public class CsvLoader extends LoadFunc {
    private RecordReader recordReader = null;
    private TupleFactory tupleFactory =
        TupleFactory.getInstance();
    private static byte DELIMITER = (byte)',';
    private ArrayList<Object> tupleArrayList = null;


    @Override
    public void setLocation(String s, Job job) throws IOException {
        FileInputFormat.setInputPaths(job, s);
    }

    @Override
    public InputFormat getInputFormat() throws IOException {
        return new TextInputFormat();
    }


    @Override
    public void prepareToRead(RecordReader recordReader, PigSplit
        pigSplit) throws IOException {
        this.recordReader = recordReader;
    }

    @Override
    public Tuple getNext() throws IOException {
        try{

            if(recordReader.nextKeyValue()){

                Text value  = (Text)
                    recordReader.getCurrentValue();
                byte[] buffer = value.getBytes();

                tupleArrayList = new ArrayList<Object>();

                int start = 0;
                int i = 0;
```

```
                int len = value.getLength();

                while(i < len){

                    if(buffer[i] == DELIMITER){

                        readFields(buffer, start, i);
                        start = i + 1;

                    }
                    i++;
                }

                readFields(buffer, start, len);

                Tuple returnTuple =
                    tupleFactory.newTupleNoCopy(tupleArrayList);
                tupleArrayList = null;

                return returnTuple;


            }

        }
        catch(InterruptedException ex){
            //Error handling

        }
        return null;


    }


    private void readFields(byte[] buffer, int start, int i){
        if(start == i){
            //Null field
            tupleArrayList.add(null);
        }
        else{
            //Read from start to i
```

```
                tupleArrayList.add(new DataByteArray(buffer, start,
                    i));
            }

        }


    }
```

The store functions

Store UDFs are similar to load UDFs. They extend the abstract class, `StoreFunc`, and deal with Hadoop's `OutputFormat` family of classes and `RecordWriter`. The methods to override in the `StoreFunc` abstract class are `putNext`, `getOutputFormat`, `setStoreLocation`, and `prepareToWrite`.

Pig performance optimizations

In this section, we will look at different performance parameters and how to tune them for optimized Pig script execution.

The optimization rules

Pig applies optimization rules on the generated logical plan for a Pig script. By default, all rules are enabled. The `pig.optimizer.rules.disabled` property can be used to disable rules. The `-optimizer_off` command-line option can also be used when executing a Pig script to disable rules. Some rules are mandatory and cannot be disabled. The `all` option disables all the non-mandatory rules:

```
set pig.optimizer.rules.disabled <comma-separated rules list>
```

Alternatively, you can use the following command:

```
pig -t|-optimizer_off [rule name | all]
```



`FilterLogicExpressionSimplifier` is turned off by default. Setting the property `pig.exec.filterLogicExpressionSimplifier` to `true` can turn it on.

Most of the optimization rules discussed in the following section are simple and borrowed from database query optimizations:

- `PartitionFilterOptimizer`: This rule pushes all filtering upstream to the +er. Many loaders are partition-aware and will be instructed to load a partition with the filter predicate.
- `FilterLogicExpressionSimplifier`: Filter statement expressions are simplified by turning on this rule. Some of the simplifications done are as follows:
    - **Constant Pre-calculations**: Any expression that evaluates to a constant is precalculated:

      `X = FILTER A BY $0 > 2*5;` is simplified to `X = FILTER A BY $0 > 10;`
    - **Eliminations of negations**: Any negations in filter expressions are removed without a change in logic:

      `X = FILTER A BY NOT(NOT ($0 > 10) OR $0 > 20);` is simplified to `X = FILTER A BY $0 > 10 AND $0 <= 20;`
    - **Elimination of implied expressions in AND**: Any redundant logical conditions in an `AND` expression is eliminated:

      `X = FILTER A BY $0 > 5 AND $0 > 10;` is simplified to `X = FILTER A BY $0 > 10;`
    - **Elimination of implied expression in OR**: Any redundant logical conditions in an `OR` expression is eliminated:

      `X = FILTER A BY $0 > 5 OR $0 > 15;` is simplified to `X = FILTER A BY $0 > 5;`
    - **Elimination of equivalence**: Any equivalence in an expression is simplified:

      `X = FILTER A BY $0 != 5 AND $0 > 5;` is simplified to `X = FILTER A BY $0 > 5;`
    - **Elimination of filtering in presence of complementary expressions in OR**: Filtering is not done when there are complementary expressions in `OR`:

      No filtering is done in the case of `X = FILTER A BY $0 <= 5 OR $0 > 5;`

- **Elimination of 'always' true expressions**: Any filtering expression that always results in a true logical evaluation is eliminated:

```
X = FILTER A BY 1 == 1;
```

- `SplitFilter`: This optimizer rule tries to split filter statements. The `SplitFilter` optimization, when combined with other filter optimizations, can be very effective in terms of performance. In the following example, the `SplitFilter` optimization will split the `joinCountryFilter` relation into two filters:

```
joinCountryFilter1 = filter joinCountry by
    INDEXOF(cc::ccode, 'a', 0) == 0;
joinCountryFilter = filter joinCountryFilter1 by population > 0;

cc = load 'countrycodes.txt' using PigStorage(',') as
    (ccode:chararray, cname:chararray);

ccity = load 'worldcitiespop.txt' using PigStorage(',') as
    (ccode:chararray, cityName:chararray,
        cityFullName:chararray, region:int,
            population:long, lat:double, long:double);
joinCountry = join cc by ccode, ccity by ccode;
store joinCountry into 'country-code-join-pig' using
    PigStorage(',');
joinCountryFilter = filter joinCountry by
    INDEXOF(cc::ccode, 'a', 0) == 0 and population > 0;
```

- `PushUpFilter`: The idea behind this optimization is to push filter statements upstream in the data pipeline. The effect of such a move is the reduction of records that are to be processed. In the `SplitFilter` example, once the filters are split, `PushUpFilter` moves `joinCountryFilter1` and `joinCountryFilter` before the `JOIN` statement and directly after the `LOAD` statements.
- `MergeFilter` : The `MergeFilter` rule is the exact complement of `SplitFilter`. The `SplitFilter` rule is applied before `PushUpFilter`, but `MergeFilter` is applied after the application of `PushUpFilter`. Multiple filters on the same dataset are combined as a single filter:

  `X = FILTER A BY $0 > 10;` and
  `Y = FILTER X BY $1 > 10;` will be combined as
  `Y = FILTER A BY ($0 > 10 AND $1 > 10);`

- `PushDownForEachFlatten`: The `FLATTEN` operation within a `FOREACH` statement generally produces more tuples than the input. Adhering to the principle of processing the least number of records in the data pipeline, the `PushDownForEachFlatten` optimization pushes these `FOREACH` statements downstream. In the following example, the `FOREACH` statement will be moved after the `JOIN` statement:

```
X = FOREACH A GENERATE FLATTEN($0), $1;
Y = JOIN X BY $1, B BY $1;
```

- `LimitOptimizer`: Similar to `PushUpFilter`, the idea here is to move the `LIMIT` operator statements upstream. This reduces the number of records processed downstream.
- `ColumnMapKeyPrune`: The idea behind this optimization is to get the loader to load only the required columns of data. If the loader is unable to do this, a `FOREACH` statement is inserted just after the load call. This optimization works on map keys as well.
- `AddForEach`: The `AddForEach` optimization is used to prune columns as soon as the script does not require it. In the following example, `column1` is no longer used after the `ORDER` statement:

```
A = LOAD 'input.txt' AS (column1, column2);
X = ORDER A by column1;
Y = FILTER X by column2 > 0;
```

  A `FOREACH` operator is added in between the `ORDER` and `FILTER` statements:

```
X1 = FOREACH X GENERATE column2;
Y = FILTER X1 by column2 > 0;
```

- `MergeForEach`: This optimization merges multiple `FOREACH` statements into one `FOREACH` statement. This saves iterating over the dataset multiple times. This optimization is only possible if the following three preconditions are satisfied:
  - The `FOREACH` has no `FLATTEN` operator within it.
  - The `FOREACH` statements are consecutive.
  - The subsequent `FOREACH` statement is not nested. This is not applicable for the first `FOREACH` in the sequence.
- `GroupByConstParallelSetter`: In a statement executing `GROUP ALL`, even if `PARALLEL` is used to set the number of Reduce tasks, only one Reduce task is used. The rest of the Reduce tasks return empty results. This optimization automatically sets the number of Reduce tasks to one.

Measurement of Pig script performance

UDFs are developer-written functions that might require performance profiling to identify hotspots. Pig gives a couple of statistics on UDFs using Hadoop counters. A setting called `pig.udf.profile` can be set to `true`. With this setting enabled, Pig tracks the time taken to execute a particular UDF, and also the frequency of UDF invocation. The `approx._microsecs` function measures the approximate time spent in UDFs, and `approx._invocations` gives the developer the number of times UDFs were called during execution.



By setting `pig.udf.profile`, counters are enabled during Hadoop job executions. As we saw in the previous chapter, counters are global, and they add overhead on the tracking of Hadoop jobs. This setting should be judiciously used, preferably in test settings only.

Combiners in Pig

In the previous chapter, we saw how Combiners reduce disk I/O and save on the amount of data sent over the network from the Map to the Reduce tasks. In Pig, Combiners are invoked based on the structure of the script. The following are a few conditions under which Combiners are invoked:

- A non-nested `FOREACH` statement is used
- All the projections in a `FOREACH` statement are expressions on the grouped columns, or any UDFs used are algebraic functions, that is, they implement the `Algebraic` interface



Combiners are used in nested `FOREACH` statements as long as `DISTINCT` is the only operator used within the nesting.

A Combiner is not used under the following conditions:

- If the script fails the rules explained previously
- If there is any statement present between a `GROUP` and `FOREACH` statement; post Pig 0.9, the `LIMIT` operator is an exception to this rule



The logical optimizer might push any `FILTER` operators that come after `FOREACH` using the `PushupFilter` optimizer. This might prevent the use of Combiners.

Memory for the Bag data type

Bags are the only complex data types that are spilt into disks if they don't fit in memory. The `pig.cachedbag.memusage` setting determines the percentage of memory allocated to bags. The default setting is 0.2, that is, 20 percent of the memory is shared between all bags in the application.

Number of reducers in Pig

Unlike bare metal MapReduce, Pig determines the number of Reduce tasks automatically based on the size of the inputs. The input data size is divided by the value in the `pig.exec.reducers.bytes.per.reducer` parameter to figure out the number of Reduce tasks. The default value for the parameter is 1000000000 (1GB). However, the maximum number of Reduce tasks is capped by the value in setting `pig.exec.reducers.max`. Its default value is 999.

The class that implements the algorithm used to calculate the number of Reduce tasks is in value of setting `pig.exec.reducer.estimator`. A custom algorithm can override this as long as the class implements the `org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.PigReducerEstimator` interface and the full class name is assigned to the setting. Arguments can be passed to this custom algorithm by providing a value to the `pig.exec.reducer.estimator.arg` setting. This value will be passed to the constructor as a string parameter.

The multiquery mode in Pig

By default, Pig executes in the multiquery mode. All the statements in a Pig script are executed as one Pig job. One such query is as follows:

```
#Multi-query execution can be turned off explicitly by the -M or -
no_multiquery switch.
pig -M <script> or pig -no_multiquery <script>
```



`DUMP` should be avoided as it disables multiquery execution, resulting in the revaluation of relations and making the Pig script inefficient. Instead, a good practice is to use `STORE`. The interactive command `DUMP` forces the Pig compiler to avoid multiquery executions.

When a multiquery execution takes place, it is important for the user to distinguish between successful and failed executions of jobs. The `STORE` command outputs will have different paths, and the result of execution can be discerned by looking at the files. Also, at the end of execution, Pig gives out a result code indicating the status of the script execution. The following table gives the different result codes and their meaning:

Return code	Meaning
0	Success
1	Retrievable errors
2	Failure (all)
3	Failure (partial)

Best practices

The optimization rules in the previous section change the logical plan of a Pig script to enhance performance. We know that these rules will help develop efficient scripts. There are a few other practices that can speed up Pig scripts. These best practices cannot be made into rules as they are application and data specific. Also, the optimization rules tend to be conservative and might not guarantee the application of the rule.

The explicit usage of types

Pig supports many types, both primitive and complex. Type usages can speed up your scripts, sometimes up to 2X. For example, in Pig, all numerical computations without type specifications are considered as double computations. The `double` type in Pig takes up 8 bytes of storage, while an `int` type takes up 4 bytes. The computation using `int` is faster than the computation involving the `double` type.

Early and frequent projection

As we saw with the `AddForEach` and `ColumnMapKeyPrune` optimizers, it is a good practice to project only fields that are absolutely necessary downstream. This helps in reducing the overall data that needs to be transferred and processed downstream. It is also good practice to check your scripts for unused fields. Projecting only the necessary fields after each operation using a `FOREACH` statement can eliminate unused fields. Projecting early and often is a Pig best practice.

Early and frequent filtering

Similar to early projection, it is efficient to filter early and often. Again, filtering reduces the amount of data being transferred and processed downstream. Filtering reduces data by reducing the number of records, while projection reduces data by reducing the number of fields in the dataset.



If filtering removes an insignificant number of records, and the operation of filtering is expensive, it might not be efficient to filter early and often. Understanding your data is important when implementing this practice.

The usage of the LIMIT operator

Very often, we are interested in sampling or perusing the top few records of our results. The `LIMIT` operator can be used to do this. As we saw in the `LimitOptimizer` rule, `LIMIT` operations will be pushed upstream, reducing the overall processing time.

The usage of the DISTINCT operator

There are two ways of finding the number of distinct elements in a field: one is to use the `GROUP` operator and generate the group key, and the other is to use the `DISTINCT` operator available in Pig. The latter is more efficient than the former.

The reduction of operations

`MergeForEach` and `MergeFilter` merge consecutive `FOREACH` and `FILTER` statements to a single `FOREACH` or `FILTER` statement. Look out for such opportunities where combining multiple operations is possible. Reducing the number of operators in the data pipeline increases the performance of a Pig script.

The usage of Algebraic UDFs

When UDFs are developed and the nature of the processing is algebraic, it is good practice to write UDFs that implement the `Algebraic` interface. Combiners are called when algebraic UDFs work on grouped data. The usage of Combiners enhance the job performance in MapReduce.

The usage of Accumulator UDFs

Accumulator UDFs can be used to reduce the amount of memory required by UDFs by taking the input in chunks.

Eliminating nulls in the data

A `JOIN` or `GROUP` operation on relations send all `NULL` keys to a single Reduce task. If `FLATTEN` is used to unwrap the grouping, all `NULL` records will be eliminated. However, this elimination happens after the Reduce task is executed. Actively filtering for nulls before a `JOIN` or `GROUP/COGROUP` operator can improve the script performance significantly by getting rid of Reduce tasks on the `NULL` keys.

The usage of specialized joins

The second input of a regular join is streamed instead of bringing it to memory. This is a regular join optimization that exists in Pig. When joining datasets of different sizes, it is more efficient to get the dataset with a larger number of records as the last input to the join:

```
C = JOIN small_file BY s, large_file by F;
```

As we saw in the *Specialized joins in Pig* section, a number of other join optimizations can be leveraged in Pig.

Compressing intermediate results

A Pig script might get compiled into a number of MapReduce jobs. Each job might produce intermediate output. The output can be compressed using the LZO compression codec. This will not only help save on HDFS storage but will also help in faster job execution by reducing load times.

The `pig.tmpfilecompression` property determines whether intermediate file compression is switched on or off. By default, the value is `false`. The `pig.tmpfilecompression.codec` property value is the codec used for compression. Currently, Pig accepts `gz` and `lzo` as possible values for this parameter. Though the GZIP compression codec provides better compression, it is not preferred because of it slowing down during execution times.

Combining smaller files

In *Chapter 2*, *Advanced MapReduce*, we saw the problems associated with small files and the usage of `CombineFileInputFormat`. Pig now has in-built support for combining smaller files. This reduces the number of splits, and in turn, the number of Map tasks.

The `pig.splitCombination` property can be set to `true` to combine smaller files. The size of each split is given by `pig.maxCombinedSplitSize`. This property value takes the suggested size in bytes of the input to each Map task. Smaller files are combined until this limit is reached.



Combining small files works well for the in-built `PigStorage` loader. If you are writing a custom loader, the loader must be stateless across invocations to the `prepareToRead` method. Also, the loader must not implement the `IndexableLoadFunc`, `OrderedLoadFunc`, and `CollectableLoadFunc` interfaces.

Summary

In this chapter, we went through the advanced features of Pig. We looked into the optimizations that Pig has to offer. The following are a few key takeaways from this chapter:

- As a rule, try to use Pig in as many situations as you can. Pig's abstractions, development aids, and flexibility can save you both time and money. Stretch Pig's capabilities before reverting to MapReduce jobs.
- The logical plan optimizations might change the order of statement execution. Use `EXPLAIN` and `ILLUSTRATE` extensively to study Pig scripts.
- Help Pig to execute your script faster by following some of the guidelines mentioned in this chapter. Try to make your UDFs implement the `Algebraic` or `Accumulator` interface, ideally both.
- Understand the data you are trying to process. Specialized support is available for some kinds of data quirks, such as Skewed joins for joins on skewed data.

In the next chapter, we will look at advanced features of a higher-level SQL abstraction on Hadoop MapReduce called Hive.

4
Advanced Hive

SQL is a popular data-processing language that has been around for four decades. There are scores of people who are already familiar with Relational Data Stores and SQL. A natural step in onboarding more users onto Hadoop is to flatten the learning curve by bringing in concepts they are well versed with. Hive introduces relational and SQL concepts into Hadoop MapReduce. In the chapter on Pig, you saw the advanced usage of Pig scripts to author MapReduce workflows. In this chapter, we will delve into the advanced usage of Hive.

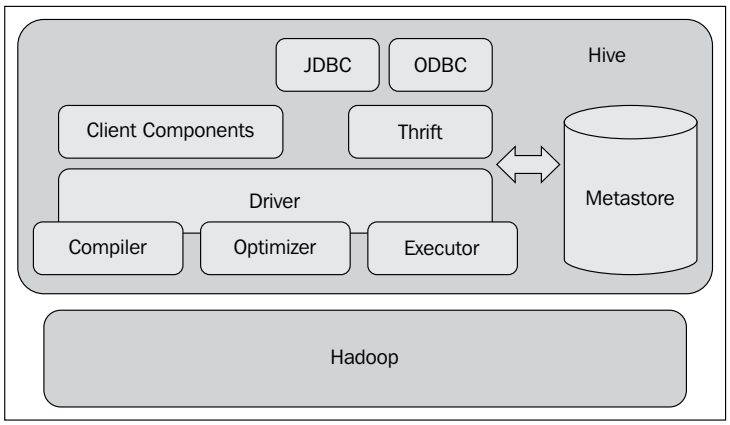
Apache Hive is often described as a data warehouse infrastructure. Traditionally, business intelligence is gathered from a data warehouse, a database that stores data from many sources within an enterprise. This database stores both historical and current data in an enterprise. This data store is primarily queried for reporting and analytics. Traditionally, infrastructure that made data warehouses consisted of **Relational Databases** (**RDBMS**), and the query language used to perform analysis and generate reports was SQL. Data warehouse infrastructure used to be made of Relational Data Stores and queried using SQL. Special star or snowflake schemas were used to model these data stores. Apache Hive continues this tradition of SQL, but it changes the underlying data store to HDFS. The queries are translated into MapReduce jobs. The variant of SQL used in Hive queries is called **HiveQL**.

In this chapter, we will discuss the following topics:

- The architecture of Hive on a Hadoop cluster
- The data types supported by Hive, underlying file formats, and data model
- The different query plan optimizers available with Hive and their significance
- The extensibility options provided by Hive, such as UDFs, UDAFs, and UDTFs

The Hive architecture

The following diagram shows the Hive architecture. We will look at each component in detail:



The Hive metastore

The metastore is a database for system-related metadata. It stores details about the tables, partitions, schemas, column types, and table locations. It can be accessed via the Thrift interface, making it possible to read this data using clients written in many different programming languages. The data is stored in a relational database system and uses an **Object-relational mapping** (**ORM**) layer to read and write data into the store. The choice of using an RDBMS for the metastore was made to reduce the latency when serving this information to the Hive query compiler.

The ORM layer of the metastore allows a pluggable model where any RDBMS can be plugged into Hive. The default RDBMS used is Apache Derby, an open source relational data store. In practice, organizations use MySQL and other popular RDBMS suites to host the metastore. The data in the metastore imposes structures on otherwise raw HDFS files. This makes it critical to protect the metastore from crashes by regular backups or replication. The metastore is only accessed during compilation and never when MapReduce jobs are run.

The Hive compiler

The compiler takes a HiveQL query and translates it into MapReduce jobs. A parser parses the query and builds an **abstract syntax tree** (**AST**). The AST is checked for types and semantic consistencies. Metadata from the metastore is used to achieve this step. The output of the checks is an operator DAG. A series of optimization transformations are then applied on the DAG. The transformations are chained, and the output is an optimized operator tree. Users are allowed to add their transformations by implementing the `Transform` interface. We will discuss some of the optimizations later in this chapter.

The optimized DAG is then translated into a physical plan. The physical plan is a set of MapReduce and HDFS jobs. A HDFS job is used to read and write data from HDFS.

The Hive execution engine

The execution engine takes the plan generated by the compiler and executes the job strictly in order of their dependencies. The plan is communicated to each task in the Hadoop cluster via a `plan.xml` file. This file is distributed across the cluster using a side channel such as DistributedCache. The job outputs are stored in temporary locations. On the completion of the entire query, if a store location is specified, these files are moved to appropriate locations as specified by the **Data Manipulation Language** (**DML**). In the case of a query without a store location, the results are served directly from the temporary location.

The supporting components of Hive

The Hive infrastructure has a number of supporting components, which are as follows:

- A **Driver** is the component that handles query submissions. It is responsible for orchestrating the life cycle of a query by invoking the components in the correct order to fulfill it. The Driver also spawns sessions and keeps track of session statistics.
- There are many client components that are used to submit queries to Hive. The notable ones are **Command Line Interface** (**CLI**), a web interface, and JDBC/ODBC connectors. Thrift serialization is used extensively as the serialization library in Hive.
- Extensibility components, such as the **SerDe** and **ObjectInspector** interfaces, are present to help users integrate with different data types and legacy data. **User-defined Functions** (**UDFs**) and **User-defined Aggregate Functions** (**UDAFs**) are custom functions that can be written by the user to extend Hive's capabilities.

Data types

Hive supports all the primitive numeric data types such as `TINYINT`, `SMALLINT`, `INT`, `BIGINT`, `FLOAT`, `DOUBLE`, and `DECIMAL`. In addition to these primitive data types, Hive also supports string types such as `CHAR`, `VARCHAR`, and `STRING` data types. Like SQL, the time indicator data types such as `TIMESTAMP` and `DATE` are present. `BOOLEAN` and `BINARY` miscellaneous types are available too.

A number of complex types are also available. Complex types can be composed from other primitive or complex types. The complex types available are as follows:

- `STRUCTS`: These are groupings of data elements similar to a C-struct. The dot notation is used to dereference elements within a struct. A field within column C defined as `STRUCT {x INT, y STRING}` can be accessed as `A.x` or `A.y`.

  The syntax for this is `STRUCT<field_name : data_type>`

- `MAPS`: These are key-value data types; providing the key within square braces can access a value. A value of a map column *M* that maps from key *x* to value *y* can be accessed by *M[x]*.There is no restriction on the type stored by the value, though the key needs to be of a primitive type.

  The syntax for this is `MAP<primitive_type, data_type>`

- `ARRAYS`: These are lists that can be randomly accessed through their position. The syntax to access an array element is the same as a map. However, what goes in the square braces is a zero-based index of the element.

  The syntax for this is `ARRAY<data_type>`

- `UNIONS`: This is a union-type available in Hive. It can hold an element of one of the data types specified in the union.

  Syntax: `UNIONTYPE<data_type1, data_type2…>`



Functions and data types in Hive are case insensitive.



Hive Version >= 0.7.0 – UNIONTYPE complex data type is available.

Hive Version >= 0.8.0 – TIMESTAMP and BINARY data types are available.

Hive Version >= 0.11.0 – DECIMAL data type is available.

Hive Version >= 0.12.0 – DATE and VARCHAR data types are available.

Hive Version >= 0.13.0 – CHAR data type is available.

File formats

Hive supports a number of file formats out of the box. In this section, we will inspect some of these file formats and their utilities.

Compressed files

For some use cases, storing files in a compressed format within HDFS is advantageous. This strategy not only uses less storage, it also can reduce query times. Hive provides importing files stored in GZIP and BZIP2 formats directly into tables. During query execution, these files are decompressed and given as inputs to Map tasks. However, files compressed with GZIP and BZIP2 compression schemes cannot be split and are processed within a single Map task.

In practice, files stored in these compressed file formats are loaded into a table whose underlying data format is a Sequence file. Sequence files can be split and distributed to different Map tasks.



The `io.seqfile.compression.type` property tells Hive how the compression of the Sequence file should happen. It can take two values, `RECORD`, where each record is compressed and, `BLOCK`, where 1MB of the file is buffered before compressing it.

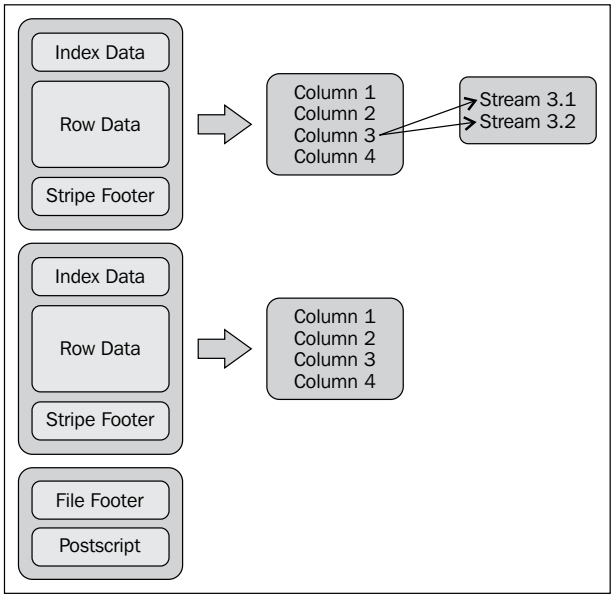
Lempel–Ziv–Oberhumer (**LZO**) compression is a lossless compression codec that trades compression for speed. If a Hadoop cluster needs to use the LZO compression, it has to be installed on each node within the cluster. By setting `mapreduce.output.fileoutputformat.compress.codec` to the LZO codec and `mapreduce.output.fileoutputformat.compress` to `true` for the Hadoop cluster, we can enable LZO compression for the output files. Setting the Hive property `hive.exec.compress.output` to `true` makes Hive query outputs to be stored in the LZO compressed format.

ORC files

ORC stands for **Optimized Row Columnar** files. This is a file format particularly attractive for Hive. These files are similar to RCFiles (Row Columnar Files), but have additional optimizations.

The following diagram shows the structure of an ORC file. It contains groups of row data called stripes. These stripes are typically 250MB in size. At the end of the file is a file footer and postscript section. The file footer contains information such as the stripe metadata, the number of rows in each stripe, and statistics such as count, minimum, maximum, and sum of data in each stripe.

Each stripe also has a footer that contains local statistics for an individual stripe. The key characteristic of an ORC file is that it stores records in a columnar format. Each column is contiguously stored for all rows. This increases the I/O efficiency for aggregate queries. The following diagram is a schematic of an ORC file:



The Parquet files

Parquet is another wide columnar store for Hive. From Hive 0.10.0 to 0.12.0, Parquet was available as an external package that needed to be explicitly installed. However, in Hive 0.13.0, it is natively available in Hive. Parquet is interesting when there is a need to store nested information in records.



SerDe stands for **Serialization and Deserialization**. They are used to read and write the rows in a Hive table. The SerDe module sits in between the file format and object representation of the row:

HDFS File -> InputFileFormat -> <key,value> -> DeSerializer -> Row Object

Row Object -> Serializer -> <key,value> -> OutputFileFormat -> HDFS file

The data model

Hive data is organized as databases. A **database** is a logical collection of Hive tables. A database within Hive assigns a namespace for its tables. If no namespace is assigned to Hive tables, it belongs to the `default` namespace. The creation of a database results in the creation of an HDFS directory for the files in the database. This directory serves as the namespace for the tables. The `CREATE DATABASE MasteringHadoop` command creates a MasteringHadoop database. When we list the HDFS directory structure, we see a directory created for this database, as shown:

```
drwxr-xr-x   - sandeepkaranth supergroup          0 2014-05-15 08:55
/user/hive/warehouse/masteringhadoop.db
```

A **table** is the basic unit of data storage similar to traditional RDBMS. It logically groups records of the same type. Records are rows corresponding to typed columns. A table maps to a single directory within HDFS. Hive also allows imposing structures on existing data locations via **external tables**. Metadata stored within Hive for each table includes the column types and list of columns. It also includes other information such as the owner of the table, serialization and deserialization information (SerDe) for the columns, data storage formats, and bucketing-related metadata. Databases and tables are stored in the HDFS location specified by the `hive.metastore.warehouse.dir` property. These properties are defined in the `hive-default.xml` or `hive-site.xml` file in the Hive installation configuration (`conf`) directory.



When specifying the `LOCATION` for an `EXTERNAL` table in HDFS, Hive expects the data files to be in a directory.

Partitions are divisions of the table based on distinct column values. When partition columns are specified, all the records corresponding to distinct values or value combinations of the columns are stored in a subdirectory within the `table` directory. Partitions are used as prefilters to prune unnecessary records from getting processed, helping decrease the query latency and I/O. It must be noted that partitions also increase the number of files in HDFS, correspondingly the number of Map tasks and intermediate outputs. The right number of partitions has to be worked out for best performance.



A `DROP` table on an `EXTERNAL` table does not delete the data in HDFS.

Buckets or **clusters** are files in the leaf-level directories that correspond to records that have the same column-value hash. A Hive user can specify the number of buckets per partition, or per table if partitions are not present. Each record is placed in this bucket, generally by taking the hash of the bucketed column modulo to the number of buckets specified. Buckets are useful to sample data.

Let's take the `worldcitiespop.txt` file and create a table out of it. The following **Data Definition Language** (**DDL**) query shows how a schema can be imposed on an external table:

```
CREATE EXTERNAL TABLE MasteringHadoop.worldcities_external(code
VARCHAR(5), name STRING, fullName STRING, region INT, population
BIGINT, lat FLOAT, long FLOAT)
COMMENT 'This is the world cities population table'
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION '/user/sandeepkaranth/worldcitiespop';
```

The keyword, `EXTERNAL`, is used to indicate an existing table. There are keywords associated with every bit of metadata that is stored in the Metastore. `ROW FORMAT` is used to specify the serialization and deserialization semantics of a table. If `ROW FORMAT` is not specified, or `ROW FORMAT DELIMITED` is specified, a Hive native SerDe is used to produce the table rows. The `STORED AS` clause specifies the underlying file format used by the table, and `LOCATION` indicates the place where the table data is stored within HDFS. When the `EXTERNAL` keyword is used, no additional HDFS directories are created.



In Hive, it is mandatory that each table maps to an HDFS directory, including an `EXTERNAL` table. In the example, `worldcitiespop` is an HDFS directory that contains the `worldcitiespop.txt` file. Hive disallows the imposing of the table structure directly on a file.

Now, let's create a table that is not external using the following DDL query:

```
CREATE TABLE MasteringHadoop.worldcities(code VARCHAR(5), name
STRING, fullName STRING, region INT, population BIGINT, lat FLOAT,
long FLOAT)
COMMENT 'This is the world cities population table'
PARTITIONED BY (region_p INT)
CLUSTERED BY (code) SORTED BY (code) INTO 2 BUCKETS
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS SEQUENCEFILE;
```

This table definition specifies a partition column within the table. It also indicates the number of buckets, that is, two, within each partition. The underlying file format of the table is the `SEQUENCEFILE` format. A partition column cannot have the same name as any other column. When loading data into the table, the partition column must be treated as a separate column.

Bucketing information is specified by the `CLUSTERED` keyword. In the preceding example, we cluster the data based on the country code and specify a sort column within each bucket. The number of buckets is specified as two. When bucketing, it is very important to set the number of Reduce tasks to be equal to the number of buckets to populate the right number of buckets. This can be done in a couple of ways. One of the ways is to set the number of Reduce tasks explicitly for every job. The other way is to let Hive automatically bucket the data by setting the `hive.enforce.bucketing` property to `true`.

Using the following DML query, we will populate the previous table using the external table we created:

```
set hive.enforce.bucketing = true;
set hive.enforce.sorting = true;
set hive.exec.dynamic.partition = true;
set hive.exec.dynamic.partition.mode=nonstrict;
set hive.exec.max.dynamic.partitions.pernode=1000;
FROM MasteringHadoop.worldcities_external
INSERT OVERWRITE TABLE MasteringHadoop.worldcities
PARTITION(region_p)
SELECT code, name, fullName, region, population, lat, long, region
WHERE region IS NOT NULL;
```



A table definition that has a column with the same name as the partition column throws the error `FAILED: SemanticException [Error 10035]: Column repeated in partitioning columns.`

The end result is a table with partitions based on the distinct value of the region code. Each partition has two buckets that are sorted on the country code. The underlying HDFS directory structure for the partitions is given below. Each partition is a separate directory. The distinct value of the column that is stored in the partitioned column is part of the directory name.

```
drwxr-xr-x   - sandeepkaranth supergroup          0 2014-05-15 11:55 /user/hive/warehouse/masteringhadoop.db/worldcities/region_p=91
drwxr-xr-x   - sandeepkaranth supergroup          0 2014-05-15 11:55 /user/hive/warehouse/masteringhadoop.db/worldcities/region_p=92
drwxr-xr-x   - sandeepkaranth supergroup          0 2014-05-15 11:55 /user/hive/warehouse/masteringhadoop.db/worldcities/region_p=93
drwxr-xr-x   - sandeepkaranth supergroup          0 2014-05-15 11:55 /user/hive/warehouse/masteringhadoop.db/worldcities/region_p=94
drwxr-xr-x   - sandeepkaranth supergroup          0 2014-05-15 11:55 /user/hive/warehouse/masteringhadoop.db/worldcities/region_p=95
drwxr-xr-x   - sandeepkaranth supergroup          0 2014-05-15 11:55 /user/hive/warehouse/masteringhadoop.db/worldcities/region_p=96
drwxr-xr-x   - sandeepkaranth supergroup          0 2014-05-15 11:55 /user/hive/warehouse/masteringhadoop.db/worldcities/region_p=97
drwxr-xr-x   - sandeepkaranth supergroup          0 2014-05-15 11:55 /user/hive/warehouse/masteringhadoop.db/worldcities/region_p=98
drwxr-xr-x   - sandeepkaranth supergroup          0 2014-05-15 11:55 /user/hive/warehouse/masteringhadoop.db/worldcities/region_p=99
```

On inspecting a partition directory, we see the buckets as files. Each of the partitions has two files, indicating two buckets.

```
-rw-r--r--   3 sandeepkaranth supergroup       3627 2014-05-15 11:55 /user/hive/warehouse/masteringhadoop.db/worldcities/region_p=99/000000_0
-rw-r--r--   3 sandeepkaranth supergroup     183750 2014-05-15 11:55 /user/hive/warehouse/masteringhadoop.db/worldcities/region_p=99/000001_0
```

Dynamic partitions

Partitioning can be of three types: static, dynamic, and hybrid. If the partition information for a column is available during compile time, it is termed a statically partitioned column. The user, when defining the tables, specifies the values taken up by the column. In contrast, for dynamically partitioned columns, the partitions are defined and determined during a query execution.



Partitions are HDFS directories and can be directly added to HDFS using the `hdfs put` command. However, the metastore has the master information for all the tables. It will not be aware of any partition that is directly added to HDFS. Hive provides a command to automatically update the metastore and recover the partition. The command is `MSCK REPAIR TABLE tableName;`. On Amazon EMR, the command is `ALTER TABLE tableName RECOVER PARTITIONS;`.

The DML query to populate the `MasteringHadoop.worldcities` table is dynamically partitioned. The values for the `region_p` column are determined when this table is populated from the `MasteringHadoop.worldcities_external` table. In the `INSERT..SELECT` DML query as the previous one, all dynamically partitioned columns should come at the end of the `SELECT` statement and appear in the same order as they appear in the `PARTITION` directive. The `region_p` partition column value comes from the region column of the `EXTERNAL` table and is specified last in the `SELECT` statement.

Semantics for dynamic partitioning

Some key semantics when using the dynamic partitioning feature in Hive are as follows:

- By default, dynamic partitioning is turned off in Hive. It can be enabled by setting the value of `hive.exec.dynamic.partition` to `true` in the `hive-default.xml` or `hive-site.xml` configuration file.
- If partitions with the same values as the dynamically loaded data already exist, the existing partitions will be overwritten.
- A `NULL` or empty partition column will be sent to a partition determined by the value of the `hive.exec.default.partition.name` property. The default value of this property is `__HIVE_DEFAULT_PARTITION__`.
- Dynamic partitions are governed and limited by the following three important properties:
    - The `hive.exec.max.dynamic.partitions` property imposes a limit on the total number of partitions that can be created by a DML query. The default value is 1,000. If the number of partitions exceeds 1,000, an exception is raised by the MapReduce job.
    - The `hive.exec.max.dynamic.partitions.pernode` property imposes a limit on the total number of partitions that can be created by a single Map or Reduce task. The default value is 100. A fatal error is raised if a task exceeds this limit.

- The `hive.exec.max.created.files` property imposes a limit on the number of files created by all Map and Reduce tasks globally when a DML query is executed. The default value is 100,000.



The following fatal error is displayed when a Reduce task exceeds the number of dynamic operations:

```
2014-05-15 08:46:27,647 FATAL [Thread-17]: ExecReducer
(ExecReducer.java:reduce(282)) -
org.apache.hadoop.hive.ql.metadata.HiveFatalException:
[Error 20004]: Fatal error occurred when node tried to
create too many dynamic partitions. The maximum number
of dynamic partitions is controlled by
hive.exec.max.dynamic.partitions and
hive.exec.max.dynamic.partitions.pernode. Maximum was
set to: 100
```

- By default, a query with all dynamically partitioned columns is not allowed. This is because the value of the `hive.exec.dynamic.partition.mode` property is set to `strict`. In the preceding example, we have no statically partitioned columns. In order for the query to work, the dynamic partition mode property has to be set to `nonstrict`.

Indexes on Hive tables

In an RDBMS, indexing is used for a faster lookup of data, which in turn relates to faster queries. Indexes are data structures that enable random and efficient access of database records based on keys. An index itself might not store the entire record, but only a pointer to the record. Hive indexes are similar to nonclustered indexes from traditional databases. They keep track of mapping between the data and relevant HDFS blocks they reside in. This enables a MapReduce job to figure out only the relevant blocks to process queries.

In the following example, an index is created using two different handlers, **compact** and **bitmap**. A Hive index is nothing but a table in HDFS. The `DEFERRED REBUILD` directive is used to instruct Hive to populate the index at a later stage. An `ALTER INDEX` command can be issued to build the index at a later point of time:

```
USE MasteringHadoop;
CREATE INDEX worldcities_idx_compact ON TABLE worldcities (name)
AS 'COMPACT' WITH DEFERRED REBUILD;
CREATE INDEX worldcities_idx_bitmap ON TABLE worldcities (name) AS
'BITMAP' WITH DEFERRED REBUILD;
DESCRIBE masteringhadoop__worldcities_worldcities_idx_compact__;
```

The output of the `DESCRIBE` operation on the compact index tables is shown as follows. For each partition and bucket, the index table holds an array of offsets. These offsets can be used to directly get the block of data:

```
hive> DESCRIBE
masteringhadoop__worldcities_worldcities_idx_compact__;
OK
name                    string
_bucketname             string
_offsets                array<bigint>
region_p                int

# Partition Information
# col_name              data_type               comment

region_p                int
Time taken: 0.078 seconds, Fetched: 9 row(s)
```

Bitmap indexes are used when the number of possible values taken by the indexed column is less. The index table structure for a Bitmap index is also similar, but the information encoded is different. The `_bitmaps` field stores a bit for each record in the table. If the value is present in the record, this particular bit is turned on, otherwise it is turned off. The index table structure of a bitmap index is shown as follows:

```
hive> DESCRIBE masteringhadoop__worldcities_worldcities_idx_bitmap__;
OK
name                    string
_bucketname             string
_offset                 bigint
_bitmaps                array<bigint>
region_p                int

# Partition Information
# col_name              data_type               comment

region_p                int
Time taken: 0.083 seconds, Fetched: 10 row(s)
```

Hive query optimizers

After type checking and semantic analysis of the query, a number of rule-based transformations are applied to optimize the query. We will discuss some of these optimizations here. Custom optimizations can be written by implementing the `org.apache.hadoop.hive.ql.optimizer.Transform` interface. This interface has one method that takes in a `ParseContext` object and returns another after the transformation. The `ParseContext` object has the current operator tree, among other information.

The following are the few optimizations that are already available with Hive 0.13.0:

- `ColumnPruner`: This operator tree is walked to determine the minimal number of columns in the base table that are required to fulfill the query. Any additional columns in the base table are pruned away by inserting a `SELECT` statement when reading the base tables. This reduces the amount of data read, processed, and written.
- `GlobalLimitOptimizer`: When a `LIMIT` operator is used in a query, this particular optimizer sets `GlobalLimitCtx`. This assists other optimizer rules that appear downstream to take smarter and efficient decisions.
- `GroupByOptimizer`: When the `GROUP BY` key is a superset of the bucketing and sorting keys, the grouping can be done on the Map side. This optimizer takes care of modifying the plan accordingly. The ordering of the keys must be the same too.
- `JoinReorder`: Based on user-specified hints, tables that are to be streamed are processed last in a join operation.
- `PredicatePushdown`: This is a term carried forward from the RDBMS world. It is a misnomer and is actually Predicate Pushup. The idea here is to move predicates that filter out data upstream close to the data source. This will enable less record processing downstream, saving both I/O and network bandwidth costs. A significant query speed can be achieved with this. By default, `PredicatePushdown` is switched off. Setting the `hive.optimize.ppd` property to `true` enables `PredicatePushdown`.
- `PredicateTransitivePropagate`: This optimization rule propagates a predicate to other relations in a join. When two tables are joined and the join key of one table is filtered, the filtering predicate can be used on the other table as well.

- `BucketingSortingReduceSinkOptimizer`: If the source and destination tables are bucketed and sorted on the same keys in the same order, no Reduce task is required to conduct join or insert operations on these tables. For example, in the `INSERT OVERWRITE A SELECT * FROM B;` query, if the `A` and `B` keys are bucketed and sorted on the same table, Map tasks are sufficient to fulfill the query completely. Eliminating Reduce means making the query faster as there is no need for the shuffle/sort step.
- `LimitPushdownOptimizer`: If no filter operators are present in a statement having the `LIMIT` operator, the Map tasks of the query can be optimized to retrieve a top K result. This top K result can then be passed to the `LIMIT` operator. This greatly reduces the number of records flowing through the shuffle/sort stage.
- `NonBlockingOpDeDupProc`: This optimization merges projections and filters them into single statements.
- `PartitionPruner`: To prevent the metastore from memory issues, partition names are fetched first, and then, each partition information is retrieved based on the need.
- `ReduceSinkDeDuplication`: If two Reduce tasks have the same partition columns and order, they can be merged into a single task.
- `RewriteGBUsingIndex`: A `GROUP BY` operation can be conducted by conducting a table scan on the index table rather than on the base table. This is possible if the key columns have an index over them. For example, `SELECT COUNT(k) FROM A GROUP BY k` can be written as `SELECT SUM(_count_of_k) FROM index_table GROUP BY k;`. This optimization might not be applicable for all `GROUP BY` queries.
- `StatsOptimizer`: There can be many queries that can be answered directly from the metastore statistics. Queries involving `MIN`, `MAX`, and `COUNT` are some examples of queries that can potentially be answered without spawning any MapReduce task. This optimizer detects this possibility and optimizes the query.

Advanced DML

The Data Manipulation Language provided by Hive is equivalent in features to any state-of-the-art SQL system. They provide standard operations such as the `JOIN`, `GROUP BY`, and `UNION` operations. The semantics might vary marginally depending on the operation. Different kinds of optimization hints are also present.

The GROUP BY operation

The `GROUP BY` operation is the same as in standard SQL, except for a few advanced features:

- **Multi-Group-By Inserts**: It is possible to have multiple `GROUP BY` clauses within a single query. The output can be written to multiple tables or HDFS files. For example, the following query is possible:

  ```
  FROM src_table INSERT OVERWRITE TABLE id_count SELECT id,
  COUNT(id) GROUP BY id INSERT OVERWRITE TABLE id_sum SELECT id,
  SUM(id_value) GROUP BY id;
  ```

- **Map-side aggregation for GROUP BY**: By setting the `hive.map.aggr` property to `true`, it is possible to enforce one level of aggregation on the Map tasks. This will yield a better-performing query.

ORDER BY versus SORT BY clauses

Hive has both an `ORDER BY` and a `SORT BY` clause to sort the output of a query. The difference between the two is that `ORDER BY` imposes a total order on query results, but `SORT BY` imposes order only on the rows in a Reduce task. If there are multiple Reduce tasks, the output data will only have a partial order when `SORT BY` is used.

It is easy to see that the `ORDER BY` clause requires a single Reduce task to achieve total order on the query, a bottleneck to sort large datasets. Hive enforces the need for a `LIMIT` operator by default when an `ORDER BY` clause is used. This is governed by the `hive.mapred.mode` setting. The default value is `strict`. By setting it to `nonstrict`, the compiler does not impose the mandatory `LIMIT` restriction. However, this is not recommended.

The JOIN operator and its types

The `JOIN` operators are very important to process data on multiple datasets or tables. A number of `JOIN` operator variants are facilitated in Hive. Hive provides a number of join optimization options based on the nature of the data. The following are some important properties and optimizations in the `JOIN` operator:

- Hive supports inner joins, outer joins, and the left semi-join. All joins are based on the equality operator. Fuzzy and theta joins are not supported.
- A multitable join is allowed. The number of MapReduce jobs depends on the number of key columns participating in the join. If the same key columns are used to join multiple tables, a single MapReduce job will be used.

- By default, the last table participating in the join is streamed to the Reduce task. The rest of the tables are buffered. It is important to place the biggest table at the end when conducting a join for better performance. The user can explicitly specify a `STREAMTABLE` hint to override the default. For example, consider the `SELECT /*+ STREAMTABLE(A)*/ A.x, B.x, C.x FROM A JOIN B ON (A.key = B.key) JOIN C ON (B.key = C.key);` query. The `STREAMTABLE` hint instructs the Hive compiler to stream table A. By default, it will have streamed table C.
- A `WHERE` clause appearing in a `JOIN` statement filters rows after the `JOIN` operator is executed. It is good practice to colocate the `WHERE` clause filters in the `JOIN` itself with the `ON` clauses, where the `JOIN` keys are specified.

Map-side joins

When one of the tables participating in a join is small, the join operation can be conducted directly on the Map task. This can be specified by the `MAPJOIN` hint in Hive. For example, `SELECT /*+ MAPJOIN(A) */ A.x, B.x FROM A JOIN B ON (A.key = B.key)` hints that A being a smaller table can be loaded into memory and a Map-side join can be performed.

If the tables participating in a join are bucketized on the join columns, and the number of buckets of one table is equal or a multiple of the number of buckets in the other table, a Map-side join can be done. Though this is not the default behavior, it can be enabled by setting the `hive.optimize.bucketmapjoin` property to `true`. This is also known as the **bucketized map-side join**.

If the tables participating in a join are bucketized on the join columns, have the same number of buckets, and the buckets are sorted on the join columns, a sort-merge operation can be done to join the two tables. However, this is again not the default behavior. In addition to setting the `hive.optimize.bucketmapjoin` property to `true`, the `hive.optimize.bucketmapjoin.sortedmerge` property has to be set to `true`, and `hive.input.format` has to be set to `org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat`. This is also known as the **bucketized sort-merge join**.

Setting the `hive.auto.convert.join` property to `true` automatically converts a join to a Map-side join. The `MAPJOIN` hint is no longer required. However, the hint is mandatory for the bucketized map-side and sort-merge joins.

Advanced aggregation support

The primary use case of Hive is analytics on data warehouses. This calls for advanced aggregation support to collect and report statistics from data on different dimensions.

Hive supports `GROUPING SETS`, where more than one `GROUP BY` operation can be done on a single table. It is equivalent to executing two different `GROUP BY` queries and then applying `UNION` on them. The following command shows the usage of `GROUPING SET`:

```
SELECT x, y, SUM(z) FROM X GROUP BY x, y GROUPING SETS( (x,y), y);
```

This query is equivalent to `UNION` of the `SELECT x, y, SUM(z) FROM X GROUP BY x,y;` and `SELECT null, y, SUM(z) FROM X GROUP BY y;` queries.

Cubes are multidimensional data structures used to drill down, roll up, and aggregate facts. Hive simulates cubes' queries by aggregating facts over all combinations of dimensions. For example, `SELECT x, y, z, SUM(a) FROM X GROUP BY x, y, z WITH CUBE;` is equivalent to the `SELECT x,y,z, SUM(a) FROM X GROUP BY x,y,x GROUPING SETS ((x,y,z), (x,y), (y,z), (x,z), (x), (y), (z), ());` query.

Hive also supports the `ROLLUP` command to compute aggregates at each hierarchy level. A query of the `SELECT x, y, z, SUM(a) FROM X GROUP BY x,y,z WITH ROLLUP;` form is equivalent to `SELECT x,y,z, SUM(a) FROM X GROUP BY x,y,z GROUPING SETS ((x,y,z) , (x,y), (x), ());`.

There is an implicit assumption of the `x` hierarchy being drilled down to `y`, and `y` can in turn being drilled down to `z`. The `ROLLUP` directive creates a number of rows for every single input row. In the preceding example, for every input row, four output rows are generated with three group keys. The higher the cardinality of the group keys, the worse it gets at the Map and Reduce task boundaries. In these cases, it is better to spawn multiple MapReduce jobs. The cardinality threshold, beyond which an additional job is launched, is given by the `hive.new.job.grouping.set.cardinality` setting.

Other advanced clauses

There are many other clauses for the advanced usage of Hive queries:

- The `EXPLODE` user-defined table generation function can be used to produce multiple rows for a single input row. For example, `EXPLODE` on an array type column produces a row for each element of the array.
- A bucket can be sampled by the `TABLESAMPLE` keyword. The query syntax is `SELECT cols FROM table_name TABLESAMPLE(i OUT OF n)`. Here, the `i` bucket is sampled in a total of `n` buckets. The `TABLESAMPLE` directive can also be used to sample at a block level. Giving a percentage value for sampling in the `TABLESAMPLE` directive does this. It must be noted that the granularity of sampling is a block in this case. `TABLESAMPLE` can be used to sample data based on the input split size and number of rows as well.
- Hive provides a few virtual columns that can be used in specialized queries. `INPUT__FILE__NAME` gives the Map tasks an input filename, and `BLOCK__OFFSET__INSIDE__FILE` gives the global position of the file.
- The `EXPLAIN` command for a query gives the AST, execution DAG with the dependencies, and description of each stage within the DAG.

UDF, UDAF, and UDTF

Like in Pig, UDFs are one of the most important extensibility features in Hive. Writing a UDF in Hive is simpler, but the interfaces do not define every override method that is needed to make the UDF complete. This is because UDFs can take any number of parameters, and it is difficult to provide a fixed interface. Hive uses Java reflection under the hood when executing the UDF to figure out the parameter list for the function.

These are the following three kinds of UDFs in Hive:

- **Regular UDFs**: These UDFs take in a single row and produce a single row after application of the custom logic.
- **UDAFs**: These are aggregators that take in multiple rows but output a single row. `SUM` and `COUNT` are examples of in-built UDAFs.
- **UDTFs**: These are generator functions that take in a single row and produce multiple rows as outputs. The `EXPLODE` function is a UDTF.

The following code example shows how a simple UDF is written. Every UDF is extended from the UDF class present in `org.apache.hadoop.hive.ql.exec package`. It has no override methods, but at least one `evaluate` method has to be part of the class. The following UDF takes in a `String` type and returns another by converting the input string to uppercase. Any number of evaluate methods can be written, and the correct one is chosen by Hive during runtime:

```
package MasteringHadoop;

import org.apache.hadoop.hive.ql.exec.UDF;

public class TOUPPER extends UDF{

    public String evaluate(String input){

        return input.toUpperCase();
    }

}
```

The following Hive statements show how to deploy and run this UDF. The JAR file containing the UDF must be registered, and the Hive metastore has to be instructed on the presence of this UDF. After this, the UDF can be used as any other built-in function within Hive:

```
add jar MasteringHadoop-1.0-SNAPSHOT-jar-with-dependencies.jar;
CREATE TEMPORARY FUNCTION MASTERINGHADOOPTOUPPER AS
'MasteringHadoop.TOUPPER';
SELECT MASTERINGHADOOPTOUPPER(name) FROM
MasteringHadoop.worldcities;
```

A UDAF is slightly more complicated to implement. As we saw in the previous two chapters, aggregations can happen either in the Map or Reduce tasks or both. The UDAF has to be ready to handle all these possibilities. The following code shows a UDAF that finds the maximum of a set of `BIGINT` numbers. As with the UDF, the UDAF extends the `UDAF` class. Instead of an evaluate method, the evaluator classes have to be declared in a UDAF. At runtime, Hive reflects on the UDAF extension and calls the methods on the evaluator classes.

Any number of evaluator classes can be present within a UDAF. They all have to extend the `UDAFEvaluator` base class. The only override the `UDAFEvaluator` base class provides is the `init()` method. In the following example, we create a `MaximumBigIntEvaluator` class to compare and select the maximum value of the `BIGINT` value.

The `init()` method is used to initialize the internal state of the evaluator. Other than the `init()` method, there is an `iterator()` method that will be called for each value that needs aggregation. The `iterator()` method has to update the state of the evaluator. Null values are ignored. The `terminatePartial()` method is called by the Hive runtime whenever it wants a partial result. This generally happens when Map-side aggregations are done. The evaluator has to return the aggregation state so far. The `merge()` function is called by Hive when two partial aggregations need to be aggregated. It generally happens on the Reduce side, where partial results from Maps are merged. Finally, the `terminate()` method is called to get the final aggregation result:

```
package MasteringHadoop;

import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;
import org.apache.hadoop.io.LongWritable;


public class BIGINTMAX extends UDAF {

    public static class MaximumBigIntEvaluator
        implements UDAFEvaluator{

        private Long max;
        private boolean empty;


        public MaximumBigIntEvaluator(){
            super();
            init();
        }


        @Override
        public void init() {
            max = (long)0;
            empty = true;
        }


        public boolean iterate(LongWritable value){
            if(value != null){

                long current = value.get();
```

```
                if(empty){
                    max = current;
                    empty = false;

                }
                else{

                    max = Math.max(current, max);
                }


            }
            return true;


        }


        public LongWritable terminatePartial(){
            return empty ? null : new LongWritable(max);

        }

        public LongWritable terminate(){
            return empty ? null : new LongWritable(max);

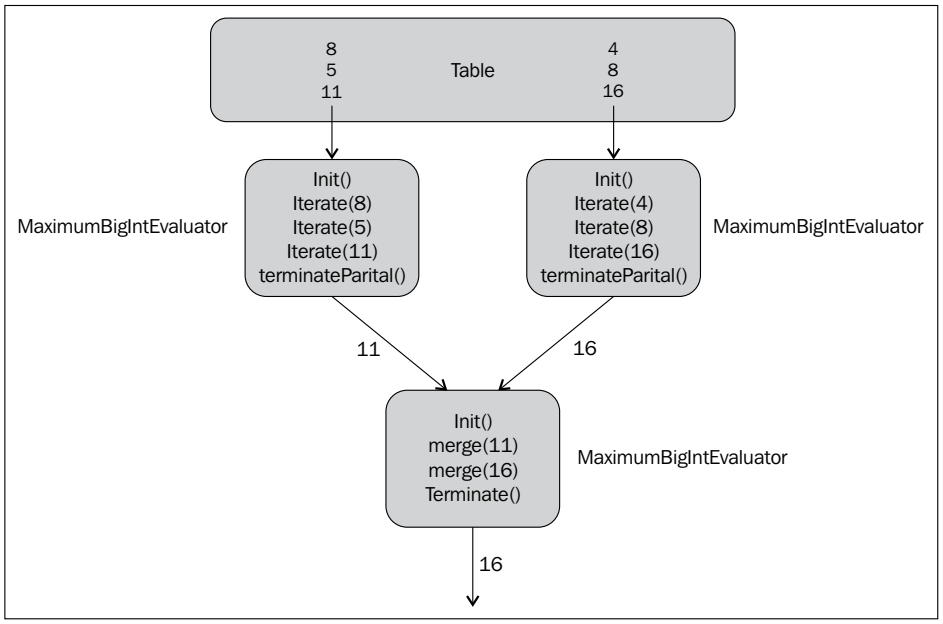
        }

        public boolean merge(LongWritable value){
          iterate(value);
          return true;
        }

    }

}
```

These methods are mandatory for a UDAF to work properly. The following diagram shows the application of the evaluator methods:



Summary

Hive, through its query language HiveQL, brings in SQL and Relational database concepts to Hadoop. The primary use case for Hive is data warehousing and analytical querying for applications such as Business Intelligence. The supporting components of Hive are built to assist this use case. For example, row-columnar file formats are very efficient when performing aggregations on columns.

The key takeaways from this chapter are as follows:

- In Hive, a close look has to be kept on the file format used by the underlying table. Text files can be inefficient. Sequence files are better off as they are compressed. Specialized files such as RC and ORC are more suited both in terms of I/O and query performance.
- Compression brings in efficiency. Both intermediate and final outputs can be compressed. It is better to avoid compression techniques such as GZIP that cannot be split. Snappy is an alternative compression technique that can be split.

- Partitioning large tables is good practice. This helps pre-filter relevant data when processing queries. However, too many partitions put pressure on HDFS.
- Use Map-side joins wherever possible. Order smaller tables to the left-hand side of your join. Settings such as `hive.auto.convert.join` help optimize joins automatically without hints. The `hive.mapjoin.smalltable.filesize` property can be set appropriately to keep the entire small table in memory. However, care has to be taken to see that the JVM has enough memory to fit the table.
- `ORDER` statements should be avoided as they use a single reducer. All the optimizations from *Chapter 2, Advanced MapReduce*, can be applied on the Hadoop cluster running Hive queries. This will have a positive influence on the performance of the query.

In the next chapter, we will look at Hadoop I/O in detail, particularly file compression and serialization/deserialization of data to and from Hadoop-based files.

5
Serialization and Hadoop I/O

Hadoop is about big data, and whenever data is handled, discussion and detailing of IO becomes an integral part of the setup. Data needs to be ingested via the network or loaded from an external persistent media. The ingested data needs to be staged during the extraction and transformation steps. Finally, the results need to be stored for consumption by downstream analysis processes for serving data, reporting, and visualization. Each of these stages involves understanding the underlying data storage structure, data formats, and data models. These aspects help in tuning the entire data-handling pipeline for efficiency of storage and speed.

In this chapter, we will look into the IO features and capabilities of Hadoop. Specifically, we will cover the following topics:

- Serialization and deserialization support and their necessity within Hadoop
- Avro—an external serialization framework
- Data compression codecs available within Hadoop and their tradeoffs
- Special file formats in Hadoop and their features

Data serialization in Hadoop

Though we see data in a structured form, the raw form of data is a sequence or stream of bits. This raw form of data is the one that travels over the network and is stored in RAM or any other persistent media. **Serialization** is the process of converting structured data into its raw form. **Deserialization** is the reverse process of reconstructing structured forms from the data's raw bit stream form.

In Hadoop, different components talk to each other via **Remote Procedure Calls** (**RPCs**). A caller process serializes the desired function name and its arguments as a byte stream before sending it to the called process. The called process deserializes this byte stream, interprets the function type, and executes it using the arguments that were supplied. The results are serialized and sent back to the caller. This workflow naturally calls for fast serialization and deserialization. Network bandwidth is at a premium and requires the serialized representation of the function name and its arguments to have the smallest possible payload. Different processes might evolve differently, and the entire serialization-deserialization process might need to be backward compatible and extensible. Processes running on different machines might have different configurations and leverage different platform components, making interoperability a desired feature of the serialization-deserialization library. These properties of serialization and deserialization are not limited to network data, but extend to storage, both volatile and persistent.

Writable and WritableComparable

Serialization and deserialization in Hadoop is done via the `Writable` interface. This interface has two methods, `void write(DataOutput out)` and `void readFields(DataInput in)`. The `write` method serializes the object into a byte stream. The `readFields` method is the deserialization method that reads off of an input byte stream and converts it into an object.

Inherited from the `Writable` interface is the `WritableComparable` interface. This interface is a combination of the `Writable` and `Comparable` interfaces. Classes implementing this interface not only facilitate serializing and deserializing, but also comparison of values. Having a Hadoop data type implement this interface can come in very handy to sort and group data objects. An example was seen in *Chapter 2*, *Advanced MapReduce*, when the join operator was implemented using a custom `WritableComparable` type.

Out of the box, Hadoop supports a number of `WritableComparable` wrappers. Each `WritableComparable` wrapper wraps a Java primitive type. For example, an `IntWritable` wrapper class wraps an `int` data point, and a `BooleanWritable` wrapper wraps a `boolean` type.



Hadoop supports the `VIntWritable` and `VLongWritable` classes. These are the variable length equivalents of the fixed length `IntWritable` and `LongWritable` types. A value between -112 and 127 is encoded as a single byte using a variable-length number type. However, larger values are encoded in a way that the first byte indicates the sign and the number of bytes that follow. Variable-length `Writable` types save space, on average, when the distribution of the numeric value has a high variance. Lesser values need lesser storage.

In *Chapter 2*, *Advanced MapReduce*, we implemented a custom `WritableComparable` class for the Reduce-side join operation. The `CompositeJoinKeyWritable` class was a composite key of the country code of the data source. Other than the `write` and `readFields` overrides, the `compareTo` function was overridden to provide a comparison of these custom types.

As we saw in the `CompositeJoinKeyWritable` class, under the hood, `Writable` types serialize their payload in a specific way. Let's take the `IntWritable`, `LongWritable`, `VIntWritable`, and `VLongWritable` classes as examples and see the raw bytes that their values are serialized to.

The following method takes a `Writable` type and serializes it as a stream of bytes. It uses the `write` method to write the payload of the `Writable` type into a byte stream. The byte stream is converted into a hexadecimal string for display on the console. The `org.apache.hadoop.util.StringUtils` utility class has some static functions that help us convert a byte array to a hexadecimal string:

```
public static String serializeToByteString(Writable writable)
  throws IOException {

    ByteArrayOutputStream outputStream = new
        ByteArrayOutputStream();
    DataOutputStream dataOutputStream = new
        DataOutputStream(outputStream);
    writable.write(dataOutputStream);
    dataOutputStream.close();
    byte[] byteArray = outputStream.toByteArray();
    return StringUtils.byteToHexString(byteArray);
    }
```

The following code instantiates each of the four classes we consider in this example to study Hadoop serialization. We will take three numbers (100 representing a small integer, 1048576 a normal integer, and 4589938592L a long integer) as the payload for these objects:

```
public static void main(String[] args) throws IOException{

    IntWritable intWritable = new IntWritable();
    VIntWritable vIntWritable = new VIntWritable();
    LongWritable longWritable = new LongWritable();
    VLongWritable vLongWritable = new VLongWritable();

    int smallInt = 100;
    int mediumInt = 1048576;
    long bigInt = 4589938592L;

    System.out.println("smallInt serialized value using
        IntWritable");
    intWritable.set(smallInt);
    System.out.println(serializeToByteString(intWritable));

    System.out.println("smallInt serialized value using
        VIntWritable");
    vIntWritable.set(smallInt);
    System.out.println(serializeToByteString(vIntWritable));

    System.out.println("mediumInt serialized value using
        IntWritable");
    intWritable.set(mediumInt);
    System.out.println(serializeToByteString(intWritable));

    System.out.println("mediumInt serialized value using
        VIntWritable");
    vIntWritable.set(mediumInt);
    System.out.println(serializeToByteString(vIntWritable));

    System.out.println("bigInt serialized value using
        LongWritable");
    longWritable.set(bigInt);
    System.out.println(serializeToByteString(longWritable));

    System.out.println("bigInt serialized value using
        VLongWritable");
    vLongWritable.set(bigInt);
    System.out.println(serializeToByteString(vLongWritable));
}
```

The program uses `IntWritable` and `VIntWritable` as the wrapper classes for the small and medium integers. `LongWritable` and `VLongWritable` are used for the large integer. The output when these numbers are serialized into a byte array is shown as follows:

```
smallInt serialized value using IntWritable
00000064
smallInt serialized value using VIntWritable
64
mediumInt serialized value using IntWritable
00100000
mediumInt serialized value using VIntWritable
8d100000
bigInt serialized value using LongWritable
000000011194e7a0
bigInt serialized value using VLongWritable
8b011194e7a0
```

The `IntWritable` class uses a fixed length of four bytes to represent an integer regardless of the value stored within it. The `VIntWritable` class is smarter, and the number of bytes it uses depends on the value of the payload. For the number 100, `VIntWritable` uses only a single byte. There is a similar difference in the `LongWritable` and `VLongWritable` serialized values too.



`Text` is a `Writable` version of the `String` type. It represents a collection of UTF-8 characters. Hadoop's `Text` class is mutable when compared to Java's `String` class.

Hadoop versus Java serialization

A question that pops up at this point is why Hadoop uses the `Writable` interface and does not rely on Java serialization. Let's try to serialize the values in the previous example using Java data types and serialization. For Java serialization, we will use the following static method:

```
public static String javaSerializeToByteString(Object o) throws
    IOException{
    ByteArrayOutputStream outputStream = new
        ByteArrayOutputStream();
    ObjectOutputStream objectOutputStream = new
        ObjectOutputStream(outputStream);
    objectOutputStream.writeObject(o);
```

```
        objectOutputStream.close();

        byte[] byteArray = outputStream.toByteArray();
        return StringUtils.byteToHexString(byteArray);

        }
```

Java provides the `ObjectOutputStream` class to serialize an object into a byte stream. The `ObjectOutputStream` class supports a `writeObject` method. The three numbers are serialized using the following code:

```
System.out.println("smallInt serialized value using Java
    serializer");
System.out.println(javaSerializeToByteString(new
    Integer(smallInt)));

System.out.println("mediumInt serialized value using Java
    serializer");
System.out.println(javaSerializeToByteString(new
    Integer(mediumInt)));

System.out.println("bigInt serialized value using Java
    serializer");
System.out.println(javaSerializeToByteString(new Long(bigInt)));
```

The output is as follows:

```
smallInt serialized value using Java serializer
aced0005737200116a6176612e6c616e672e496e746567657212e2a0a4f7818738
    02000149000576616c7565787200106a6176612e6c616e672e4e756d62657
        286ac951d0b94e08b020000787000000064
mediumInt serialized value using Java serializer
aced0005737200116a6176612e6c616e672e496e746567657212e2a0a4f7818738
    02000149000576616c7565787200106a6176612e6c616e672e4e756d6265
        7286ac951d0b94e08b020000787000100000
bigInt serialized value using Java serializer
aced00057372000e6a6176612e6c616e672e4c6f6e673b8be490cc8f23df020001
    4a000576616c7565787200106a6176612e6c616e672e4e756d62657286ac9
        51d0b94e08b02000078700000000011194e7a0
```

Very clearly, the serialized value is way bigger than the serialized values of a `Writable` class. Hadoop is all about serializing and deserializing either on disk or on the wire, and compactness is highly valued. Java serialization takes way more bytes to represent an object.

Java serialization's inefficiency stems from the fact that Java does not make any assumption about the class of the serialized value. This entails tagging every serialized value with class-related metadata. `Writable` classes, on the other hand, read the fields from the byte stream and assume that the byte stream is of its type. This leads to higher performance due to the compactness of the representation. The cost is a steeper learning curve for the Hadoop newbie. Another downside is that `Writable` classes are locked into the Java programming language.

Creating custom `Writable` classes is tedious as the developer has to take care of the class format on the wire. Briefly, **Record IO** was introduced within Hadoop. This feature came with a record definition language and a compiler that could translate record specifications to `Writable` classes. Eventually, this has been deprecated, and Avro has been suggested as the alternative.



Before Hadoop 0.17, any MapReduce program had to use Writable classes for Map and Reduce task keys and values. However, post this release, any serialization framework can be integrated with MapReduce jobs in Hadoop. This has led to the usage of a number of alternate serialization frameworks. Each framework brought in performance gains either in terms of representation compactness or speed of serialization and deserialization, or both.

Avro serialization

Avro is a popular data serialization framework that is part of Apache Software Foundation. Its key features are as follows:

- It supports a number of data structures for serialization.
- It is neutral to particular programming languages and provides fast and compact binary serialization.
- Code generation is optional in Avro. Data can be read, written, or used in RPCs without having to generate classes or code.

Avro uses **schemas** during the reading and writing of data. Schemas make the compact representation of the serialized object conducive. The self-describing capability of schemas makes it possible to get rid of object-type metadata to be present along with the serialized byte stream, the method used in Java serialization. The schemas are described in the **Javascript Object Notation** (**JSON**) format that has evolved as a popular object description notation on the Web. Schema changes can be handled by having both the old and new schema available when processing data.

The following are two schema files used in Avro. The first file is the schema of the `worldcitiespop.txt` file, and the second file is the schema of the `countrycodes.txt` file:

```
{"namespace": "MasteringHadoop.avro",
 "type": "record",
 "name": "City",
 "fields": [
     {"name": "countryCode", "type": "string"},
     {"name": "cityName",  "type": "string"},
     {"name": "cityFullName", "type": "string"},
     {"name": "regionCode", "type": ["int","null"]},
     {"name": "population", "type": ["long", "null"]},
     {"name": "latitude", "type": ["float", "null"]},
     {"name": "longitude", "type": ["float", "null"]}
 ]
}

{"namespace": "MasteringHadoop.avro",
 "type": "record",
 "name": "Country",
 "fields": [
     {"name": "countryCode", "type": "string"},
     {"name": "countryName",  "type": "string"}
 ]
}
```

Schema files are self-explanatory and the JSON notation makes them readable. Avro supports all the standard primitive data types. In addition, Avro also supports complex data types such as unions. Null value fields are unions of the null and field types. Unions are syntactically represented as JSON arrays.

Let's take the `worldcitiespop.txt` file, a file in CSV text format, and convert it into an Avro file using the `City` schema specified previously. The following code gives the important steps to write Avro files. Most of the conversion happens in the static method, `CsvToAvro`. This method takes in `csvFilePath`, the output of `avroFilePath`, and the path to `schemaFile`. There is a special `Schema` class in Avro, and parsing the schema file initializes an object of this class. The schema is not code generated, so we use the `GenericRecord` class to initialize the schema and write the data points. If the schema is used to generate code, the result will be a `City` class that can be imported directly in the following code, like any other Java class.

The `DataFileWriter` class is used to write the actual records into the file. It has a `create` method that creates the output Avro file. Using a `BufferedReader` object, we read each city record from the CSV file one line at a time. The `getCity` helper method takes the line, splits it into tokens separated by a comma, and generates a `GenericRecord` object. The `GenericData.Record` class is used to instantiate an Avro record. This class constructor takes in a `Schema` object.

Writing to a `GenericRecord` object requires a `put` method that takes in the name of the record field and the corresponding value. The `isNumeric` method is used to validate the tokenized `String` to see whether it is a number or not. Bad records are skipped and not written into the Avro file. If `put` is not used on a field, this particular field is assumed to be `null`:

```
public static void CsvToAvro(String csvFilePath, String
    avroFilePath, String schemaFile) throws IOException{

        //Read the schema
        Schema schema  = (new Schema.Parser()).parse(new
            File(schemaFile));
        File avroFile = new File(avroFilePath);

DatumWriter<GenericRecord>datumWriter = new
    GenericDatumWriter<>(schema);
DataFileWriter<GenericRecord>dataFileWriter = new
    DataFileWriter<>(datumWriter);
dataFileWriter.create(schema,avroFile);


BufferedReader bufferedReader = new BufferedReader(new
    FileReader(csvFilePath));
        String commaSeparatedLine;
while((commaSeparatedLine = bufferedReader.readLine()) != null){

GenericRecord city = getCity(commaSeparatedLine, schema);

dataFileWriter.append(city);
        }

dataFileWriter.close();

    }


private static GenericRecord getCity(String commaSeparatedLine,
    Schema schema){

GenericRecord city = null;
```

```
String[] tokens = commaSeparatedLine.split(",");

        //Filter out the bad tokens
if(tokens.length == 7){
city = new GenericData.Record(schema);
city.put("countryCode", tokens[0]);
city.put("cityName", tokens[1]);
city.put("cityFullName", tokens[2]);

if(tokens[3] != null && tokens[3].length() > 0 &&isNumeric(tokens[3]))
{
city.put("regionCode", Integer.parseInt(tokens[3]));
             }


if(tokens[4] != null && tokens[4].length() > 0
    &&isNumeric(tokens[4])){
city.put("population", Long.parseLong(tokens[4]));
             }


if(tokens[5] != null && tokens[5].length() > 0
    &&isNumeric(tokens[5])){
city.put("latitude", Float.parseFloat(tokens[5]));
             }

if(tokens[6] != null && tokens[6].length() > 0
    &&isNumeric(tokens[6])){
city.put("longitude", Float.parseFloat(tokens[6]));
             }
         }

return city;

         }


public static boolean isNumeric(String str)
    {
try
        {
double d = Double.parseDouble(str);
        }
catch(NumberFormatException nfe)
        {
return false;
        }
return true;
    }
```

Avro and MapReduce

There is extensive support for Avro serialization and deserialization in a Hadoop MapReduce job. In Hadoop 1.X, there were the `AvroMapper` and `AvroReducer` specialized classes that needed to be used. However, in Hadoop 2.X, the built-in `Mapper` and `Reducer` classes can be reused. `AvroKey` can be used as the input or output types to both the `Mapper` and `Reducer` classes.

There is a special `InputFormat` class called `AvroKeyInputFormat`, which can be used to read `AvroKey` from the input files. The following code finds the population of each country using the `worldcitiespop.avro` file that was generated using the previous program. The `Mapper` code is given in the following code. We pass the schema as a string using a side channel. In the following code, it is passed through the `Configuration` object by setting a key on it. `DistributedCache` can also be used to pass the schema file around. The `setup` method is overridden to read the schema in the Map task.

The `map` method reads the `GenericRecord` datum object based on the schema passed to it:

```
package MasteringHadoop;

import org.apache.avro.Schema;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.mapred.AvroKey;
import org.apache.avro.mapreduce.AvroJob;
import org.apache.avro.mapreduce.AvroKeyInputFormat;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

import java.io.File;
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;

public class MasteringHadoopAvroMapReduce {

private static String citySchema = "{\"namespace\":
    \"MasteringHadoop.avro\",\n" +
```

```
        " \"type\": \"record\",\n" +
        " \"name\": \"City\",\n" +
        " \"fields\": [\n" +
                "     {\"name\": \"countryCode\", \"type\":
            \"string\"},\n" +
                "     {\"name\": \"cityName\",  \"type\":
            \"string\"},\n" +
                "     {\"name\": \"cityFullName\", \"type\":
            \"string\"},\n" +
                "     {\"name\": \"regionCode\", \"type\":
             [\"int\",\"null\"]},\n" +
                "     {\"name\": \"population\", \"type\": [\"long\",
            \"null\"]},\n" +
                "     {\"name\": \"latitude\", \"type\": [\"float\",
            \"null\"]},\n" +
                "     {\"name\": \"longitude\", \"type\": [\"float\",
            \"null\"]}\n" +
        " ]\n" +
                "}";


    public static class MasteringHadoopAvroMapper extends
        Mapper<AvroKey<GenericRecord>, NullWritable, Text,
            LongWritable>{

    private Text ccode = new Text();
    private LongWritable population = new LongWritable();
    private String inputSchema;

            @Override
    protected void setup(Context context) throws IOException,
        InterruptedException {
    inputSchema = context.getConfiguration().get("citySchema");
             }

            @Override
    protected void map(AvroKey<GenericRecord> key, NullWritable value,
        Context context) throws IOException, InterruptedException {

    GenericRecord record = key.datum();
                String countryCode = (String)
                record.get("countryCode");
                Long cityPopulation = (Long) record.get("population");

    if(cityPopulation != null){
```

```
ccode.set(countryCode);
population.set(cityPopulation.longValue());
context.write(ccode, population);

            }

        }
    }
```

The following `Reducer` code reduces on the country code and sums up the population. The `main` function sets up the `Job` configuration. There is a specialized `AvroJob` class that can be used to specify Avro-specific properties on the `Job` configuration:

```
public static class MasteringHadoopAvroReducer extends
    Reducer<Text, LongWritable, Text, LongWritable>{

private LongWritable total = new LongWritable();


        @Override
protected void reduce(Text key, Iterable<LongWritable> values,
    Context context) throws IOException, InterruptedException {
long totalPopulation = 0;

for(LongWritable pop : values){
totalPopulation += pop.get();
            }

total.set(totalPopulation);
context.write(key, total);
        }
    }




public static void main(String args[]) throws IOException,
    InterruptedException, ClassNotFoundException,
        URISyntaxException{

GenericOptionsParser parser = new GenericOptionsParser(args);
        Configuration config = parser.getConfiguration();
```

```
String[] remainingArgs = parser.getRemainingArgs();

config.set("citySchema", citySchema);

        Job job = Job.getInstance(config, "MasteringHadoop-
            AvroMapReduce");

job.setMapOutputKeyClass(AvroKey.class);
job.setMapOutputValueClass(Text.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);

job.addCacheFile(new URI(remainingArgs[2]));

job.setMapperClass(MasteringHadoopAvroMapper.class);
job.setReducerClass(MasteringHadoopAvroReducer.class);
job.setNumReduceTasks(1);

        Schema schema  = (new Schema.Parser()).parse(new
            File(remainingArgs[2]));
AvroJob.setInputKeySchema(job, schema);

job.setInputFormatClass(AvroKeyInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);

AvroKeyInputFormat.addInputPath(job, new Path(remainingArgs[0]));
TextOutputFormat.setOutputPath(job, new Path(remainingArgs[1]));

job.waitForCompletion(true);

    }
}
```

Avro and Pig

Pig has been extended to support Avro. `AvroStorage` implements both the `LoadFunc` and `StoreFunc` interfaces to support loading from and writing to Avro files. However, Pig's Avro integration has some limitations and assumptions, which are as follows:

- Nested record types are not supported in `AvroStorage`.
- Union support is only for nulls.
- It is assumed that all files in a directory and subdirectories have the same schema.

- When `AvroStorage` is used to store data in the Avro format, all fields will be null-valued unions. This is because there are no non-null-valued fields in Pig.
- `TUPLE` wrappers might be present in the Avro file when `STORE` is called on a Pig relation.
- JSON-encoded Avro files are not supported.
- `AvroStorage` does not implement map data types.
- The column-pruning optimization is not present when `AvroStorage` is used.

The following example shows the loading of the `countrycodes.avro` file into a Pig relation. It is important to register a number of JAR files for `AvroStorage` to work well in Pig:

```
REGISTER avro-1.4.0.jar
REGISTER json-simple-1.1.jar
REGISTER piggybank.jar
avroCountry = LOAD 'countrycodes.avro' USING
AvroStorage('{"namespace": "MasteringHadoop.avro",
 "type": "record",
 "name": "Country",
 "fields": [
     {"name": "countryCode", "type": "string"},
     {"name": "countryName",  "type": "string"}
 ]
}');
```

Avro and Hive

Hive has a SerDe module called `AvroSerde`, which can read and write Hive tables using Avro. It automatically infers the schema of the Hive table from the Avro input. For most Avro types, there are corresponding Hive table types. If some Avro types do not exist in Hive, they are automatically converted to a type that is available in Hive.



Avro has the concept of enums, while Hive does not. All enum types in Avro are converted to a string type in Hive.

Let's build a Hive table on the external Avro file with the `country.avschema` schema. The Hive DDL statement is as follows:

```
CREATE EXTERNAL TABLE avrocountry
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS
```

```
INPUTFORMAT
'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
LOCATION '/user/sandeepkaranth/avrocountrydata'

TBLPROPERTIES ( 'avro.schema.literal'='
{"namespace": "MasteringHadoop.avro",
 "type": "record",
 "name": "Country",
 "fields": [
     {"name": "countryCode", "type": "string"},
     {"name": "countryName",  "type": "string"}
 ]
}')
;
```

The key pieces of the DDL statement are as follows:

- The usage of `AvroContainerInputFormat` as `InputFormat` of the table.
- The usage of `AvroContainerOutputFormat` as `OutputFormat` of the table.
- The specification of the schema in `TBLPROPERTIES`. The schema can be specified in a file either by a link to the schema file or literally, as shown in the DDL statement. If specified by a link or URL, the property name changes to `avro.schema.url` instead of `avro.schema.literal`.

The describe of the `avrocountry` table shows the interpreted Hive table schema from the Avro schema, as shown:

```
hive> describe avrocountry;
OK
countrycode      string                    from deserializer
countryname      string                    from deserializer
Time taken: 0.155 seconds, Fetched: 2 row(s)
```



When writing tables, all null-valued columns should be specified as a union of the column type and null in the Avro-schema definition.

Comparison – Avro versus Protocol Buffers / Thrift

Avro has its share of competing serialization/deserialization libraries. The popular libraries among them are **Thrift** and **Protocol Buffers**. Avro differs from these frameworks in the following ways:

- Avro supports dynamic typing and can support static typing if performance is the need of the hour. Protocol Buffers and Thrift have **Interface Definition Languages** (**IDLs**) to specify schemas and their types. These IDLs are used to generate code for serialization and deserialization. Using IDLs brings down the flexibility in building generic data-processing pipelines.
- Avro is built into Hadoop, while the rest are not. The Hadoop ecosystem components also support Avro, as we saw in the case of Hive and Pig.
- Avro's schema definition is in JSON and not in any proprietary IDL. This makes it popular among developers as JSON has evolved as the object notation for the Web. It also makes Avro a language neutral.

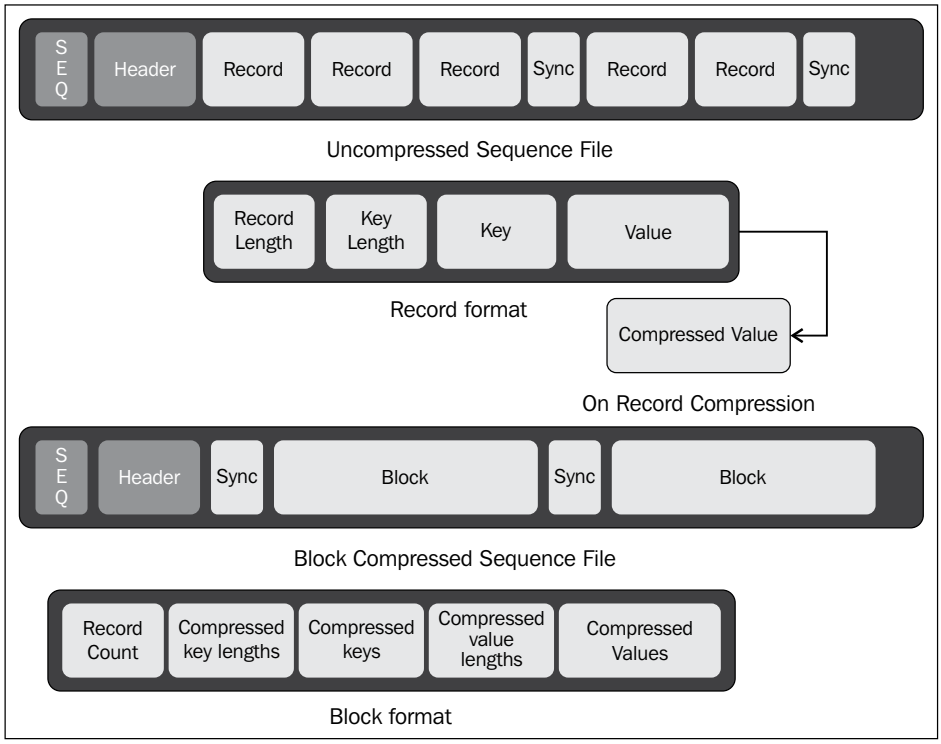
File formats

There are a number of file formats that are data structures by themselves. In the chapter on Hive, we saw ORC files, an optimized form of record columnar file storage. There are a few other popular file container formats supported by Hadoop. We will look at them in this section.

The Sequence file format

A **Sequence** file is a container format for binary key-value pairs. Each record in a Sequence file contains a key and its associated value. Sequence files are used to combine smaller files into a single large file to alleviate the small file problem in Hadoop. In this situation, the filename forms the key and the file contents are the values associated with the key. Sequence files have a broader applicability as they can be split into configurable blocks. They can be combined with fast compression methods such as LZO or Snappy and can provide speed as well as storage and bandwidth.

The next image shows the format of a Sequence file. The start of the file has a magic number, a binary representation of the letters SEQ. A version byte and header follow it. The header stores the metadata of the file, such as the key and value class names, as a string. If the key or value is of the `Text` class, a `org.apache.hadoop.io.Text` string is embedded in the header. The class names are followed by Booleans that indicate whether compression is enabled, and block compression is enabled in this order. The compression codec class name to be used is then specified, followed by user-related metadata as key-value pairs. The header ends with a sync marker to denote the end of the header. These sync markers are allowed to get a record boundary in the file. They are randomly generated markers. The overhead due to the sync markers is kept below 1 percent of the total file size. This means the markers appear at the end of a group of records.



Each record contains record-related metadata, such as the record and key lengths. The actual key and value bytes follow the metadata. The lengths are 4-byte integer values serialized using the `IntWritable` class. When record compression is enabled, the value bytes are compressed using the codec specified in the header. There is no change to the record structure.



The keys are not compressed when compression is enabled.

During block compression, records are grouped into blocks. The minimum block size is determined by the property `io.seqfile.compress.blocksize` parameter. A sync marker is written at the beginning of each block. The sync marker is 16-bytes long, and is generated by taking a hash of the `(UID() + '@' + time or internet address)` expression. Block compression compresses the keys too. Blocks use a `VIntWritable` serialization to store the counts, key lengths, and keys.

Reading and writing Sequence files

The following code illustrates reading and writing using the `SequenceFile` format. The `writeSequenceFile` method takes a path to the file to be converted and the output file path. The `SequenceFile` class has a `createWriter` static method to create a writer handle. The `append` method on the writer takes in a key and value and appends it to the file. The following code takes in a CSV file and writes a line number as the key and line string as the value:

```
package MasteringHadoop

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.util.ReflectionUtils;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URI;


public class MasteringHadoopSequenceFile {

public static void writeSequenceFile(String textFile, String
    seqFile) throws IOException{

        Path readPath = new Path(textFile);
        Path writePath = new Path(seqFile);
        Configuration conf = new Configuration(false);
```

```
FileSystem fs = FileSystem.get(URI.create(textFile), conf);
BufferedReaderbufferedReader = null;
SequenceFile.WritersequenceFileWriter = null;

try{

    bufferedReader = new BufferedReader
                        (newInputStreamReader
                                (fs.open(readPath)));

    sequenceFileWriter = SequenceFile.createWriter(conf,
    SequenceFile.Writer.file(writePath),
    SequenceFile.Writer.keyClass(LongWritable.class),
    SequenceFile.Writer.valueClass(Text.class));
            String line = null;
    LongWritable key = new LongWritable();
            Text value = new Text();
    long lineCount = 0;

    while((line = bufferedReader.readLine()) != null){
        key.set(lineCount);
        lineCount++;
        value.set(line);
        sequenceFileWriter.append(key, value);

            }


        }
catch(IOException ioEx){
ioEx.printStackTrace();
        }
finally{
if(sequenceFileWriter != null)
sequenceFileWriter.close();

if(bufferedReader != null)
bufferedReader.close();
        }
   }
```

The following function reads the Sequence file using `SequenceFile.Reader`. From the Sequence file headers, we can infer the type of the key and value. `ReflectionUtils` has some utility methods to create objects based on these types. The `syncSeen` method gives an indication about the sync markers within the file as the file is being read:

```
public static void readSequenceFile(String seqFile) throws
    IOException{

       Path readPath = new Path(seqFile);
       Configuration conf = new Configuration(false);
FileSystem fs = FileSystem.get(URI.create(seqFile), conf);

SequenceFile.Reader reader = null;

try{
reader = new SequenceFile.Reader(conf,
    SequenceFile.Reader.file(readPath));
         Writable key =
     (Writable)ReflectionUtils.newInstance(reader.getKeyClass(),
        conf);
         Writable value =
     (Writable)ReflectionUtils.newInstance(reader.getValueClass(),
        conf);

while(reader.next(key,value)){
System.out.println("key: " + key.toString());
if(reader.syncSeen()){
System.out.println("sync: ");
             }

          }
       }
catch(IOException ioEx){
ioEx.printStackTrace();
       }
finally{
if(reader !=  null){
reader.close();
            }
       }

   }


public static void main(String[] args){

try{
```

```
    writeSequenceFile(args[0], args[1]);
    readSequenceFile(args[1]);
           }
    catch(IOException ioEx){
    ioEx.printStackTrace();
           }

        }

    }
```

A Sequence file can also be read using the following Hadoop command:

```
hadoop fs -text /user/sandeepkaranth/countrycodes.seq
```

The MapFile format

`MapFile` is the same as `SequenceFile` in structure. Additionally, it provides an index for the keys in the file. `MapFile` keys have to be of the `WritableComparable` type and values have to be of the `Writable` type. In `SequenceFile`, any serialization framework can be used to serialize the keys and values.

When `MapFile` is created, it has two associated files, one for the data and another for the index. Both these files are of the `SequenceFile` type. The data `SequenceFile` contains all the data records sorted by the key. The index `SequenceFile` contains the key and the file offset where the key is present. The keys in the index file are sampled. Not every key occurs in the index file. The interval of the sample is given by the value in the `io.map.index.interval` property. The following example illustrates the data and index files in the `countrycodes.map` file:

```
hadoop fs -ls countrycodes.map/
Found 2 items
-rw-r--r--   3 sandeepkaranth supergroup      10033 2014-06-08 14:20
countrycodes.map/data
-rw-r--r--   3 sandeepkaranth supergroup        166 2014-06-08 14:35
countrycodes.map/index


hadoop fs -text countrycodes.map/index
127    5088


hadoop fs –text countrycodes.map/data
241  vi,Virgin Islands (USA)
242  vn,Vietnam
```

```
243  vu,Vanuatu
244  wf,Wallis and Futuna Islands
245  ws,Samoa
246  ye,Yemen
247  yt,Mayotte
248  yu,Yugoslavia
249  za,South Africa
250  zm,Zambia
251  zr,Zaire
252  zw,Zimbabwe
```

The `MapFile` format can be useful to process Map-side joins. The sorted nature of the data and index files can be used to force splits of the datasets participating in a join into a single Map task. The APIs to create files in the `MapFile` format are similar to the APIs for `SequenceFile` creation.

The following code shows how `SequenceFile` can be converted into `MapFile`. The `MapFile.fix()` static method shown in the following code is used to achieve this:

```
    public static void writeMapFile(String seqFile) throws
IOException {

        Path readPath = new Path(seqFile);
        Path mapPath = new Path(readPath, MapFile.DATA_FILE_NAME);

        Configuration conf = new Configuration(false);
        FileSystem fs = FileSystem.get(URI.create(seqFile), conf);

        SequenceFile.Reader reader = null;

        try{
            reader = new SequenceFile.Reader(conf,
                SequenceFile.Reader.file(mapPath));
            Class keyClass = reader.getKeyClass();
            Class valueClass = reader.getValueClass();

            MapFile.fix(fs, readPath, keyClass, valueClass, false,
                conf);

        }
        catch(IOException ioEx){
            ioEx.printStackTrace();
        }
```

```
        catch(Exception ex){
            ex.printStackTrace();
        }
        finally{
            if(reader !=  null){
                reader.close();
            }
        }

    }
```

Other data structures

Hadoop also supports other persistent data structures that are variants of `MapFile`. Some of them are as follows:

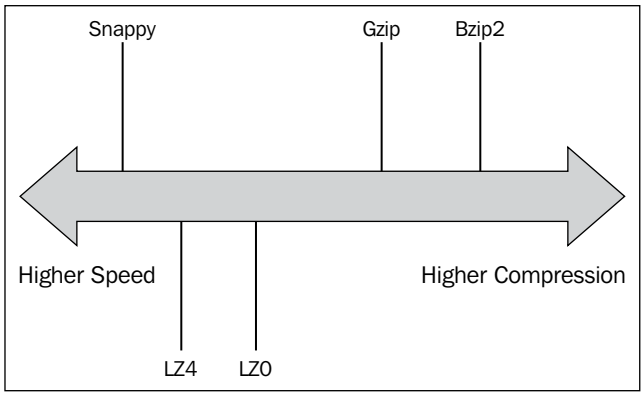
- `SetFile`: This file format stores a set of keys and allows set operations on the keys. The key difference in the `SetFile` API when compared to the `MapFile` API is that the `append` method of `SetFileWriter` takes in only a key and no value. Under the hood, the value is `NullWritable`. The file structure remains the same, as in, it has both the index and the data `SequenceFile` methods.
- `ArrayFile`: This particular file format can be thought of as the complement of `SetFile`. It stores only values and no keys. Like an array, the key for a particular value is `LongWritable`, which contains a record number. The API method, `append`, takes in only a value.
- `BloomMapFile`: This file format is a variant of `MapFile`. In addition to the index and data files, it has a bloom file. This bloom file is a file that encodes a Dynamic Bloom filter. For large files in the key-value format with sparse keys, the lookup of the key on the index might not be fast enough. A Bloom filter is a probabilistic data structure that encodes the presence of keys in a few bits and can provide quick answers to the `MapFileget()` method.

Compression

A recurring theme that appears in this book is the need to save storage and network data transfer. When dealing with large volumes of data, anything that reduces these two properties gives an efficiency boost both in terms of speed and cost. Compression is one such strategy that can help make a Hadoop-based system efficient.

All compression techniques are a tradeoff between speed and space. The higher the space savings, the slower the compression technique, and vice versa. Each compression technique is also tunable for this tradeoff. For example, the gzip compression tool has options -1 to -9, where -1 optimizes for speed and -9 for space.

The following figure shows the different compression algorithms in the speed-space spectrum. The gzip tool does a good job of balancing out both storage and speed. Techniques such as LZO, LZ4, and Snappy are very fast, but their compression ratio is not very good. Bzip2 is a slower technique, but has the best compression.



Codecs are concrete implementations of these compression techniques. All compression codecs in Hadoop have to implement the `CompressionCodec` interface in their implementing classes. Codecs are found in the `org.apache.hadoop.io.compress` package. There is a default codec in Hadoop, the compression of which is based on the `DEFLATE` algorithm.



The `DEFLATE` compression is similar to gzip, but does not contain additional headers and footers.

Splits and compressions

Map tasks act on each split of data, generally a file block stored in HDFS. However, the majority of the compression algorithms do not allow you to read the file at arbitrary points. Though implementations such as gzip are block-based compression techniques, these blocks are no way related to or aware of HDFS blocks. In these situations, Hadoop does not try to split the file, and it provides the entire file to a single Map task. This can turn unwieldy in many situations.

For some compression formats such as LZO, there are indexing tools that can process the LZO file and build an index of the compressed blocks. This index can be used by an appropriate `InputFormat` method to determine the number of splits and their offsets. For example, `LzoTextInputFormat` has the capability to read a LZO-based file index and decide the Map task input splits.

However, compression techniques such as bzip2 support splitting natively. These split points are indicated by means of synchronization markers. Hadoop recognizes the different compression formats of files using the extension of the file.

There are a number of strategies that can be used when compression is enabled:

- The application can split the files as a preprocessing step and use well-known compression techniques such as gzip on each file split. These compressed file chunks can be stored in HDFS. For optimality, the post-compressed size of a file chunk has to be nearly equal to the HDFS block size. In this case, it does not matter whether the compression algorithm is splittable or not.
- Splittable compressions such as bzip2 can be applied on the file. However, this technique is the slowest of the supported compression codecs. An alternative is to use LZO, and then build an index on top of it.
- There are a number of file formats, such as `SequenceFile`, `MapFile`, and `RCFile`. These file formats support splitting natively. They can be compressed as well, as we saw in the File Formats section of this chapter.
- The preferred method in the industry is to store data in specialized file formats. They provide a balanced view between speed and compression.

Scope for compression

In *Chapter 2*, *Advanced MapReduce*, we saw a number of places in the MapReduce pipeline where compression increases job speed and reduces storage needs. We will summarize it in this subsection:

- All compressed inputs are decompressed and processed within the Map task. The codec to be used is determined by the file extension. In some cases, such as LZO compression with indexing, appropriate `InputFormat` classes should be used to supply the appropriate splits.
- The `mapreduce.map.output.compress` property can be set to `true` to enable compression of intermediate outputs. The codec to be used for compression can be set using the `mapreduce.map.output.compress.code` property. The default is `org.apache.hadoop.compress.DefaultCodec`.

- The compression of job outputs can be enabled by setting the `mapreduce.output.fileoutputformat.compress` property to `true`. The codec can be specified by setting `mapreduce.output.fileoutputformat.compress.codec`. For `SequenceFile` outputs, there is a special `mapreduce.output.fileoutputformat.compress.type` property, which determines the granularity at which the compression should happen. The default value for this is `RECORD`, indicating that each record will be compressed. Record groupings can also be compressed by setting this value to `BLOCK`.

Summary

Big data processing involves data representation either in storage or in transit over the network. Compact representation, fast transformations, extensibility, and backward compatibility of the data representation are desired properties. Some key takeaways from this chapter related to data representation are as follows:

- Hadoop provides inbuilt serialization/deserialization mechanisms using the `Writable` interface. The `Writable` classes are serialized more compactly than Java serialization.
- Avro is a flexible and extensible data serialization framework. It serializes data in binary and is supported by Hadoop, MapReduce, Pig, and Hive.
- Avro provides dynamic typing, eliminating the need for code generation. The schema can be stored with the data and read by any subsystem.
- Compression techniques trade speed and storage savings. Hadoop supports many compression codecs along this tradeoff spectrum. Compression is a very important optimization parameter for big data processing.
- Hadoop supports specialized container file formats such as `SequenceFile` and `MapFile`. These formats support splitting and compression. Hadoop also supports persistence of specialized data structures such as `ArrayFile`, `SetFile`, and `BloomMapFile`.

In the next chapter, we will look at YARN, the heart of resource management in Hadoop 2.X, and how it generalizes the Hadoop platform.

6
YARN – Bringing Other Paradigms to Hadoop

Yet Another Resource Negotiator (**YARN**) is a cluster resource management layer that was introduced in Hadoop 2.0. As we saw briefly in *Chapter 1*, *Hadoop 2.X*, YARN separates out the responsibilities of the JobTracker daemon. JobTracker was responsible for:

- Resource arbitration within a Hadoop cluster
- MapReduce job management

The problem with the JobTracker model was that it became the single point of failure in the compute layer of a Hadoop cluster. Any failure in JobTracker meant trashing the running jobs and starting all over again. JobTracker's singular nature also became a scaling bottleneck. All job communications, scheduling, and resource management were controlled by the JobTracker master daemon.

The tightly coupled functions of JobTracker made it rigid, allowing a single computing paradigm, MapReduce, to be onboarded onto the cluster. MapReduce is not suitable for a variety of emerging applications and force-fitting solutions to all problems using this paradigm is not prudent.

YARN takes care of cluster resource management and application scheduling. It is agnostic to the kind of application that is executing or any of its internals. Resource negotiation happens strictly through protocols. MapReduce becomes an application in YARN. Like MapReduce, other applications can be written to run on a cluster that can request for CPU, memory, and other resources from YARN via defined protocols and execute.

In this chapter, we will be:

- Delving into the architecture of YARN
- Building a simple YARN non-MapReduce application and looking at:
    - The modules that constitute a YARN application
    - The core steps to build out each module
    - The protocols used to communicate with YARN components
- Discussing YARN scheduling variants
- Glancing at YARN commands

The YARN architecture

The following figure illustrates the architecture of a YARN-based cluster. There are five major component types in a YARN cluster. They are as follows:

- **Resource Manager** (**RM**): This is a per-cluster daemon that is solely responsible for allocating and managing resources available within the cluster.
- **Node Manager** (**NM**): This is a per-node daemon that is responsible for local resource management. It is also the node-local representative of the RM.
- **Application Master** (**AM**): This is a per-application daemon that encapsulates all application-specific logic and libraries. The AM is responsible for negotiating resources from the RM and working with the NM to execute them to completion.
- **Container**: This is an abstract representation of a resource set that is given to a particular application. The AM is a specialized container that is used to bootstrap and manage the entire application's life cycle.
- **Client**: This is the entity in the cluster that can submit applications to the RM and specify the type of AM that needs to be spawned to execute the application.

Resource Manager (RM)

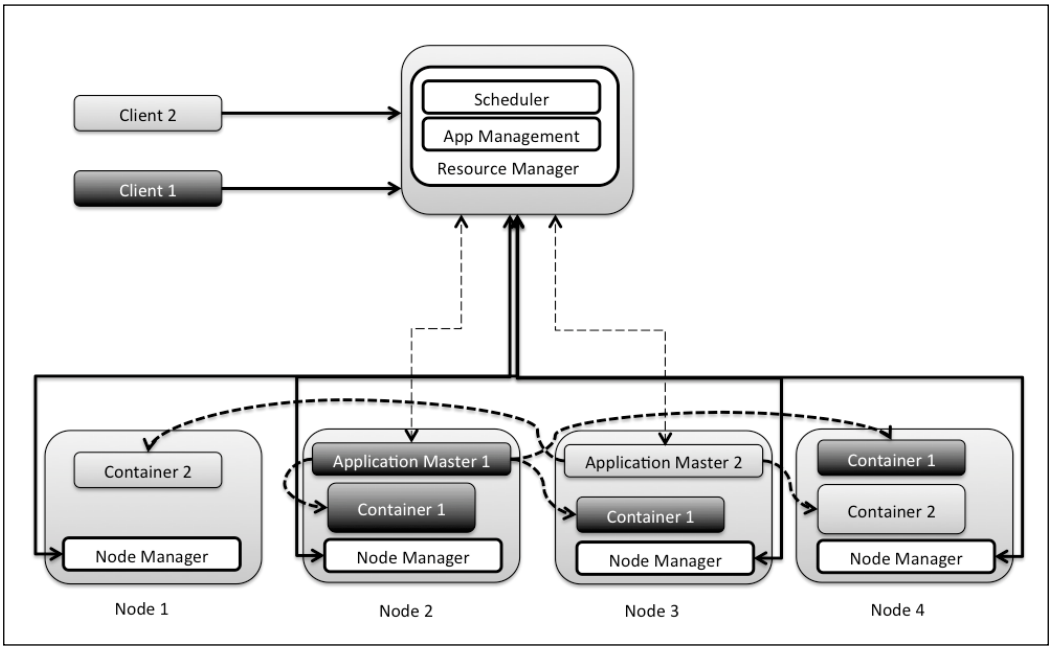
The Resource Manager has the following two main components:

- **Scheduler**
- **ApplicationsManager**

The Scheduler is responsible for allocating resources to the various applications that are running in the cluster. It is a pure entity and does not have any insight into the status of the application. The Scheduler does not guarantee restarts on application or hardware failures. Scheduling is done based on the global model of the cluster the RM is aware of. It uses queues and capacity parameters during the allocation process.

The scheduling policy can be plugged into the Scheduler. The two popular scheduling policies in Hadoop 1.X were **CapacityScheduler** and **FairScheduler**. These policies continue to exist in Hadoop 2.X.

The ApplicationsManager is the component responsible for handling application submissions made by clients. In addition, it also bootstraps applications by negotiating the container on behalf of the application for the Application Master. The ApplicationsManager also provides the services of restarting the Application Master in case of failures. The following image illustrates the architecture of a YARN-based cluster:



The RM is loosely coupled and interacts via a couple of public interfaces and one private interface. It has the following interfaces:

- A public interface for clients to submit jobs (**Application-Client Protocol**)
- A public interface for AMs to request for resources (**Application-Master Protocol**)
- An internal interface for NM interactions

Resource allocation is dynamic and is agnostic to the internals of the application or its optimizations. This makes efficient resource utilization within clusters possible. AMs send resource requests with the following parameters:

- Number of containers that will be required, for example, 100 containers
- The specifications of resources in each container, for example, 2 CPUs and 4GB RAM
- The locality preferences for the container at the node or rack level
- Priorities of requests within an application

The RM's Scheduler gets these requests, and based on the cluster state image it has built using heartbeats from NMs, it allocates containers to AMs. The container exits are relayed back to the AMs. In the case of scarcity of cluster resources, the RM might request AMs to give back a few containers. If no containers are released after a certain timeout, the RM might terminate the container. The RM's request for resource release can be treated as a warning to save any critical data and work state that the AM might be executing.

Application Master (AM)

On application submission, the ApplicationsManager negotiates a container with the Scheduler. The container is used to bootstrap the AM for this particular application. Once spawned, the AM will periodically send heartbeats to the RM. The heartbeats are used to perform the following actions:

- Report whether the AM is alive
- Request resources for the application

In response to these heartbeats, the RM allocates containers and the AM is free to use them. The onus to interpret and handle container terminations and other application-related faults is completely up to the AM.

The AM interacts with the RM using the Application-Master protocol. The AM gathers the status of its containers directly from the NM. It can also start and stop containers allocated to it by interacting with the NM. The NM-related interactions are done via the **ContainerManager** protocol.

In YARN, the resource model follows the concept of late binding. The container spawned might not be related to the AM's request. It is only bound to a lease sought out by the AM. The state at which the AM requests resources might change by the time the resource is allocated. The allocated resource can be used for purposes other than what it is originally intended for.

Let's illustrate late binding by means of a hypothetical example using the MapReduce Application Master. We know that HDFS replicates each block of a file among nodes in a cluster. A Map task preferably runs on the same node as the input block. When the MR AM requests for containers, it assigns a Map task to the container whose data is local or close to the allocated container node. The decision happens only after the AM receives the containers, and in a dynamic fashion.



Hadoop 1.X had a web interface for the JobTracker. This web interface (generally listening on port 50030) is no longer available in Hadoop 2.X because the JobTracker is absent.

Node Manager (NM)

The NMs are per-node daemons that do local container management, ranging from authentication to resource monitoring. They report to the RM using heartbeats. A **Container Launch Context** (**CLC**) record is used to specify container metadata such as dependencies, data file paths, and environment variables. Based on the values in the CLC, an NM spawns a container.

Resources might be shared between containers by the same tenant. Download of resources and dependencies from external sources is also possible by providing their URLs. NMs are responsible for termination of containers, either on request by the AM or on decree from the RM. An NM has the authority to terminate a container if the container breaches its lease. Termination includes cleanup, such as deletion of any local data that the container might have.

Monitoring of local physical resources such as CPU, memory, and disk health falls in the NM's purview. It reports these parameters to the RM. The RM scheduler can take decisions on container scheduling based on the load or health of the node.

The NM provides services such as log aggregation to the application. Standard output and error logs spewed by an application are uploaded to HDFS upon application completion. NMs can also be configured to have pluggable **auxiliary services**. For example, an auxiliary service to persist local data until an application's termination, rather than container termination, can be useful in some scenarios. In the MapReduce use case, map outputs need to be transferred to the reducers. Auxiliary services can be used to achieve this. Any additional configuration required for these services can be specified by the CLC.

YARN clients

The YARN client is responsible for submitting an appropriate CLC for the AM. As we discussed earlier, the AM itself runs in a container whose resources are negotiated by the RM. Registering the AM is also the responsibility of the client. The client is free to provide other services to its consumers.

Developing YARN applications

YARN can bring in other computing paradigms to Hadoop. In Hadoop 2.X, MapReduce, Pig, and Hive are all Application Master libraries and their corresponding clients. Developers can write their own applications using the YARN API and leverage the existing infrastructure running Hadoop. Also, enterprises can have lots of data assets in HDFS already, and writing custom applications can leverage this without a need to provision new clusters or migrate the existing data.

Storm is a real-time stream-processing engine that has been ported onto YARN, bringing in the paradigm of moving data to compute nodes. **Spark** is another project that is on YARN and can leverage the existing Hadoop infrastructure to provide in-memory data transformations, including MapReduce. There are a number of projects in development that exhibit Hadoop's capability as a generic cluster-computing platform.

In this section, let's look at how to write a simple YARN application. The application takes in a shell command and executes it on a predefined number of nodes on the Hadoop cluster. We will need to write both the Application Master and Client programs.

Writing YARN clients

YARN clients submit applications to the RM via `ApplicationClientProtocol`. The result is the assignment of `ApplicationId` to the client. The client then needs to communicate the specifications of the container that deploys the Application Master. The Application Master is like a program that needs to be started and executed independently. These specifications include the location of the Application Master libraries, any environment variables required for the execution of the Application Master, and arguments to actually run the program.

The following code is a snippet from the shell command application. We write a run method that will be called from the `DistributedShellClient main` method. Arguments are passed to the `main` method from the command line. The arguments are the Application Master JAR path, shell command, and number of containers that execute this command. Let us see how to write a YARN client with the following steps:

1. The first step is to create a `YarnConfiguration` object. The `YarnConfiguration` class is the subclass of the `Configuration` class that is used in Hadoop MapReduce. The successful creation of the `YarnConfiguration` object means that the application is able to read the necessary config files, such as the `yarn-site.xml` file. The defaults of properties are present in the `yarn-default.xml` file. The `yarn-site.xml` file is generally found in the `etc/hadoop` folder relative to the Hadoop installation directory.
2. The application client now has to initialize a `YarnClient` object. This is done via a factory method called `createYarnClient`. The `YarnClient` object is initialized using the configuration that was created earlier. Based on the configuration passed, the `YarnClient` object resolves the RM end point. During initialization, `YarnClient` creates a proxy for the RM. All communication happens via the proxy. The proxy encapsulates the `ApplicationClientProtocol` object. A start call on the `YarnClient` object is called to get the client machinery up and running.
3. An alternative approach will be for developers to create the proxies by themselves and manage it. The `ApplicationClientProtocol` object is a proxy type that can be created and used. However, it is recommended to use the former method.
4. The `YarnClient` has a `createApplication` method that is used to get the `YarnClientApplication` object. Since `YarnClient` encapsulates a proxy for the RM, it also contains methods to retrieve properties about the RM and manage applications submitted to the RM.

   The code is as follows:

```
package MasteringYarn;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.yarn.api.ApplicationConstants;
import org.apache.hadoop.yarn.api.records.*;
import org.apache.hadoop.yarn.client.api.YarnClient;
import
org.apache.hadoop.yarn.client.api.YarnClientApplication;
import org.apache.hadoop.yarn.conf.YarnConfiguration;
```

```
import org.apache.hadoop.yarn.exceptions.YarnException;
import org.apache.hadoop.yarn.util.Apps;
import org.apache.hadoop.yarn.util.ConverterUtils;
import org.apache.hadoop.yarn.util.Records;
import java.io.File;
import java.io.IOException;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class DistributedShellClient {
private Configuration conf = new YarnConfiguration();

    public void run(String[] args) throws YarnException,
        IOException, InterruptedException {

YarnConfiguration yarnConfiguration = new
    YarnConfiguration();
YarnClient yarnClient = YarnClient.createYarnClient();
        yarnClient.init(yarnConfiguration);
        yarnClient.start();

YarnClientApplication yarnClientApplication =
    yarnClient.createApplication();
```

5. Once `YarnClientApplication` is created, the next step is to request a container in order to bootstrap the Application Master. Container specifications are described in a `ContainerLaunchContext` class in YARN. In the `org.apache.hadoop.yarn.util` package, there is a special `Records.newRecord` static factory method that instantiates different classes.
6. Going through the documentation of the `ContainerLaunchContext` class will give a glimpse of the properties that can be specified when launching a container. ACLs, commands, environment variables, local resources, binary service data, and security token setters are present in any `ContainerLaunchContext` object. In the following code, the `ContainerLaunchContext` object is instantiated and the `setCommands` method is called to set the list of commands that need to be executed upon the container launch. In our case, we will specify the command to launch the Application Master present in the `DistributedShellApplicationMaster` class that we will define later.

7. Launching the Application Master requires the necessary classes or JAR files to be present locally. The next step is to specify the JAR file containing the logic of the Application Master using the `setLocalResources` method on the `ContainerLaunchContext` object. In this example, the HDFS path to the JAR file will be taken as the local resource. The path of this file is specified as a command-line argument. Other side channels can also be used to distribute resources locally to containers.
8. Similarly, if any environment variables need to be set up for the smooth functioning of the container, it can be set up using the `setEnvironment` method on the `ContainerLaunchContext` object.
9. The most important pieces of specification needed by the RM to launch any container are CPU and memory requirements of the container. In this example, the Application Master container needs around 100MB of memory and a single core to execute. This is specified using a `Resource` object. The `Resource` object is an abstract representation of the container's compute requirements. The `Resource` object can currently model CPU and memory. CPU is modeled in units called virtual cores. It is an integer value, and the configuration has to map a virtual core to the actual physical core. Usually, this mapping is 1:1. The memory is modeled in megabytes (MB). The `setVirtualCores` and `setMemory` methods on the `Resource` object are used to specify them:

```
    //container launch context for application master
ContainerLaunchContext applicationMasterContainer =
    Records.newRecord(ContainerLaunchContext.class);
        applicationMasterContainer.setCommands(
                Collections.singletonList("$JAVA_HOME/bin/java
    MasteringYarn.DistributedShellApplicationMaster " +
                                           args[2]  +
                                           " "      +
                                           args[3]  +
                                           " "      +
                                           "1>"     +
                                           ApplicationConstants.
LOG_DIR_EXPANSION_VAR + "/stdout "  +
                                           "2>"     +

ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stderr")
        );


LocalResource applicationMasterJar =
    Records.newRecord(LocalResource.class);
```

```
        setupJarFileForApplicationMaster(new Path(args[1]),
    applicationMasterJar);
        applicationMasterContainer.setLocalResources(
                Collections.singletonMap("MasteringYarn.jar",
    applicationMasterJar)
        );

        Map<String, String> appMasterEnv = new HashMap<>();
        setupEnvironmentForApplicationMaster(appMasterEnv);
        applicationMasterContainer.setEnvironment(appMasterEnv);

        Resource resources = Records.newRecord(Resource.class);
        resources.setVirtualCores(1);
        resources.setMemory(100);
```

10. The final step is to submit the application to the ApplicationManager in the RM. The submission parameters are bundled in an `ApplicationSubmissionsContext` object. The `YarnClientApplication` class holds a reference to this context. The `ApplicationSubmissionContext` object is given the container specifications, the submission queue, a friendly name for the application, and the `Resource` object needed to bootstrap the container. The `ApplicationSubmissionContext` object also gives `ApplicationId`. This `ApplicationId` object can be used to reference the application in the management APIs. In the following example, we will set the application to be in the default queue with a friendly name, `MasteringYarn`. Finally, the `YarnClient` object is used to submit the application. Internally, the proxy is used to post the application request to the RM. The Scheduler then kicks in and schedules the application on the cluster. The Application Master is the first container that will be spawned:

```
ApplicationSubmissionContext submissionContext =
    yarnClientApplication.getApplicationSubmissionContext();
        submissionContext.setAMContainerSpec(applicationMaster
    Container);
        submissionContext.setQueue("default");
        submissionContext.setApplicationName("MasteringYarn");
        submissionContext.setResource(resources);

ApplicationId applicationId =
    submissionContext.getApplicationId();
```

```
            System.out.println("Submitting " + applicationId);
            yarnClient.submitApplication(submissionContext);
            System.out.println("Post submission " +
                applicationId);
```

Once the submission of the application finishes, the progress of the application can be monitored using the `getApplicationReport` method on the `YarnClient` object. The `ApplicationReport` object contains useful information about the application that can be used to determine the success or failure of the application. It also contains a diagnostic field that can help the developer gain insight in the case of failures.

The `ApplicationReport` object has the `getYarnApplicationState` method that gives the current state of the application. In the following code, we poll the application state every 1 second and see whether it is terminated. An application is terminated if it is in the `KILLED`, `FINISHED`, or `FAILED` state. The `getDiagnostics` function is used to print diagnostic information in the event of failures:

```
ApplicationReport applicationReport;
YarnApplicationState applicationState;

do{
            Thread.sleep(1000);
            applicationReport =
                yarnClient.getApplicationReport(applicationId);
            applicationState =
                applicationReport.getYarnApplicationState();

            System.out.println("Diagnostics " +
                applicationReport.getDiagnostics());

        }while(applicationState != YarnApplicationState.FAILED &&
               applicationState != YarnApplicationState.FINISHED
                &&
               applicationState != YarnApplicationState.KILLED );

        System.out.println("Application finished with " +
            applicationState + " state and id " + applicationId);
    }
```

A couple of helper methods complete the client. The first method, `setJarFileForApplicationMaster`, sets up the Application Master JAR file with appropriate properties, most of which are self-explanatory. Similarly, all the necessary environment variables are packaged in the `setEnvironmentForApplicationMaster` method. This method also illustrates the use of `YarnConfiguration` to read off of the `yarn-site.xml` file. Finally, the main driver method instantiates the `DistributedShellClient` object and calls the run method on it:

```
    private void setupJarFileForApplicationMaster(Path jarPath,
        LocalResource localResource) throws IOException {
  FileStatus jarStat = FileSystem.get(conf).getFileStatus(jarPath);
        localResource.setResource(ConverterUtils
            .getYarnUrlFromPath(jarPath));
        localResource.setSize(jarStat.getLen());
        localResource.setTimestamp(jarStat.getModificationTime());
        localResource.setType(LocalResourceType.FILE);
        localResource.setVisibility(LocalResourceVisibility.PUBLIC);
    }

    private void setupEnvironmentForApplicationMaster(Map<String,
        String> environmentMap) {
        for (String c : conf.getStrings(
                YarnConfiguration.YARN_APPLICATION_CLASSPATH,
                YarnConfiguration.DEFAULT_YARN_APPLICATION_CLASSPATH))
  {
            Apps.addToEnvironment(environmentMap,
                ApplicationConstants.Environment.CLASSPATH.name(),
                    c.trim());
        }
        Apps.addToEnvironment(environmentMap,
                ApplicationConstants.Environment.CLASSPATH.name(),
                ApplicationConstants.Environment.PWD.$() +
                    File.separator + "*");
    }

    public static void main(String[] args) throws Exception {
  DistributedShellClient shellClient = new DistributedShellClient();
        shellClient.run(args);
    }
  }
```

Writing the Application Master entity

Application Master is the leader of the application. This entity encapsulates all the logic for the application and requests for resources from the RM as and when it is appropriate. Unlike the Client, Application Master has to keep contact with the following two entities:

- **ResourceManager**: This is used for communications regarding the global state of the application. This is also known as **ApplicationMasterProtocol**.
- **NodeManager**: This is used for communications regarding containers allocated to the application. This protocol is also called **ContainerManager**.

Writing the Application Master is similar to writing a client. We start off creating a `YarnConfiguration` object. To facilitate communication with the RM, `AMRMClient` is created. This client encapsulates the proxy object that is needed to talk to the RM. Again, the proxy can be explicitly created, though this is a simpler and recommended way to do it. `AMRMClient` has many methods; the most important methods deal with the registration of the AM (`registerApplicationMaster`) and container allocation request (`addContainerRequest`) methods.

To communicate with the NM, an `NMClient` object is created that encapsulates the communication proxy. The important methods on `NMClient` are the `startContainer` and `stopContainer` methods that are used to launch and terminate containers on nodes. The following code snippet illustrates this:

```
package MasteringYarn;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.yarn.api.ApplicationConstants;
import
org.apache.hadoop.yarn.api.protocolrecords.AllocateResponse;
import org.apache.hadoop.yarn.api.records.*;
import org.apache.hadoop.yarn.client.api.AMRMClient;
import org.apache.hadoop.yarn.client.api.NMClient;
import org.apache.hadoop.yarn.conf.YarnConfiguration;
import org.apache.hadoop.yarn.exceptions.YarnException;
import org.apache.hadoop.yarn.util.Records;

import java.io.IOException;
```

```
import java.util.Collections;

public class DistributedShellApplicationMaster {


    public static void main(String[] args) throws YarnException,
IOException, InterruptedException {

        Configuration configuration = new YarnConfiguration();
int numberOfContainers = Integer.parseInt(args[1]);
        String command = args[0];

        System.out.println("Starting Application Master");

        AMRMClient<AMRMClient.ContainerRequest>
            resourceManagerClient = AMRMClient.createAMRMClient();
        resourceManagerClient.init(configuration);
        resourceManagerClient.start();


        System.out.println("Started AMRMClient");

NMClient nodeManagerClient = NMClient.createNMClient();
        nodeManagerClient.init(configuration);
        nodeManagerClient.start();

        System.out.println("Started NMClient");
```

Both the `AMRMClient` and `NMClient` classes have corresponding asynchronous versions. Asynchronous APIs are resource efficient as threads don't block themselves waiting for responses. After the method call is invoked, threads are free to take up some other task. When the results of the APIs are ready, registered callback handler methods are called based on the nature of the result.

The following code outline illustrates the usage of the `AMRMClientAsync` class for communication with the RM:

```
class AMRMClientCallbackHandler implements
AMRMClientAsync.CallbackHandler {
public void onContainersAllocated(List<Container>
containers) {
     //the container is allocated and relevant tasks
can
be executed.
}
public void onContainersCompleted(List<ContainerStatus>
statuses){
      //the container has completed. The application
status needs to be updated.
}
public void onNodesUpdated(List<NodeReport> updated) {}
        public void onReboot() {}
}
AMRMClientAsync asyncClient =   AMRMClientAsync.
createAMRMClientAsync(appId, 1000, new
AMRMClientCallbackHandler ());
//init the client with the configuration and start the
proxy.
asyncClient.addContainerRequest(container)
```



The `AMRMClientCallbackHandler` object is passed on async client creation. The appropriate methods on this handler are called whenever an event happens on the container. For example, when a container is allocated, the `onContainersAllocated` callback method is called. A similar API is present for the `NMClient` object as well.

The `AMRMClient` class is used to register the AM with the RM. On successful registration, the AM starts a heartbeat thread that periodically informs the RM that it is alive. The `registerApplicationMaster` method also supplies the host and port on which the Application Master is listening. Clients can use the AM's host and port to get information about the application.

Containers now have to be allocated based on the arguments specified when starting our DistributedShell application. The two important properties that need to be set is the priority of the container and the amount of resources to be allocated for the container.

The `Priority` class is instantiated to set the priority of the container. In the following example, we use the priority of 0. The `Priority` object is applicable within this particular application only.

As we did for the Application Master, we set the resource requirements for each worker container using the `Resource` class. To recall, the `setMemory` method sets the memory requirement of the container in MB, and `setVirtualCores` sets the number of cores required.

The `Priority` and `Resource` objects are then assigned to an `AMRMClient.ContainerRequest` object to be dispatched to the RM for resource allocation. The second and third parameters of the constructor are null. These correspond to any nodes and racks we would like the containers to be allocated on. This is particularly useful for applications such as MapReduce, where data locality needs to be exploited. The data type of these parameters is `String[]`.The racks corresponding to any nodes that are listed in the second parameter are automatically added to the list of racks.

The `ContainerRequest` objects are now added to the `AMRMClient` proxy object using the `addContainerRequest` method on the client:

```
resourceManagerClient.registerApplicationMaster("localhost",
    80010, "myappmaster");

        System.out.println("Registration done");

        // Priority for worker containers - priorities are intra-
            application
        Priority priority = Records.newRecord(Priority.class);
        priority.setPriority(0);

        // Resource requirements for worker containers
        Resource capability = Records.newRecord(Resource.class);
        capability.setMemory(128);
        capability.setVirtualCores(1);


        for(int i=0; i < numberOfContainers; i++){
```

```
            AMRMClient.ContainerRequest containerRequest = new
                AMRMClient.ContainerRequest(capability, null,
                    null, priority);
            resourceManagerClient.addContainerRequest(containerReque
st);

        }
```

The `allocate` method on the `AMRMClient` class indicates to the RM to allocate containers. It also acts like a heartbeat to the RM. The return value of this method is an `AllocateResponse` object. This object contains information about newly allocated containers, completed containers, and cluster-related information. It also indicates the remainder resources that are available for this particular application within the cluster.

`AllocateResponse` also has `ResponseId` that can be used to disambiguate duplicate requests. The parameter in the `allocate` method is a progress indicator of the float type. The AM can indicate the progress of the application via this parameter.

In the following code snippet, we use the `getAllocatedContainers` method to get the entire `Container` object corresponding to the newly allocated containers. For each of these containers, we have to launch the shell command that was specified in the command line.



Concurrent allocate request calls should be avoided. This might lead to request loss.

Container launch is always done using the `ContainerLaunchContext` object. It is very similar to the launch of the AM container where we specify the commands, environment variables, local resources, and other parameters that are needed by the program to execute in the container.

The container is then launched using the `NMClient` object. This informs `NodeManager` to launch the container and execute the relevant commands. The `startContainer` also takes in the `Container` object that was returned by the RM. The `Container` object contains identifiers, tokens, and the node where the container is allocated:

```
int completedContainers = 0;
int containerId = 0;
while(completedContainers < numberOfContainers){

  AllocateResponse allocateResponse = resourceManagerClient.
allocate(containerId);
```

```
    containerId++;

    for(Container container : allocateResponse.getAllocatedContainers())
{

      ContainerLaunchContext shellContainerContext = Records.newRecord(C
ontainerLaunchContext.class);
      shellContainerContext.setCommands(
      Collections.singletonList(command +
      " 1>"  +
      ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stdout "  +
      " 2>"  +
      ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stderr")
      );

      nodeManagerClient.startContainer(container,
shellContainerContext);

    }
```

Once the containers are launched, the allocate call can be used to monitor the completed containers, as shown in the next code snippet. The `getCompletedContainersStatuses` method on the `AllocateResponse` object gives the status of each completed container. On completion, the AM can unregister from the RM by calling the `unregisterApplicationMaster` call using the `AMRMClient` object. The status of the application can be communicated to the RM. This in turn can be reported back to the client or any other process that monitors the application.

The `FinalApplicationStatus` enum has the `FAILED`, `KILLED`, `SUCCEEDED`, and `UNDEFINED` values. The unregister call also take in any diagnostic message that needs to be communicated and any new URL the client can use to get information about the Application Master on termination:

```
for(ContainerStatus containerStatus : allocateResponse.
getCompletedContainersStatuses()){
                completedContainers++;
                System.out.println("Completed Container " +
completedContainers + " " + containerStatus);

          }

            Thread.sleep(1000);
```

```
        }


          resourceManagerClient.unregisterApplicationMaster(FinalApplica
tionStatus.SUCCEEDED, "", "");


      }


    }
```

The application can now be executed using the following commands. The date command will be run on two containers in the cluster:

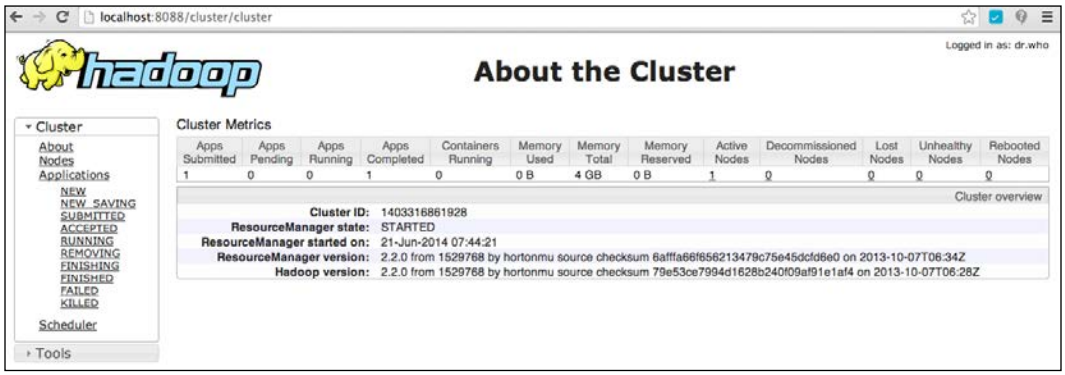
```
hadoop fs -copyFromLocal MasteringYarn-1.0-SNAPSHOT.jar

hadoop jar MasteringYarn-1.0-SNAPSHOT.jar
MasteringYarn.DistributedShellClient
hdfs://localhost/user/sandeepkaranth/MasteringYarn-1.0-SNAPSHOT.jar
date 2
```

Monitoring YARN

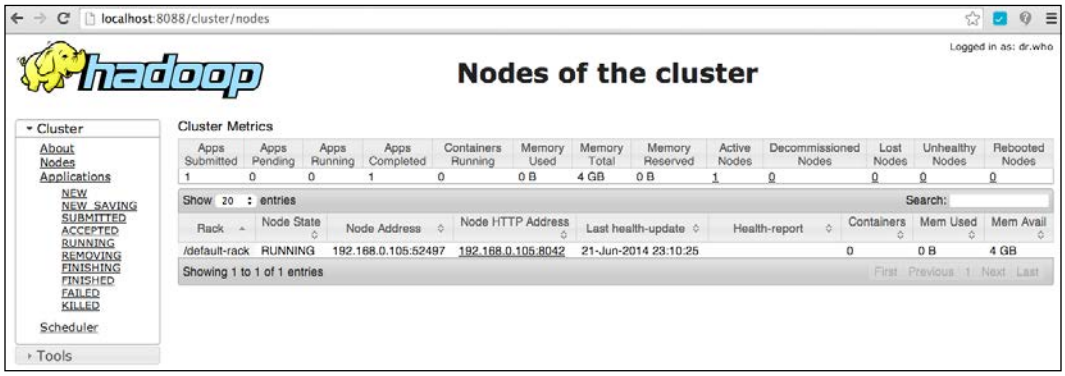
The RM provides a friendly web interface to view the cluster and its resources. The home page of this interface gives details about the cluster, such as the RM state, number of applications, the total memory available, total number of nodes, and node status, among other details. The next screenshot shows the home page.

On the left-hand side of the screen, there are links to navigate and get different kinds of details of the cluster.



Clicking on the **Nodes** link on the left-hand pane gives the details of the nodes in the YARN cluster. The following screenshot shows an example of a single node cluster. For each node in the cluster, the screen gives details on the rack it belongs to, the node state, resource consumption (memory for now) on the node, HTTP address of the node, and last heartbeat details of the node, among other details.

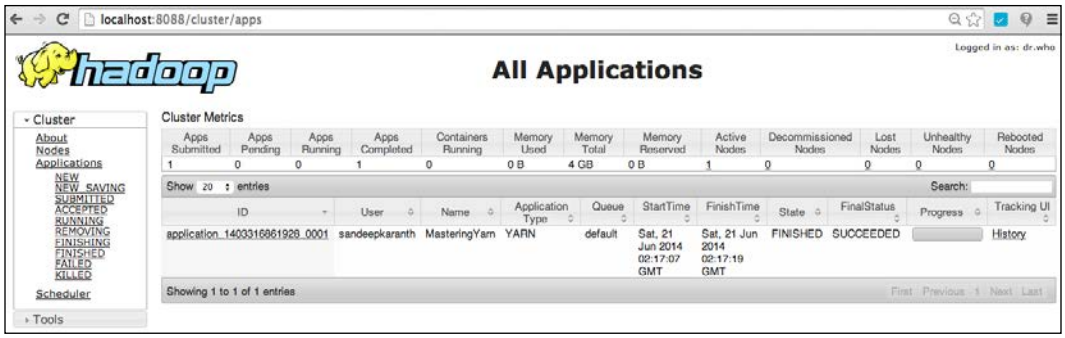
The last-health update column in the nodes grid tells when the RM received the last heartbeat from the NM.



The **Applications** link on the left-hand pane gives details of all the applications, as shown in the following screenshot. Applications can be filtered based on the state they are in. The list of filters, such as **NEW**, **NEW_SAVING**, **SUBMITTED**, **ACCEPTED**, **RUNNING**, **REMOVING**, **FINISHING**, **FINISHED**, **FAILED**, and **KILLED** are present on the left-hand pane. By clicking on each filter, applications present in a particular state will be visible.

Application details such as the kind of application, the queue it belongs to, its state and final state, start and end times, and progress, among other details, are visible. Though the Web UI does not give any way to actually execute a command, the application ID can be used to execute commands using the YARN script.

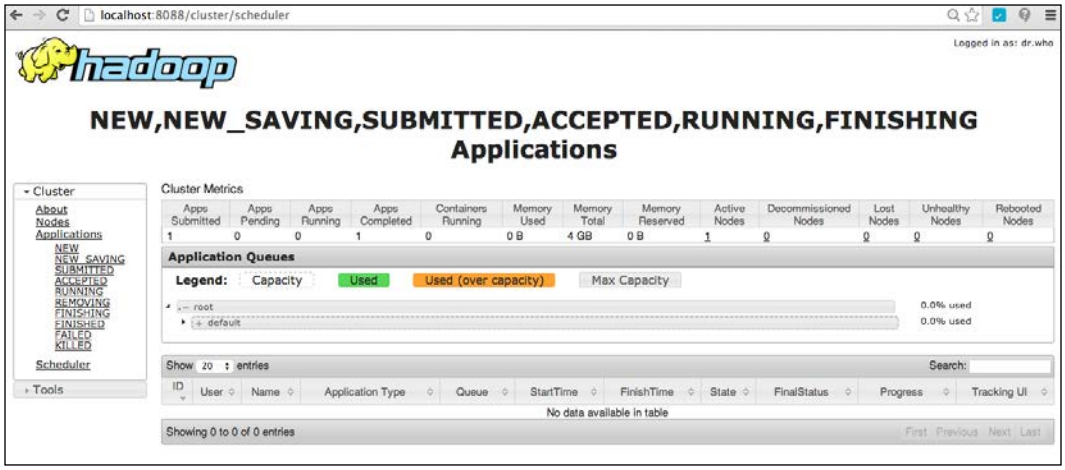
We will look at some of the YARN commands to manipulate applications in the following sections in this chapter.



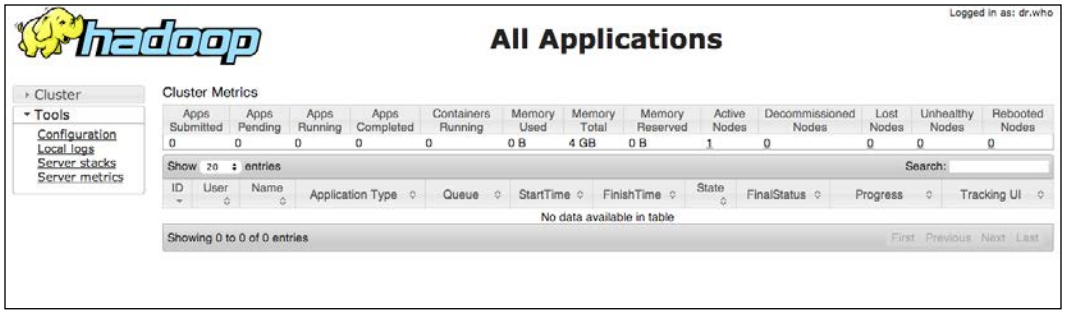
Clicking on the **Scheduler** link shows the details of the scheduler being used by the RM. The Web UI shows the hierarchy of queues and color-codes the capacity, maximum capacity, used capacity, and used capacity (over capacity) for each queue. The next section on scheduling gives a better sense of these application queues.

The details of any running applications are also given here. The heading of the page gives details on the states of applications that are shown on this page. Only **NEW**, **NEW_SAVING**, **SUBMITTED**, **ACCEPTED**, **RUNNING**, and **FINISHING** applications are shown.

The cluster metrics section is a common UI section across all the pages of the Web UI that RM exposes. The following screenshot shows an example scheduler page:



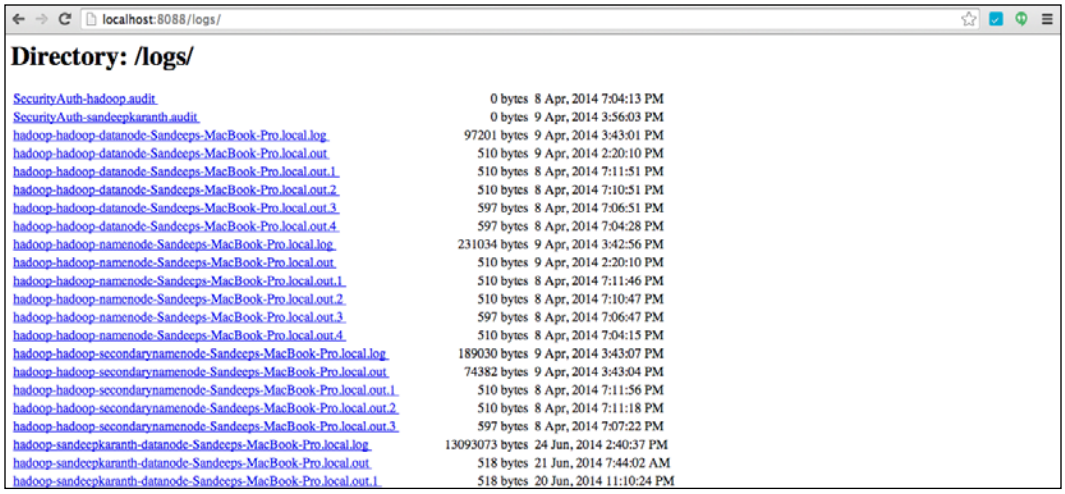
There is a **Tools** link on the left-hand pane, as shown in the following screenshot. On expansion, it has a few menu items. These are the tools that help an administrator and developer of a YARN cluster and application to debug.



The **Configuration** hyperlink takes the user to the configuration that is used by YARN. This provides a quick-and-easy way to go through the values of different properties in the YARN cluster. The following screenshot shows the configuration page. The configuration is displayed in XML.



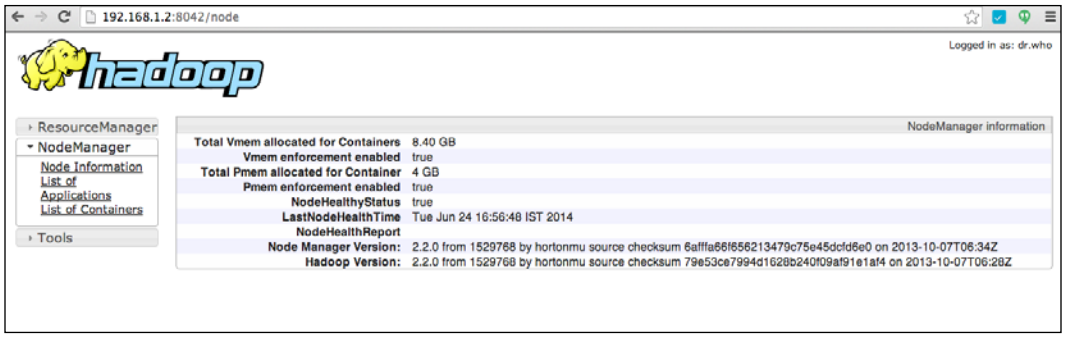
The **Local logs** link opens up the local log directory. Clicking on each log opens up the log in the browser. The following screenshot shows the local logs on a single machine cluster deployment:



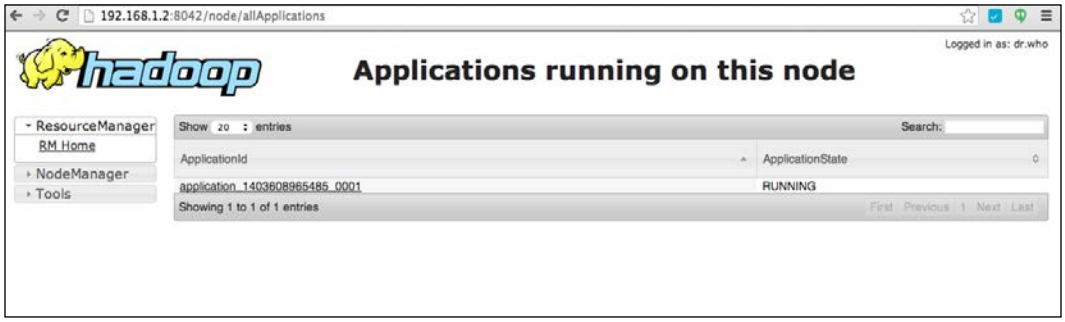
The **Server stacks** link opens up the exception stacks the server throws along with the threading information. The screenshot of the server stack link is as follows:



There is a **Server metrics** link that opens up the metrics page. Clicking on a node host takes the user to the **NodeManager** section. The left-hand pane of the page is now introduced with NodeManager-related links, as shown in the following screenshot. This screen gives the information of a particular node.



Clicking on the **List of Applications** link takes the user to a screen that looks something like the following one. Here, the list of applications running on this node is displayed with its application state.



Clicking on the **List of Containers** link in the **NodeManager** menu item gives the details of the containers currently running on the NodeManager, with the state of each container. There is a link to open the logs directory as well.



Job scheduling in YARN

Most cluster resources are multitenant in nature, that is, a number of teams or people share the cluster resources. Allocation of resources to satisfy the needs of all these tenants becomes important and is the responsibility of the scheduler. Individual clusters per team or person is not viable as they render poor utilization.

YARN provides a pluggable model to schedule policies. The initial versions of Hadoop had a simple **First in First Out** (**FIFO**) scheduler. However, FIFO was found to be inadequate in dealing with the complexities of multitenancy. We will discuss two other scheduling strategies that are used in Hadoop today, CapacityScheduler and FairScheduler.

CapacityScheduler

The concept behind CapacityScheduler is to guarantee a tenant-promised capacity on a shared cluster. If other tenants utilize less than the requested capacity, the scheduler allows the tenant to tap into these unused resources. The number one goal of CapacityScheduler is not to allow a single application or user to hog the cluster resources. The scheduler enforces strict limits on the resource usage of tenants sharing the cluster.

CapacityScheduler manages scheduling based on queues. Administrators configure these queues based on the requirements of the tenants. Hierarchical queues are used to share the underutilized part of the cluster. Hierarchies ensure that first preference is given to the tenant who has requested for the capacity before others can use it.

Each queue has a capacity that the administrator can configure. The sum of the capacities of all the queues in the cluster determines the cluster's capacity. The capacity of a queue is elastic, so the scheduler can transfer the unused capacity from one queue to another. This redistributed capacity can be reclaimed to satisfy the capacity guarantee made on a queue. A maximum capacity for a queue can also be enforced. In addition, each queue can also support per-user limits.

In addition to queues and their hierarchies, CapacityScheduler also has the following features:

- CapacityScheduler has a set of security features. Each queue has ACLs that authorize users to submit jobs into the queue. User jobs are isolated, preventing other users from modifying jobs other than their own. The scheduler also introduces the concept of per-queue and system administrator roles.
- CapacityScheduler is dynamic, that is, its properties, such as queue definitions and ACLs, can be changed at runtime. Delete operations on the queues are not permitted. Adding new queues is allowed.

- Administrators are allowed to stop queues, preventing new jobs from getting submitted onto the queue and its children queues. Existing jobs are allowed to continue, though without pre-emption. The administrator is allowed to start the queues once the jobs from the queue have drained.
- Applications that require higher resources such as multiple maps and reduce slots for a job, are allowed by CapacityScheduler. CapacityScheduler does a resource-based scheduling of the jobs as long as they do not exceed the capacity constraints that are set on the queues.

In YARN, CapacityScheduler can be plugged in using following methods:

- The ResourceManager can be directed to use CapacityScheduler by setting the `yarn.resource.manager.scheduler.class` property to `org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler`. This setting is declared in the `yarn-site.xml` config file.
- Queues can be set up by adding entries to the `capacity-scheduler.xml` file. This is the configuration file of CapacityScheduler. A predefined queue called as the **root** is present. Any queue that is created is a child of the root queue.
- The `yarn.scheduler.capacity.root.queues` property is used to define additional queues. Queues are specified as comma-separated lists of queue names. Queue paths specify hierarchies of queues. Queue paths are special property names that start from the root queue and list the tree path using a dot notation. The following configuration file snippet describes two levels of queues. The `x`, `y`, and `z` queues are under the root queue. The `x1` and `x2` queues are under the `x` queue. The queue path for `x1` and `x2` is given by `root.x`:

```
<property>
<name>yarn.scheduler.capacity.root.queues</name>
<value>x,y,z</value>
<description>The queues at the this level (root is the root
  queue).   </description>
</property>
<property>
<name>yarn.scheduler.capacity.root.x.queues</name>
<value>x1,x2</value>
<description>The queues at the this level (root is the root
  queue).   </description>
</property>
```

- The resource allocation characteristics for each queue is characterized by the following properties:
    - `yarn.scheduler.capacity.<queue-path>.capacity`: This property is used to set the capacity of each queue as a percentage of the cluster capacity. The property is of the `float` type. At each level in the queue hierarchy, these queue values must add up to be a 100 percent. This is a soft limit. If unused capacity is available, the jobs in the queue can use it, providing elasticity to the queue.
    - `yarn.scheduler.capacity.<queue-path>.maximum-capacity`: This property is used to set a hard limit on the capacity used by a queue. This hard limit on the queue capacity is a float value and limits the elasticity of the queue. The default value is -1, that is, there is no limit on the elasticity.
    - `yarn.scheduler.capacity.<queue-path>.minimum-user-limit-percent`: This is an integer value property that enforces the percentage of resources allocated to a single user in the queue. The limit kicks in only when there is a demand for resources. For example, if we set this value to 50, a single user can be allocated 100 percent of the capacity, and two users can be allocated 50 percent of the capacity. However, beyond two users, the scheduler waits for the existing user applications to complete before scheduling the next one. The default value for this property is 100.
    - `yarn.scheduler.capacity.<queue-path>.user-limit-factor`:This property dictates multiple queue capacities that can be used by the user. For example, if this property has a value 2, a user on this queue can be allocated twice as many resources as the queue capacity. This can happen only if the cluster has the capacity and is idle. This is a `float` value type and the default is 1.
- CapacityScheduler supports the following properties to control the properties of running applications:
    - `yarn.scheduler.capacity.maximum-applications`: This property determines the maximum number of active applications in the cluster. This is a hard limit and submissions of applications beyond this limit are not allowed. The default value is 10,000.
    - `yarn.scheduler.capacity.<queue-path>.maximum-applications`: This property is a per-queue override for the preceding property. Both these properties are of `integer` type.

    - `yarn.scheduler.capacity.maximum-am-resource-percent`: This property determines the percentage of resources in the cluster that is allocated for Application Masters alone. The default value is 0.1, that is, Application Master containers can utilize 10 percent of the cluster resources.
    - `yarn.scheduler.capacity.<queue-path>.maximum-am-resource-percent`: This property is a per-queue setting for resources allocated to AMs.

- CapacityScheduler supports the following properties that help in setting cluster authorization and queue runtime parameters:
    - `yarn.scheduler.capacity.<queue-path>.state`: This property sets the queue state. It can be in the **RUNNING** or **STOPPED** state. In the **STOPPED** state, application submissions to this queue or any of its child queues are not allowed. In the **STOPPED** state, existing applications are allowed to execute and finish.
    - `yarn.scheduler.capacity.<queue-path>.acl_submit_applications`: This property determines the users who can submit applications to the queue and all its children. ACLs are inherited from parent queues. ACLs are comma-separated lists of users or groups. A * wildcard can be used to specify anyone.
    - `yarn.scheduler.capacity.<queue-path>.acl_administer_queue`: This property determines the users who can administer the queue and all its children queues.

- The `yarn rmadmin` command can be used to refresh the ResourceManager (RM) with the new properties. The refresh happens without having to restart the ResourceManager.

FairScheduler

As the name suggests, the concept behind FairScheduler is to provide, on an average, equal resources to all running applications over time. FairScheduler organizes applications into pools or queues and shares time between the different application pools. Periodically, the scheduler checks each application for the compute time it has received in the cluster and the amount of time it would have received under ideal conditions.

The applications are sorted in descending order of deficits. The next application to be scheduled will be the one with the largest deficit. The concept of hierarchical pools exists in FairScheduler as well.

To configure FairScheduler in YARN, the following points needs to be considered:

- The RM can be made to use FairScheduler by setting the `yarn.resourcemanager.scheduler.class` property to `org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler`. This configuration value is set in the `yarn-site.xml` config file.
- The other properties can be set in two files:
    - The `yarn-site.xml` file is used to define global scheduler properties.
    - An allocation file is used to specify properties such as weights and capacities for each queue or pool. This file is loaded every 10 seconds. Changes to this file take effect whenever the file is loaded.
- The important global properties that can be placed in the `yarn-site.xml` file are as follows:
    - `yarn.scheduler.fair.allocation.file`: This property contains the path to the allocation file. This file is in the XML format and specifies the properties of each pool or queue. The value defaults to the `fair-scheduler.xml` file.
    - `yarn.scheduler.fair.use-as-default-queue`: This property has a Boolean value. If set to `true`, it uses the username associated with the allocation as the pool or queue name. If set to `false`, there is a shared queue called **default**, and all jobs are allocated to this queue. The default value is `true`.
    - `yarn.scheduler.fair.sizebasedweight`: This is a Boolean property that suggests whether all apps have to be given equal share. It defaults to `false`, that is, all apps are given equal share. If set to `true`, the applications are weighted by the logarithm to base two of one plus the requested memory of the application.
    - `yarn.scheduler.fair.locality.threshold.node`: This property is of the `float` type between 0 and 1. When requesting containers on specific nodes to exploit locality, the app might want to delay allocation if containers cannot be allocated on these specific nodes. This particular property value determines the amount of delay before allocating the container on a nonlocal node. The value is the fraction of the cluster size. It defaults to -1.0, indicating to the scheduler to allocate containers without any delay.
    - `yarn.scheduler.fair.locality.threshold.rack`: This property is very similar to the previous property. However, unlike the previous property, this one deals with the rack-local placement of containers.

    - `yarn.scheduler.fair.allow-undeclared-pools`: This is a Boolean property that determines whether new queues can be created when the application is submitted to the RM. If it is set to `false`, any application that does not belong to a pool specified by the allocations file will reside in the default pool.

- The allocation file defines the pools or queues in the cluster. It is an XML manifest file that has the following elements:

Queue element	Description
`minResources`	This is the value is of the form `A mb, B vcores`, indicating the minimum number of resources for this particular queue. If this condition cannot be satisfied, resources from parent pools are reallocated.
`maxResources`	This is the value corresponding to this tag gives the maximum resources a queue can consume. No containers are allocated to the queue if it is deemed to use more resources.
`maxRunningApps`	This is the upper limit on the number of applications that can run simultaneously from this particular queue.
`weight`	This defines the proportion of resources this queue can use when compared to the default. The default is 1. If the weight is 2, it can use twice the number of resources when compared to the default.
`schedulingPolicy`	The allowed values are `fifo`, `drf`, or `fair`.
`aclSubmitApps`	This is the ACL for the users and groups that can submit jobs to this queue. The ACL format is the same as in the case of CapacityScheduler.
`aclAdministerApps`	This is the ACL description for the list of users and groups that can do administrative functions on this queue.
`minSharePreemptionTimeout`	In conditions where the queue is not given its due share of resources (`minResources`), it waits for the time defined by this property. After this timeout, it pre-empts containers from other queues.

User Element	Description
maxRunningApps	This is the upper limit on the number of applications that can be run by a single user in the queue.

queuePlacementPolicy element	Description
rule	This particular node in the XML file contains the rules on how a submitted application should be placed within the queue. There can be a number of rules, and each rule is executed in the order of declaration. For example, the rule specified places the application in the queue that was specified during submission. If no queue is specified, it is placed in the default queue. The "user" rule places the app in the queue with the username.

The outline XML of the allocation file is given as follows:

```
<?xml version="1.0"?>
<allocations>
  <queue name="">
    <minResources></minResources>
    <maxResources>A mb, B vcores</maxResources>
    <maxRunningApps></maxRunningApps>
    <weight>1.0</weight>
    <schedulingPolicy>fair</schedulingPolicy>
    <queue name="sub_queue_name">
      <aclSubmitApps>username</aclSubmitApps>
     <!—other queue properties can appear à
   </queue>
  </queue>

  <user name="username">
    <maxRunningApps></maxRunningApps>
  </user>
<queuePlacementPolicy>
    <rule name="specified" />
    <rule name="user" />
    <rule name="primaryGroup" create="false" />
    <rule name="default" />
    <rule name="reject" />
  </queuePlacementPolicy>
</allocations>
```

YARN commands

Like Hadoop, YARN has a script that provides commands to manage YARN. The commands are of the following two kinds:

- **User command**s: These are commands for the cluster user
- **Administration commands**: These are commands for the cluster administrator

In Hadoop deployment, the YARN script is found in the same directory as the Hadoop scripts. The general syntax of the YARN script is as follows:

```
yarn [--config <config directory>] command [options]
```

The `–config` option can be used to override the default configuration. The default configuration directory is picked up from the environment variable `$HADOOP_PREFIX/conf`.

User commands

The following are the important user commands in YARN:

- The `jar` command is used to run a custom, user-built JAR file. In the previous distributed shell example, we use the following command to run the YARN job. The syntax for the command is:

  ```
  yarn jar <jar file path> [main class name] [arguments…]
  ```

- The `application` command is used to manipulate a running application in YARN. It has three verbs: to list all the running applications in the cluster, to get the status of the application, and to kill a running application. The list verb can be filtered on the application state and application type:

  ```
  yarn application -list [-appStates <state identifiers> | -
  appTypes <type identifiers>] | -status <application id> | -
  kill <application id>
  ```

- The `node` command is used to report the status on the nodes in a cluster. It has two verbs: to list all the status and to find the status of a particular node. The `list` command can also be used to filter nodes that are in particular states:

  ```
  yarn node –list [-all | -states <state identifiers> | -status
  <node id>
  ```

- The `logs` command is used to dump logs of completed applications. It has two verbs: to dump logs of a particular user or to dump based on container identifier and node address values. The application ID is the mandatory parameter:

  ```
  yarn logs -applicationId <application Id> -appOwner <appOwner>
  | (-nodeAddress <node address> & -containerId <container Id>)
  ```

Administration commands

The important administration commands in YARN are as follows:

- The `resourcemanager`, `nodemanager`, and `proxyserver` parameters start the respective daemons:

  ```
  yarn resourcemanager | nodemanager | proxyserver
  ```

- The administrator can manipulate the ResourceManager by using the `rmadmin` command. This command has the following verbs:
  - `-refreshQueues`: This refreshes all the queue ACLs, states, and scheduler properties
  - `-refreshNodes`: This refreshes node-specific information in the ResourceManager
  - `-refreshUserToGroupMappings`: This refreshes all mappings about user memberships in groups
  - `-refreshSuperUserGroupsConfiguration`: This refreshes superuser-specific mappings
  - `-refreshAdminAcls`: This refreshes ACLs, determining the RM administrator access
  - `-refreshServiceAcl`: This reloads the authorization file in the RM

- The administrator uses the `daemonlog` command to get and set the log levels of the YARN daemons:

  ```
  yarn [-getLevel <daemon host:port> <name>| -setLevel <daemon
  host:port> <name> <level>]
  ```

Summary

YARN has opened up the Hadoop ecosystem to a wide range of applications. It has not only alleviated scaling bottlenecks that were present in traditional MapReduce-based Hadoop but also aided in improving infrastructure efficiency in an organization. This was made possible by:

- Separating out application-specific logic from resource management. The ResourceManager is solely responsible for cluster resource management and is agnostic of any application.
- Providing common and generic abstractions for resource specifications. Resources are specified in terms of cores and memory.
- Maintaining backward compatibility with existing Hadoop APIs. Existing Hadoop programs work on YARN on recompilation, without any code changes.
- Providing a variety of pluggable scheduling policies such as FairScheduler and CapacityScheduler. Pluggable policies make it easy for other paradigms to come onboard.

Development of newer computing paradigms on Hadoop is as simple as implementing a client and Application Master. These components interact with the ResourceManager and NodeManager to achieve their goals. Like MapReduce, applications such as Spark and Storm aspire to become first-class citizens in the Hadoop ecosystem. YARN has made this aspiration a reality.

In the next chapter, we will look at Storm and how it can be integrated with Hadoop. Storm is a real-time stream processing cluster-based engine that can operate on streaming data.

7
Storm on YARN – Low Latency Processing in Hadoop

Hadoop MapReduce builds on the concept of moving computation to data. Data is significantly larger than the instructions to manipulate it. The network is the slowest component in any distributed data processing system, so it is natural to move the smaller piece around, that is, the program itself. With assistance from the NameNode, Hadoop knows exactly how the data resides in a cluster of computers. It uses this data locality information to schedule tasks on appropriate nodes, putting in the best effort to locate the task very close to the data needed by the task.

In this chapter, we will discuss the opposite paradigm, that is, moving data to the compute, also known as the **streaming** paradigm. There are many frameworks that facilitate streaming, Apache Storm being a popular one. Apache Storm integrates with Hadoop YARN, bringing the streaming paradigm to Hadoop. In this chapter, we will cover the following topics:

- Comparing and contrasting a batch-processing paradigm such as MapReduce with the streaming paradigm
- The key concepts of Apache Storm
- Walking through application development using Apache Storm
- Walking through Apache Storm installation on Hadoop by running YARN

Batch processing versus streaming

MapReduce is a batch-processing model. The data is allowed to accumulate before processing is done on it. This leads to larger turnaround times. It can also lead to pressures on storage, memory, and compute resources of the system. A batch of data needs to be staged till analysis begins and ends, thus occupying storage resources. Analyzing a large piece of data will mean a peak load for a short amount of time on the nodes of the compute cluster.

Batch models also lead to poor utilization of the cluster resources. During data accumulation, the cluster compute and memory are idle. However, during analysis, they have peak load. Provisioning of the cluster must cater to the peak load.

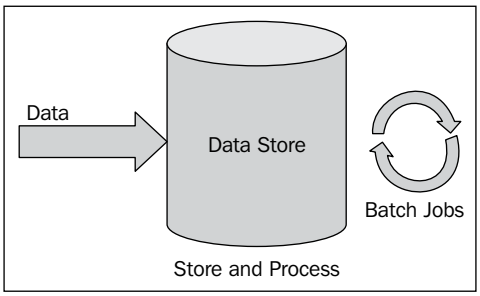
The disadvantages of batch-processing systems are overcome by using **streaming computation models**. Instead of moving the computation to the data, data is streamed through computation nodes. Each compute node operates on the data point or a small window of data to analyze and output a result or update its internal state. The computation nodes form a topology that can be viewed as a continuously executing query. The turnaround time is now smaller because analysis is not done at the end of the batch, but continuously. Also, the analysis is done on very small sets of data. These systems provide near real-time analysis, that is, analysis as soon as the data is received by the system.

Let's illustrate this with an example. Let's say we are given a task to bucket vehicles passing at a point on the interstate based on the color of the vehicle and report the count of vehicles per color in one hour. In the batch approach, this will involve stopping all the vehicles that come in during the hour and parking them in a chartered parking lot nearby. At the end of one hour, we get to the parking lot and take the count of the vehicles for each color.

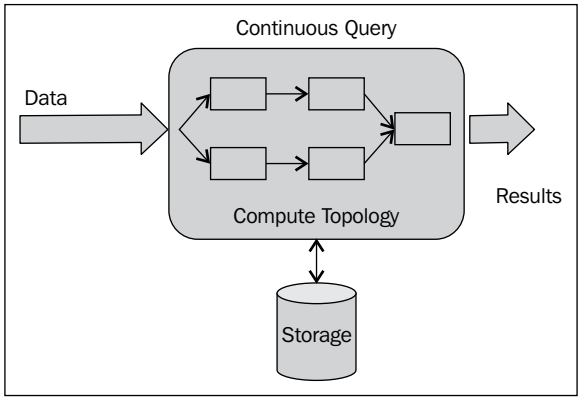
By contrast, the streaming approach will involve determining the color of the vehicle passing by and incrementing a counter based on the color. At the end of the hour, we have the counts per color. On the downside, if we make an error in counting or determining the color, we cannot recover from it. In the batch approach, we have the opportunity to double-check our computation.

Streaming systems thrive in applications where low latency is one of the primary goals. However, they come with a set of tradeoffs when accomplishing this goal. The ability to make decisions by seeing a few points makes accuracies of the results suffer. Also, not every analysis algorithm can be implemented using the streaming model. Streaming models are not suitable for algorithms that make multiple passes over the data and cannot be used if a decision can be made only by looking at the entire dataset.

The following diagrams contrast batch and stream processing of data. The first diagram shows batch processing:



The following is a figure for stream processing of data:



Apache Storm

Apache Storm is one of the most popular open source streaming engines and provides real-time analytics on unbounded streams of data. It is a distributed framework that can work on multiple nodes, providing both fault tolerance and horizontal scalability. Another primary feature of Apache Storm is that it provides guaranteed event processing, that is, every event that enters the system is processed without a loss of events. Apache Storm applications can be deployed with any programming language of the developer's choice, making it extremely attractive for usage in low-latency analytics.

MapReduce provides the Map and Reduce function primitives to build batch applications. Similarly, Storm provides its own set of primitives to support real-time analytics. If a framework such as Apache Storm is not available, writing real-time applications will mean a lot of complexity. It will involve adding and maintaining processing queues to guarantee processing of all events in the system. It will also involve writing worker programs that can read off the queue, do processing, and then enqueue the program elsewhere for downstream processing. Failure handling and synchronization between the workers and queues will be the onus of the developer.

Maintaining queues and workers will involve much greater effort than the data processing logic itself. Partitioning of the data stream when the throughput is extremely high will add additional overhead on the developer's time and effort, posing a threat on the scalability of the system. Providing fault tolerance is not an easy task too. Apache Storm tries to abstract these complexities and provide features that improve the reliability and availability of a real-time system.

Architecture of an Apache Storm cluster

A Storm cluster runs long-standing queries instead of jobs. Batch-processing systems, on the other hand, have jobs as their basic units. The key difference is that jobs finish executing eventually, but long-standing queries do not finish (unless explicitly killed). These long-standing queries are called **topologies**.

These are the two different kinds of nodes in a Storm cluster:

- **Master node**: This runs the Nimbus daemon, similar in function to a JobTracker in Hadoop MRv1
- **Worker node**: This runs the Supervisor daemon, similar in function to a TaskTracker in Hadoop MRv1

The Master node is the central node that takes care of three key functions. They are as follows:

- Distributing the code across the different worker nodes in the cluster for execution
- Scheduling by assigning tasks to available worker nodes within the Apache Storm cluster
- Monitoring for failures within the cluster and taking actions on them

The Supervisor daemon is present per node in the cluster. Its responsibilities are as follows:

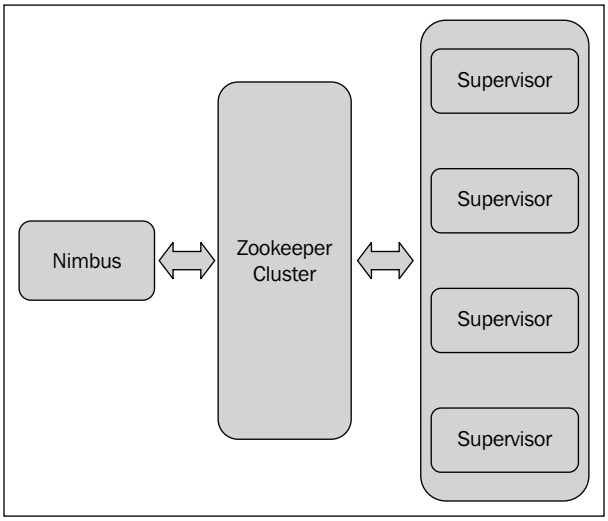
- Listening to directions from the Master Nimbus daemon
- Starting and stopping worker processes based on the directions from Nimbus; each worker process executes a subset of the topology

The actual coordination between the Nimbus and Supervisor daemons happens through a Zookeeper cluster.



Zookeeper is an open source service that takes care of configuration management, synchronization of nodes, and naming services within a distributed system. Though it was under the Hadoop project initially, it is now a top-level project within the Apache Software Foundation.

The following diagram shows a high-level view of an Apache Storm cluster:



Computation and data modeling in Apache Storm

The computation is modeled as topologies, a graph of computations. Each node in this graph of computations contains logic to process data. The links between these compute nodes in the topology indicate the nature of data transfers between the nodes.

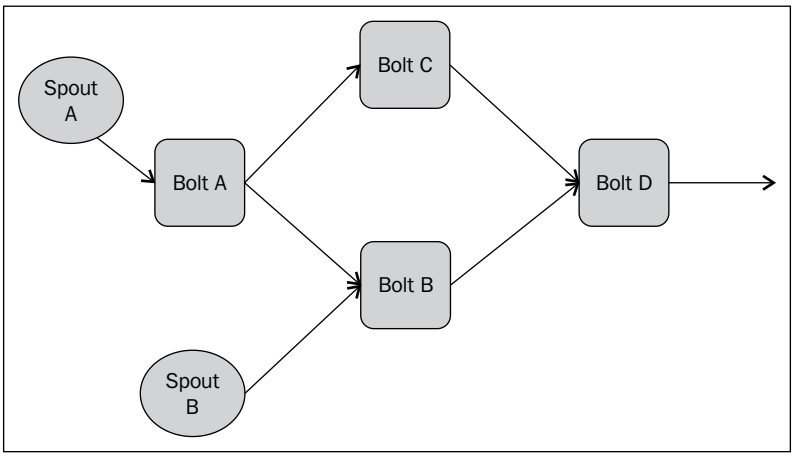
Topologies are defined as Thrift structs. In *Chapter 5*, *Serialization and Hadoop I/O*, we saw that Avro schemas can be defined in a data definition language such as JSON, making it agnostic to any programming language. Similarly, Thrift allows for topologies to be defined using its own **Interface Definition Language** (**IDL**), making the Apache Storm language agnostic.

Data abstraction used in Apache Storm is known as a **stream**. A Stream is a sequence of tuples that is unbounded. The Apache Storm cluster takes in one Stream and produces another Stream. For example, if a vehicle passing along the interstate is encoded as a tuple, in the example we used to compare batch and stream processing, the input stream contains a sequence of vehicle tuples. The output stream is a set of tuples giving us the count of vehicles per color. The output stream is unbounded, but the tuples come out of the cluster every hour. The Storm cluster transforms the input stream to produce an output stream. The number of streams that are ingested or egested to and from Apache Storm can be many.

There are three abstractions in Storm: spouts, bolts, and topologies, which are explained as follows:

- **Spouts**: A spout can be visualized as a data adapter, which converts the data source into a Stream that can be processed by Storm. A spout is the source for all the streams used within Storm. For example, a spout can connect to the Twitter API and produce a stream of tweets, or it can connect to a Kafka queue and produce a stream of system logs where each log entry is a tuple.
- **Bolts**: A bolt is a primitive that consumes multiple streams from the spouts and other bolts to emit new streams of data. A single topology in a Storm cluster might require a number of interconnected bolts to achieve the desired transformation. Bolts can do many kinds of transformations such as filtering, aggregation, stream joins, writing to data stores, or simple function execution. A bolt can subscribe to a set of streams from spouts or other bolts. This subscription sets up the links in the topology.
- **Topology**: A topology represents a network of spouts and bolts, with each edge in the network representing a bolt subscribing to the output stream of some other spout or bolt. A topology is an arbitrarily complex multistage stream computation. Topologies run indefinitely when deployed.

The following diagram illustrates a possible topology. **Bolt A** subscribes to **Spout A** and **Bolt C** subscribes to **Bolt A**. **Bolt B** is an example of a bolt subscribing to two streams, one stream from **Spout B** and another from **Bolt A**. Again, **Bolt D** subscribes to streams from **Bolt C** and **Bolt B**:



Use cases for Apache Storm

Streaming frameworks such as Apache Storm has a number of use cases. Some of the practical ones are as follows:

- **Algorithmic trading in stock markets**: Algorithmic trading requires low-latency decisions based on the performance of the stock, the market, and even external conditions such as events. Storm can parallelize decision-making and deliver distributed low-latency results.
- **Analytics from social network feeds**: Social networks such as Twitter and Facebook have a continuous stream of updates flowing into the system. A lot of analysis needs to be done at real time. For example, trending topics from Twitter feeds are low-latency applications. Trends change quickly and need to be reported as it happens.
- **Smart advertising**: Advertising is a major revenue generator for Internet companies and search engines. Advertisements have higher click-through rates if they are relevant to the user's browsing context. Smart selections and placements of advertisements is another application where frameworks such as Apache Storm can add a lot of value. Smart ads involve inferring the users' intent in real time.

- **Location-based applications**: A user's location can be used to target advertising, promotions, and services. This is a low-latency application as the target user might be at a location briefly. Storm can gather location data and make targeting happen in real time.
- **Sensor network-based applications**: Explosions of sensors in areas such as manufacturing, disaster monitoring, security, and so on has made it necessary to react and respond to anomalous events in real time. A sensor that can sense a disaster, such as an earthquake, can notify officials in real time, which can help save lives by timely evacuations and safety measures.

Developing with Apache Storm

Now, let's develop an Apache Storm topology using Java. Let's take `worldcitiespop.txt`, a CSV file containing city information along with the country code, population, and latitude/longitude information of each city. Though the example might not be an ideal streaming application, as the stream is bounded, it illustrates two patterns of stream filtering and grouping:

1. Let's develop a simple spout that reads each line from the `worldcitiespop.txt` CSV file and emits records as a stream of tuples. All spouts implement the `IRichSpout` interface. The abstract class, `BaseRichSpout`, also implements this interface. We will extend this class and override the abstract methods, as shown in the following code snippet. The three main methods to be overridden are:
    - `open`: This method is used to initialize and start the spout. In our case, we open the file and store a handle to the output collector. The `SpoutOutputCollector` object is used to emit the tuples into the output stream. The key feature of the `SpoutOutputCollector` object is that the messages can be tagged with IDs to acknowledge or fail the messages. The configuration information is also available in this method.
    - `nextTuple`: This method is called repeatedly to request the spout to emit tuples. This is a nonblocking call, that is, if there is no tuple to be emitted, the call will return. In the following example, we read one line of the file and emit it to the `SpoutOutputCollector` object. A `Values` object is used to package the tuple.

- `declareOutputFields`: This method is used to specify the schema and IDs for the message. In the following example, we indicate that our tuple has a single field, `city`, through the `Fields` object:

```
package MasteringStorm;

import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Map;


public class ReadCitySpout extends BaseRichSpout {

    private SpoutOutputCollector spoutOutputCollector;
    private BufferedReader cityFileReader;

    @Override
    public void open(Map map, TopologyContext
        topologyContext, SpoutOutputCollector
            spoutOutputCollector) {
         this.spoutOutputCollector = spoutOutputCollector;

         try{
              cityFileReader = new BufferedReader(new
                 FileReader((String)map.get("city.file")));
         }
         catch(Exception ex){
             ex.printStackTrace();
         }

    }

    @Override
    public void nextTuple() {

        String city = null;
```

```
            if(cityFileReader != null){

                try {
                    city = cityFileReader.readLine();
                }
                catch(Exception ex){
                    ex.printStackTrace();
                }

            }


            if(city != null){
                spoutOutputCollector.emit(new Values(city));
            }

        }

        @Override
        public void declareOutputFields(OutputFieldsDeclarer
            outputFieldsDeclarer) {
              outputFieldsDeclarer.declare(new Fields("city"));
        }
    }
```

2. Now that we have a stream of city tuples, we want to filter tuples that contain a valid population entry. If the `worldcitiespop.txt` file is observed closely, many cities do not have a population entry. We will filter out these records using bolts. The preceding code snippet illustrates the filtering pattern.

   All bolts have to implement the `IRichBolt` interface. The abstract `BaseRichBolt` class is extended, and three methods are overridden here too, which are as follows:

   - `prepare`: This method is used to initialize the bolt and make it ready to accept streams. We store the `OutputCollector` object so that we can write to the output stream.
   - `execute`: This method is the override where all the processing logic for the tuple goes. In the preceding code, we extract the `city` field from the tuple and split it on the comma character. We check for the population field of the record. If it is empty or throws `NumberFormatException`, we discard the tuple by not emitting anything.

- `declareOutputFields`: This method is again used to specify the schema of the output tuple. This time, our tuple has two fields, `countryCode` and `city`. The former contains the country code, and the latter contains the entire city record.

```
package MasteringStorm;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;

import java.util.Map;

public class FilterCityBolt extends BaseRichBolt {

    OutputCollector collector;

    @Override
    public void prepare(Map map, TopologyContext
        topologyContext, OutputCollector outputCollector) {
         this.collector = outputCollector;
    }

    @Override
    public void execute(Tuple tuple) {

        String city = tuple.getString(0);

        String[] tokens = city.split(",");

        //Filter cities that have a population number.
        if(tokens != null && tokens.length >= 7 &&
            tokens[4] != null && tokens[4].length() > 0){

            try {
                Long population =
                    Long.parseLong(tokens[4]);
            }
            catch(NumberFormatException ex){
                city = null;
            }

            if(city != null)
```

```
                    collector.emit(new Values(tokens[0],city));
                }

            }

            @Override
            public void declareOutputFields(OutputFieldsDeclarer
                outputFieldsDeclarer) {
                outputFieldsDeclarer.declare(new
                    Fields("countryCode","city"));

            }
        }
```

3. Our next bolt is one that sums up all the city populations to get the country's population. The following code snippet exhibits this. As before, we extend the `BaseRichBolt` abstract class. In the `prepare` method, we initialize `HashMap` to store intermediate population values for each country. The `execute` method updates these intermediate population values and emits a tuple with the country code and the intermediate population value:

```
package MasteringStorm;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;

import java.util.HashMap;
import java.util.Map;


public class SumPopulationForCountryBolt extends
    BaseRichBolt {

    private HashMap<String, Long> countryCodePopulationMap;
    private OutputCollector outputCollector;


    @Override
    public void prepare(Map map, TopologyContext
        topologyContext, OutputCollector outputCollector) {

        this.outputCollector = outputCollector;
```

```
        this.countryCodePopulationMap = new HashMap<>();
    }

    @Override
    public void execute(Tuple tuple) {

        String countryCode = tuple.getString(0);
        String city = tuple.getString(1);
        String[] tokens = city.split(",");
        Long population = Long.parseLong(tokens[4]);

        if(countryCodePopulationMap.containsKey(countryCode)) {
            Long savedPopulation =
                countryCodePopulationMap.get(tokens[0]);
            population += savedPopulation;
            countryCodePopulationMap.remove(countryCode);
        }

        countryCodePopulationMap.put(countryCode,
            population);
        outputCollector.emit(new Values(countryCode,
            population));

    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer
        outputFieldsDeclarer) {
        outputFieldsDeclarer.declare(new
            Fields("countryCode", "population"));

    }
}
```

4. Now that we have a spout that emits tuples, a bolt to filter out bad tuples, and an aggregation bolt that sums up the population, let's build a topology. As we saw previously, building a topology is to set up links between the spouts and bolts. The following code illustrates the way to construct and submit a topology of spouts and bolts.

   The topology is built using the `TopologyBuilder` object. It has the methods `setSpout` and `setBolt` to set the spouts and bolts in the topology. The names of the spouts and bolts can be specified. The `setBolt` method returns a `TopologyBuilder.BoltGetter` object. This object permits different kinds of groupings. Groupings can be thought off as stream-partitioning directives.

In the preceeding code snippet, we use the `shuffleGrouping` method to link `FilterCityBolt` to `ReadCitySpout`. However, we use the `fieldsGrouping` method to link `SumPopulationForCountryBolt` to `FilterCityBolt`. This is because we want to aggregate the population by country and will like tuples of the same country to arrive at the same bolt task.

The `StormSubmitter` helper class is then used to submit the topology to the Apache Storm cluster. We use the `Config` object to specify any additional configuration values. In this case, we set `debugging` and the filename that we got from the command line.



These are the following seven stream-partitioning schemes that are native to Apache Storm:

- **Shuffle grouping**: Each tuple is randomly distributed to the Bolt tasks. This scheme tries to guarantee that each Bolt task gets an equal number of tuples.
- **Fields grouping**: Based on the value of a particular field, tuples will be sent to a single Bolt task. In the code used in this section, we group based on the `countryCode` field, which guarantees that tuples that share the same country code are sent to the same task.
- **All grouping**: Every generated tuple is duplicated and sent to all the Bolt tasks.
- **Global grouping**: The entire stream goes to a single Bolt task. This is useful when calculating global metrics.
- **None grouping**: This defaults to Shuffle grouping for now. In the future, this will give a provision to push down to execute in the same thread of the Bolt or Spout.
- **Direct grouping**: In this case, the Bolt or Spout generating the tuple decides the Bolt it goes to.
- **Local or Shuffle grouping**: If the Bolt receiving the tuple happens to have some tasks in the same worker process as the producer, the tuples will be shuffled in-process. If this is not the case, it will fallback to the Shuffle grouping.

```
package MasteringStorm;

import backtype.storm.Config;
import backtype.storm.StormSubmitter;
import backtype.storm.topology.TopologyBuilder;
```

```
import backtype.storm.tuple.Fields;

public class MasteringStormTopology {

    public static void main(String[] args){

        Config config = new Config();
        config.setDebug(true);

        if(args != null){
             System.out.println(args.length);
        }

        System.out.println(args[0]);
        config.put("city.file", args[0]);

        TopologyBuilder topologyBuilder = new
            TopologyBuilder();

        topologyBuilder.setSpout("cities", new
            ReadCitySpout(), 3);
        topologyBuilder.setBolt("filter", new
            FilterCityBolt(), 3).shuffleGrouping("cities");
        topologyBuilder.setBolt("group", new
            SumPopulationForCountryBolt(),
                3).fieldsGrouping("filter", new
                    Fields("countryCode"));

        try {
            StormSubmitter.submitTopology("test-filtering-
                storm", config,
                    topologyBuilder.createTopology());

        }
        catch(Exception ex){
            ex.printStackTrace();
        }

    }


}
```

The topology can be submitted into a Storm cluster using a command line similar to the following:

```
storm jar MasteringStormOnYarn-1.0-SNAPSHOT-jar-with-dependencies.jar
MasteringStorm.MasteringStormTopology worldcitiespop.txt
```



If you use Maven to build your topology JAR file, use the `<scope>provided</scope>` tag when specifying the Apache Storm dependency. If this is not used, the Apache Storm JAR files and `default.yaml` file are packaged into the JAR file. This leads to duplicate `default.yaml` files, which results in a runtime error. The following code snippet shows the dependency section in `pom.xml`:

```
<dependencies>
   <dependency>
      <groupId>storm</groupId>
      <artifactId>storm-core</artifactId>
      <version>0.9.0</version>
      <scope>provided</scope>
   </dependency>
</dependencies>
```

Apache Storm 0.9.1

Apache Storm 0.9.1 was released in February, 2014. It brings in a number of enhancements when compared to the older versions of Storm. The latest releases of Apache Storm can be found at `http://storm.apache.org/downloads.html`. The important enhancements are as follows:

- **Netty-based transport**: Previously, Apache Storm came with 0MQ as its transport. 0MQ needed native binaries to be installed on the cluster, which was tedious. Netty is a Java-based transport that ensures good portability across nodes in the cluster. It also comes with superior performance characteristics, providing almost two times more improvement in message throughput.
- **Windows support**: Apache Storm can now be run on the Windows platform. This is significant for the number of clusters running on Windows.
- **Apache Software Foundation**: Apache Storm is an incubator project within the Apache Software Foundation. This ensures greater community reach and provides distribution and licensing structure to the software.

- **Maven Integration**: Apache Storm is primarily written in a JVM-based Lisp language called Clojure. Leiningen used to be a popular build tool for Clojure and Storm builds as they used to be based on Leiningen. However, as Apache Storm moved as an incubator project within the Apache Software Foundation, the choice of build tool became an important feature for release management. Apache Storm now uses the Maven build system, which makes it easy to release early and often.

Storm on YARN

In *Chapter 6*, *YARN – Bringing Other Paradigms to Hadoop*, we built a YARN application that executes distributed shell commands. Storm is one such application that has been brought to YARN by Yahoo!. Any Hadoop cluster running YARN can now execute streaming workloads for low-latency real-time applications. The Application Master and client program to execute Storm are available for deployment. It is open sourced on GitHub at `https://github.com/yahoo/storm-yarn`.

Installing Apache Storm-on-YARN

Apache Storm-on-YARN can currently be installed from GitHub. This section assumes that the Hadoop 2.2.0 cluster is available.

Prerequisites

The following prerequisites are necessary to install Storm-on-YARN:

- Java 7
- Maven: This needs to be installed on the gateway machine to help compile and deploy the Storm-on-YARN Application Master and client:
    - `wget http://mirror.symnds.com/software/Apache/maven/maven-3/3.1.1/binaries/apache-maven-3.1.1-bin.tar.gz`
    - `tar –zxvf apache-maven-3.1.1-bin.tar.gz`
    - `mkdir -p /usr/lib/maven`
    - `mv apache-maven-3.1.1 /usr/lib/maven`
    - `vi ~/.bash_profile and add $PATH=$PATH:/usr/lib/maven/bin`

Installation procedure

Perform the following steps for installation:

1. A copy of the Storm-YARN repository can be downloaded from GitHub. If you have Git installed, the repository can be cloned locally:

   ```
   wget https://github.com/yahoo/storm-yarn/archive/master.zip
   ```

2. The downloaded `master.zip` file needs to be decompressed:

   ```
   unzip master.zip
   ```

3. Now, the Maven config file, `pom.xml`, has to be edited to reflect the Hadoop version being used. We are using 2.2.0, so mention it in the `hadoop.version` XML tag:

   ```
   <properties>
         <storm.version>0.9.0-wip21</storm.version>
         <hadoop.version>2.2.0</hadoop.version>
         <!--hadoop.version>2.1.0.2.0.5.0-67</hadoop.version-->
     </properties>
   ```

4. The Storm-YARN project comes with Storm binaries in the `lib` directory of the downloaded project files:

   ```
   mkdir ~/working-dir
   ```

5. Go to the `storm-yarn-master` directory:

   ```
   cd storm-yarn-master
   cp lib/storm.zip ~/working-dir
   ```

6. The `storm.zip` file has to be placed within HDFS so that it can be deployed on all the nodes of the Hadoop cluster. For now, this path is hardcoded to `/lib/storm/<storm-version>` within the Storm-YARN AM:

   ```
   hadoop fs -mkdir /lib
   hadoop fs -mkdir /lib/storm
   hadoop fs -mkdir /lib/storm/0.9.0-wip21
   hadoop fs -put storm.zip /lib/storm/0.9.0-wip21
   ```

7. The `storm.zip` file has to be unpacked in the working directory and added to the path:

```
unzip storm.zip
vi ~/.bash_profile
```

8. Add the `storm-yarn-master` and `storm` bin paths into the `PATH` environment variable:

```
export STORM_HOME="<your path>/working-dir/storm-0.9.0-wip21"
export STORM_YARN_HOME="<your path>/storm-yarn-master"
export PATH=$PATH:$STORM_HOME/bin:$STORM_YARN_HOME/bin
```

9. Go to `storm-yarn-master` and execute the Maven package command:

```
cd storm-yarn-master
mvn package
```

10. The Maven package command will build the Application Master and Client. Additionally, it will run tests to verify that everything went well. It is highly recommended that tests be run to catch any issues early. You can skip tests by running the following command:

```
mvn package -DskipTests
```

11. As we saw in the section on Storm architecture, Zookeeper is used to coordinate communication between the Nimbus and Supervisor daemons. It is important to install a Zookeeper cluster before starting off on Storm:

```
wget
http://www.gtlib.gatech.edu/pub/apache/zookeeper/zookeeper-
3.4.6/zookeeper-3.4.6.tar.gz
```

12. You need to unzip and untar the package and store it in a location of your choice:

```
tar zxvf zookeeper-3.4.6.tar.gz
```

13. Zookeeper stores all configuration information on disk. Go to the `conf` folder of the Zookeeper installation, create a `zoo.cfg` file from the template that is given to you, and check the settings. Most importantly, the directory pointed by the `dataDir` setting must exist:

    ```
    cd conf
    cp zoo_sample.cfg zoo.cfg
    vi zoo.cfg
    ```

    The Zookeeper configuration on my machine is given as follows:

    ```
    # The number of milliseconds of each tick
    tickTime=2000
    # The number of ticks that the initial
    # synchronization phase can take
    initLimit=10
    # The number of ticks that can pass between
    # sending a request and getting an acknowledgement
    syncLimit=5
    # the directory where the snapshot is stored.
    # do not use /tmp for storage, /tmp here is just
    # example sakes.
    dataDir=/data/zookeeper
    # the port at which the clients will connect
    clientPort=2181
    ```

To start Zookeeper, go to the `bin` folder and execute the following command:

```
cd bin
./zkServer.sh start
```

Once Zookeeper is started, it is time to submit the Apache Storm application to our Hadoop YARN cluster. This can be done using the following command:

```
storm-yarn launch
```

It can take a few minutes to launch. To check if all the necessary services are running, you can execute the `jps` command:

```
jps
```

The following screenshot shows all the necessary services that need to run:



The **QuorumPeerMain** service is the Zookeeper service. Since the cluster I am running on has a single node, we see a single supervisor. The Nimbus daemon is also visible. The MasterServer is the Application Master for Storm. It is the container that spawns Nimbus. There are a number of worker processes. These are the actual topologies that run in Storm. We will shortly see how to run a topology. For now, you might not be able to see the worker processes.

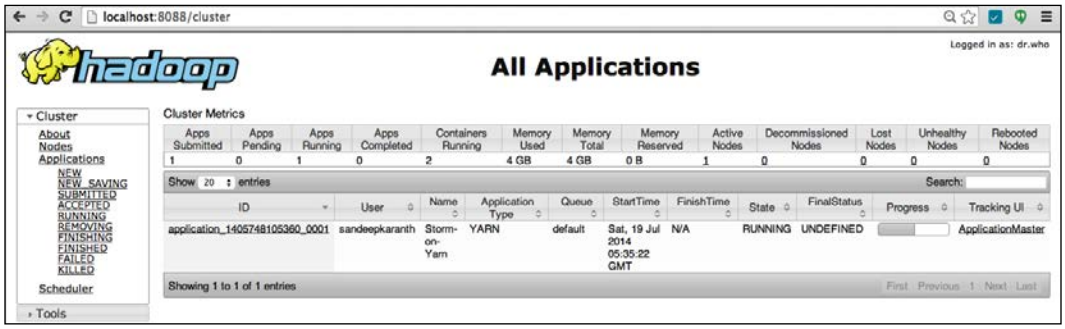
You can check to see the application ID within YARN using the following command:

```
yarn application -list
```

The following screenshot gives the output of the `list` command from YARN RM. The application ID for the Storm application is `application_1404566721714_0004` in this case:



You can also check the application ID by connecting to the RM web UI, as shown in the following screenshot:



The Apache Storm configuration can be stored with the `.storm` directory in the user's home. In this way, whenever topologies are submitted to the cluster, Storm can automatically pick the configuration from this place:

```
storm-yarn getStormConfig –appId <appId from YARN> -output ~/.storm/
storm.yaml
```

The `storm.yaml` file has the configuration of the Apache Storm cluster. It is important to review it for the correct information. For example, the `nimbus.host` property shows the host where the Nimbus daemon is running. All the Zookeeper configurations, timeouts, and other properties can be set in this config file.

Now that the Apache Storm cluster is up and running, it is time to submit some topologies for testing. Storm YARN comes with a couple of test topologies, such as `storm.starter.WordCountTopology` and `storm.starter.ExclamationTopology`. These can be run to check whether the cluster is deployed correctly.

To run the topologies, use the following command:

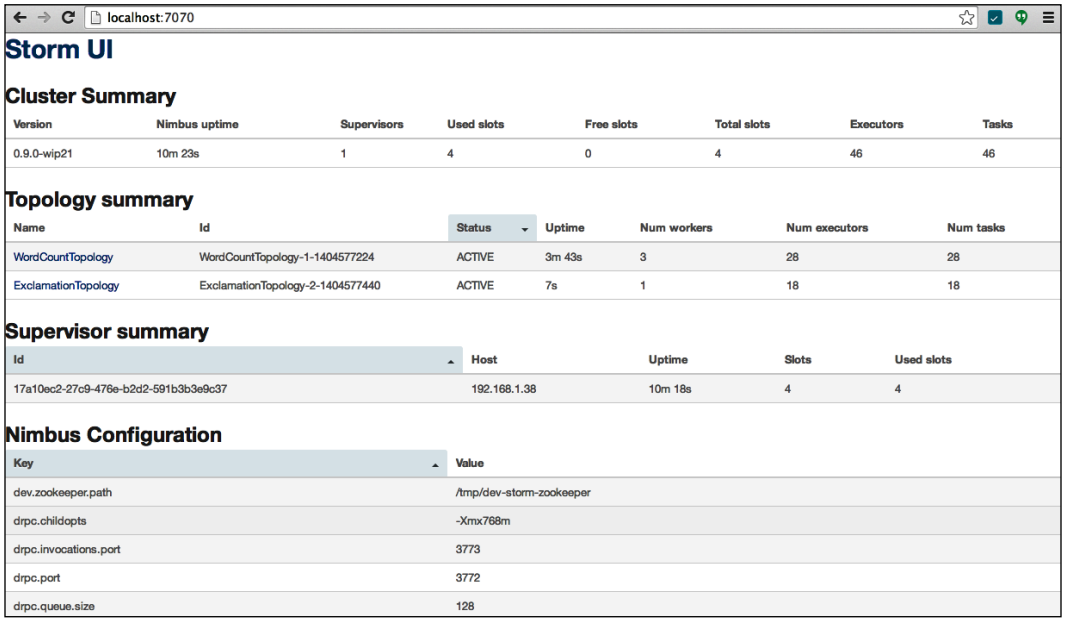
```
storm jar storm-starter-0.0.1-SNAPSHOT.jar
storm.starter.WordCountTopology
```

Also run the following command:

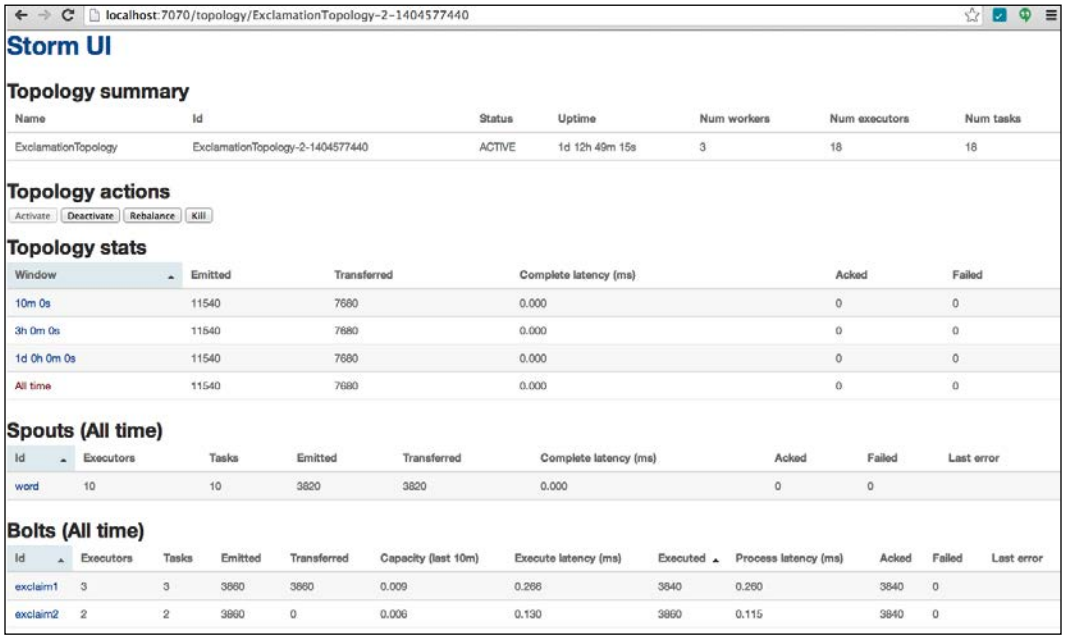
```
storm jar storm-starter-0.0.1-SNAPSHOT.jar
storm.starter.ExclamationTopology
```

Apache Storm comes with its Web UI to monitor topologies. The web endpoint is on the gateway machine at port 7070. The following screenshot shows a Storm cluster running both topologies.

The `jps` command execution on each node can be used to check whether worker processes are running on the nodes.

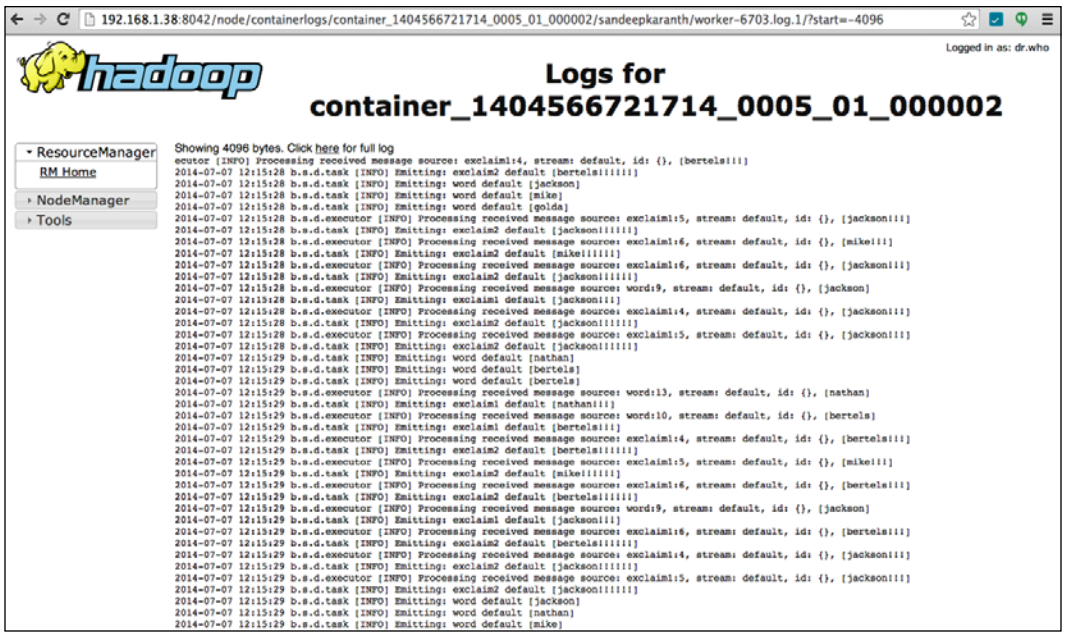


Clicking on a particular topology gives us its details. The following screenshot gives the details of `ExclamationTopology` running on the cluster. It provides the details of the number of bolts and spouts that are running. Actions can be taken on a particular topology as well. The currently permitted actions are to activate/deactivate, kill, and rebalance the topology.



The following screenshot gives a snapshot of the logs from `ExclamationTopology`. These logs were gathered from a container that was running the topology. `ExclamationTopology` adds the string `!!!` to every word that was generated by the spout. The spout, in this case, selects a word randomly from a list of words and sends it to the bolts. The `exclaim1` bolt adds the three exclamation marks to the word and forwards it to the `exclaim2` bolt that adds three more marks.

In the logs, the `exclaim1` output always has three exclamations, while the `exclaim2` output has six exclamations. The word `spout` emits the word randomly from a set of words defined in the spout.



There are Apache Storm commands to kill a topology instead of using the Web UI. The killing process takes place in several steps. First, the spouts in the topology are deactivated. Then, Apache Storm waits for a timeout before it can terminate the workers and clean up all the states. During this interval, workers can complete processing the tuples they have received. The `-w` flag can be used to vary this particular timeout:

```
storm kill <topology-name> [-w wait_for_seconds]
```

Similarly, Apache Storm provides other commands to manage the topologies. We previously encountered the `jar` command that can be used to instantiate topologies within the cluster. The key here is that it automatically picks up the Storm configuration from the `.storm` directory within `home`.

The entire Apache Storm application can be stopped on the cluster using the `Storm-YARN shutdown` command. The usage is given as follows:

```
storm-yarn shutdown -appId <application id>
```

Both Apache Storm and Storm-YARN have many other commands. The Apache Storm commands manage a deployed instance of Storm, while the Storm-YARN command talks to the RM to manage the Storm application with YARN.

A comprehensive set of Apache Storm commands is given using the following command:

```
storm help [command]
```

For a comprehensive set of Storm-YARN commands and options, you can run the following command:

```
storm-yarn help
```

Some important Storm commands are as follows:

- `activate`: This command activates a spout. It is analogous to turning on a faucet at a sink to get a stream of water. Activating a spout starts the stream of tuples.
- `deactivate`: This command deactivates a spout. It is analogous to turning off the faucet. This command stops the stream of tuples.
- `dev-zookeeper`: This particular command can be used to instantiate a `Zookeeper` cluster during development, debugging, and testing. It is used when we do tests on the Storm-on-YARN package. The properties of this transient cluster are given by the `dev.zookeeper.path` property that specifies the path for the Zookeeper data directory and the `storm.zookeeper.port` property that specifies the port of the Zookeeper process.
- `drpc`: This command launches a distributed RPC daemon. The distributed RPC is a special pattern that is possible via Storm. RPC calls that are intensive can be parallelized using an Apache Storm topology. The function name and arguments form the tuple in the input stream.
- `list`: This command lists the topologies running in the Apache Storm cluster.



- `localconfvalue`: This command prints out the configuration value for a specified property. This value is taken from the `storm.yaml` file in the `.storm` directory merged with the properties from the `default.yaml` file.
- `logviewer`: This command launches a web endpoint from where logs can be viewed.
- `nimbus`: This command launches a Nimbus daemon.
- `rebalance`: Addition of nodes to the cluster might call for redistributing the workload in the cluster. This can be done in two ways; the first is to kill and restart the topology, and the second is to use the `rebalance` command. The `rebalance` command first deactivates the spouts in the system, then redistributes the workload and activates the spouts. The `rebalance` command can also be used to modify the parallelism of the workers in the cluster.
- `remoteconfvalue`: This command prints the configuration value for a property on a cluster machine. The path to the `storm.yaml` file is `$STORM-PATH/conf/storm.yaml`. Again, it merges the properties with the `default.yaml` file.
- `supervisor`: This command launches the supervisor daemon.
- `ui`: This command launches the UI daemon. The port number of the endpoint is given in the `storm.yaml` file.

Summary

The number one goal of the streaming paradigm is to cater to low-latency applications. The Storm-on-YARN project has brought this paradigm to Hadoop. Stakeholders can now multiplex streaming and batch processing on a single Hadoop cluster and cater to different kinds of applications.

There are a number of streaming frameworks available, such as Microsoft SQL Server StreamInsight, S4, and Apache Storm, among others. Apache Storm is open source, part of the Apache Software Foundation, Hadoop integrated, and has a large community behind it, making it attractive for distributed stream processing.

Some key takeaways from this chapter are as follows:

- The basic data model in Apache Storm is an unbounded sequence of tuples called Streams.
- Long-standing queries are modeled as computational topologies. The data stream flows through these topologies.
- Apache Storm provides the following primitives:
    - Spouts: They convert input data into streams
    - Bolts: They take an input stream, do some processing on the stream, and output another stream
- Apache Storm provides guaranteed message delivery and fault tolerance in a distributed setting.
- Apache Storm uses Thrift to specify topologies, making it language-agnostic.
- Storm-on-YARN is an open source under-development project initiated by Yahoo! to bring Apache Storm to Hadoop clusters running YARN.

In the next chapter, we will look at Hadoop's cloud support, particularly the one by Amazon, which is one of the biggest cloud vendors.

8
Hadoop on the Cloud

Cloud computing is the paradigm that has made computing a utility. Just like electricity grids and water supply channels bring electricity and water, to individual homes, cloud computing allows individuals and businesses—small, medium, and large—to tap into pooled computing resources connected via a network to execute desired tasks and run their applications.

In this chapter, we will be:

- Looking at the characteristics and advantages of cloud computing
- Comparing and contrasting the *Hadoop on the cloud* offering by Amazon AWS and Microsoft Azure, two of the leading players in the cloud computing space
- Delving into the details of Amazon's managed Hadoop service called **Elastic MapReduce** (**EMR**)
- Looking at how to provision an EMR cluster within minutes and run MapReduce jobs

Cloud computing characteristics

The **National Institute of Standards and Technology** (**NIST**) (`www.nist.gov`) defines the following five important characteristics as the essence of cloud computing:

- **On-demand self service**: Consumers of computing utility can provision and deprovision resources on the fly. The provisioning action can happen unilaterally in a self-serve manner without any human intervention with the service provider. For example, cloud computing has made it possible for an organization to provision a Hadoop cluster of a desired configuration from a console in their office, without having to call Amazon and letting them know about it.

- **Broad network access**: The self-service part of cloud computing is facilitated via the Internet, allowing heterogeneous clients ranging from mobile phones to desktop computers to interact with the cloud computing service. Standard communication protocols such as HTTP are used to allow communication between these diverse sets of clients and the service provider.
- **Resource pooling**: The entire setup is multitenant, that is, a number of consumers operate their workloads on a pooled set of computing resources of the service provider. Dynamic adjustments of physical resources among the consumers happen as the demand varies.
- **Rapid elasticity**: The resources utilized by a single consumer can be scaled outward and inward as demanded in a very short amount of time. Most cloud computing providers also provide an auto-scaling feature, where the resources scale outwards or inwards based on a set of conditions defined by the consumer. A feeling of infinite capacity is given to the consumer using the cloud-based service.
- **Measured service**: The cloud service provider measures, monitors, and reports the usage of each of its tenants. The measurement is transparent to the consumer and is used for billing. The cloud computing service always follows a pay-as-you-go model, where the consumer pays only for what has been used. For example, if a consumer provisions a Hadoop cluster of three nodes, runs a MapReduce job for one hour, and turns the cluster off once the job is done, the cloud computing service provider bills the consumer only for one hour of compute time on three nodes.

The reasons why the cloud makes sense to run Hadoop clusters are as follows:

- **Lower costs**: Operationalizing analytic workloads in an organization is always the last step in the data processing pipeline. Before operationalizing, iterations of refinements happen in the analyses and trials on different datasets. During the pre-operationalization phase, it is not prudent to provision an in-house Hadoop cluster, which involves a huge capital expenditure. Any in-house cluster that is provisioned will either be overprovisioned or underprovisioned at this stage. With the cloud, this situation is alleviated as organizations can lease out clusters based on their needs, without having to pay for the capital costs of the cluster. Also, traditionally, it will take months to set up the hardware and software for such a cluster. With the cloud model, this can happen in minutes.
- **Elasticity**: Experimentation and prototyping involves variable workloads. The cloud infrastructure is known for its elasticity. Hadoop clusters of different sizes can be provisioned, and nodes added or removed based on the requirement of the job. The inward or outward expansion of the cluster can be done dynamically too.

- **Administration**: The self-service model of the cloud makes it easy to administer and maintain a cluster. This has significant bearing not only in terms of administrative costs but also in terms of time taken to recover from failures.

Cloud-based software can be categorized into three service models:

- **Infrastructure as a Service** (**IaaS**): This cloud service provider provides physical or virtual machines as a service.
- **Platform as a Service** (**PaaS**): This cloud service provider provides a computing platform as a service. A computing platform can be an execution runtime, a database, a Hadoop cluster, or a web server.
- **Software as a Service** (**SaaS**): This cloud service provider provides a software application as a service.

As we move from IaaS to SaaS, the flexibility of application configuration decreases with the decreasing cost of service. Hadoop on the cloud is a PaaS offering. It is sometimes termed **Hadoop as a Service** (**HaaS**). This distributed computing framework along with HDFS is available for a consumer as a service.

Hadoop on the cloud

All major cloud service providers have Hadoop as a PaaS offering. Amazon with Elastic MapReduce, Microsoft with their HDInsight offering, and Google with their Hadoop on the Google Cloud platform are the frontrunners in this space. The first to offer Hadoop on the cloud, way back in 2009, was Amazon.

We will briefly compare and contrast EMR and HDInsight in the following table:

Amazon AWS EMR	**Microsoft Azure HDInsight**
It was released in 2009. It has more than five years of service and technology maturity.	It was released in 2012. It has around two years of service and technology maturity.
The popularity of AWS makes the learning curve less steep for a new user. EMR is integrated with the popular AWS console.	Microsoft Azure is picking up, but it's not yet as popular as AWS. HDInsight is also integrated with Microsoft Azure dashboards.
It has the ability to deploy clusters with the MapR Hadoop distribution.	Hadoop distribution is limited to Microsoft's distribution in partnership with Hortonworks.
No Microsoft Windows support is available.	Hadoop distribution is tailored to run on Microsoft Windows.

Amazon AWS EMR	Microsoft Azure HDInsight
It is marginally less expensive.	It is slightly more expensive when compared to EMR.
It lags behind in end-user tooling.	It integrates better with the Microsoft Office suite. For example, it provides a Hive ODBC driver and Hive Excel add-on for frontend analysis and visualization.
It provides native support for Java, Pig, and Hive. Other executables/scripts can be executed using Hadoop Streaming.	It provides native support for C#, Java, Javascript, Pig, and Hive. Other executables/scripts can be executed using Hadoop Streaming.

Amazon Elastic MapReduce (EMR)

Amazon AWS offers Hadoop as a PaaS. Organizations and individuals can provision Hadoop clusters on the fly, run their workloads, and download results. Provisioning a Hadoop cluster using EMR takes a few minutes and is a few clicks away.



Usage of any Amazon Web Service requires an Amazon account. Visit `http://aws.amazon.com` and register for a free account. A credit card is mandatory to register for an Amazon account. However, it will be charged only on usage beyond the free tier offered by Amazon. The registered e-mail is subsequently used as the username.

The general steps to create and run workloads on EMR are as follows:

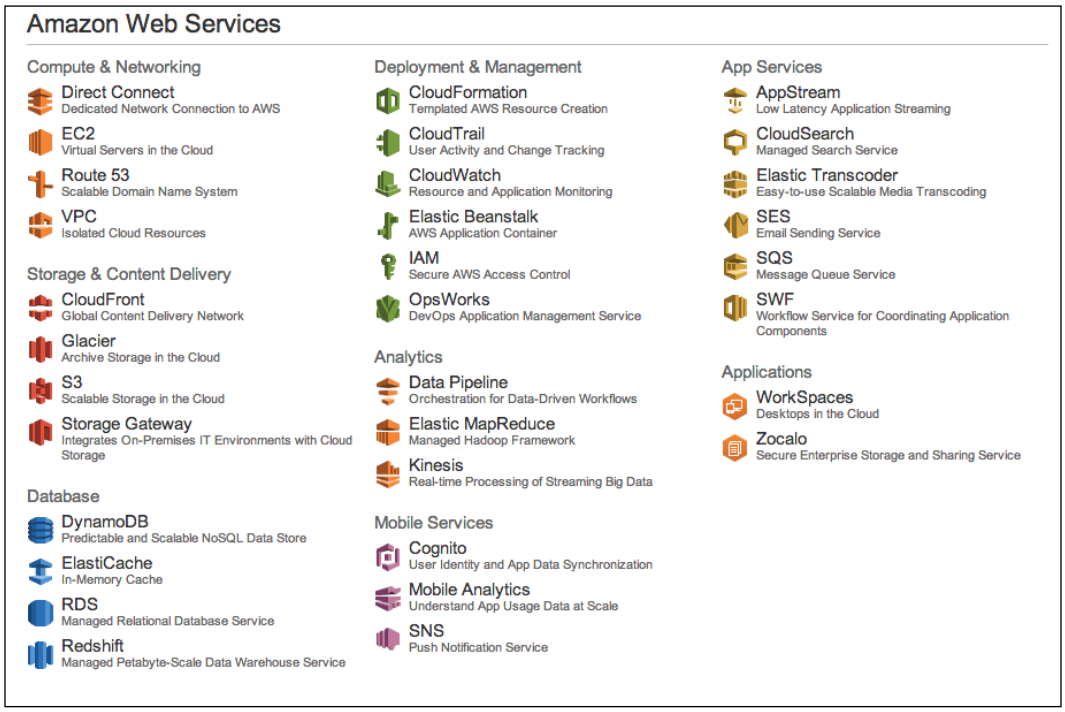
1. The application is developed locally in Java using Hadoop's MapReduce APIs, Hive, Pig, or a language of the user's choice. Non-Java-based languages can be executed in a Hadoop cluster using Hadoop Streaming. A developer guide can be found at `http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-what-is-emr.html`.
2. The application and relevant data are stored in Amazon S3. S3 is a scalable storage service provided by Amazon. There are many ways of placing data in Amazon S3. A number of clients for data upload are available, or a web interface can also be used. Data can directly be written to HDFS on the EMR cluster as well.

3. Using the management console, the cluster configuration is specified and launched. The cluster configuration includes the kind of machines in the cluster, the version of Hadoop to use, and additional applications that need to be installed on the cluster. The actions to be performed once the cluster is provisioned are also part of this step.
4. The cluster is then launched, the data processing is done, and the results are either moved into S3 or can be read directly off HDFS on the cluster.

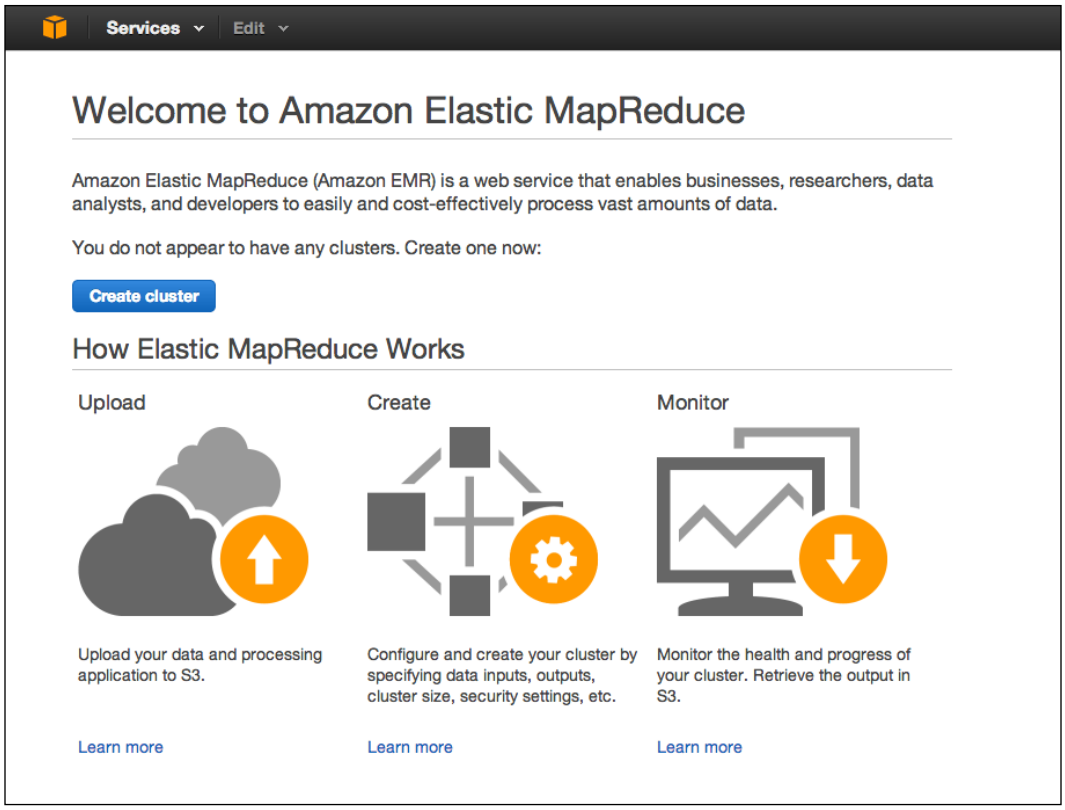
Provisioning a Hadoop cluster on EMR

To provision a Hadoop cluster on EMR, perform the following steps:

1. Once you have procured an Amazon account, visit `https://console.aws.amazon.com` and log in with your Amazon account credentials. The following screenshot shows an AWS console page. It lists all the different cloud services Amazon has to offer. The services of interest to us will be S3, a scalable storage service on the cloud, and Elastic MapReduce, a managed Hadoop service.



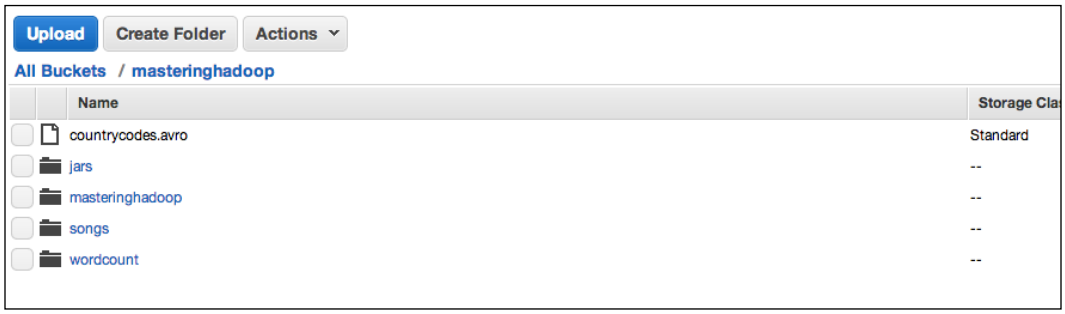
2. Clicking on the **Elastic MapReduce** link takes the user to the EMR service management page. The following screenshot shows the EMR page. It gives you a brief introduction to EMR and the main steps to launch a cluster. The **Create cluster** button is used to launch the Hadoop cluster wizard.



3. Before launching the cluster, we have to upload the data and corresponding application onto S3. Using the **Services** dropdown on the top navigation bar, a user can quickly navigate to the management console of any of Amazon's cloud services. S3 can be chosen to go to the S3 management console. The following screenshot shows the S3 management console:



4. It is mandatory to have a bucket in S3 before any file can be uploaded. Files can be uploaded to S3 using the **Upload** button once a particular bucket is selected. There is a button to create folders within the bucket. The **Actions** dropdown lists many other actions, such as copy, move, download, and setting access control on the file or folder. The files display metadata properties such as size, its storage class, and the last modified date and time of a particular file. It must be noted that the folders do not have these metadata properties as they do not have any underlying structure. Files can be uploaded to S3 using a number of file manager programs available for S3, or via the web interface shown in the following screenshot:





S3 stores files in containers called **buckets**. A bucket can be part of a single account and needs to be unique. Buckets can be assigned to different regions to cater to low-latency serving of files for users close to the region.

Buckets can contain folders and files. Folders are pseudo-structures and are used to prefix the filenames. A bucket is a flat container and does not contain any folder hierarchies. The following screenshot shows the MasteringHadoop bucket with folders and files in an S3 account.

Amazon EMR provides a number of sample jobs that can be run on the provisioned clusters. One such JAR is a Hadoop Streaming program for word count. The program is written in Python and counts the words in a document. The files whose words need to be counted, and the program, are already on a public bucket on S3 and can be used by anyone to test out EMR.



Hadoop Streaming is a utility that comes with every Hadoop distribution. It can be used to run Hadoop jobs with any executable or script. The executable can be written in any language of the developer's choice. Hadoop Streaming is particularly useful for executing legacy applications in a Hadoop environment. Hadoop Streaming should not be confused with the Stream computing paradigm we saw in the previous chapter.

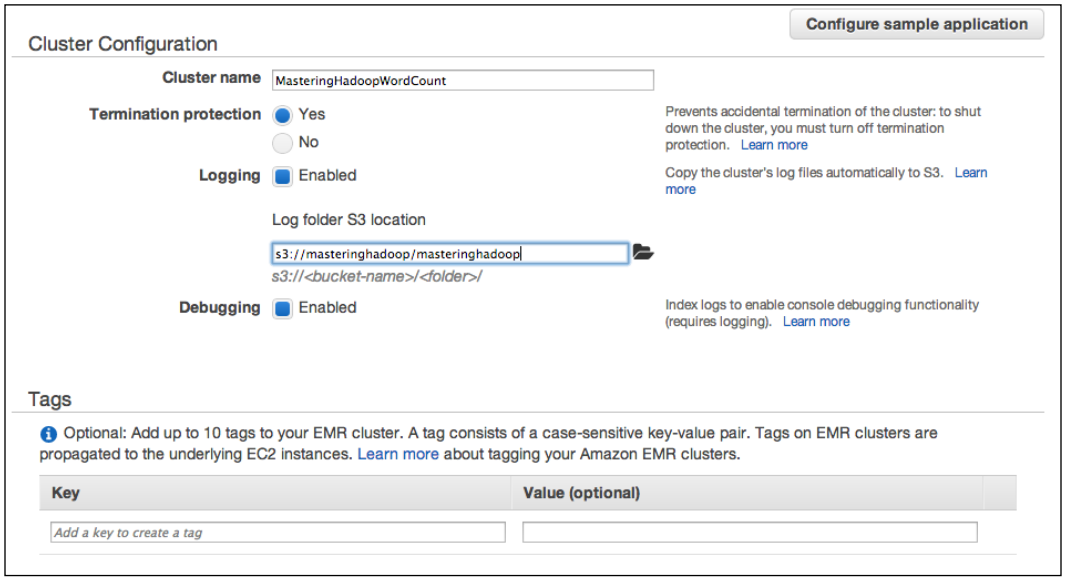
5. We will start off by going back to the EMR console using the **Services** navigation dropdown. The next step is to click on the **Create cluster** button on the console.

   The **Create Cluster** page has multiple sections. Each section configures a particular aspect of the cluster.

6. The first section is the **Cluster Configuration** section where the cluster properties are mentioned. The next screenshot shows this section on the **Create Cluster** page. Some of the properties are as follows:
   - **Cluster name:** This is a friendly name for the cluster. This name will help identify and manage a cluster based on the name. In the example, it has been set to `MasteringHadoopWordCount`.

- **Termination protection:** This is set to **Yes**, as shown in the next screenshot. When turned on, it prevents the cluster from terminating when failures are encountered. If the cluster needs to be terminated, it has to be explicitly set to **No** before termination. It is recommended to turn on this property as it might be necessary to back cluster instance data before terminating the cluster. All cluster instance data will be lost otherwise.
- **Logging:** Logging can be turned on and a S3 path can be specified to dump the logs. Logs are written on the `/mnt/var/log` directory of the Master Node. These are copied onto S3 at intervals of 5 minutes.
- **Debugging**: By enabling debugging, an index of the logfiles is created in SimpleDB.

7. The next section is the **Tags** section. Up to 10 key-value strings can be associated with the EMR cluster. These tags are persisted in the underlying EC2 instances that run the Hadoop cluster. Tags can be powerful metadata that can help classify and manage EMR clusters and EC2 instances.



8. The next sections configure the software and hardware of the EMR cluster we want to provision. The next screenshot shows the page with both these sections.

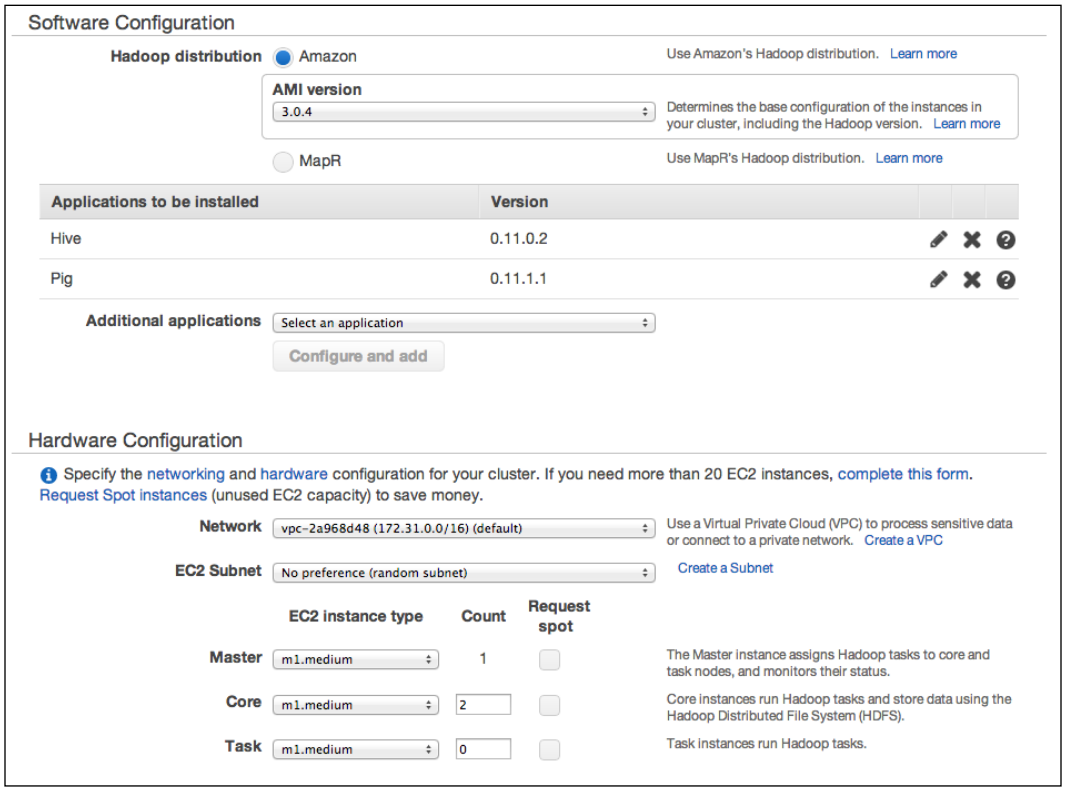
   In the **Software Configuration** section, the fields are as follows:

   - **Hadoop distribution**: The Hadoop distribution to be used in the cluster can be set here. Amazon has its own Hadoop distribution. This distribution is optimized to run on Amazon's EC2 instances. It also supports MapReduce's Hadoop distribution. The **AMI version** dropdown gives the versions of Hadoop that are present within the distribution. Hadoop versions 2.4.0, 2.2.0, 1.0.3, and 0.20.205 are present. Each version of Hadoop has different AMIs corresponding to it. In the preceding screenshot, we choose Hadoop 2.2.0 that is installed on AMI Version 3.0.4 to be consistent throughout the book.
   - **Additional applications**: Additional applications can be added to the cluster. By default, Hive and Pig are installed. If it is not required for your workloads, they can be removed. HBase, Impala, and Ganglia are three other applications that are available with this version of Hadoop.

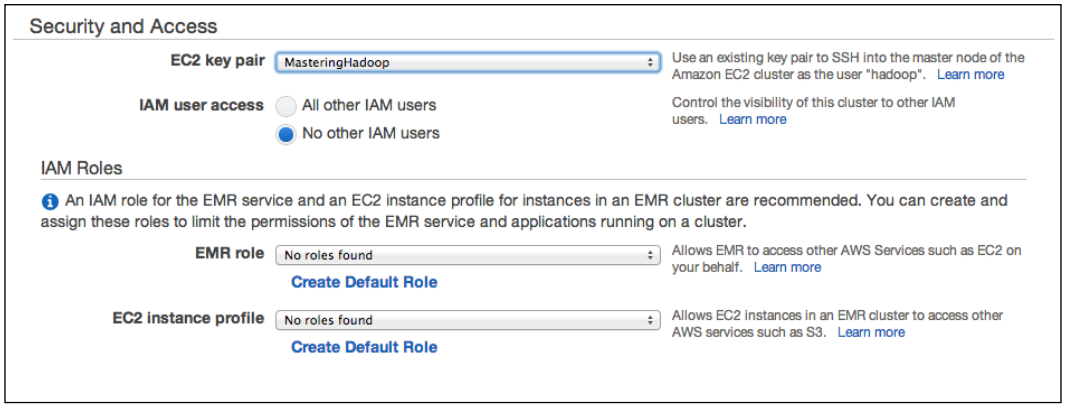
   In the **Hardware Configuration** section, the fields are as follows:

   - **Network**: A **Virtual Private Cloud** (**VPC**) can be used to connect to a private cloud to process sensitive data. As shown in the next screenshot, we will choose a default VPC.
   - **EC2 Subnet**: The EC2 subnet can be chosen using the dropdown. All subnets available within your region are shown. There is an option to choose a random subnet, as shown in the next screenshot.
   - **Instance Information**: There are three types of EC2 instances that can be specified:
   - **Master**: This EC2 instance is responsible for assigning tasks to the different cores and task nodes. There mandatorily has to be one master instance.

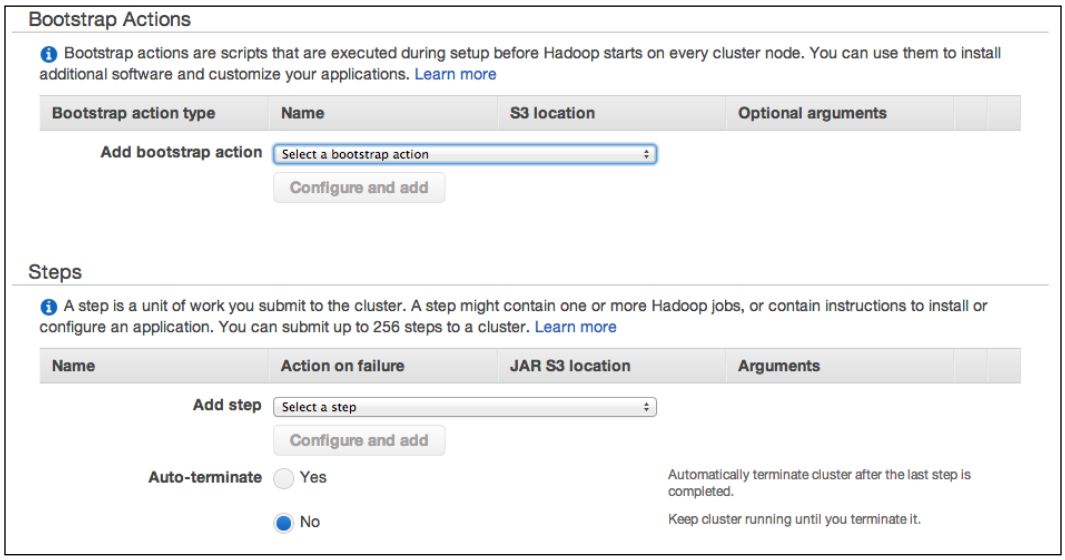
- **Core**: These are the nodes that execute tasks as well as the ones that act as data nodes. In the example, we set the number of **Core** instances to **2**. We choose the smallest possible VM, **m1.medium**. Amazon offers many other VMs with different CPU and memory parameters.
- **Task**: These are nodes that can only execute tasks. They do not have the DataNode component, and therefore, are not part of the HDFS.



9. The next section is the **Security and Access** section. It allows the user to set access control on the cluster and specify the keys for access. If you want to securely log in into any of the EC2 instances, it requires an Amazon EC2 key pair. The PEM file associated with this key pair is used when connecting to EC2 via ssh. The instructions on how to set up a key pair are given at `https://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-plan-access-ssh.html`. In the example, we use the `MasteringHadoop` key pair to access the cluster via ssh. Also, we don't allow access to any other AWS user by selecting **No Other IAM users** in the **IAM user access** subsection. EMR allows role-based access control, and there are two access control grants that can be specified.
10. Setting a role in the **EMR role** dropdown allows the application using this role to access other AWS services such as EC2. Similarly, setting a role in **EC2 instance profile** allows EC2 instances within EMR to access other AWS services:

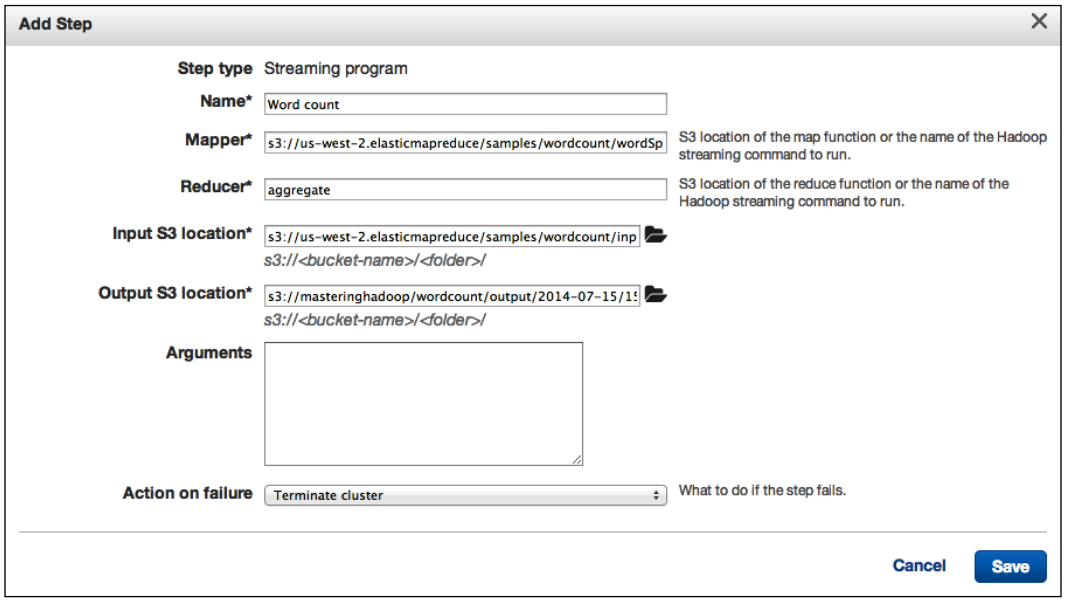


11. The next section shows how to set **Bootstrap Actions** for the cluster. The next screenshot shows an image of this section. Setup scripts can be specified here to instruct any special configuration that will be required before starting the cluster. The Bootstrap action dropdown has options to configure Hadoop, configure daemons, execute scripts on predicates, or conduct some custom actions. In the example, we will omit having any bootstrap actions.
12. The final section is the **Steps** section. This is the section where jobs can be submitted to the Hadoop cluster. In the example, the dropdown has options to execute a Hive program, Pig program, Streaming program, Impala program, or a custom MapReduce Java JAR file. We will see how we can add a streaming program from the AWS EMR samples that are already available. This section also has an autoterminate action that terminates the cluster once the last step has been executed. In the example, we set the **Auto-terminate** radio button to **No** as we wish to explicitly terminate the cluster.

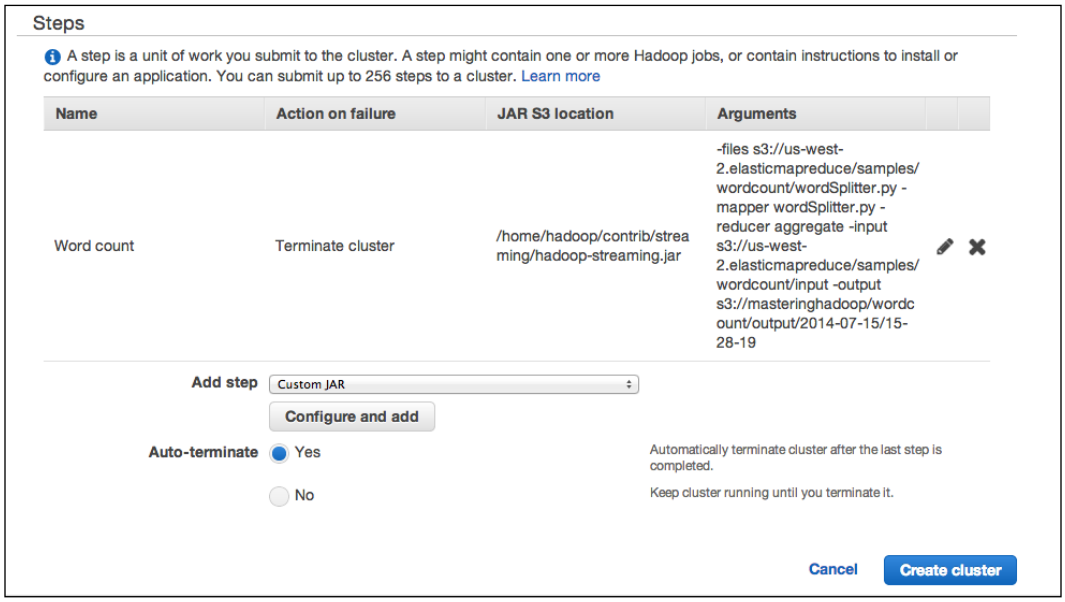


In the example, let's select a Hadoop Streaming program step by choosing it from the dropdown. Clicking on the **Configure and add** button opens up the **Add step** wizard, as shown in the following screenshot. We enter a friendly name for the step. The Mapper task is set to the Python program given by the S3 path, `s3://us-west-2.elasticmapreduce/samples/wordcount/wordSplitter.py`.

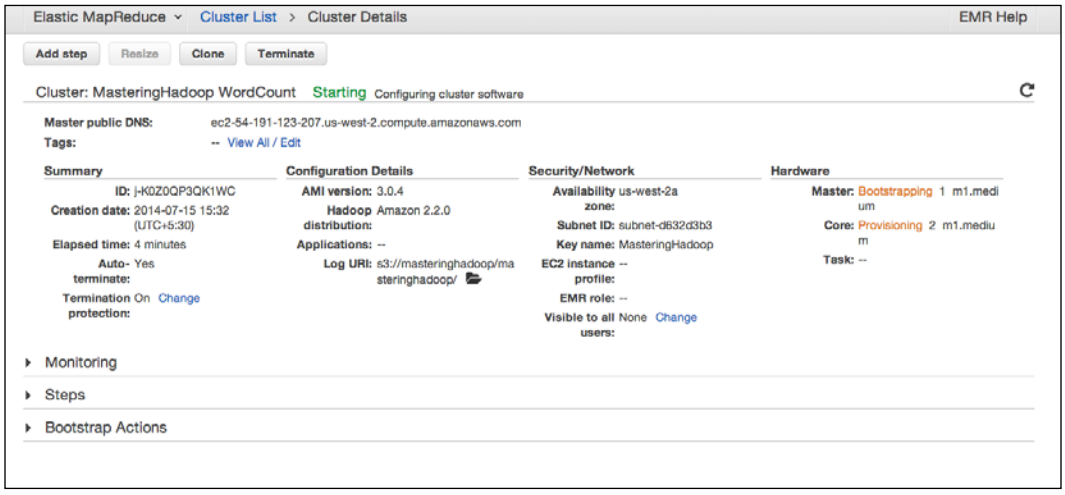
The **Reducer** field is set to `aggregate`. This is a built-in reducer that sums the values corresponding to each key. The folder containing the files on which we run the word count is given by the S3 path, `s3://us-west-2.elasticmapreduce/samples/wordcount/input`. This is specified in the **Input S3 location** path. The output S3 path is specified by the `s3://masteringhadoop/wordcount/output/2014-07-15/15-28-19` path. The word count output is put in this folder. Any additional arguments can be specified in the **Arguments** box. We can also specify the steps to be undertaken if a failure is encountered. In the example, we have chosen to terminate the cluster on any failure. The other options are to continue executing subsequent steps or cancel and wait for user intervention. We then click on the **Save** button. We are now ready to review the cluster before running it.



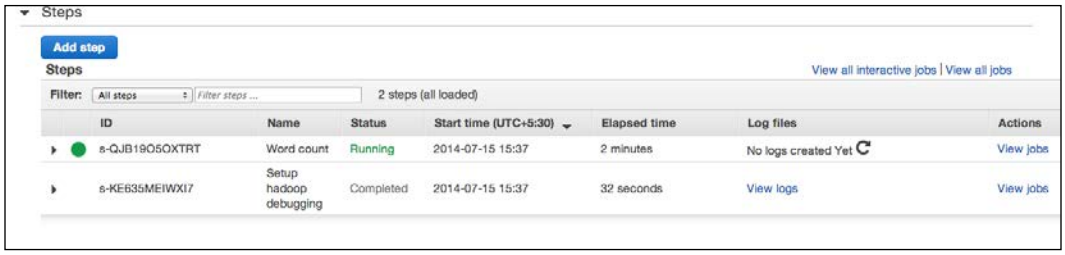
13. The **Steps** section now looks similar to what is shown in the following screenshot. Clicking on the **Create cluster** button starts provisioning the cluster:



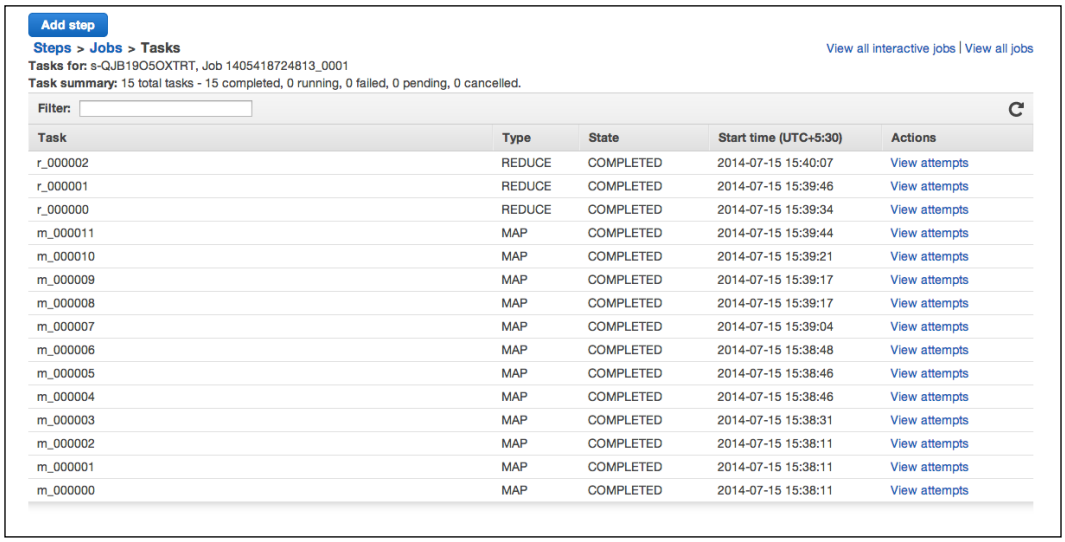
14. The EMR dashboard now shows the list of clusters that are running. Clicking on a cluster of interest shows the details of the cluster. The following screenshot shows the status of the executing cluster. The configuration status and state of each cluster component are given. In the screenshot, the Master Node is **Bootstrapping** while the two core nodes are in the **Provisioning** state.



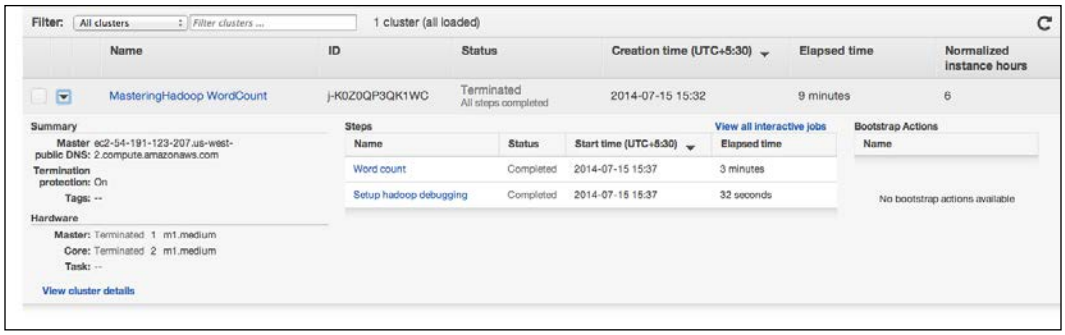
15. The page has expandable sections that can be expanded to see the details of each section. When we expand the **Steps** section, the details are as shown in the following screenshot. It can be seen that the Hadoop setup step is completed, and word count streaming is currently being executed.



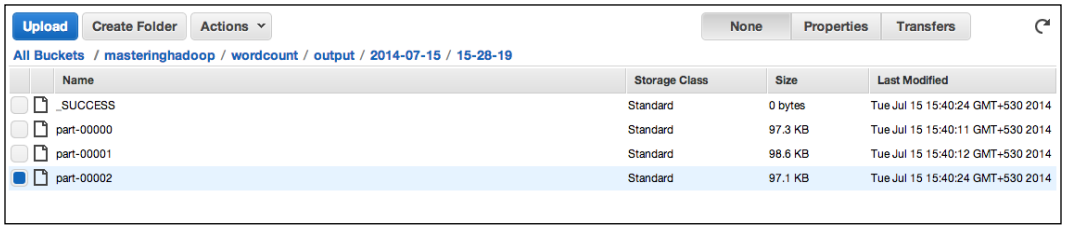
16. The link to the right gives details on the jobs being executed in the step. Clicking on the **View jobs** link of the currently executing step shows the details of the job and its tasks, as shown in the following screenshot. Three Reduce tasks were executed along with twelve Map tasks to complete the program. The attempts by the tasks can be analyzed by clicking on the **View attempts** link. The status of each task is also given.



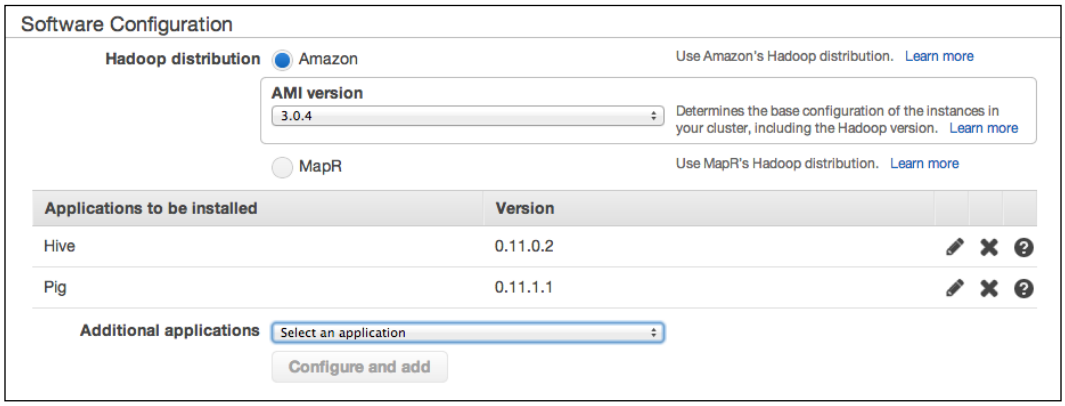
17. Expanding the cluster information on the EMR dashboard can give a quick summary of the cluster. Details such as elapsed time are given here. A step-by-step breakdown of the elapsed time is also visible. The following screenshot gives this view:



18. Finally, when the job is complete, the output files can be seen in S3. The output folder was specified when initiating the job. The following screenshot shows the output folder in S3. We had three Reduce tasks in the job, and it promptly has three output files in S3, one file per Reduce task.



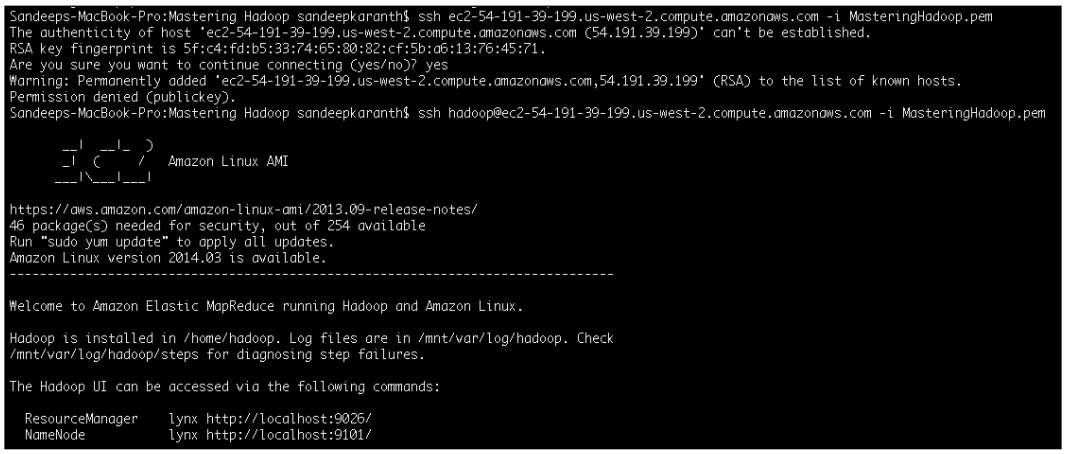
We can also run Hive and Pig scripts via the management console. The scripts need to be uploaded in S3. During the software configuration step, we need to specify the install of Hive and/or Pig, based on the requirement. The following screenshot shows the **Software Configuration** section of a cluster with Hive and Pig installed. Based on the AMI chosen, an appropriate version of Hive and Pig are available:



We will look at how to run Hive and Pig interactively on EMR, instead of a batch mode, via the management console. Once the cluster is provisioned with Hive and Pig installed, we can securely log in to the Master Node. To reiterate, it is very important to get a key-value pair from Amazon and assign it to the cluster. If this is not done, it is not possible to securely log in to the cluster.

The following screenshot shows how to securely log in to the cluster. The DNS name of the Master Node can be copied from the cluster status page. In the example, the Master Node DNS name is `ec2-54-191-39-199.us-west-2.compute.amazonaws.com`.

It is also important to provide a username when logging in. All Hadoop services run under the `hadoop` user.



Providing the respective program names on the command line of the Master Node can start the Hive and Pig (Grunt) shells. The following screenshots illustrate this:





Interactive commands can now be executed at these prompts. File locations can be specified with the `s3://<bucket name>/<folder name>` paths to read them from S3.

Summary

Cloud is a cost-efficient and effective way of developing pre-operationalized analytics. The self-serve, pay-as-you-go, and elastic deployment features of the cloud are reasons for the cost benefits. Many companies such as Yelp and Netflix run massive analytic workloads using the cloud infrastructure. Apache Hadoop is available as a PaaS offering on all major cloud service providers.

Some key takeaways from this chapter are as follows:

- Amazon's Hadoop offering is called Elastic MapReduce (EMR), and it has been around since 2009. Microsoft launched its Hadoop offering in 2012, which is known as HDInsight on Microsoft Azure.
- Using an AWS account, a Hadoop cluster can be launched in a matter of minutes. The number of EC2 instances in the Hadoop cluster is currently limited to 20. For more instances, a special request needs to be mailed to Amazon. Here's a word of caution for you: remember to terminate your Hadoop EMR cluster after use. If this is not done, charges will be incurred even if the cluster is idle.
- EMR provides many Hadoop versions, the latest being 2.4. It also provides the MapR Hadoop distribution.
- For now, in EMR, a custom JAR, Hive queries, Pig scripts, and Hadoop Streaming programs can be run. The input and the MapReduce program are generally stored in S3, the scalable storage offering from Amazon AWS.
- EMR provides granular access control to Hadoop clusters. Roles can be defined and assigned appropriately.

In the next chapter, we will see how HDFS can be replaced with other filesystems. With HDFS replacement strategies, the file load time before job execution can be brought down significantly, thus bringing down the latency of the job.

9
HDFS Replacements

The parallelism and scalability of the MapReduce computing paradigm are greatly influenced by the underlying filesystem. HDFS is the default filesystem that comes with most Hadoop distributions. The filesystem automatically chunks files into blocks and stores them in a replicated fashion across the cluster. The information of the distribution pattern is supplied to the MapReduce engine that can then smartly place tasks so that movement of data over the network is minimized.

However, there are many use cases where HDFS may not be ideal. In this chapter, we will look at the following topics:

- The strengths and drawbacks of HDFS when compared to other POSIX filesystems.
- Hadoop's support for other filesystems. One of them is Amazon's cloud storage service known as **Simple Storage Service** (S3). Reading and writing files from and to the S3 services is permitted within Hadoop.
- Hadoop HDFS has extensibility features. Extending the framework can be of two kinds: by providing a new object storage interface or by providing a drop-in replacement for HDFS. The former would mean changing the MapReduce layer by providing the ability to read via the new interface, while the latter requires no change to existing jobs.
- How to extend HDFS to support S3's native filesystem.

HDFS – advantages and drawbacks

HDFS has its advantages and drawbacks. Some of its advantages are as follows:

- HDFS is inexpensive because of two reasons. Firstly, the filesystem relies on commodity storage disks that are much less expensive than the storage media used for enterprise grade storage. Secondly, the filesystem shares the hardware with the computation framework as well, in this case, MapReduce. Also, HDFS is open source and does not levy licensing fee on the user.
- HDFS has been around for more than 7 years and is considered mature technology. There is a large community behind it and a broad range of organizations that are storing petabytes of data on HDFS.
- HDFS is optimized for MapReduce workloads. It provides very high performance for sequential reads and writes, which is the typical access pattern in MapReduce jobs.

But, HDFS does not cater to all the data needs that may arise in an enterprise. The main drawback of HDFS is that it is not POSIX compliant. This means:

- HDFS is immutable, that is, files cannot be modified. They rather need to be created from scratch. The **append** action is the only action possible on the files and was introduced much later in the evolution of HDFS.
- HDFS is not mountable. Unlike POSIX-compliant filesystems, HDFS cannot be mounted and operated upon. This eliminates the employment of many familiar and popular filesystem tools that are used to search, browse, and manipulate data.
- HDFS is optimized for streaming reads and not for random access of files. This feature of HDFS in combination with the preceding two properties makes it tedious for the user to get a holistic view of the files in HDFS.
- Though HDFS is optimized for MapReduce jobs, Hadoop has moved on to become a generic cluster compute framework with the introduction of YARN. Other computing paradigms might have different expectations from the underlying filesystem, and HDFS falls short of meeting these expectations.

Amazon AWS S3

S3, short for Simple Storage Service, is Amazon's storage as a service offering. It provides reliable storage for data by providing redundancy. The consumer is charged for storage of data on S3 based on the amount of storage used. Any download of data from S3 is also charged, but data upload and transfer of data between AWS properties are free of charge. This makes it extremely attractive for the user to run EMR (Elastic Map Reduce) on AWS and have data stored on S3.

S3 can be used as the input and output data store for MapReduce jobs. The intermediate files can be stored on local disks or the HDFS of the EMR cluster. This also allows easy sharing of input and results among different people in the organization without fearing data loss, with high data security. If an EMR cluster gets terminated accidentally, all of the HDFS data will be lost unless it is moved out. Using S3 for input and output mitigates such risks.

However, S3 is significantly slower because it does not provide data locality and should be used wisely. The best practice is to get the initial dataset for MapReduce jobs from S3 and move the final results back to S3. All intermediate output from the cascaded MapReduce job should be kept within HDFS.

Hadoop support for S3

Hadoop supports transfers to and from a cluster and S3. Two kinds of file storage support are provided by Hadoop, which are as follows:

- **S3 native filesystem** (**s3n**): The native S3 filesystem object within Hadoop allows reading and writing of files as S3 objects. The files are stored in S3's native form. This makes it possible to read files using other S3 tools. However, S3 imposes a limit of 5 TB on the file objects.
- **S3 block filesystem** (**s3**): This is similar to how a file is stored in HDFS. The file is broken into blocks and all the blocks are stored using S3. S3 is purely a storage layer for the file blocks. Though the block-based filesystem on S3 allows storage of files greater than 5 TB, it imposes a requirement of dedicating an entire bucket as the Hadoop store. Storage of other non-block-based file types is not permitted. Unlike the S3 native filesystem, it does not allow reading data from other standard S3 tools.

The S3 block filesystem comes close to being a drop-in replacement for HDFS. However, it has some limitations. The biggest limitation other than data locality is the problem of eventual consistency that is inherent to a distributed store. Changes made to the filesystem might not be visible immediately and might take an indefinite time to converge.

A few configuration changes such as providing the credentials for the S3 bucket are needed to connect and move files to and from the S3 bucket using the mentioned filesystem drivers. The `core-site.xml` file in the Hadoop `conf` directory can be modified as shown in the following XML snippet:

```
<property>
  <name>fs.s3n.awsAccessKeyId</name>
  <value><Your access id></value>
```

```
</property>

<property>
  <name>fs.s3n.awsSecretAccessKey</name>
  <value><Your secret key></value>
</property>
```

The preceding code tells Hadoop's S3 native filesystem driver to use the specified credentials to connect to S3. If the S3 block-based filesystem driver is being used, the property names change to `fs.s3.awsAccessKeyId` and `fs.s3.awsSecretKeyId`.

Any HDFS command can now be executed with the `s3` or `s3n` scheme in the URL to conduct file operations on the specified S3 path. For example, the following command lists all the files in the `masteringhadoop` AWS bucket:

```
hadoop fs -ls s3n://masteringhadoop/
```

The output of this command is as follows:

```
Found 10 items
-rw-rw-rw-   1   43736787 2014-07-31 16:44
s3n://masteringhadoop/HDFSReplacements-1.0-SNAPSHOT-jar-with-
dependencies.jar
-rw-rw-rw-   1       3875 2014-06-08 20:06
s3n://masteringhadoop/countrycodes.avro
-rw-rw-rw-   1       3787 2014-07-19 10:18
s3n://masteringhadoop/countrycodes.txt
drwxrwxrwx   -          0 1970-01-01 05:30
s3n://masteringhadoop/jars
drwxrwxrwx   -          0 1970-01-01 05:30
s3n://masteringhadoop/logs
drwxrwxrwx   -          0 1970-01-01 05:30
s3n://masteringhadoop/masteringhadoop
drwxrwxrwx   -          0 1970-01-01 05:30
s3n://masteringhadoop/songs
drwxrwxrwx   -          0 1970-01-01 05:30
s3n://masteringhadoop/user
drwxrwxrwx   -          0 1970-01-01 05:30
s3n://masteringhadoop/wordcount
```

The key part to note here is that Hadoop is able to connect to S3 out of the box because the S3 credentials were specified in the `core-site.xml` file. The S3 filesystem is addressed by a scheme, `s3n`, indicating that we are using Hadoop's S3 native filesystem driver. The scheme can be changed to `s3` if we use the S3 block filesystem driver.

Instead of specifying the access and secret key in `core-site.xml`, it can be directly specified in the path of the file as follows:

```
s3n://AWS-ACCESS-ID:AWS-SECRET-KEY@masteringhadoop/
```

The AWS secret key and access key can be obtained in the accounts section of the AWS management console.

Implementing a filesystem in Hadoop

Based on the situation, it might be a necessity to replace HDFS with a filesystem of your choice. Hadoop provides out-of-the-box support for a few filesystems such as S3. HDFS replacement can be done either as a drop-in replacement or, as in the case with S3, seamless integration with the S3 file store for input and output.

In this section, we will re-implement the S3 native filesystem and extend Hadoop. The code in this section illustrates the steps on how HDFS replacement can be done. Error handling and other features related to S3 have been omitted for brevity.

The major steps in implementing a filesystem for Hadoop are as follows:

1. The `org.apache.hadoop.fs.FileSystem` abstract class needs to be extended and all the abstract methods need to be overridden. There are out-of-the-box implementations for `FilterFileSystem`, `NativeS3FileSystem`, `S3FileSystem`, `RawLocalFileSystem`, `FTPFileSystem`, and `ViewFileSystem`.
2. The `open` method returns an `FsDataInputStream` object. A backing `InputStream` object that can read from the underlying filesystem whose support the user wishes to incorporate in Hadoop needs to be created.
3. The `create` and `append` methods return an `FsDataOutputStream` object. A backing `OutputStream` object that can write to the underlying filesystem whose support the user wishes to incorporate within Hadoop needs to be created.
4. An `org.apache.hadoop.fs.FileStatus` object needs to be created whenever the status of a particular file given by a `Path` object is specified.
5. The JAR file containing this implementation has to be placed in the `$HADOOP_HOME/share/hadoop/hdfs/lib` directory so that it can be picked up when the distributed filesystem is started. The `core-site.xml` file must have the `fs.<scheme>.impl` property value set to the fully-qualified class name for the filesystem implementation. Additional properties can also be provided to configure the filesystem.

Implementing an S3 native filesystem in Hadoop

Let's first create `InputStream` and `OutputStream` for the filesystem. In our example, we have to connect to the AWS to read and write files to S3.

Hadoop provides us with the `FSInputStream` class to cater to custom filesystems. We extend this class and override a few methods in the example implementation. A lot of private variables are declared along with the constructor and helper methods to initialize the client as illustrated in the following code snippet. The `private` variables contain objects that are used to configure and retrieve data from the filesystem. In this example, we use objects such as `AmazonS3Client` to call REST web APIs on AWS, `S3Object` as a representation of the remote object on S3, and `S3ObjectInputStream` representing the object stream to perform the read operation. All the AWS-related classes are present in the `com.amazonaws.services.s3` and `com.amazonaws.services.s3.model` packages. There are a few other private variables such as the S3 bucket name and the S3 key. The Hadoop `Configuration` object is passed to help read any configuration properties the user might have defined.

The constructor ensures that all the private variables are correctly initialized. In the example, we open the S3 object stream lazily. Therefore, there are no stream initialization calls in the constructor. To lazily initialize the object and the object stream, we have the `openObject` and `openS3Stream` calls. If the object is not initialized, `openObject` calls `openS3Stream` by setting the stream to the beginning of the file. The `openS3Stream` call aborts any open stream and reinitializes a new object and stream:

```
private class S3NFsInputStream extends FSInputStream{

    private AmazonS3Client s3Client;
    private Configuration configuration;
    private String bucket;
    private String key;
    private long length;

    private S3ObjectInputStream s3ObjectInputStream;
    private S3Object s3Object;
    private long position;

    public S3NFsInputStream(AmazonS3Client s3, Configuration
        conf, String bucket, String key, long length) {
        super();

        this.s3Client = s3;
```

```
        this.configuration = conf;
        this.bucket = bucket;
        this.key = key;
        this.length = length;

        this.s3Object = null;

    }

    private void openObject(){

        if(s3Object == null){
            openS3Stream(0);
        }

    }

    private void openS3Stream(long position){

        if(s3ObjectInputStream != null){
            s3ObjectInputStream.abort();
        }

        GetObjectRequest objectRequest = new
            GetObjectRequest(this.bucket, this.key);
        objectRequest.setRange(position, length - 1);
        this.s3Object =
            this.s3Client.getObject(objectRequest);
        this.s3ObjectInputStream =
            this.s3Object.getObjectContent();

        this.position = position;

    }
```

In the preceding example, we go ahead and implement the methods that mandatorily have to be overridden. The code snippet that follows gives the override methods and their implementations. The `read` override method is used to read a byte from the input stream. The position of the input stream is incremented. The `read` override is overloaded with another variant that fills a byte buffer and returns the number of read bytes. All these methods call the `openObject` call before reading from `S3ObjectInputStream` to lazily initialize the stream.

The `close` method is used to clean up and the `seek` method is used to get to a particular position in the file. We do not support any markers in this simple implementation. You can recall markers as points in the file that can be used for splittable compression:

```
@Override
public int read() throws IOException {

    openObject();
    int readByte = this.s3ObjectInputStream.read();

    if(readByte >= 0){
        this.position++;
    }

    return readByte;
}

@Override
public int read(byte[] b, int off, int len) throws
IOException {

    openObject();
    int readByte = this.s3ObjectInputStream.read
        (b, off, len);

    if(readByte >= 0){
        this.position+=readByte;

    }
    return readByte;
}

@Override
public void close() throws IOException {
    super.close();

    if(s3Object != null){
        s3Object.close();
    }
}


@Override
```

```
        public boolean markSupported() {
            return false;
        }

        @Override
        public void seek(long l) throws IOException {

            if(this.position == l){
                return;
            }
            openS3Stream(l);

        }

        @Override
        public long getPos() throws IOException {
            return this.position;
        }

        @Override
        public boolean seekToNewSource(long l) throws IOException
        {
            return false;
        }


    }
```

Next, we will implement the output stream to write files to the object. We extend the `java.io.OutputStream` package and override the abstract methods. As with the input stream, we have a bunch of private variables such as `AmazonS3Client`, the Hadoop `Configuration`, and the backing `OutputStream` object. The strategy here is to write to a local file, and when the stream is closed, the file is uploaded onto S3. To facilitate writing to the local file, we create the `BufferedOutputStream` and `LocalDirAllocator` objects.

When the client writes to this `OutputStream` object, a temporary file is created and written into using the `BufferedOutputStream` object. The file is created in a local directory based on the specification in the configuration. We use the temporary directory given by the value `hadoop.tmp.dir` in the configuration. The example code is for illustrating the concepts behind extending HDFS and cannot be used as is in production. For example, cleanup of the temporary directories has to be done manually. Also, we use the same backing filename, `temp`, which may lead to thread safety issues in multithreaded environments.

A constructor is used to create the temporary file in the local directory. `BufferedOutputStream` is also initialized. The `write` and `flush` overrides are operations on `BufferedOutputStream`. The `close` method override is where we upload the file from the local store to S3. The `PutObjectRequest` S3 class is used to specify the properties of the object that needs to be uploaded to S3. The `AmazonS3Client` object is then used to upload the local file:

```
    private class S3NFsOutputStream extends OutputStream{

        private OutputStream localFileStream;
        private AmazonS3Client s3Client;
        private LocalDirAllocator localDirAllocator;
        private Configuration configuration;
        private File backingFile;
        private BufferedOutputStream bufferedOutputStream;
        private String bucket;
        private String key;


        public S3NFsOutputStream(AmazonS3Client s3, Configuration
            conf, String bucket, String key) throws IOException{
            super();
            this.s3Client = s3;
            this.configuration = conf;
            this.localDirAllocator = new
                LocalDirAllocator("${hadoop.tmp.dir}/s3mh");

            this.backingFile =
               localDirAllocator.createTmpFileForWrite("temp",
                    LocalDirAllocator.SIZE_UNKNOWN, conf);
            this.bufferedOutputStream = new
                BufferedOutputStream(new
                    FileOutputStream(this.backingFile));
            this.bucket = bucket;
            this.key = key;
        }


        @Override
        public void write(int b) throws IOException {
            this.bufferedOutputStream.write(b);
        }

        @Override
        public void write(byte[] b) throws IOException {
```

```
            this.bufferedOutputStream.write(b);
        }

        @Override
        public void write(byte[] b, int off, int len) throws
            IOException {
            this.bufferedOutputStream.write(b, off, len);
        }

        @Override
        public void flush() throws IOException {
            if(this.bufferedOutputStream != null){
                this.bufferedOutputStream.flush();
            }
        }

        @Override
        public void close() throws IOException {

            if(this.bufferedOutputStream != null){
                this.bufferedOutputStream.close();
            }


            try {
                PutObjectRequest putObjectRequest = new
                    PutObjectRequest(bucket, key, backingFile);
                putObjectRequest.setCannedAcl(CannedAccessControlList.
Private);

                s3Client.putObject(putObjectRequest);
            }
            catch(AmazonServiceException ase){
                ase.printStackTrace();
            }

        }
    }

}
```

Now we can extend the `FileSystem` class and implement the override methods. The following code snippet gives the class declaration and the `initialize` method override. The `initialize` method takes in the `URI` of the file or directory and the Hadoop `Configuration` object. It then constructs the `AmazonS3Client` based on the access and secret keys present within the configuration. To recall, the configuration values are taken from `core-site.xml` and its override files.

We define `fs.s3mh.access.key` as the property name that contains the access key and `fs.s3mh.secret.key` as the property name whose value contains the secret key. We use the `BasicAWSCredentials` object to encapsulate the credentials before constructing the `AmazonS3Client` object.

The `getScheme` and `getUri` methods are overridden. We use the scheme for our filesystem implementation as `s3mh`. Our filesystem class itself is called `S3NFileSystem`, indicating that we will use the Amazon S3 native object model:

```
package MasteringHadoop;

import com.amazonaws.AmazonServiceException;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.services.s3.model.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.fs.permission.FsPermission;
import org.apache.hadoop.util.Progressable;

import java.io.*;
import java.net.URI;
import java.util.ArrayList;

public class S3NFileSystem extends FileSystem {

    private URI uri;
    private AmazonS3Client s3Client;
    private Configuration configuration;
    private String bucket;

    public S3NFileSystem(){
        super();
    }

    @Override
```

```
public void initialize(URI name, Configuration conf) throws
    IOException {

    super.initialize(name, conf);
    this.uri = URI.create(name.getScheme() + "://" +
        name.getAuthority());

    String accessKey = conf.get("fs.s3mh.access.key");
    String secretKey = conf.get("fs.s3mh.secret.key");

    System.out.println("Access Key: "  + accessKey);

    s3Client = new AmazonS3Client(new
        BasicAWSCredentials(accessKey, secretKey));

    this.bucket = name.getHost();

    if(!s3Client.doesBucketExist(this.bucket)){
        throw new IOException("Bucket " + this.bucket + " does
            not exist!");
    }

    this.configuration = conf;


}

@Override
public String getScheme() {
    return "s3mh";

}

@Override
public URI getUri() {
    return uri;
}
```

Next, we override the `open`, `create`, and `delete` methods. We do not support `append` and `rename`. In S3, `rename` is implemented by doing a copy followed by the `delete` operation. The `open` method returns the `FSDataInputStream` object, whose class implementation we saw previously. S3 has a concept of folders, but folders do not translate into actual objects in the filesystem and will have a length of 0. The length parameter for the custom implementation of the `FSDataInputStream` object is obtained from the `getFileStatus` method. The `delete` method deletes a file object from S3. For directories, the `delete` call is ignored, as they are pseudo-objects:

```
@Override
public FSDataInputStream open(Path path, int i) throws
    IOException {
    FileStatus fs = getFileStatus(path);
    return new FSDataInputStream(new
        S3NFsInputStream(this.s3Client, this.configuration,
            this.bucket, pathToKey(path), fs.getLen()));
}

@Override
public FSDataOutputStream create(Path path, FsPermission
    fsPermission, boolean b, int i, short i2, long l,
        Progressable progressable) throws IOException {
    String key = pathToKey(path);
    return new FSDataOutputStream(new
        S3NFsOutputStream(this.s3Client, this.configuration,
            this.bucket, key), null);
}

@Override
public FSDataOutputStream append(Path path, int i,
    Progressable progressable) throws IOException {
    throw new IOException("Append functionality is not
        supported");
}

@Override
public boolean rename(Path path, Path path2) throws
    IOException {
    throw new IOException("Rename is copy followed by
        delete");
}

@Override
```

```
    public boolean delete(Path path, boolean b) throws IOException
        {

        FileStatus fs = getFileStatus(path);

        if(b){
            throw new PathIOException("Recursive delete is not
supported");
        }

        if(!fs.isDirectory()){
            s3Client.deleteObject(this.bucket, pathToKey(path));
        }

        return false;
    }
```

The `listStatus` override lists all the objects at a given path as illustrated in the following code snippet. A `ListObjectRequest` object is created and `AmazonS3Client` is used to retrieve the object summaries. The metadata is extracted and then stored in a `FileStatus` object before it is returned. AWS can list objects in batches. In this example, we use a batch file of size 1,000. If the number of objects is greater than this number, the first 1,000 objects are fetched. In actual production code, all objects are retrieved in batches.

The other important override method is the `getFileStatus` method. It takes in a path and returns a single `FileStatus` object for that path. The path can be a folder or a file. Helper methods are also used, such as `pathToKey`, which returns keys from a path object, and `isADirectory`, which checks whether an object is a directory or not based on its name and size:

```
    @Override
    public FileStatus[] listStatus(Path path) throws
        FileNotFoundException, IOException {

        ArrayList<FileStatus> returnList = new ArrayList<>();
        String key = pathToKey(path);
        FileStatus fs = getFileStatus(path);
        if(fs.isDirectory()){

            if(!key.isEmpty()){
                key = key + "/";
            }
```

```
                ListObjectsRequest listObjectsRequest = new
                    ListObjectsRequest(this.bucket, key,
                        null, "/", 1000);
                ObjectListing objectListing =
                    s3Client.listObjects(listObjectsRequest);

                for(S3ObjectSummary summary :
                    objectListing.getObjectSummaries()){

                    FileStatus fileStatus;
                    if(isADirectory(summary.getKey(),
                        summary.getSize())){
                        fileStatus = new FileStatus(summary.getSize(),
                            true, 1, 0, 0, new Path("/" + key));
                    }
                    else{
                        fileStatus = new FileStatus(summary.getSize(),
                            false, 1, 0, 0, new Path("/" + key));
                    }

                    returnList.add(fileStatus);

                }

            }
            else{
                returnList.add(fs);
            }

            return returnList.toArray(new
                FileStatus[returnList.size()]);
        }

        @Override
        public void setWorkingDirectory(Path path) {}
    @Override
        public Path getWorkingDirectory() {return null; }
        @Override
        public boolean mkdirs(Path path, FsPermission fsPermission)
            throws IOException { return false;}
        @Override
        public FileStatus getFileStatus(Path path) throws IOException
        {
```

```
            String key = pathToKey(path);
            System.out.println("Key : " + key);
            System.out.println("Bucket : " + this.bucket)  ;
            if(key.isEmpty()){
                throw new IOException("File not found.");
            }
            ObjectMetadata objectMetadata =
                s3Client.getObjectMetadata(this.bucket, key);
            if(isADirectory(key, objectMetadata.getContentLength())){
                return new FileStatus(0, true, 1, 0, 0, path);
            }
    return new FileStatus(0, false, 1, 0,
        objectMetadata.getLastModified().getTime(), path);
        }


        private String pathToKey(Path path) {
            return path.toUri().getPath().substring(1);
        }

        private boolean isADirectory(String name, long size) {
            return !name.isEmpty()
                   && name.charAt(name.length() - 1) == '/'
                   && size == 0L;
        }
```

Once all the code snippets are compiled and made into a JAR file, the JAR file can be placed in the HDFS library directory. The following snippet shows the configuration information that needs to go into the `core-site.xml` file in order to indicate the JAR file for the filesystem driver, as well as other credential information required by the implementation. Any HDFS command can now be executed, with the path indicating the `s3mh` scheme used to invoke this `FileSystem` implementation:

```
<!-- omit for IAM role based authentication -->
<property>
  <name>fs.s3mh.access.key</name>
  <value><!-- Your Amazon AWS key --></value>
</property>

<!-- omit for IAM role based authentication -->
<property>
  <name>fs.s3mh.secret.key</name>
```

```
    <value><!—Your Amazon AWS Secret à</value>
  </property>

  <!-- necessary for Hadoop to load our filesystem driver -->
  <property>
    <name>fs.s3mh.impl</name>
    <value>MasteringHadoop.S3NFileSystem</value>
  </property>
```

Summary

HDFS is a great filesystem for MapReduce workloads. But its sequential access pattern and non-compliance with POSIX interfaces make it tedious to work with in certain situations. Hadoop allows its users to extend HDFS or provide drop-in replacements. The key takeaways from this chapter are as follows:

- There are a number of implementations that extend or provide drop-in replacements for HDFS. CephFS, MapRFS, GPFS from IBM, and Cassandra by DataStax are some examples of such extensions.
- Interface to the Amazon S3 storage service is available out of the box in Hadoop. Both a native-storage S3 filesystem interface and a block-storage filesystem interface are available.
- Extending Hadoop to incorporate other filesystems is done by extending the `FileSystem` abstract base class. The `FSDataInputStream` and `FSDataOutputStream` objects are used to wrap the input and output streams of the underlying filesystem respectively.
- The security and access control mechanisms of the underlying filesystem can be left intact within Hadoop by allowing configurations to be specified via the Hadoop `Configuration` files and classes.

In the next chapter, we will continue our study of HDFS by looking at the federation aspects of HDFS and how it changed when it moved from Hadoop 1.X to 2.X.

10
HDFS Federation

The NameNode component of HDFS was the central point of failure in the initial versions of Hadoop. In the later versions, a secondary NameNode was introduced as a backup for the primary NameNode. Until Hadoop 2.X, the NameNode component could only handle a single namespace, making it less scalable and difficult to isolate in a multitenant HDFS environment. Scalability and isolation were the two most desired requirements for Hadoop enterprise deployments. Most organizations shared infrastructure among their different teams with varying degrees of availability and authorization aspirations.

HDFS Federation is a feature that enables Hadoop to have multiple namespaces, making it easy to use for shared cluster scenarios. This feature brings about a separation between the storage and namespace management. Similar to YARN, this separation helps onboard other applications and use cases on to HDFS, making Hadoop move away from a MapReduce-only platform to a more generic cluster-computing platform.

In this chapter, we will be:

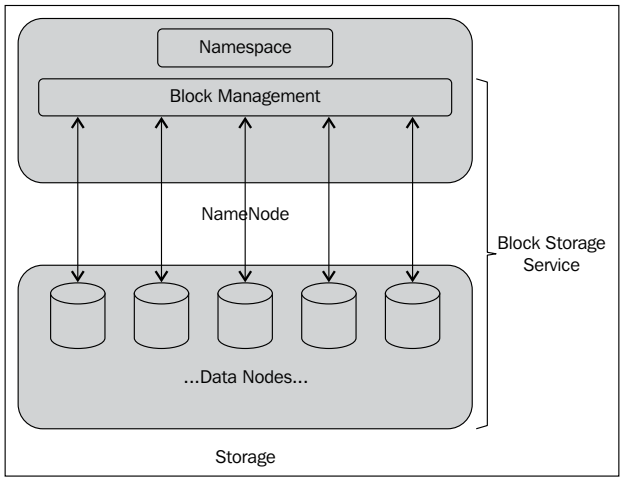
- Looking at the necessity behind HDFS Federation
- Studying HDFS Federation and its architecture
- Understanding the steps to deploy federated NameNodes
- Understanding backup and recovery options for the active NameNode
- Studying the strategies and commands that support NameNode high availability
- Looking into some useful tools and commands in HDFS
- Looking into HDFS block placement strategies in the MapReduce environment

Limitations of the older HDFS architecture

The older HDFS architecture had two main components, which are as follows:

- **Namespace**: This is the HDFS component that handles building blocks such as directories, files, and the actual file blocks. The Namespace component supports the create, delete, list, and update/modify operations on these building blocks. The Namespace component is within the NameNode daemon.
- **Block Storage Service**: This is the HDFS component that handles file block management. The block storage component is divided between the NameNode and DataNode. The DataNode part of the block storage service takes care of block storage on the local machine in a cluster. It provides read and write services for the blocks. The NameNode part of the block storage performs the following actions:
    - Taking care of DataNode registrations, monitoring, and health reports.
    - Digesting block reports from DataNodes and storing the location of the file blocks in memory.
    - Dealing with the create, delete, list, and update operations at the block level. As we saw earlier, at the file level, the Namespace component performs these operations.
    - Facilitating replica placement algorithms and heuristics. It can manage the placement of replicated blocks. It can also replicate underreplicated blocks and delete overreplicated ones.

The following diagram illustrates the older HDFS architecture:



This architecture has a number of limitations, which are as follows:

- From the figure, it is apparent that the block storage service component intrudes into the NameNode. This leads to tight coupling between the Namespace and Block Management Service components. All block management functions need to go through the NameNode. The DataNodes are not capable of being an independent block storage service.
- The architecture allows for a single NameNode. This NameNode stores all directories, files, and block-level metadata in memory. Unlike the DataNodes, the NameNode cannot be scaled horizontally. It has to be scaled vertically by adding more memory to the machine where the NameNode runs. The memory of the NameNode becomes the limiting factor behind the scalability of the cluster.
- With a single NameNode, about 60K tasks can be managed within the cluster. However, with changes in the Hadoop stack and the introduction of YARN, this can go up to 100K tasks and beyond. This kind of task explosion puts a lot of pressure on the request servicing capabilities of the NameNode. A single NameNode might not be able to handle so many requests without adversely affecting the performance of the tasks.
- Larger organizations require a certain degree of isolation between the different teams within them. The reasons for isolation can be for confidentiality, performance, or availability reasons. A single Namespace will not satisfactorily guarantee any of the three reasons. A shared Namespace requires rigorous security measures. Performance and availability depend a lot on the other workloads that are already running on the cluster.

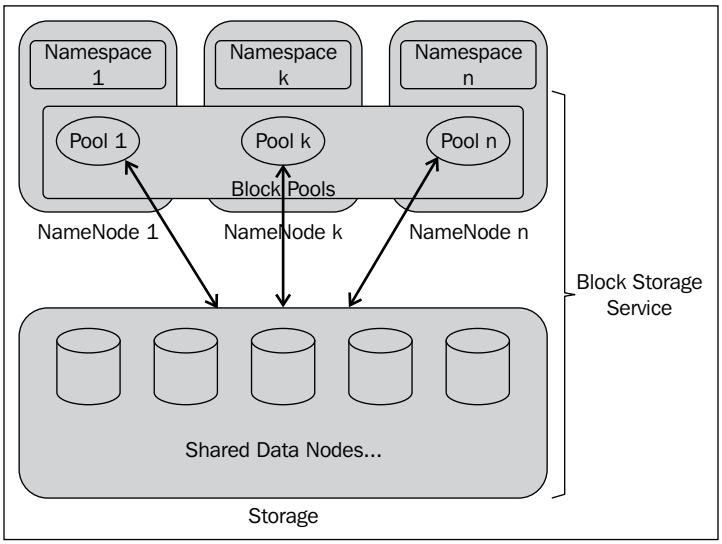
The preceding limitations call for the separation of the Namespace from the Block Storage service component. They also call for a capability to run many instances of the NameNode, particularly in multitenant environments. Horizontal scaling of the NameNode will also help in performance by load balancing.

Architecture of HDFS Federation

The crux of the HDFS Federation feature is that it allows for multiple NameNodes to run on a cluster. These NameNodes are independent and do not have any dependency on each other. However, the DataNodes are shared between all the NameNodes in the system. The NameNodes are said to be federated because they can be run independently without coordination.

Each DataNode sends heartbeats and block report information to all the NameNodes in the cluster. DataNodes also receive instructions from all the NameNodes. They are the common shared storage resource in the cluster and still run on commodity hardware. However, they cater to different NameNodes, and in turn, facilitate different Namespaces. These independent Namespaces provide isolation guarantees in a multitenant environment. By running many NameNodes, the cluster can be horizontally scaled and requests can be load-balanced among these NameNodes.

The following diagram shows the architecture of a federated HDFS cluster:



A concept called **block pools** is used to provide block federation. A block pool is a set of blocks that belong to a single NameNode. DataNodes store blocks that can potentially belong to different block pools. Each block pool is independent and management of one block pool does not affect any other block pools. The Namespace based on its block pool can independently generate Block IDs.

A Namespace, along with its block pool, is termed as the **Namespace Volume**. When a NameNode is decommissioned or a Namespace is deleted, DataNodes delete all the blocks related to the block pool that belongs to the Namespace Volume.

Another parameter, **ClusterId**, is used to identify all the nodes in the cluster. Any new NameNode joining the cluster is given a Cluster ID to correctly identify the cluster to which it belongs. This parameter can either be specified manually or generated automatically. The default behavior is to automatically generate this ID.

Benefits of HDFS Federation

HDFS Federation overcomes the limitations of a single NameNode and provides the following benefits:

- The most important benefit is the horizontal scalability it provides for the NameNode. Clusters with a large number of smaller files benefit immensely from this. Every file occupies memory in the NameNode to store its metadata. The Namespace can now be divided into many Namespaces, either based on function or organizational divisions. The load can now be distributed across many NameNodes rather than a single NameNode.
- Read-and-write throughput scaling is a benefit that is hindered with a single NameNode. Organizations can now divide the Namespace among different NameNodes and keep the throughputs at a desirable level. Information about the kind of workloads that run on the cluster can be used to determine the number of NameNodes that will be required in a federated HDFS deployment.
- Having different NameNodes and Namespaces makes it natural and easy for isolation. Organizations can now isolate different organizational datasets. NameNodes can also be divided based on functions such as development, test, or production. All of the data can be multiplexed onto a bunch of DataNodes promoting efficient sharing. The isolation property ensures that the NameNode does not become the performance bottleneck when jobs of varying needs are executed on the cluster. For example, a job that overloads the NameNode need not affect another job if the data required by them reside in different Namespaces.
- The Federation feature also makes it possible to treat the Block Storage Service as a generic block store. The block pool abstraction can be used to build newer filesystems or HDFS APIs with different characteristics. This generic nature of storage can save costs for the enterprise by increasing the efficiency of existing hardware in their clusters.
- A side effect of the HDFS Federation architecture's simplicity is the backward compatibility of the feature. Existing single-NameNode deployments do not break. They are a special case of the federated approach. Most of the code that change to the Hadoop implementation to support this feature actually happened on the DataNode. This helped in keeping the entire NameNode stable from a Hadoop testing perspective. The DataNode has an additional level of indirection, indicating the block pool where the block belongs.

Deploying federated NameNodes

In this section, we'll see how to deploy multiple NameNodes. To try out these steps, you will need at least two machines with different addresses. The configuration for federated deployment is backward compatible. It supports previous installations with a single NameNode.

To support federation, **NameServiceId** is introduced. The secondary backup and checkpoint nodes belong to a particular NameServiceId. The NameServiceId is suffixed to all the properties of the NameNode and its associated components when specifying in the configuration.

The important configuration steps are as follows:

1. Specifying the NameServiceIds for the different NameNodes in a cluster. This is specified by adding the `dfs.nameservices` property to the configuration. The value of this property is a comma-separated list of all the NameServiceIds.
2. All the other properties that are specified for a particular NameNode should be prefixed with an appropriate NameServiceId. This should be one of the NameServiceIds specified in the `dfs.nameservices` property.
3. A NameNode can be formatted using `bin/hdfs namenode –format [-clusterid clustered]`. If no `clusterId` is specified, an ID is automatically generated.
4. All additional NameNodes can be formatted using the same command as the preceding one, but the `clusterid` parameter becomes mandatory. If the `clusterid` parameter is not specified, then the NameNodes will not be federated. Now, the command is `bin/hdfs namenode –format –clusterid <specify clusterid that was given in the previous step>`.
5. Older releases of Hadoop can be upgraded to the newer release. After upgrading, the NameNode can be started using `bin/hdfs namenode –config <new configuration directory> -upgrade –clusterid <clusterid>`.
6. Adding another NameNode to the system is simply adding the new configuration parameters for the new NameNode and propagating it throughout the cluster. The important parameters are the NameServiceIds and related NameNode properties using the NameServiceId suffix. The NameNodes are then started. Now, the DataNodes have to be informed about the new NameNode. This is done by the `bin/hdfs dfadmin –refreshNameNode <datanode host and port>` command. This command must be executed for all DataNodes that form the block storage layer for the new NameNode.

The following snippet shows a sample `hdfs-site.xml` configuration file that supports two NameNodes. The NameServiceIds are `ns1` and `ns2`:

```
<property>
  <name>dfs.namenode.name.dir.ns1</name>
  <value>[path to namenode store]</value>
  <description>Path on the local filesystem where the NameNode
      stores the namespace and transaction logs
          persistently.</description>
</property>
<property>
  <name>dfs.namenode.name.dir.ns2</name>
  <value>[path to namenode store]</value>
  <description>Path on the local filesystem where the NameNode
      stores the namespace and transaction logs
          persistently.</description>
</property>
<property>
  <name>dfs.nameservices</name>
  <value>ns1,ns2</value>
</property>
<property>
  <name>dfs.namenode.rpc-address.ns1</name>
  <value>[ip:port]</value>
</property>
<property>
  <name>dfs.namenode.http-address.ns1</name>
  <value>[ip:port]</value>
</property>
<property>
  <name>dfs.namenode.secondaryhttp-address.ns1</name>
  <value>[ip:port]</value>
</property>
<property>
  <name>dfs.namenode.rpc-address.ns2</name>
  <value>[ip:port]</value>
</property>
<property>
  <name>dfs.namenode.http-address.ns2</name>
  <value>[ip:port]</value>
</property>
<property>
  <name>dfs.namenode.secondaryhttp-address.ns2</name>
  <value>[ip:port]</value>
</property>
```

HDFS high availability

NameNodes are the heart of an HDFS Namespace. The availability of any cluster using HDFS is directly related to the availability of the NameNode.

Secondary NameNode, Checkpoint Node, and Backup Node

In Hadoop 1.X, the concept of a **Secondary NameNode** was introduced. The Secondary NameNode is a shield against disasters. On the failure of a NameNode, the Secondary NameNode can be used to recover the NameNode. The term Secondary NameNode is a misnomer. It is a cold standby and cannot service requests on its own. The NameNode can, however, read from the Secondary NameNode when encountered with failures.

The NameNode writes all HDFS updates to the `edits` log in the native filesystem. The log is written in an append-only fashion. The NameNode owns another file called the `fsimage` file that contains the image of HDFS. A NameNode starting up, reads the `edits` file and applies all the edits one by one to the `fsimage` file. During this time, no writes are allowed on HDFS. The NameNode is said to be in **Safe Mode**. The Safe Mode ends once the NameNode receives block reports from the DataNodes about the health of the blocks. Writes are allowed only after the NameNode is confident about the health of HDFS. Before starting normal service, the NameNode begins with an empty `edits` file and an updated `fsimage` file.

The longer a NameNode runs, the bigger its `edits` file. This directly translates to longer startup times for the NameNode when it is restarted. The Secondary NameNode periodically takes the `fsimage` and `edits` files and merges them. Generally, to maximize the probability of recovery from a disaster, the Secondary NameNode is run on a different machine. The NameNode can query the Secondary NameNode for the `fsimage` and `edits` files when recovering from a failure. The Secondary NameNode mimics the directory structure of the NameNode when storing checkpoints. This makes it easy for the NameNode to read data when recovering from failures.

The concept of a **Checkpoint Node** is also introduced. It is very similar in function to the Secondary NameNode, with an additional function. Periodically, the Checkpoint Node not only gets the `fsimage` and `edits` file updates from the NameNode, but also folds the edits into the `fsimage` file and uploads it back on to the NameNode. This helps the NameNode to recover fast from failures. The Checkpoint Node can be considered as the Secondary NameNode with a feature to upload the updates back to the NameNode. Again, the checkpoints are stored in the same directory structure as the NameNode.

The Checkpoint Node is started using the following command:

```
hdfs namenode -checkpoint
```

The `dfs.namenode.backup.address` and `dfs.namenode.backup.http-address` properties in the configuration are used to specify the location of the Checkpoint Node and its HTTP endpoint.

A **Backup Node** is more live than the Checkpoint Node and the Secondary NameNode. It streams in updates from the NameNode and updates its own copy of the `fsimage` and `edits` files. The Checkpoint Node and the Secondary NameNode, on the other hand, download the `fsimage` and `edits` files from the active NameNode. A Checkpoint Node is not allowed to run along with a Backup Node. The memory requirements of the Backup Node are the same as that of the NameNode as it stores all information that a NameNode would store.

The Backup Node is started using the following command:

```
hdfs namenode -backup
```

The configuration parameters for the Backup Node are the same as that of the Checkpoint Node.

High availability – edits sharing

In *Chapter 1*, *Hadoop 2.X*, we briefly saw how to provide NameNode high availability using the Quorum Journal Manager and the NFS strategy. Having a hot standby, a NameNode that can instantly switch over to becoming an active NameNode, is the key to ensure **High Availability** (**HA**). The standby node maintains enough information about the active NameNode so that it can provide fast failover. The standby can also perform check-pointing to aid in disaster recovery.

The general strategy behind HA of the NameNode is to share the `edits` file between the active and standby NameNodes. The Quorum Journal Manager is a way to do this using an array of Journal nodes. An NFS share can also be used to achieve the same result.

Hadoop supports the following two failover modes:

- **Manual failover**: Here, the administrator of the Hadoop cluster can execute commands to make the standby active. Since this is a deterministic action, the time for the standby to become active is a matter of about 5 seconds. The command used is as follows:

  ```
  hdfs haadmin -failover <standby-namenode> <active-namenode>
  ```

- **Automatic failover**: When there are systems in place to monitor the health of the active NameNode, the monitoring system might find enough evidence to do an automatic switch between the active and standby NameNodes. This is based on heuristics, and the failover process might take a few seconds longer (order of tens of seconds). Zookeeper is a tool that can help in automatic failover along with the ZKFailoverController module.

If there is a failure in the ZKFailoverController module, there is a possibility that both NameNodes, active and standby, will think that they are in the active state. This scenario is called the *split-brain* scenario. A split-brain scenario can leave the Namespace in an inconsistent state as both NameNodes can make conflicting changes. The solution to this problem is to have the active NameNode stop making changes to the system. The QJM strategy for failover has fencing built in as there is an array of Journal nodes that allow writing only from a single NameNode.

Useful HDFS tools

A number of useful tools are provided to check the health of HDFS. These tools are as follows:

- **Rebalancer**: The distribution of blocks among the DataNodes might not be uniform. These skews in distribution appear when new DataNodes are commissioned in the cluster. To help administrators view the distribution and redistribute the blocks, a `balancer` option is available in the Hadoop command, as shown:

  ```
  hadoop balancer [-threshold threshold]
  ```

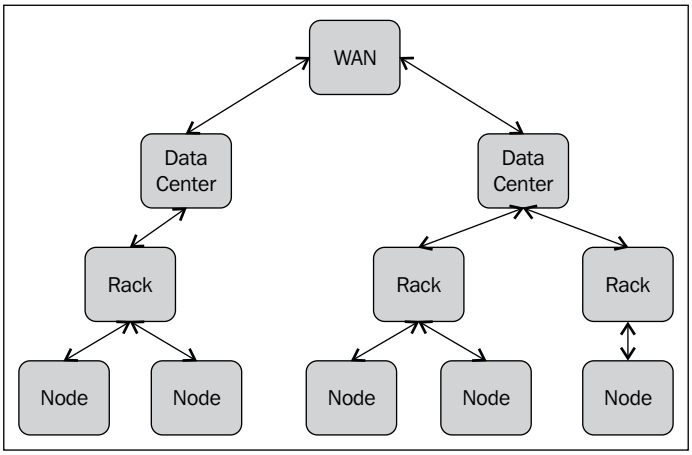
- **fsck**: Like the native filesystem, the `fsck` command goes through the files in the filesystem and provides a report on the health of the blocks and files. Unlike the `fsck` utility of the native filesystem, the Hadoop `fsck` tool takes no action. It is purely a reporting tool. Most of the errors in HDFS are generally taken care of by Hadoop:

  ```
  hadoop fsck
  ```

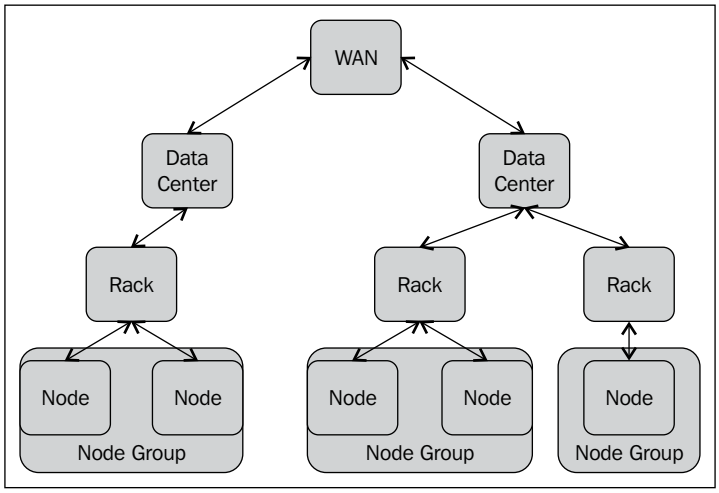
- **Import checkpoint**: It is possible for the NameNode to import checkpoints from the Backup or Checkpoint Node. All these nodes store the checkpoints in the same directory structure as the NameNode. This can be achieved by configuring the `dfs.namenode.checkpoint.dir` property to the directory where the checkpoint resides, and using the `-importcheckpoint` flag when starting the NameNode.

Three-layer versus four-layer network topology

Traditionally, Hadoop topologies follow a three-layer architecture, as shown in the following diagram. The leaves of the hierarchy are the data nodes. Data nodes are combined into racks and racks form the data center. A Hadoop cluster can also span multiple data centers connected via a WAN.



With the advent of virtualization, a single physical machine can now run multiple nodes as virtual machines. This leads to an additional layer in the Hadoop topology. This layer is called the Nodegroup layer. All virtual machines running on the same physical machine belong to a single Nodegroup. In other words, a Nodegroup is per hypervisor. Communication between nodes in a Nodegroup does not have to go through the network and can give rise to some interesting block placement policies. The following diagram illustrates the four-layer architecture:



HDFS block placement

Replication is an important feature in HDFS; it ensures data reliability against loss and high availability in the face of failures. The default replication factor is 3, though this parameter can be tuned using the `dfs.replication` configuration parameter. HDFS does not replicate the file as a whole; rather, it chunks the file into fixed size blocks and stores it across the cluster.

The replication factor can be specified at file creation time. It can be changed as and when desired. The salient feature of HDFS is the smart placement of blocks, and this feature distinguishes it from other distributed filesystems. The placement policy is said to be rack-aware, that is, it is cognizant of the physical location of where the block resides. This not only aids in fault tolerance but can also be instrumental in making network bandwidth utilization efficient. Any computing paradigm running on HDFS can make use of this information to minimize the amount of data that needs to be moved across the network.

A rack is a bunch of machines stacked up. Generally, a single switch serves all the machines in a single rack. A data center is made of many such racks. The intra-rack network bandwidth is higher than the inter-rack network bandwidth. Therefore, it is potentially faster to move data within the same rack than between racks.

The NameNode is aware of the rack on which the DataNode resides. When placing replicas of a single block, the NameNode can decide which rack each block should reside in. To increase fault tolerance, the NameNode might decide to place each replica of a block in a separate rack. This helps in load balancing and data loss on rack failures. However, writing a block can take additional time as it has to span different racks.

The default rack placement policy, when the replication factor is 3, is to place a replica on a node in a rack, the second replica on a different node on the same rack, and the third replica on a node in a separate rack altogether. Generally, during a write, the first block is written on the same node as the client (for clients within the cluster). The next two blocks are written on the same random node off-rack. This helps in increasing write throughput as the writes are local. It does suffer a certain amount of skew as two-thirds of the replicas reside on the same rack. However, it again does not affect reliability and availability as the probability of rack failure is much less than the probability of a node failure. In the rare scenario of a rack failure, there is an additional copy on a different rack that can come to the rescue. If the client is not present on the cluster, a random first node is chosen to write the first copy of the block.

When a read is initiated, the NameNode tries to point the reader to the node that is closest to it. The hierarchy of selection in order of preference is to first try something on the same node as the reader first; if it is not present, it tries to get a read replica from the same rack as the reader. If it is not present in the same rack, it goes to another rack in the same data center, before moving on to another data center.

The block placement policies are now pluggable in Hadoop 2.X. This enhancement was done as a fix for ticket HDFS-385.

Pluggable block placement policy

HDFS now provides a pluggable block placement policy. This can be achieved by overriding the `BlockPlacementPolicy` abstract class in the `org.apache.hadoop.hdfs.server.blockmanagement` package. This abstract class has a few override methods. The `chooseTarget` override tells HDFS the placement choices. The `chooseReplicaToDelete` override is used to decide whether deleting a specific replica makes all the blocks conform to the block placement policy. There is a third override, `verifyBlockPlacement`, which verifies whether the block is present in `minRacks`. An `initialize` method is also provided to set up private variables of the `BlockPlacementPolicy` object.

Once the class derived by `BlockPlacementPolicy` is created, it is built into a JAR. The JAR can be placed in the Hadoop classpath. Hadoop then needs to be informed about the new block-placement policy. Introducing a property in the `hdfs-site.xml` file does this. A configuration property with the `dfs.block.replicator.classname` key is created, whose value contains the fully qualified class name of the `BlockPlacementPolicy` custom class, as shown in the following config file snippet:

```
<property>
    <name>dfs.block.replicator.classname</name>
    <value><Fully qualified class name of the custom block
        placement implementation class</value>
</property>
```

Out of the box, Hadoop has two block-placement-policy-derived classes that can be used. They are as follows:

- The `BlockPlacementPolicyDefault` class that provides a policy as discussed in this section
- The `BlockPlacementPolicyWithNodeGroup` class that deals with topologies that have the Nodegroup layer

Summary

Hadoop is made of the compute and storage layers. The compute layer has been replaced by YARN in Hadoop 2.X, helping other paradigms to co-exist on the Hadoop cluster hardware. The storage layer is making rapid progress towards a similar goal. Features such as HDFS Federation are one step closer in making the storage layer generic. By loosely coupling Block Storage from the Namespace, this can become a reality soon.

The key takeaways from this chapter are as follows:

- With HDFS Federation, it is possible to run multiple NameNodes. This not only helps in isolation, but it can also aid in performance by load balancing. Horizontal scaling of the NameNode is easier.
- Block pools are the abstractions that facilitate federation. Blocks from a single Namespace belong to a single pool. Each pool is given an identifier for addressability. The DataNodes remain shared among the different NameNodes.
- In Hadoop 2.X, there are a number of different options to ensure NameNode recovery from failures. Secondary NameNodes were the only option previously. Now, there are Checkpoint and Backup Nodes. All three strategies preserve the NameNodes directory structure and can be used by the NameNode to recover.
- The modern virtualized data center takes a four-layered approach for the Hadoop network topology. Communication between different nodes on the same physical hardware does not need to go over the network and can help optimize block placements.
- HDFS allows for pluggable block-placement strategies by overriding the `BlockPlacementPolicy` abstract class.

Hadoop and data security are important aspects of Hadoop. In this chapter, we saw isolation as one way to ensure separation of concerns. However, this does not suffice in large organizations with regulatory compliance requirements on data. More rigorous data security safeguards are required. We will look at Hadoop security in the next chapter.

11
Hadoop Security

Data is an asset to any organization. In this millennium, large companies such as Google have shown that gathering data and analyzing it can in itself be a product that can lead to an extremely successful business. This demonstration led to an explosion in data-driven decision-making for businesses and personalized experiences for the consumer. Data essentially became a high-value property for an organization. Just like any other asset, data needs to be protected.

Data security is the area that looks into protection of data. Security threats to data can come from outside the organization or from within it. Data theft is one of the highest reported cyber crimes. Recent studies have shown that data security threats are more common from within the organization, either by disgruntled personnel or inadvertently by benign users. A security feature such as authorization has become a baseline security feature for any data product.

In this chapter, we will look at the following topics:

- The security pillars
- Authentication in Hadoop
- Authorization in Hadoop
- Data confidentiality in Hadoop
- Audit logging in Hadoop

The security pillars

The four pillars of data security are as follows:

- **Authentication**: This refers to challenging a system or user to prove their identity. Only authenticated identities are allowed to gain access into the data system. Authentication in Hadoop can be of two major kinds, simple authentication and pseudo-authentication. The former is a loose kind of security where trust is placed on the user's assertion about their identity. In the latter, systems such as Kerberos are used for authenticating a user. In the industry, the latter is recommended as a best practice. Hadoop even supports seamless integration with a number of user stores such as LDAP and Active Directory. With the help of these stores, Kerberos can be implemented as an authentication mechanism.
- **Authorization**: Authorization refers to granting authenticated users access to data resources. In a multitenant system, or a multiteam organization sharing a single data cluster, policies, compliance, and regulatory norms might prohibit one team from accessing data belonging to another team. In such situations, it is very important to fence sensitive data resources from inadvertent or malicious access. Hadoop supports authorization at different levels. In HDFS, Hadoop provides fine-grained access control at the file level. The access control is very similar to any UNIX-based filesystem. The MapReduce compute layer also has **Access Control Lists** (**ACLs**) at a resource level. Hadoop services are allowed to have their own authorization features. For example, Hive tables can be protected using coarse-grained access control mechanisms, as in SQL.
- **Auditing**: Auditing is a mechanism to look into the usage patterns of a data system. A fundamental requirement to conduct any audit is to provide an accounting feature. All accesses and manipulations need to be recorded in an audit log to audit at a later point in time. Auditing is important to ensure compliance within an enterprise. Routine audits are conducted to ensure compliance with data policies. Some situations might call for ad hoc auditing, particularly when a security breach has happened in the system. Auditing can reveal forensic information that can help penalize the guilty and estimate the extent of damage caused by the breach. At the platform level, Hadoop supports auditing. At a service level, services such as Hive record all user-related actions in the metastore.

- **Data protection**: Big data systems are distributed across a number of machines. This mandates for the movement of data from one node to another. It might also involve storing data in untrusted domains, such as the cloud, for example. These two scenarios can lead to compromises in privacy and confidentiality. During transit, a man in the middle can sniff out the data that is being sent across the wire. An extremely malicious attacker can even manipulate the data. At rest, an untrusted party can snoop the data or modify it. Protection against such attacks can be done through cryptographic techniques. Data can be encrypted during transit and when at rest. Digital signatures can be generated to protect the data against modifications. Hadoop transports can be encrypted to ensure confidentiality over the wire. OS-level encryption can be used to protect confidentiality of HDFS data when at rest.

Hadoop addresses all these pillars to a satisfactory extent. We will look at some of these pillars in depth.

Authentication in Hadoop

Authentication in Hadoop can either be simple or in Kerberos. Hadoop also allows you to have your custom authentication scheme. In this section, we will look at Kerberos authentication and how the HTTP Hadoop interfaces can be secured via authentication.

Kerberos authentication

Kerberos is a network authentication protocol. It uses cryptography to provide a highly secure authentication mechanism. This authentication mechanism is popular because of its features, which are as follows:

- **Mutual authentication**: Both the client and server can mutually authenticate each other before proceeding with a session.
- **Single login per session**: Once a login happens, tokens with certain time validities are issued for usage. The duration of token validity defines the maximum length of the session.
- **Protocol message encryption**: All protocol messages during authentication are encrypted. It is not possible to conduct any man-in-the-middle or replay attacks by an adversary.

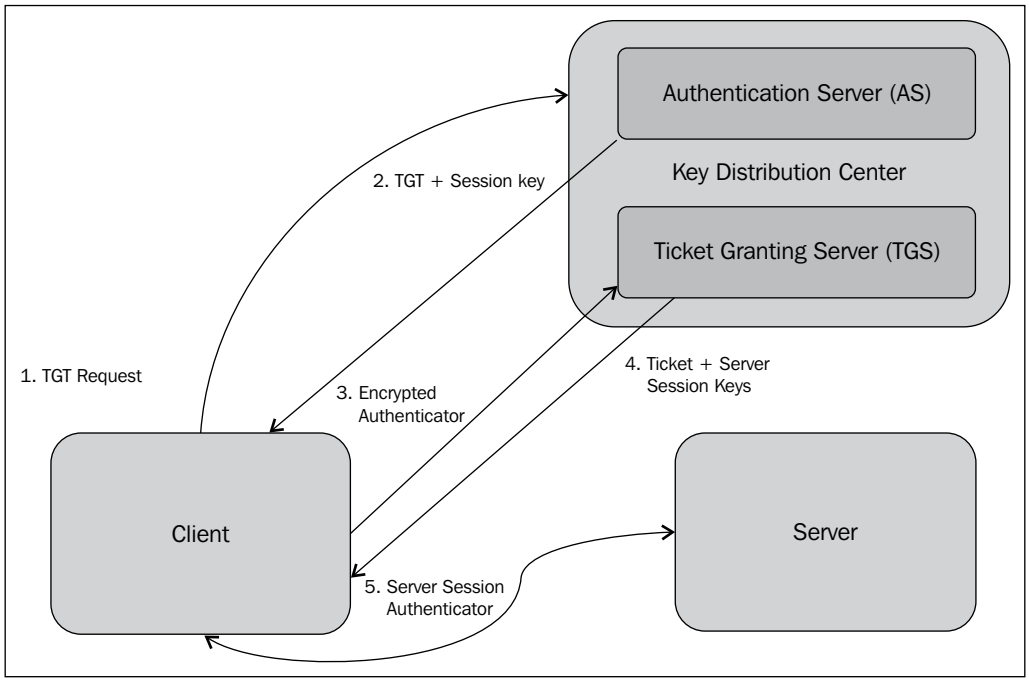
The Kerberos architecture and workflow

Central to Kerberos is the Kerberos **Key Distribution Center** (**KDC**). The KDC has the following two important components:

- The **Authentication Server** (**AS**)
- The **Ticket Granting Server** (**TGS**)

The Authentication Server is responsible for authenticating the validity of clients. The Ticket Granting Server is the one that issues tokens or tickets that are time-bound cryptographic messages, which can be used by the grantee to authenticate itself.

The following diagram shows the general protocol workflow when a client authenticates using Kerberos:



The authentication steps are as follows:

1. The protocol starts with the client requesting a **Ticket Granting Ticket** (**TGT**) with the AS in Kerberos.
2. The AS checks for the client in its database and sends back two messages. The first message is a session key, and the second is a TGT. Both messages are encrypted using the client's password as the key. The client can use the session key and the TGT only if the passwords match with what is stored by the AS.
3. A client wanting to access a service has to first go to the TGS with its identity. To prove its authenticity, the client has to send an authenticator encrypted by the session key that was received by it from the AS.
4. The TGS receives this request, decrypts it, and checks for validity of the client and request. On successful validation, the TGS grants a ticket with a validity period. A server session key is also returned to the client. The server session key comes in two copies—one encrypted with the client's secret and the other with the server's secret.
5. The client now presents the ticket, server session keys, and authenticator to the service that it needs access to. The server hosting the service validates the session key and grants access based on the result of the validation. If mutual authentication is desired, the server too sends back an authenticator that the client can check for validity. This is possible because the session key comes in two copies encrypted using the server and client secrets.

Kerberos authentication and Hadoop

An authenticating client within Hadoop requires a password to conduct the Kerberos authentication workflow. This might not be feasible for long-running MapReduce jobs that extend beyond the ticket validity time period. The `kinit` command can be used to initialize a client with a password in Hadoop. Though a ticket validity might be for a few hours, it is better to put the password in a `keytab` file for long-running jobs.

A `keytab` file can be created using the `ktutil` command. The `keytab` file can be given to the `kinit` command using the `-t` option. The `klist` command can be used to see the different tickets owned by a user. The `kdestroy` command can be used to expire tickets that are no longer in use.

Authentication via HTTP interfaces

By default, all HTTP web endpoints within a Hadoop cluster do not have authentication enabled. This means that anyone who knows the endpoint address can browse through the different services in the cluster. HTTP web interfaces can be explicitly configured to require Kerberos authentication using the HTTP SPNEGO protocol. This protocol is well supported on all major browsers.

Simple authentication can also be enabled. This involves appending the username in the web endpoint address as a query string parameter. The value of this parameter is the identity name of the user. Custom authentication schemes are also possible. All web endpoints within Hadoop support this extensibility as long as the `AuthenticatorHandler` class is overridden appropriately.

To configure HTTP authentication, the following properties in the `core-site.xml` file can be used:

- `hadoop.http.filter.initializers`: The `org.apache.hadoop.security.AuthenticationInitializer` class name needs to be put as the value of this property.
- `hadoop.http.authentication.type`: This property defines the authentication type. It can take the values `simple` or `kerberos`, or the class name of the custom `AuthenticatorHandler` derived class. By default, this is set to `simple`.
- `hadoop.http.authentication.token.validity`: The value of this property indicates the duration of validity of an authentication token. The value is in seconds, and the default is 36,000. After this duration, the token has to be renewed and presented.
- `hadoop.http.authentication.signature.secret.file`: The web endpoint secrets are kept in this file. The secret key is used to provide a digital signature for the authentication tokens.
- `hadoop.http.authentication.cookie.domain`: This is the whitelist of domains from where authentication tokens can be presented via cookies. There is no default value for this property.
- `hadoop.http.authentication.simple.anonymous.allowed`: This permits anonymous users to connect to the HTTP endpoint when set to `true`. The default value is `true`, and this works only when the authentication type is set to `simple`.
- `hadoop.http.authentication.kerberos.principal`: This value indicates the Kerberos principal name to be used when the authentication type is set to `kerberos`.
- `hadoop.http.authentication.kerberos.keytab`: This is the `keytab` file location for the Kerberos principal that is used in the HTTP endpoint.

Authorization in Hadoop

Authorization involves restricting access to resources. Hadoop provides authorization for both HDFS and all Hadoop services. In this section, we will look at how authorization can be enabled in Hadoop to secure shared resources against illegitimate access.

Authorization in HDFS

The HDFS authorization model is very similar to the authorization model in a POSIX system. In POSIX, each resource—files and directories—is associated with an owner user and a group. HDFS is similar to this. Permissions are given to each of these identities separately. There are separate permissions for:

- The owner of the resource
- The users of the group that are associated with the resource
- All other users within the system

There are two permissions levels, read and write. In contrast with POSIX, there is no execute permission on files in HDFS as files are not executables. Any user, or a user belonging to a group that has the read permission `r`, is only allowed to read the contents of a file from HDFS. Similarly, any user, or a user belonging to a group that has write permissions `w`, is allowed to write or append to existing files. A user or group can be given both read and write permissions `rw`.

For directories, the semantics change a bit. A read permission allows the user or users belonging to the group to list the contents of the directory. Write permissions allow the users or groups to create files and directories or append to files residing within the directory. Directories have a special execute permission `x`. This allows the user or group to access the children of a directory.

HDFS files do not have the concept of `setuid` and `setgid`, which are present in POSIX. Again, HDFS files are not executables, and it does not make sense to have these operations. In this case, even directories do not have `setuid` and `setgid`. HDFS directories do allow setting the sticky bit. This prevents any other user, other than the superuser, from manipulating the directory or its contents. However, setting the sticky bit on a file has no effect.

Like in any UNIX-based operating system, permissions are encoded as three octal numbers. The first octal number indicates the `rwx` bits for the owner, the second for the group, and the third for all other users. For example, the following command gives all permissions to the owner, read permissions to the group, and no permissions to other users in the system:

```
hadoop fs -chmod 740 masteringhadoop
```

The set of permissions for a file or directory is called a mode. It can be manipulated using the `chmod` command, which means change mode. When a process creates a file or directory in HDFS, it automatically assumes the identity of the process owner. However, the group is inherited from the parent directory.

A client operating on an HDFS file or directory presents the username and groups the user belongs to. HDFS first matches the username with the owner of the file or directory. If it matches, then a permissions check is done on the resource. If it does not match, a check is done to find out whether the user belongs to the group that is specified by checking the list of groups presented with the group associated with the resource. Again, a match makes a permission check against the requested operation. If both these matches fail, other permissions are checked for the user. If the permissions don't permit the operation in any of the three cases, the operation is rejected.

Identity of an HDFS user

As seen in the overview, Hadoop supports two mechanisms to authenticate a user. This is determined by the value in the `hadoop.security.authentication` property. The values can be of two kinds, which are as follows:

- `Simple`: This indicates that the identity of the user is determined and presented by the OS that runs the client process
- `kerberos`: This indicates that the identity of the user is determined by its Kerberos credentials

A key point to note is that HDFS cannot create, modify, or delete any identities. All identity management happens outside HDFS, either in the OS, as with simple authentication, or with Kerberos. HDFS simply uses the identity presented to it and performs authorization checks on the identity.

Group listings for an HDFS user

The user to group mapping services is determined by the value in the `hadoop.security.group.mapping` property. By default, this value is `org.apache.hadoop.security.ShellBasedUnixGroupsMapping`. When this particular implementation of the group mapping is used, the username is sent to a UNIX shell command to determine the list of groups the user belongs to. For example, with the bash shell, it is as follows:

```
bash -c groups
```

Enterprises can have their own user profile stores such as **Lightweight Directory Access Protocol** (**LDAP**) or **Active Directory**. In such cases, groups can be determined by asking these directory services for the group members. Hadoop has a built-in group-mapping service to connect to LDAP data stores to determine groups. Setting the `hadoop.security.group.mapping` property to the `org.apache.hadoop.security.LdapGroupsMapping` class does this.

The NameNode is responsible for invoking the appropriate API to determine the groups listing for a particular user. It is then presented to the data nodes. Also, all groups and usernames are stored as strings and not as numbers when compared to UNIX-based systems.



Revoking permissions in the middle of a client operation lets the client access the blocks of the file that are already known to it.

HDFS APIs and shell commands

All HDFS APIs throw an `AccessControlException` exception on the failure of permission checks. The `FsPermission` class in the `org.apache.hadoop.fs.permission` package is used to encapsulate the necessary permission-related information for a file or directory. `FileStatus` includes the `FsPermission` object. The `getFileStatus` method can be used to get the status of the files.

Additionally, the `FileSystem` class provides a couple of methods to set the mode and owner/group of a file or directory. The signature of the `setPermission` method is given as follows:

```
public void setPermission(Path path, FsPermission permission)
    throws IOException
```

It takes in the `Path` object to the file or directory and the permissions that need to go on it. Similarly, the signature of the `setOwner` method is given as follows:

```
public void setOwner(Path p, String username, String group)
    throws IOException
```

The `setOwner` method takes in the owner and group that need to be set on the file or directory specified in the `Path` object.

A few Shell commands are also present to change the authorization parameters for files and directories. Consider the following command:

```
hdfs chmod [-R] <octal mode> <file path>
```

This command can be used to change the mode of the file or directory. The `octal mode` parameter is given to specify the desired permissions for the owner, group, and others. The `-R` flag can be used to recursively apply the mode on all descendent files and directories until the leaf files or directories. Strictly, the owner or superuser is allowed to change the mode of a particular file or directory.

Similarly, there is a `chown` command that can be used to set the owner or the group for a particular file or directories. The command is as follows:

```
hdfs chown [-R] [owner][:[group]] <filepath>
```

The owner name and group can be specified along with the file path. The owner of the file can only be changed by the superuser and no one else. Again, the `-R` flag can be used to recursively change the owner of a directory and all its descendents.

The `chgrp` command can also be used to change the group a file belongs to. The command is specified as follows:

```
hdfs chgrp [-R] <group> <filepath>
```

Only an owner of the file or directory who belongs to the specified group is allowed to change the group. The superuser is also allowed to conduct this operation.

Specifying the HDFS superuser

In Hadoop, the HDFS superuser is the user under whom the NameNode runs. The superuser has ultimate privileges in HDFS. No permission checks fail for the superuser, and the superuser identity is allowed to execute all operations.

There is no permanent superuser identity. It is strictly determined by the identity that starts the NameNode service. It is not necessary for the superuser identity to be the administrator of the host that runs the NameNode though.

However, the administrator might also specify a group of users as superuser identities, though they might not be running the NameNode. Setting the configuration property `dfs.permissions.superusergroup` to the group of users that need to have superuser privileges does this.

HDFS provides a web interface to browse the filesystem. The web server hosting this interface runs under an identity. This identity is specified by the `dfs.web.ugi` configuration property. The value is a comma-separated list of the user and group that run the web interface. The user specified in this property can be the superuser as well. If the web server runs under superuser privileges, it has access to view the entire namespace. If it is set to a user and group that is not a superuser, access might be limited based on the permissions that are granted to the specified user or group. More than one group can also be specified in the comma-separated list.

Turning off HDFS authorization

The entire authorization feature is controlled by the `dfs.permissions` property. If it is set to `true`, all the permission-related rules and checks apply on each operation. If set to `false`, authorization is disabled. Turning off permissions does not alter the mode, user, or groups for the files and directories in the filesystem.

However, the `chmod`, `chown`, and `chgrp` commands that were discussed previously are not affected by turning permissions off. When these commands are executed, permissions are checked mandatorily.

Limiting HDFS usage

Even with adequate authentication and authorization setup, there is a possibility that a user or a group of users exceeds their fair share of resource usage. This can either be inadvertently due to faulty processes or through a compromised user trying to mount a denial of service on the Hadoop cluster.

HDFS provides quotas to limit usage. Quotas are imposed on the number of names and amount of space that can be used. These quotas are fixed at a directory level and applied to all files and directories that are descendants of the directory.

Name quotas in HDFS

The number of directories and filenames within a directory can be limited using the name quota. The name quota with a value 1 at a directory implies that no files or directories can be created below this particular directory. By default, a directory that has been created has no quota associated with it. The maximum quota that can be associated is `Long.Max_Value`. Setting a quota on a directory will succeed even if the directory violates the quota. The following command is used to set a quota on a directory or set of directories:

```
hdfs dfsadmin -setQuota <Quota> <dir1>….<dirn>
```

A quota can be removed from a directory or set of directories using the `-clrQuota` command. Its usage is as follows:

```
hdfs dfsadmin -clrQuota <dir1> ….. <dirn>
```

Space quotas in HDFS

The space usage under each directory can be limited using the space quota. The unit of space quota is in bytes. If the block allocation of a file within a directory exceeds this quota, the write fails. A zero quota allows a file to be created, but the file cannot be filled. This is because the metadata of files do not come under the quota. Even directories are exempt from being counted within space quotas. The maximum number of bytes that can be specified in the quota is the `Long.Max_Value` value.

File replications are accounted within quotas. A 1 TB file with a replication factor of three accounts for 3 TB of the quota. This has to be kept in mind when setting up quotas.

The space quotas in HDFS can be set using the following command:

```
hdfs dfsadmin -setSpaceQuota <Quota> <dir1>….<dirn>
```

Space quotas can be reset using the `-clrSpaceQuota` command, as shown:

```
hdfs dfsadmin -clrSpaceQuota <dir1>…. <dirn>
```

The `count` command on HDFS can be used with the `-q` switch to list all the quotas for the files and directories. If no quotas are set, the command displays `none` for the name quota value for this directory and `inf` (infinity) for the space quota value:

```
hdfs -count -q <dir1>…<dirn>
```

Service-level authorization in Hadoop

Hadoop has many services running in tandem, processing submitted applications and jobs. YARN has the Resource Manager (RM) that can run submitted applications. Application Masters (AMs) that take jobs as inputs and process them can be spawned. Similarly, HDFS has the NameNode service that provides a metadata store and directory service for HDFS. Authorization to access services is a mandatory security component in any framework.

In the Hadoop configuration directory, the `hadoop-policy.xml` file describes the authorization policies for service access. Authorization is defined by the use of ACLs that define the user or group and the type of access granted or denied to the user or group. These ACLs are checked at the beginning, much before the other authorization permission checks, such as the HDFS authorization.

Service-level authorization can be enabled by setting the `hadoop.security.authorization` property in the `core-site.xml` file to `true`. The Hadoop service-level authorization feature has a number of ACLs that can be defined to grant or restrict access to services. They are as follows:

- `security.client.protocol.acl`: This property determines the permissions on usage of the distributed filesystem client via the Hadoop APIs. A user or group who is granted access via an **access control entry** (**ACE**) can make calls to the NameNode service.
- `security.client.datanode.protocol.acl`: This property determines the users and groups who can access the DataNode within the Hadoop cluster. A user or group who is granted access via an ACE can call APIs on the DataNode. This is generally done in block recovery scenarios.
- `security.datanode.protocol.acl`: This property determines the ACEs that grant permissions for the DataNodes to communicate and access the NameNode.
- `security.inter.datanode.protocol.acl`: This property determines the ACEs that grant permissions for the DataNodes to communicate with other DataNodes. They are generally used to update generation timestamps. Generation timestamps are used in block write failure scenarios.
- `security.namenode.protocol.acl`: This property determines the kind of permissions the secondary NameNode has when communicating with the primary NameNode.
- `security.inter.tracker.protocol.acl`: This property determines the kind of permissions the task tracker has when communicating with the job tracker.

- `security.job.submission.protocol.acl`: This property determines the permissions job submission clients have. These clients are used to submit jobs or query job statuses.
- `security.task.umbilical.protocol.acl`: This property determines the permissions the Map or Reduce tasks have when communicating with the parent `TaskTracker` process.
- `security.refresh.policy.protocol.acl`: This property determines the permissions on the `dfsadmin` and `mradmin` commands for policy refresh.
- `security.ha.service.protocol.acl`: This property determines the permissions on the HAAdmin for active and standby NameNode management. This protocol ACL deals with NameNode high availability only.

Each ACL is a list of users followed by a list of groups. Each user list is comma separated. Each group list is comma separated as well. The user and group lists are separated by a space. For example, `u1, u2 g1, g2` represents an ACL for users `u1` and `u2` and groups `g1` and `g2`. A `*` can be specified as a wildcard for all users.

Any service-level authorization can be refreshed without having to restart the NameNode or any other daemons. Both the `dfsadmin` and `mradmin` commands have the `-refreshServiceAcl` option to reload the configuration.

The following XML is a snippet from a sample `hadoop-policy.xml` file:

```
<property>
    <name>security.job.submission.protocol.acl</name>
        <value>u1,u2 g1</value>
</property>
<property>
    <name>security.client.protocol.acl</name>
        <value>* </value>
</property>
```

The first ACL permits the users `u1` and `u2` to submit jobs in the cluster. In addition, all users belonging to group `g1` can also submit jobs. The second ACL grants all users to access HDFS.

Data confidentiality in Hadoop

Hadoop is a distributed system. All distributed systems are interconnected via a network. Networks are vulnerable to malicious sniffing of data. Data at rest can also be read if they are not protected via encryption.

Data confidentiality for data at rest is delegated to the OS that hosts the DataNode. Most modern OSes provide encryption schemes to protect data on disks under their purview. In this section, we will look at confidentiality over the wire and how to enable encryption when data is in transit.

HTTPS and encrypted shuffle

Encrypted shuffle is a feature that facilitates data confidentiality in the shuffle process. To recap, the shuffle step is where data moves from the Map to the Reduce tasks in a MapReduce job life cycle. The movement of data occurs across machines through the network. The transport used to move this data across the network is HTTP.

HTTP, by itself, sends data in clear text, that is, in an unencrypted form. This can lead to information leak when an adversary snoops in the network. HTTPS is the secure form of HTTP, where all packet payloads between the HTTP endpoints are encrypted using **Secure Socket Layer** (**SSL**). Hadoop allows for encrypted shuffle by facilitating HTTPS communication between the Map and Reduce task nodes.

Optionally, Hadoop also allows for client authentication. Encrypted shuffle comprises configuration settings to achieve:

- Switching off the shuffle process between HTTP and HTTPS
- Specification of a keystore and truststore to facilitate HTTP encryption
- Reload of trust stores when nodes are added or decommissioned

SSL configuration changes

The encrypted shuffle configuration requires SSL. To enable SSL, the changes required are as follows:

- In the `core-site.xml` file, the `hadoop.ssl.require.client.cert` property is set to `true` if client certificates are used. By default, this value is `false`.
- The `hadoop.ssl.hostname.verifier` property is used to specify the level of strictness when making an SSL connection. The `HttpsUrlConnection` class in Java uses this value to determine whether connections should be allowed to go through. The framework compares the identity of the server in the authentication scheme with the actual server identity to decide granting or rejecting the connection. This property can take the values `DEFAULT`, `STRICT`, `STRICT_I6`, `DEFAULT_AND_LOCALHOST`, and `ALLOW_ALL`. The default value is `DEFAULT`. `ALLOW_ALL` is the weakest form of verification. This value is specified in `core-site.xml`.

- The `hadoop.ssl.keystores.factory.class` property indicates the class name to be used to implement and manage keystores. By default, the value is `org.apache.hadoop.security.ssl.FileBasedKeyStoresFactory`. This value is specified in `core-site.xml`.
- The `hadoop.ssl.server.conf` property indicates the configuration file that is used to configure SSL on the server side. The default value is `ssl-server.xml`. This file is looked in the class path for availability. The values of this configuration file configure the keystore and other SSL properties. This value is specified in `core-site.xml`.
- The `hadoop.ssl.client.conf` property is similar to the preceding property, but defines the client-side SSL properties. The default value is `ssl-client.xml`, and it needs to be present in the class path.

All of the previous properties have to be marked as `final`, indicating that they cannot be overridden in any other configurations either by the framework or the user. The previous properties have to be set on all the nodes of the cluster.

The following configuration snippet shows a sample `core-site.xml` configuration:

```
<property>
    <name>hadoop.ssl.require.client.cert</name>
    <value>false</value>
    <final>true</final>
</property>
<property>
    <name>hadoop.ssl.hostname.verifier</name>
    <value>DEFAULT</value>
    <final>true</final>
</property>
<property>
    <name>hadoop.ssl.keystores.factory.class</name>
    <value>org.apache.hadoop.security.ssl
            .FileBasedKeyStoresFactory</value>
    <final>true</final>
</property>
<property>
    <name>hadoop.ssl.server.conf</name>
    <value>ssl-server.xml</value>
    <final>true</final>
</property>
<property>
    <name>hadoop.ssl.client.conf</name>
    <value>ssl-client.xml</value>
    <final>true</final>
</property>
```

The preceding properties set up SSL between the nodes so that HTTPS can be used as the transport for communication. To enable HTTPS in the encrypted shuffle process, the `mapreduce.shuffle.ssl.enabled` property can be set to `true` in `mapred-site.xml`. By default, this property is set to `false`. Again, this property is not overridable and has to be set as `final`. The following code snippet shows the default configuration for this property in `mapred-site.xml`:

```
<property>
    <name>mapreduce.shuffle.ssl.enabled</name>
    <value>true</value>
    <final>true</final>
</property>
```

Configuring the keystore and truststore

`FileBasedKeyStoreFactory` is the only keystore implementation available in Hadoop out of the box. The settings for truststores and keystores are in the files specified as values for the properties `hadoop.ssl.server.conf` and `hadoop.ssl.client.conf`.



Keystores and **truststores** have very similar structures. They are used to store private keys and certificates. Functionally, though, they serve different goals. A keystore is used to store credentials that need to be presented during an SSL connection. Generally, a keystore is used to store private keys and public key certificates that can be used to initiate a secure remote connection. If an SSL server is being launched or a server does client authentication, a keystore is a must to store the necessary keys and certificates.

A truststore, in contrast, is used to verify credentials when a connection is established. They generally contain third-party certificates such as root certificates or certificates that are signed by certificate authorities that identify and endorse endpoints.

The keystore and truststore can be the same file. However, generally, it is good practice to keep them separate.

The `ssl-server.xml` file can be configured with the properties listed in the following table:

Property name	Description
`ssl.server.keystore.type`	This is the type of keystore file. Java keystores are of type `jks`. The default value for this parameter is `jks`.
`ssl.server.keystore.location`	This is the location of the keystore file on the local node. The user running any MapReduce jobs needs at least read access to this file.
`ssl.server.keystore.password`	Each keystore and truststore file is password protected. The password for the keystore is specified here.
`ssl.server.truststore.type`	This is the type of truststore file. The default value is `jks`.
`ssl.server.truststore.location`	This is the file path of the truststore.
`ssl.server.truststore.password`	This is the truststore password.
`ssl.server.truststore.reload.interval`	This is the number of milliseconds after which a reload of the certificates must happen from the truststore. The default value is 1,000, indicating 10 seconds.

A sample `ssl-server.xml` configuration is given as follows:

```
<configuration>
<!-- Keystore Configurations -->
<property>
    <name>ssl.server.keystore.type</name>
    <value>jks</value>
</property>
<property>
    <name>ssl.server.keystore.location</name>
    <value>${user.home}/keystores/certstore.jks</value>
   </property>
<property>
    <name>ssl.server.keystore.password</name>
    <value><your keystore password></value>
</property>

<!-- Truststore configurations -->
<property>
```

```
    <name>ssl.server.truststore.type</name>
    <value>jks</value>
</property>
<property>
    <name>ssl.server.truststore.location</name>
    <value>${user.home}/keystores/castore.jks</value>
</property>
<property>
    <name>ssl.server.truststore.password</name>
    <value><your truststore password></value>
</property>
<property>
    <name>ssl.server.truststore.reload.interval</name>
    <value>10000</value>
</property>
</configuration>
```

The `ssl-client.xml` file can be configured with the properties in the given following table:

Property name	Comments
`ssl.client.keystore.type`	This is the type of keystore file. Java keystores are of type `jks`. The default value for this parameter is `jks`.
`ssl.client.keystore.location`	This is the location of the keystore file on the local node. A user running any MapReduce jobs needs at least read access to this file.
`ssl.client.keystore.password`	Each keystore and truststore file is password protected. The password for the keystore is specified here.
`ssl.client.truststore.type`	This is the type of truststore file. The default value is `jks`.
`ssl.client.truststore.location`	This is the file path of the truststore.
`ssl.client.truststore.password`	This is the truststore password.
`ssl.client.truststore.reload.interval`	This is the number of milliseconds after which a reload of the certificates must happen from the truststore. The default value is 1,000, indicating 10 seconds.

A sample `ssl-client.xml` configuration file is given as follows:

```
<configuration>
<!—Keystore configuration settings -->
<property>
    <name>ssl.client.keystore.type</name>
    <value>jks</value>
</property>
<property>
    <name>ssl.client.keystore.location</name>
    <value>${user.home}/keystores/clientcertstore.jks</value>
</property>
<property>
    <name>ssl.client.keystore.password</name>
    <value><your keystore password></value>
</property>
<!—Truststore configuration settings -->
<property>
    <name>ssl.client.truststore.type</name>
    <value>jks</value>
</property>
<property>
    <name>ssl.client.truststore.location</name>
    <value>${user.home}/keystores/clientcastore.jks</value>
</property>
    property>
    <name>ssl.client.truststore.password</name>
    <value><Your truststore password></value>
</property>
<property>
    <name>ssl.client.truststore.reload.interval</name>
    <value>10000</value>
</property>
</configuration>
```

Once the settings are in place, the encrypted shuffle can be activated by restarting all the `NodeManagers` (NMs) in the cluster. Encrypted shuffle will add processing overheads as the shuffle steps will have to do encryption and decryption in addition to its existing duties.

The SSL connections can be debugged on the Reduce task nodes. This is done by setting the `mapreduce.reduce.child.java.opts` property value to the `javax.net.debug=all` Java option. This can be done on a per-job basis or in `mapred-site.xml`, so all the jobs in the cluster can be debugged. The following snippet shows how this option can be set in `mapred-site.xml`:

```
<property>
    <name>mapred.reduce.child.java.opts</name>
    <value>-Djavax.net.debug=all</value>
</property>
```

The debug property should be used prudently only for debugging. When used, it slows down the job executing with this option. Debugging can also be enabled on the `NodeManager` by setting the following environment variable:

```
YARN_NODEMANAGER_OPTS="-Djavax.net.debug=all"
```

Audit logging in Hadoop

Audit logging is an accounting process that logs all operations happening in Hadoop. HDFS and the MapReduce engine logging are already present in Hadoop via the `log4j` properties. Audit logs use the same framework, but they log more events and give higher resolution into Hadoop operations. The file that is used to configure logging is the `log4j.properties` file.

By default, the `log4j.properties` file has the log threshold set to `WARN`. By setting this level to `INFO`, audit logging can be turned on. The following snippet shows the `log4j.properties` configuration when HDFS and MapReduce audit logs are turned on:

```
#
# hdfs audit logging
#
hdfs.audit.logger=INFO,NullAppender
hdfs.audit.log.maxfilesize=256MB
hdfs.audit.log.maxbackupindex=20
log4j.logger.org.apache.hadoop.hdfs.server.namenode.FSNamesystem
    .audit=${hdfs.audit.logger}
```

```
log4j.additivity.org.apache.hadoop.hdfs.server.namenode
    .FSNamesystem.audit=false
log4j.appender.RFAAUDIT=org.apache.log4j.RollingFileAppender
log4j.appender.RFAAUDIT.File=${hadoop.log.dir}/hdfs-audit.log
log4j.appender.RFAAUDIT.layout=org.apache.log4j.PatternLayout
log4j.appender.RFAAUDIT.layout.ConversionPattern=%d{ISO8601} %p
    %c{2}: %m%n
log4j.appender.RFAAUDIT.MaxFileSize=${hdfs.audit.log.maxfilesize}
log4j.appender.RFAAUDIT.MaxBackupIndex=${hdfs.audit.log
    .maxbackupindex}

#
# mapred audit logging
#
mapred.audit.logger=INFO,NullAppender
mapred.audit.log.maxfilesize=256MB
mapred.audit.log.maxbackupindex=20
log4j.logger.org.apache.hadoop.mapred.AuditLogger=${mapred.
    audit.logger}
log4j.additivity.org.apache.hadoop.mapred.AuditLogger=false
log4j.appender.MRAUDIT=org.apache.log4j.RollingFileAppender
log4j.appender.MRAUDIT.File=${hadoop.log.dir}/mapred-audit.log
log4j.appender.MRAUDIT.layout=org.apache.log4j.PatternLayout
log4j.appender.MRAUDIT.layout.ConversionPattern=%d{ISO8601} %p
    %c{2}: %m%n
log4j.appender.MRAUDIT.MaxFileSize=${mapred.audit.log.maxfilesize}
log4j.appender.MRAUDIT.MaxBackupIndex=${mapred.audit.log
    .maxbackupindex}
```

The `hdfs.audit.logger` and `mapred.audit.logger` properties are turned on by setting the level to `INFO`. These properties are then assigned to `log4j` properties such as `log4j.logger.org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit`. Other properties can be set appropriately to control the logging.

Summary

Security becomes a primary feature in a multitenant and distributed environment. Networks and resource sharing can potentially lead to information leaks via unauthorized access, malicious modifications, or even denial of service. Preempting attacks on Hadoop clusters can be done by enabling security features such as authentication, authorization, data protection, and data auditing.

The key takeaways from this chapter are as follows:

- Post 0.20, Yahoo! introduced Hadoop security-related features for compliance, confidentiality, and fair usage in shared enterprise clusters.
- Hadoop can now be configured for Kerberos-based authentication or simple authentication based on the topology and compliance requirements. User information can be retrieved from enterprise user stores such as LDAP or Active Directory.
- Hadoop has both service-level and resource-level authorization built in. HDFS authorization is very similar to the UNIX-based file authorization model.
- Hadoop provides data confidentiality by facilitating HTTPS in MapReduce shuffle and web endpoints. HTTPS can be turned on by tweaking a few parameters. Data confidentiality for data at rest is delegated to the node's operating system.
- Accounting is very important for compliance and forensics in an enterprise. Hadoop uses the log4j logging framework to provide audit logs.

In the next chapter, we will look at Hadoop applications, particularly in the field of big data analytics.

12
Analytics Using Hadoop

Hadoop has come to the fore because of its capability to aid in data analytics. As data grows in the dimensions of volume, velocity, and variety, there needs to be systems that are capable of analyzing this data efficiently and effectively. Vertically scaling hardware to handle this data is not viable because it is expensive and difficult to manage. Distributed computing and horizontal scaling are good options, and frameworks such as Hadoop automatically cater to the fault tolerance, scaling, and distribution needs of such a system.

Analytics is all about data. A question that frequently arises is when does Hadoop become overkill? Typically, it is recommended that you use Hadoop for datasets of 1 TB and upwards. However, when it becomes difficult to predict the rate of data growth, it may be a good idea to use Hadoop MapReduce because of its attractive "code once, deploy at any scale" characteristic.

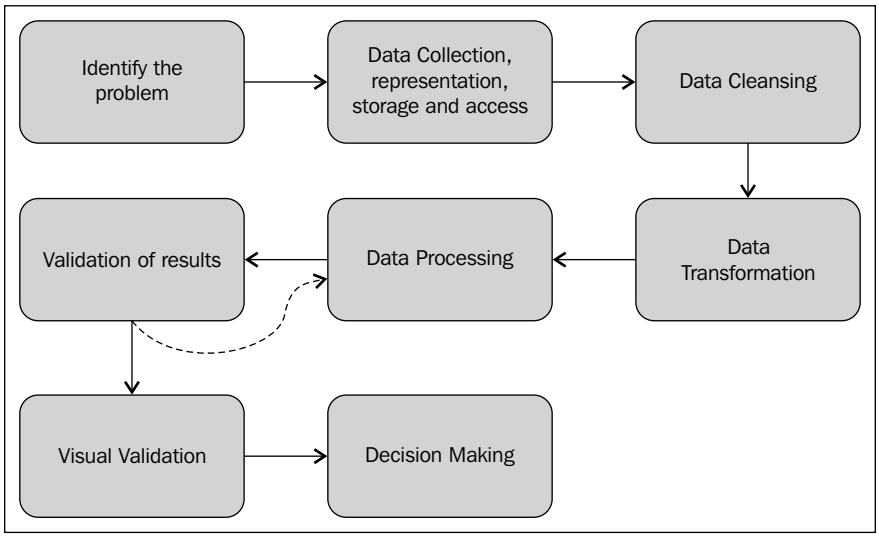
There are organizations that use Hadoop to analyze a few hundreds of gigabytes of data as well. The smaller the dataset the better, since the user has to take into account latency costs due to long startup times and disk access associated with Hadoop jobs. The functional aspect of Hadoop MapReduce makes it easy to code and port over complex analytics functions. In some situations, when datasets are smaller and traditional SQL becomes unwieldy due to the nature of the analysis, it may be prudent to use Hadoop and directly interact with the filesystem.

In this chapter, we will look at the following topics:

- The workflow for data analytics
- A brief introduction to machine learning
- The basics of Apache Mahout
- Document analysis as a data analytics case study using Pig and Mahout

Data analytics workflow

Data analytics involves transforming and inspecting data to figure out the inherent meaningful information from it. The information extracted is used in decision making or suggesting conclusions. The analytics workflow is shown in the following diagram:



The steps involved in the analytics workflow are as follows:

1. The first step is to identify the problem to be solved. This is important as the decisions in the rest of the steps hinge on it. For example, the problem statement will dictate what kind of data to collect and what the important features that represent the solution to the problem are. A lot of domain expertise is required in data analytics, and a problem space where expertise is accessible is almost mandatory.
2. Once the problem is identified, appropriate data needs to be collected. The collected data needs to be represented in a format that optimizes on space without losing resolution in information. Enterprises now need to be aware of compliance and security. Access to the data might need to be restricted to authorized personnel and data could be confidential in some cases.
3. The stored data needs to be cleansed. Cleansing involves removal of outliers, missing values, and bad records. The result of the data analysis depends much on cleansing of the data. Data that is not cleansed might lead to skewed analysis.

4. The cleansed data is transformed into a representation that can be used for analysis. One example of transformation is normalization of data to a range between 0 and 1. Another example is changing the scale of the data to ease computation.
5. The transformed data is then analyzed using algorithms. Machine learning algorithms are a class of analysis algorithms that offer solutions based on previously known empirical evidence.
6. Once the data is transformed and results are obtained, they need to be validated. Validation can occur by either consulting with domain experts or by deploying it to a test set of users. Post-validation, if the results can be used to make meaningful decisions, it ends the process of analysis. Otherwise, the data scientist and associates get back to the drawing board and tweak the parameters in the pipeline.
7. The validated results are visually represented for the stakeholders (which could include users) to validate. The apt visual representation is decided at this stage.
8. Finally, the results are used to drive decisions.

Machine learning

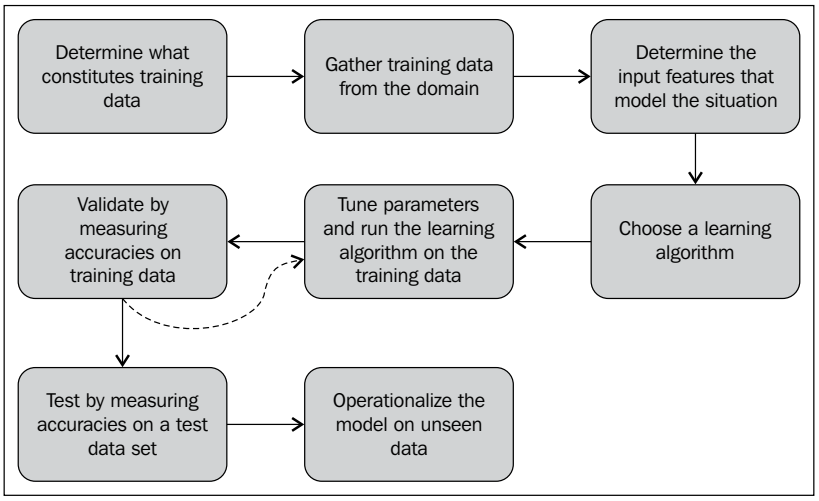
Machine learning is about programming computers to optimize a function based on previous experience. The computer is given empirical data to analyze and build a model function that can predict the output on unseen data that it might encounter in the real world. The computer builds a function based on the parameters and the empirical data supplied to it. This function evolves as more empirical data is given or when there is a change in the data characteristics. When this function is applied on unseen data at a later point, it predicts the output based on the model function. The empirical data supplied to learn this function is termed as **training data**.

The following are the kinds of machine learning algorithms:

- **Supervised learning**: The training data supplied to supervised learning methods is labeled. Each data point in the training dataset is a pair of objects, the actual data point representing the situation, which is generally a vector of values, and the desired output value for this situation. An expert who understands the domain of the data annotates the desired output value for the situation. This desired output is also called a **label** or a supervisory signal. The algorithm runs on the set of training data to infer a mathematical function. The function is as generalized as possible and is termed as a model. When this function is applied on any unseen data, it gives an output value. This model function's accuracy in prediction determines its strength.

- **Unsupervised learning**: The training data supplied to unsupervised learning methods is unlabeled. The learning algorithm is asked to determine the underlying structure in the data. Clustering is an example of an unsupervised learning method, where groups are determined from unlabeled data based on a distance metric between the data points.
- **Semi-supervised learning**: This takes a hybrid approach wherein unlabeled empirical data is given to the computer along with a few labeled data points.

The following diagram shows the machine learning process:



The steps in machine learning process are as follows:

1. The first step in machine learning is to determine a problem to be solved and what constitutes training data for the problem in hand. This involves figuring out how granular a training data point should be and also the number of points. Expertise in the domain of the problem can be very useful in determining the granularity and size of the training data set.
2. Once it is determined, the next step would be to actually gather the training data from the real world. Training data gathered may need to be annotated or labeled by an expert in the case of supervised learning. The optimal size of the labeled training data might become crucial in determining the accuracies of the modeling function. In general, getting an expert to label training data is very expensive and needs proper planning. Also, because of the manual nature of labeling, it does not scale well to a large number of data points. Semi-supervised techniques are becoming popular because of problems associated with getting labeled training data.

3. The training data point now needs to be broken down into a set of features. These are characteristics of the data point that accurately represent the situation when the data point was collected. Selection of the right features is critical to get good model functions. A lot of features could mean slower processing and too little features may lead to less accuracy.
4. The next step is to choose a good learning algorithm to learn the function. Depending on the nature of the problem, there are a number of algorithms. Classification algorithms output model functions that can determine the class to which a particular data point belongs. A clustering algorithm takes a set of data points and groups them into a number of groups based on a measure of distance.
5. Once a suitable algorithm is chosen, it is fed with the training data and parameters. The output of a learning algorithm is the model function. The learning parameters can be used to tune the characteristics of the model function. For example, a regularization parameter can be used to generalize the model function to solve the problem of **overfitting**. In a clustering scenario, a parameter can determine the number of clusters that need to be determined as the output of the learning algorithm.
6. Validation is one of the most important steps in machine learning workflows. It involves determining the strengths and weaknesses of the learned model. Using the learned model function to predict on random subsets of the training dataset itself does validation. In a supervised setting, since we know the labels beforehand, this information can be used to evaluate the accuracies of the learned model. If it is found to be less accurate, we can go back to step 4 or 5 to either change the algorithm or tune the parameters for a better model function.
7. A validated machine learning model is then used to predict the output on a test dataset. This is also a labeled dataset outside the training dataset. Operating parameters and characteristics of the model can be determined on this dataset. The operating parameters can then be used to predict unseen data points in the real world.
8. The final step is to deploy and operationalize the learned model so that it operates with the operating parameters determined in step 7. Unseen data is supplied to the learned model function and it is asked to predict the output. The predicted output can be used to drive business decisions.
9. Periodically, the models are updated either using newly gained knowledge from the field or in the form of feedback from users and stakeholders. Newer training data could be collected and steps 1 through 8 can be performed to update the model.



Often, the terms machine learning and data mining appear in the same context. **Data mining** is a field that involves discovering patterns from large sets of data. The differences between machine learning and data mining are as follows:

- Machine learning methods can be used as tools in the data mining process
- Machine learning solves specific tasks while data mining is exploratory in nature
- Machine learning deals with accurately identifying known information on unseen data, while data mining deals with discovering unknown information on data

Apache Mahout

Apache Mahout is a scalable machine learning library. It is an open source library under the Apache Software Foundation. It supports algorithms for clustering, classification, and collaborative filtering on distributed platforms. Apache Mahout welcomes contributors to contribute any algorithm to the library. The algorithm coded may not always be distributed and can run on a single machine as well.



As Apache Mahout allows developers to introduce single-machine algorithms, it is recommended that you study the implementation before running it on Hadoop.

Apache Mahout has a few algorithms that are implemented as MapReduce. These algorithms can be run in Hadoop to exploit the parallelism on a distributed cluster. Again, a word of caution for you is to study the implementation of an algorithm before using it in your Hadoop deployments. A non-MapReduce algorithm may not yield any speedup when run on a Hadoop cluster.



In a recent change, since April 2014, Mahout has stopped accepting algorithms that are programmed in the MapReduce model. However, Mahout has made a commitment to support all the algorithms that are already in its library and programmed using the MapReduce model.

The following is a list of use cases supported by Apache Mahout and the algorithms that can be run on Hadoop and exploit the parallelism provided by any Hadoop cluster categorized by the use case:

- **Classification**: This is a supervised method that learns how to place data points in different classes. Unseen data is then put into one of these classes. In the classification use case, Mahout supports parallelized implementation for the Bayesian classifier and the Random Forest classifier. The Bayesian classifier uses the Bayes rule and conditional probabilities to do binary classification. The Random Forest classifier is a decision-tree-based classifier at its core, but uses a collection or ensemble of decision trees.
- **Clustering**: This is an unsupervised method that categorizes training data points into coherent groups. Mahout supports distributed implementations of many clustering algorithms. K-means, a very popular clustering algorithm, has an implementation that is parallel and distributed. The core of K-means is to group data points such that the mean distance between the points is minimum. The fuzzy k-means clustering algorithm is also compatible with Hadoop. In this clustering method, the clusters are soft and allow multiple cluster memberships for a particular data point. Hierarchical, **Latent Dirichlet Allocation** (**LDA**), Mean shift, MinHash, Dirichlet Process, Canopy, and Spectral clustering have got distributed implementations in the Mahout library.
- **Collaborative filtering**: This makes recommendations based on the user data that is available. Distributed-item-based collaborative filtering and parallel-matrix-factorization-based collaborative filtering algorithms have Hadoop-compatible implementations. The former uses the user's preferences for other items to predict the preference for a similar item. The latter predicts the preference of the user from a matrix of unseen items.
- **Frequent itemset mining**: This is also called market-basket analysis where the algorithm analyzes which other item typically appears along with the item in hand. There is a parallel implementation of the Parallel FP growth algorithm to determine item associations.

Document analysis using Hadoop and Mahout

In this section, we will take an example of document analysis to illustrate analytics using Hadoop and Mahout. We will be using Pig as the higher-level abstraction for Hadoop MapReduce. We will be calculating the distance between documents using a score called **Tf-idf**. This distance metric is very popular in the field of information retrieval and text analytics. It is based on the statistics of words occurring in a document.

Tf-idf is used to rank documents based on query terms. It is extensively used in text search scenarios. The distance between the query terms and the document terms determines how close the query is with respect to the document. This distance can be used to rank documents.

For this particular example, we will be using the NSF grants abstracts that are available at `http://kdd.ics.uci.edu/databases/nsfabs/nsfawards.html`. The dataset consists of about 120,000 abstracts and comes in three parts. Each grant abstract is a separate text file.

Tf-idf stands for **Term Frequency – Inverse Document Frequency**. It is the product of two metrics: term frequency (*tf*) and inverse of the document frequency (*df*).

Term frequency

As the words suggest, term frequency is the number of times a word occurs in a particular document. The more the occurrences of a word in a document, the stronger the association of that word with the document. For example, if the word "Hadoop" appears 10 times in document A and 15 times in document B, document B is more relevant in the context of the word Hadoop. This simple intuition drives the calculation of the term frequency metric.

Documents may have varying numbers of words in them. A larger document, say of 1,000 words, may have the word "Hadoop" appear 10 times, while a smaller document, say of 100 words, may have it five times. It might be unfair to say that the term Hadoop is more relevant in the larger document, because the percentage of occurrences of the term Hadoop relative to all the terms in the smaller document is larger. Therefore, when the term frequency is calculated, dividing it by the number of terms in the document normalizes it.

In essence, the term frequency for a term `t` and document `d` is given by the following formula:

```
Term Frequency (t,d) = Number of times t appears in d / Number of
terms in d
```

Document frequency

Using the term frequency alone might not do justice in determining the importance of a term relative to a document. There are terms in the English lexicon that occur very frequently in documents. For example, terms such as "and", "the", and "in" occur very often compared to other words. These words are called stop words.

There could be other words that occur frequently based on where the document repository is coming from. For example, documents coming from an organization might have the name of the organization appearing all over them. These terms, like the stop words, do not contribute to any signal when comparing the importance of a set of terms with a document.

Document frequency is used to eliminate or reduce the effect of these frequently occurring terms from the calculation of the distance measure. Calculating the number of documents in which a term appears in the entire document corpus or repository does this. More the value, less is the signal the term adds in differentiating the document. Therefore, the inverse of the document frequency is taken to reduce the effect of such terms appearing very frequently within the document.

In essence, the inverse document is calculated as follows:

```
inverse document frequency (t) = log(number of documents in the
corpus/number of documents the term t appears in)
```

The document frequency is divided by the number of documents in the corpus to normalize the values and make it a number between 0 and 1. The logarithm of the entire score is taken to keep the value within a reasonable range as the number of documents could be huge, like in a web corpus.

Term frequency – inverse document frequency

The product of the term frequency and the inverse document frequency for a given term and document gives the importance of the term in that particular document:

```
Tf-idf for term t and document d = tf for term t and document d *
idf for term t
```

It is important to understand that there is a Tf-idf score for each term and each document of a document corpus. These Tf-idf scores are stored in inverted indexes of full-text search engines and are used to measure distances between the query terms and the documents. The inverted index stores these values in sparse format, that is, a Tf-idf score is associated with a document only if a term is present in it. A term not appearing in a document has a Tf-idf score of zero as the term frequency in the document is zero. These zero-valued Tf-idf scores are not stored within the index.

Tf-Idf in Pig

The following steps illustrate how to calculate Tf-idf for the NSF grants abstract document corpus that was described previously:

1. The prerequisites are loading all the documents into the HDFS. This can be done using the `hadoop fs -cp` command. The next step will be to load the files into a Pig relation so that we can run data calculations and transformations on the relation. The `PigStorage` class is used to read a document into a relation that has the filename and the sentence in the document as `chararray` (`file_and_sentence`). We use the `-tagsource` directive to inform the `PigStorage` class to tag the filename along with the tuple in the relation. This is needed to identify a particular document. Since the Tf-idf scores are term and document dependent, the filename acts as the document identifier.
2. Once the relation is loaded, the next step is to tokenize the sentence into its words. These words are used as terms for further calculations. We use the `TOKENIZE` Pig function to split the sentence into words. A more sophisticated regular expression can also be used to tokenize the sentence. The output is another relation, `file_and_words`, that contains the filename and the terms associated with that sentence in the file. We will get a number of such tuples for a single file depending on the number of sentences that are present in that file.

3. These tuples are now passed through a `filter` statement to remove all words that don't contain alphanumeric characters. The regular expression `\w+` is used to do this filtering. In reality, a stop words list is also used to filter out commonly occurring words.
4. Next, all the remaining terms are converted to lowercase strings so that the same terms are uniformly represented. This is a very simplistic kind of transformation on the terms. In practice, operations such as **stemming** and **lemmatization** that help in representing variants of a word as the same word or term are done at this stage. For example, swimmers or swimmer can be represented as a single word, that is, swimmer. There are many stemming algorithms, and most well-known natural language processing libraries have built-in stemmers. **Porter Stemmer** is a very popular stemmer:

```
/* This is a Pig template file to get tf-idf calculated for
all the NSF grants. Before starting off,
* 1) Unzip the zip file in your local directory
* 2) Use bin/hadoop fs -cp
* /

/* We have to load all the files in hdfs grants directory.
This can be done using the command below. *The nice thing
about the PigStorage class is that it helps us get the file
name along with the *sentence in the file. At the end of
this command, you will have (file_name, sentence) tuples to
go *further. Please take care to change the hdfs load
directory if necessary*/


file_and_sentence = load 'grants/*' using PigStorage('\t',
    '-tagsource') as (file_name: chararray, sentence:
        chararray);


/*We now split each sentence using the TOKENIZE Pig
function and flatten out the tokens we get from the split.
At the end of this step, we get (filename, word1,
word2,….). The tuple of words are broken down from the
sentence.
*/

file_and_words = foreach file_and_sentence generate
    file_name as file_name,flatten(TOKENIZE(sentence)) as
        words;


filtered_file_and_words = filter file_and_words by (words
```

```
    matches '\\w+');


/*We now group each file and the words of the sentence that
it follows. A group operation in Pig gives us (group,
{members of the group}). A flatten on the group will
produce more tuples. Please refer to this document for the
details. We are doing the following step for us to get neat
tuples of the form (filename, word, count of
word). http://pig.apache.org/docs/r0.9.1/basic.html#flatten
*/

lowercased_file_and_word = foreach filtered_file_and_words
    generate file_name as file_name, LOWER(words) as word;
```

5. Once we have cleansed and transformed the data appropriately, the `GROUP BY` operator in Pig is used to group on filename and the term. This yields a grouping of all terms in a particular file. The key to this grouping is the filename and is represented by the relation `file_and_words_groups` in the following Pig code snippet.
6. We now take the term counts per document by counting the number of terms in the grouping. The `file_and_word_and_count` relation represents this. The relation has tuples that have the filename, the word, and the number of times the word appears in the file. These are the term frequencies.
7. By grouping `file_and_word_and_count` differently, that is, by the filename, we can get the number of terms in a particular document. Summing up the counts for the terms in each and every group does this calculation. The `unnormalized_term_counts` relation gives the number of terms or words in each file.
8. These term counts in each document can then be used to normalize and calculate the term frequencies. In the example, the `term_frequencies` relation represents the term frequencies. It is calculated by dividing the term frequency by the number of terms in the document:

```
file_and_words_groups = group lowercased_file_and_word by
    (file_name, word);

file_and_word_and_count = foreach file_and_words_groups
    generate flatten(group) as (file_name:chararray,
        word:chararray),
            COUNT_STAR(lowercased_file_and_word) as count;



/* Now that we have the data massaged in the form of
```

```
(filename, word, count). I will let you proceed with the
rest of the exercise*/


/* We can now group appropriately and get different
statistics. For example, we are getting the number of terms
per document using the Pig commands below. JOINS maybe
important things to take note of when trying to normalise
the tf or the idf scores*/

group_file_and_word_and_count = group
    file_and_word_and_count by file_name;


/* Add doc sizes to the term count tuple */
unnormalized_term_counts = foreach
    group_file_and_word_and_count generate group as
        file_name, flatten(file_and_word_and_count.(word,
            count)) as (word, count),
                SUM(file_and_word_and_count.count) as
                    doc_size;


/* Generate the tf scores */
term_frequencies = foreach unnormalized_term_counts
    generate file_name as file_name, word as term,
        ((double)count / (double)doc_size) as term_freq;
```

9. By grouping `term_frequencies` by each term and counting the number of elements in the group, we get the document frequency for that particular term. In the next code snippet, the relation `doc_term_count` represents the number of documents containing the term.
10. The document frequencies now have to be normalized by the number of documents in the corpus. Grouping the `file_and_sentence` relation by the filename does this. The count of the groups represents the number of files in the corpus.
11. The final Tf-idf scores are then calculated per term and per file using the formula discussed previously. We then order the scores to validate our calculations. The term-document pairs with the highest scores are the most relevant terms for the document:

```
/* Generate the document frequencies */
group_term_frequencies = group term_frequencies by term;

doc_term_count = foreach group_term_frequencies generate
    FLATTEN(term_frequencies) as (file_name, term,
```

```
            term_freq), COUNT_STAR(term_frequencies) as
                doc_freq;


/* Generate the doc count in the corpus */
doc_groups = foreach (group file_and_sentence by file_name)
    generate group as file_name;

doc_count = foreach(group doc_groups all) generate
    COUNT(doc_groups) as n_docs;


/* Generate the final tf-idf scores */
scores = foreach doc_term_count generate file_name as
    file_name, term as term, term_freq *
        LOG((double)doc_count.n_docs/(double)doc_freq)
            as tf_idf;

ordered_scores = order scores by tf_idf ;
```

Cosine similarity distance measures

In the previous section, we saw how Tf-idf scores are calculated. A document can be represented as a vector of Tf-idf scores for each term that occurs in it. For non-occurring terms, the Tf-idf score is zero. Given this vector representation of a document, a question arises as to how we can find the distance between two documents or, in the case of a search engine, the distance between a document and a query. The least distance between two documents or a document and a query deems them most similar or relevant.

There are many distance measures. A commonly used distance measure involves finding the **Euclidean** distance or the vector difference between the two documents or the document and the query. The resultant vector is dependent on the length of the two vectors involved in the subtraction. Euclidean distance leads to longer documents being closer to each other than documents of different sizes. This might not be a very accurate way of measuring distance between two documents, particularly in the text analytics setting:

$$|D1 - D2|$$

or

$$|D1 - Q|$$

By taking into consideration the angle between the two vectors, document distance can be represented more correctly. In text analytics, document distances are calculated as the cosine of the angle between the two Tf-idf document vectors. The same calculation is made for the distance between a query and a document as the query is treated as a small document.

From elementary trigonometry, we know that two documents are similar if the cosine of the angle between their vectors is large. The cosine of zero is one, representing documents that are the same or very similar. Documents represented by orthogonal vectors have a value close to zero as the cosine of 90 degrees is equal to zero.

The cosine of two vectors can be calculated by the inner product of the two vectors as shown in the following formula:

```
Cosine similarity of document i and j = d1i * d1j + d2i * d2j……. +
dki * dkj
```

Dividing it by the length of the two vectors normalizes the cosine similarity.

Clustering using k-means

k-means is a popular clustering algorithm. It is built into the Apache Mahout library and can be run on a Hadoop cluster. It is an unsupervised learning method that groups data points by minimizing their distance from the cluster center.

The k-means algorithm is as follows when it is specified that the points have to be put into k-clusters:

1. The first step is to initialize *k* cluster centers. These centers are generally initialized randomly. In some cases, if prior knowledge of the clusters is available, then these clusters can be placed intelligently to bring down the computation time of the algorithm.
2. Each data point—in our example, the Tf-idf vector for a document—is assigned the closest cluster center. This notion of closeness is via different distance or similarity measures. We studied two distance measures in the previous section—the Euclidean distance and the cosine similarity distance.
3. Once all the data points are assigned, the next step is to readjust the cluster centers. The cluster centers are readjusted by taking the average of all the points that were assigned to the cluster center in step 2.
4. The steps 2 and 3 are repeated till convergence is reached or a preset number of iterations happen.

K-means clustering using Apache Mahout

In this section, we will see how to run k-means clustering using Hadoop and Apache Mahout. We will be running it on the grant proposal extracts discussed in the previous section.



Installing Apache Mahout involves downloading the binaries from the site `http://mahout.apache.org`. The downloaded archive file is extracted. The following environment variables are set to inform Mahout about the Hadoop installation:

```
export HADOOP_HOME=<Path to Hadoop installation folder>
export MAHOUT_HOME=<Path to Mahout installation folder>
export PATH=$PATH:$HADOOP_HOME/bin:$MAHOUT_HOME/bin
```

The Mahout binary has a number of interesting command-line options.

We will examine the different steps and options that are provided by Mahout by executing the following commands:

```
#Convert the grant proposals into a sequence file. It combines all
the files into a single file
bin/mahout seqdirectory -i /user/hadoop/grants -o
    /user/hadoop/grants-seqdir -c UTF-8 -chunk 5

#Use seqdumper to visualize the sequence files. Observe the
    combining of all the files
bin/mahout seqdumper -i /user/hadoop/grants-seqdir/part-m-00000


#Generate all statistics like tf, df and tf-idf from the corpus.
Generate ids for each term and construct the dictionary. All the
vectors are in a sparse format
bin/mahout seq2sparse -i /user/hadoop/grants-seqdir/ -o
    /user/hadoop/grants-seqdir-sparse --maxDFPercent 85 --
        namedVector

#Examine the tf-idf vectors
bin/mahout seqdumper -i /user/hadoop/grants-seqdir-sparse/tfidf-
    vectors/part-r-00000
# Examine the dictionary file termid to term mappings

bin/mahout seqdumper -i /user/hadoop/grants-seqdir-
    sparse/dictionary.file-0

# Run kmeans for 3 clusters with cosine of tf-idf as the distance
    metric
```

```
bin/mahout kmeans -i /user/hadoop/grants-seqdir-sparse/tfidf-
    vectors/ -c /user/hadoop/grants-kmeans-clusters -o
        /user/hadoop/grants-kmeans -dm
            org.apache.mahout.common.distance.CosineDistanceMeasure
                -x 10 -k 3 -ow --clustering

# Use cluster dump to get metrics about the cluster
bin/mahout clusterdump -i /user/hadoop/grants-kmeans/clusters-*-
    final -o clusterdump -d  /user/hadoop/grants-seqdir-
        sparse/dictionary.file-0 -dt sequencefile -b 100 -n 20 --
            evaluate -dm
                org.apache.mahout.common.distance.
CosineDistanceMeasure -sp 0 --
    pointsDir /user/hadoop/grants-kmeans/clusteredPoints
```

The steps to run K-means clustering using Apache Mahout are as follows:

1. The Mahout binary has an option to create `SequenceFile` from a directory. The `seqdirectory` option can be used to create it. The `seqdirectory` command has many options such as the encoding to be used, the chunk size in MB, and the class name for file parsing. In the following example, we create a `grants-seqdir` sequence file using UTF-8 encoding and a chunk size of 5 MB.
2. The `seqdumper` command in the Mahout binary is a very useful tool for visualizing the sequence file. In the following example, we observe one of the parts of the sequence file. The filename forms the key of each record and the contents of the file form its value in the sequence file. An example grants proposal looks similar to the following snippet:

```
Key: /a9996416.txt: Value: Title       : Inverse Diffraction
Problems in Optics
Type        : Award
NSF Org     : DMS
Latest
Amendment
Date        : September 13,  1999
File        : a9996416
Award Number: 9996416
Award Instr.: Standard Grant
Prgm Manager: Deborah Lockhart
             DMS  DIVISION OF MATHEMATICAL SCIENCES
             MPS  DIRECT FOR MATHEMATICAL & PHYSICAL SCIEN
Start Date  : August 16,  1999
Expires     : June 30,  2001       (Estimated)
```

3. The `seq2sparse` command is another useful utility that creates vectors from the supplied sequence files. There are two kinds of vectors that are output by this command. The first is a sequence file with the document ID and a vector of tokenized document terms. The second is a sequence file containing the document ID and a vector of Tf-idf scores. The `seq2sparse` command eliminates the need for calculating the Tf-idf scores separately. The command has many options such as `-namedVector` that creates a named vector, `-maxDFPercent` that sets a limit on the maximum percentage of documents that should considered for document frequency measurements, and `-minDF` that sets the minimum document frequency for a term. In the following example, we create vectorized representations of the grants files using named vectors and a `maxDFPercent` value of `85`. The following is a listing of the grants-seqdir-sparse directory in HDFS:

```
Found 2 items
-rw-r--r--   3 sandeepkaranth supergroup          0 2014-09-09
15:14 grants-seqdir-sparse/df-count/_SUCCESS
-rw-r--r--   3 sandeepkaranth supergroup     159253 2014-09-09
15:14 grants-seqdir-sparse/df-count/part-r-00000
Found 1 items
-rw-r--r--   3 sandeepkaranth supergroup     162107 2014-09-09
15:14 grants-seqdir-sparse/dictionary.file-0
Found 1 items
-rw-r--r--   3 sandeepkaranth supergroup     159233 2014-09-09
15:14 grants-seqdir-sparse/frequency.file-0
Found 2 items
-rw-r--r--   3 sandeepkaranth supergroup          0 2014-09-09
15:14 grants-seqdir-sparse/tf-vectors/_SUCCESS
-rw-r--r--   3 sandeepkaranth supergroup     646642 2014-09-09
15:14 grants-seqdir-sparse/tf-vectors/part-r-00000
Found 2 items
-rw-r--r--   3 sandeepkaranth supergroup          0 2014-09-09
15:14 grants-seqdir-sparse/tfidf-vectors/_SUCCESS
-rw-r--r--   3 sandeepkaranth supergroup     646642 2014-09-09
15:14 grants-seqdir-sparse/tfidf-vectors/part-r-00000
Found 2 items
-rw-r--r--   3 sandeepkaranth supergroup          0 2014-09-09
15:14 grants-seqdir-sparse/tokenized-documents/_SUCCESS
-rw-r--r--   3 sandeepkaranth supergroup     884092 2014-09-09
15:14 grants-seqdir-sparse/tokenized-documents/part-m-00000
Found 2 items
-rw-r--r--   3 sandeepkaranth supergroup          0 2014-09-09
15:14 grants-seqdir-sparse/wordcount/_SUCCESS
-rw-r--r--   3 sandeepkaranth supergroup     193944 2014-09-09
15:14 grants-seqdir-sparse/wordcount/part-r-00000
```

4. We examine the created files using the `seqdumper` command. Both the Tf-idf vectors and the dictionary file that contains the term ID for terms mapping are examined using the `seqdumper` command. The following outputs give the Tf-idf vectors for a single file and a sample dictionary file snippet:

```
Key: /a9996454.txt: Value: /a9996454.txt:{3050:4.144606113433838,2
77:2.0784096717834473,501:3.9535505771636963,190:1.200166940689087
,6974:2.745239496231079,998:3.8460910320281982,6977:1.971051096916
1987,2819:2.483874797821045,2496:1.2779039144515991,1185:4.5240955
3527832,2039:1.704549789428711,4493:2.9781711101531982,4418:3.8701
69162750244,4868:3.5626842975616455,5574:4.786459922790527,5556:6.
918078899383545,5802:4.449987411499023,779:5.297285556793213,1037:
3.028601884841919,5024:3.655057668685913,6496:2.589235305786133,12
46:5.009603500366211,7356:3.793208122253418,4662:3.541300296783447
3,6829:3.756840467453003,4325:2.8123786449432373,2121:4.4499874114
99023,6497:5.292787075042725,2640:3.2604033946990967,1045:4.316456
317901611,1542:5.075188636779785,643:5.143134593963623,2411:4.3164
56317901611,5123:7.907101154327393,5565:6.330745697021484,4773:2.5
56445360183716,7500:3.1770219802856445,6687:5.080615043640137,2683
:4.211770057678223,321:5.143134593963623,850:3.451458692550659,580
7:3.1000609397888184,7750:5.479607105255127,6370:3.157219171524048
,2868:5.9904327392578125,5561:2.276860475540161,3510:4.95756196975
708,7066:4.691149711608887,5721:3.655057668685913,2673:4.786459922
790527,2397:4.604138374328613,5208:2.784979820251465,195:1.6337237
358093262,7737:3.3048553466796875,1856:8.246277809143066,1854:5.47
9607105255127,3564:4.449987411499023,1402:3.451458692550659,7533:2
.2746293544769287,1881:5.297285556793213,5236:2.493925094604492,55
95:3.756840467453003,4947:2.589235305786133,5707:3.081711769104004
,6532:5.9904327392578125,4031:4.144606113433838,7249:2.84054970741
27197,4208:2.252763032913208,1902:4.997402191162109,5624:4.8918204
30755615,4676:1.9650808572769165,7765:2.4770045280456543,1638:12.7
9179573059082,1637:4.604138374328613,2995:2.930161714553833,4099:3
.157219171524048,2778:5.143134593963623,5874:2.930161714553833,248
3:8.76198959350586,574:5.143134593963623,3847:3.2199668884277344,6
704:4.786459922790527,3485:2.9146575927734375,3529:7.0053181648254
395,7574:5.143134593963623,7608:3.4781270027160645,2697:4.19867324
8291016,3597:2.84804368019104,5083:1.8686890602111816,2435:3.91099
09534454346,2896:3.756840467453003,7386:5.479607105255127,6678:5.9
904327392578125,4613:4.255831718444824,1526:4.691149711608887,4517
:5.26334810256958,4218:7.2734904289245605,2561:2.5044660568237305,
2425:2.611708164215088,7065:4.255831718444824,387:5.143134593963623
,6800:3.687847375869751,6244:2.635207414627075,3846:3.68784737586
9751,5904:2.352846384048462,3954:4.144606113433838,197:2.406913757
3242188,6774:4.604138374328613,3235:3.756840467453003,7205:5.14313
4593963623,1224:4.38099479675293,6898:2.473924398422241,32:1.26304
47149276733,601:5.143134593963623,3943:5.9904327392578125,2509:4.8
91820430755615,7181:3.4781270027160645,3337:4.188774108886719,3860
:2.6702041625976562,6963:4.198673248291016,7216:3.5336968898773193
,3925:4.152933120727539,1863:5.7027506828308105}
```

```
Input Path: /user/sandeepkaranth/grants-seqdir-sparse/dictionary.
file-0

……

Key: zirconia: Value: 7871

Key: znati: Value: 7872

Key: zoe: Value: 7873

Key: zone: Value: 7874

Key: zones: Value: 7875

Key: zooplankton: Value: 7876

Key: zygotes: Value: 7877

Key: zygotic: Value: 7878
```

5. Once we have the vectorized representations of our grants files, we then run k-means clustering. The Mahout binary has the `kmeans` command that is used to run this clustering algorithm. The distance metric we choose is the cosine distance measure and can be specified by the `-dm` option. It is implemented by the `org.apache.mahout.common.distance.CosineDistanceMeasure` class. In the following example, we specify the number of clusters as `3` with the `-k` switch. The `-x` switch can be used to specify the maximum number of iterations. In our case, we have set it to `10`.
6. We can view the cluster output by the k-means clustering using the `clusterdump` command supported by Mahout. The `clusterdump` command has some elaborate options such as the `-b` option that allows the user to choose the number of characters in the file to be displayed, the `-n` option that shows the topmost *n* terms based on the Tf-idf scores, and an`-evaluate` option to evaluate the input. The following snippet shows the output of the `clusterdump` command:

```
14/09/09 15:45:51 INFO evaluation.ClusterEvaluator: Scaled
Inter-Cluster Density = 0.6053257638783347

14/09/09 15:45:51 INFO evaluation.ClusterEvaluator: Intra-
Cluster Density[277] = 0.6828160148795702

14/09/09 15:45:51 INFO evaluation.ClusterEvaluator: Intra-
Cluster Density[423] = 0.6729720492208191

14/09/09 15:45:51 INFO evaluation.ClusterEvaluator: Intra-
Cluster Density[97] = 0.6610114589088609

14/09/09 15:45:51 INFO evaluation.ClusterEvaluator: Average
Intra-Cluster Density = 0.6722665076697502

14/09/09 15:45:51 INFO clustering.ClusterDumper: Wrote 3
clusters
```

RHadoop

R is a programming language used for statistics, data science, and visualization. It has a number of packages that can be imported to perform some specialized or custom tasks. It has more than 5,000 data analysis algorithms implemented as libraries. These algorithms can be used to facilitate a wide variety of data analysis tasks, much more than those supported by Apache Mahout. The community using R as a language is very big and vibrant.

However, R has two drawbacks: it executes in memory and its support for multithreading is minimal. These drawbacks make R unsuitable for big data crunching where disk-based analysis and distribution are mandatory. One alternative would be using R programs by using Hadoop Streaming. But this is a tedious proposition, and RHadoop had to be envisioned. RHadoop also uses Hadoop Streaming as its underlying mechanism to run R scripts in Hadoop, but alleviates some of the pain points that native streaming has. Some of the advantages of RHadoop are as follows:

- It eliminates the need to script R functions manually in the MapReduce paradigm. In-built library functions do this automatically for the user.
- It allows access—both reading and writing data from and to HDFS.
- It allows the same R script to run locally and on cluster environments.

RHadoop is a set of five R packages that can be used to analyze data in Hadoop. The following are the constituents of RHadoop:

- `ravro`: This is the R package that helps in serializing and deserializing data that is present in the Avro data format.
- `rmr`: This is the R package that provides Hadoop MapReduce functionality within R.
- `rhdfs`: This is the R package that provides functions to manage the data resident in HDFS from within R.
- `rhbase`: This is the R package that provides functions to manage an HBase database from within R.
- `plyrmr`: This is the R package used for structured data processing similar to `plyr`. This package uses `rmr` as the underlying framework.

Summary

Hadoop is a very useful tool for big data transformation and processing. It can come in handy at almost all the stages of the data analytics workflow. Data analytics is not about the algorithms but more about the data. Larger data can yield almost two-fold improvements in prediction. A data scientist should worry more about the cleansing, transformation, feature engineering, and validation of results rather than the actual algorithm that will be used to do the analysis. This does not mean that the analysis algorithm choice is not important. Instead, it means that there are other players that are equally important and vital for healthy decision making.

In this chapter, the key takeaways are as follows:

- Hadoop is generally used for analytics on data sizes of 1 TB and above. However, the ease of use brought about by functional programming concepts in Hadoop tempts people to use it for smaller data sizes. There is nothing wrong with this approach as long as they are cognizant of the fact about higher latency in doing so.
- Data mining is the branch that deals with discovering patterns and knowledge from data. Machine learning provides the tools for data mining.
- Machine learning algorithms are of the supervised, unsupervised, and the semi-supervised kind. Supervised learning requires labeling from domain experts and can be expensive. There are newer crowd-sourced methods that can gather labeled data. One such approach is to use Mechanical Turk by Amazon.
- Apache Mahout and RHadoop are popular data analysis libraries that have extensive support on Hadoop. Apache Mahout has stopped accepting algorithms in the MapReduce paradigm since April 2014. However, the older entries in the library still support Hadoop. It is important for the individual using these algorithms to check whether the implementation is parallelized or not as these libraries accept non-parallelized single-machine implementations as well.
- Tf-Idf is a popular metric used in text analytics. It takes into account both the popularity of a term in a document and penalizes non-differentiating terms by looking at the corpus that the documents come from. Cosine distance is used to measure similarity between documents. Euclidean distances fail because of variations in document lengths.

Hadoop for Microsoft Windows

Traditionally, Hadoop has been supported on Unix-based operating systems. Installation on Microsoft Windows was tedious and not consistent. It involved installing Unix-based emulators such as Cygwin and carrying out installation steps similar to Hadoop installations on Unix systems. Other alternatives were to run a Linux virtual machine on Windows hosts and install Hadoop on them. But Hadoop was still not natively available on the Microsoft Windows operating system until Hadoop 2.0 arrived.

With all major players moving into the cloud, the **Hadoop as a Service** (**HaaS**) offering is becoming popular. It offers an easy and cost-efficient way of analyzing big data on the cloud. Microsoft also joined the cloud bandwagon with the Azure suite of services on the cloud. The Microsoft Azure cloud not only supports Linux Virtual Machines, but also provides Hadoop as a service. Players such as Hortonworks collaborated with Microsoft to bring Hadoop to Windows.

Hadoop's native support on Microsoft Windows becomes immensely important because:

- Microsoft Windows is known for having excellent tooling for business intelligence. Microsoft Excel, PowerPivot for Excel, and PowerView are a few examples of tools that can facilitate powerful decision making by analyzing and visualizing data from enterprise data sources. The power of these native Windows tools can be unleashed on big data stored and processed by Hadoop.
- SQL Server and related technologies are native Windows database solutions that are widely deployed in many enterprises. They cater to the enterprise needs of storing and managing structured data. With Hadoop on Windows natively, unstructured data can also be added into the mix for insightful decision making. This involves almost zero migration and learning costs for the enterprise.

In this chapter, we will look at single-node deployment of Hadoop on Microsoft Windows.

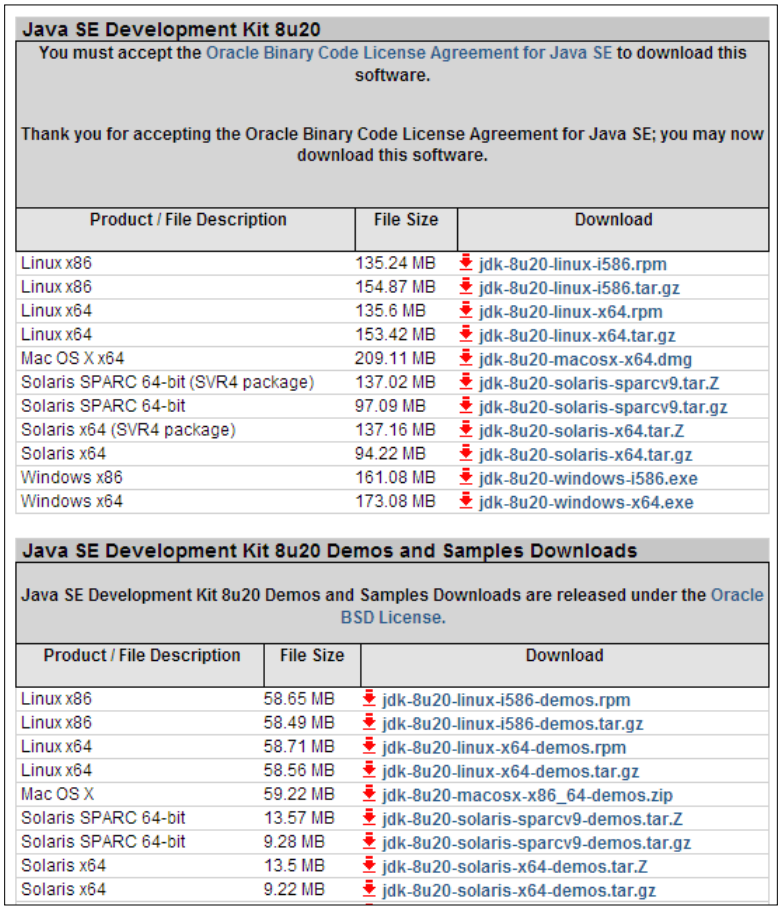
Deploying Hadoop on Microsoft Windows

In this section, we will look in detail as to how we can build and install Hadoop natively on a Windows system. We will be using Windows 8 to install Hadoop. The same steps can be employed to install Hadoop on Windows Server 2008 or Windows 7. We will be installing Hadoop on a 64-bit Windows OS running on 64-bit hardware.

Prerequisites

Installing Hadoop on Windows requires the following platforms, software, and tools installed:

- **Java JDK**: Java is the soul of Hadoop and requires it to be installed on the machine. Java from Oracle comes with the **Java Runtime Environment** (**JRE**) and the **Java Development Kit** (**JDK**). Hadoop installation requires the JDK. The JDK can be obtained from Oracle's website. To reiterate, it is very important to choose the JDK that is higher than 1.6. We will choose the latest—JDK 1.8. The following screenshot shows the page from where the JDK can be downloaded. In this example, we choose the **Windows x64** product. A 32-bit user can choose the **Windows x86** product. The download is about 170 MB in size. Once downloaded, it can be installed using the installer in the package. It is very important to choose the right processor architecture and OS for the JDK. Otherwise, there could be undesirable results.



Java SE Development Kit 8u20

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

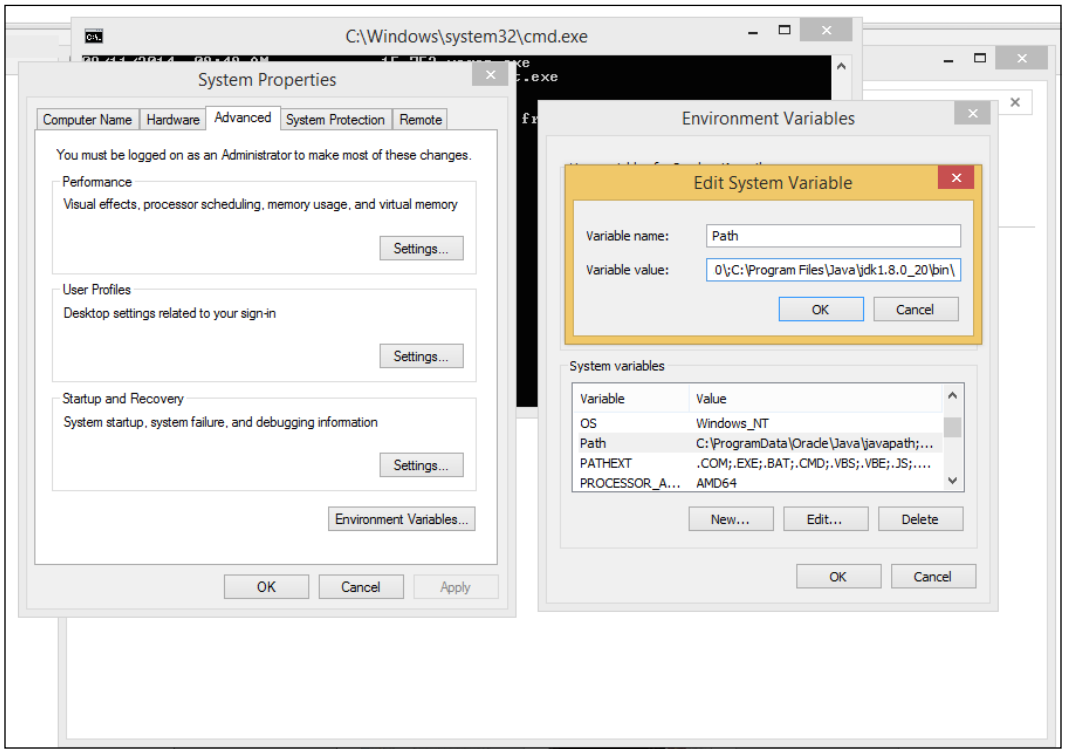
Product / File Description	File Size	Download
Linux x86	135.24 MB	jdk-8u20-linux-i586.rpm
Linux x86	154.87 MB	jdk-8u20-linux-i586.tar.gz
Linux x64	135.6 MB	jdk-8u20-linux-x64.rpm
Linux x64	153.42 MB	jdk-8u20-linux-x64.tar.gz
Mac OS X x64	209.11 MB	jdk-8u20-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	137.02 MB	jdk-8u20-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	97.09 MB	jdk-8u20-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	137.16 MB	jdk-8u20-solaris-x64.tar.Z
Solaris x64	94.22 MB	jdk-8u20-solaris-x64.tar.gz
Windows x86	161.08 MB	jdk-8u20-windows-i586.exe
Windows x64	173.08 MB	jdk-8u20-windows-x64.exe

Java SE Development Kit 8u20 Demos and Samples Downloads

Java SE Development Kit 8u20 Demos and Samples Downloads are released under the Oracle BSD License.

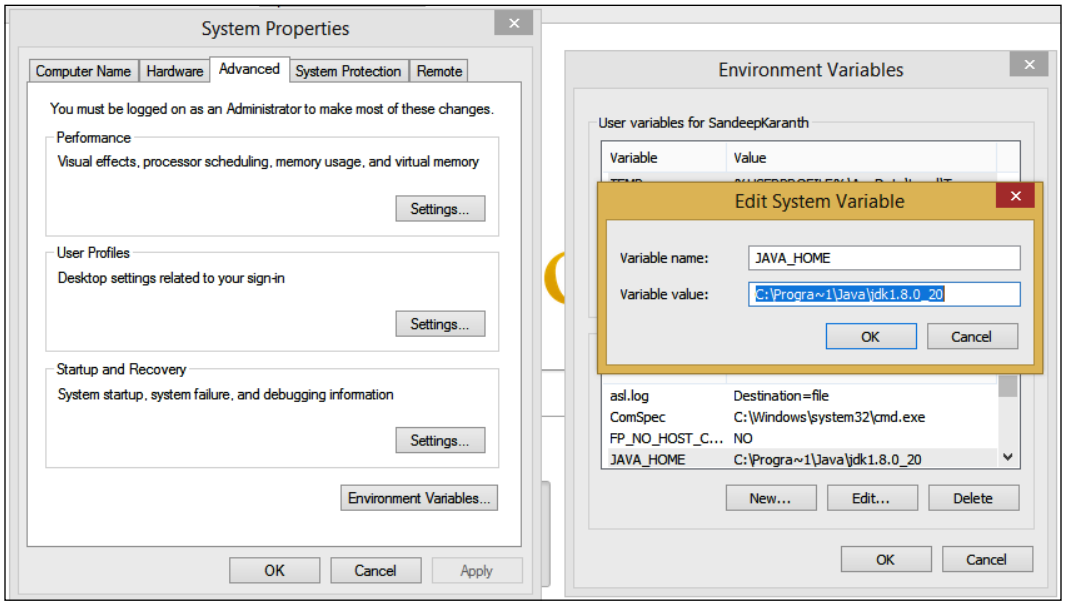
Product / File Description	File Size	Download
Linux x86	58.65 MB	jdk-8u20-linux-i586-demos.rpm
Linux x86	58.49 MB	jdk-8u20-linux-i586-demos.tar.gz
Linux x64	58.71 MB	jdk-8u20-linux-x64-demos.rpm
Linux x64	58.56 MB	jdk-8u20-linux-x64-demos.tar.gz
Mac OS X	59.22 MB	jdk-8u20-macosx-x86_64-demos.zip
Solaris SPARC 64-bit	13.57 MB	jdk-8u20-solaris-sparcv9-demos.tar.Z
Solaris SPARC 64-bit	9.28 MB	jdk-8u20-solaris-sparcv9-demos.tar.gz
Solaris x64	13.5 MB	jdk-8u20-solaris-x64-demos.tar.Z
Solaris x64	9.22 MB	jdk-8u20-solaris-x64-demos.tar.gz

- **Setting the Path variable**: Now the Windows `Path` environment variable has to be set so that the command-line tools can directly pick the Java executable from the path. In the Windows **Control Panel**, under **System Properties**, a button is provided for **Environment Variables**. Clicking on this button opens up the **Environment Variables** dialog. The appropriate environment variable can be chosen for editing if it is already present, or a new one can be created. We use the existing `Path` variable and add the `bin` folder for the Java binaries. In the preceding example, it is `C:\Program Files\Java\jdk1.8.0_20\bin\`. It is very important to separate the paths using a semicolon. The path can be tested by opening the command prompt and typing `java -version`. This should give the version number of the installed Java software. The following screenshot shows the setting of the `Path` environment variable:



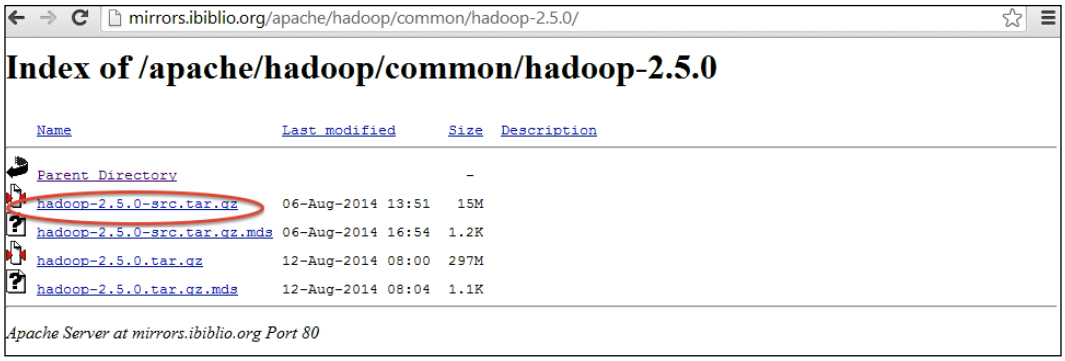
- **Setting the JAVA_HOME environment variable**: All Hadoop binaries choose the version of Java by looking up the `JAVA_HOME` directory. It is important to set this variable before starting off with Hadoop deployment and usage. Again, we use the **Environment Variables** dialog to do this. This time, instead of editing, we click on the **New** button to add a variable. In the following example, we set `JAVA_HOME` as `C:\Progra~1\Java\jdk1.8.0_20`. It is important to note that the `Program Files` folder has been shortened to a 8-letter path called `Progra~1`. This is because Hadoop does not handle spaces in the paths. Windows OS understands this 8-letter scheme, as it is a legacy feature.

  The following screenshot shows the actual setting of the `JAVA_HOME` environment variable:

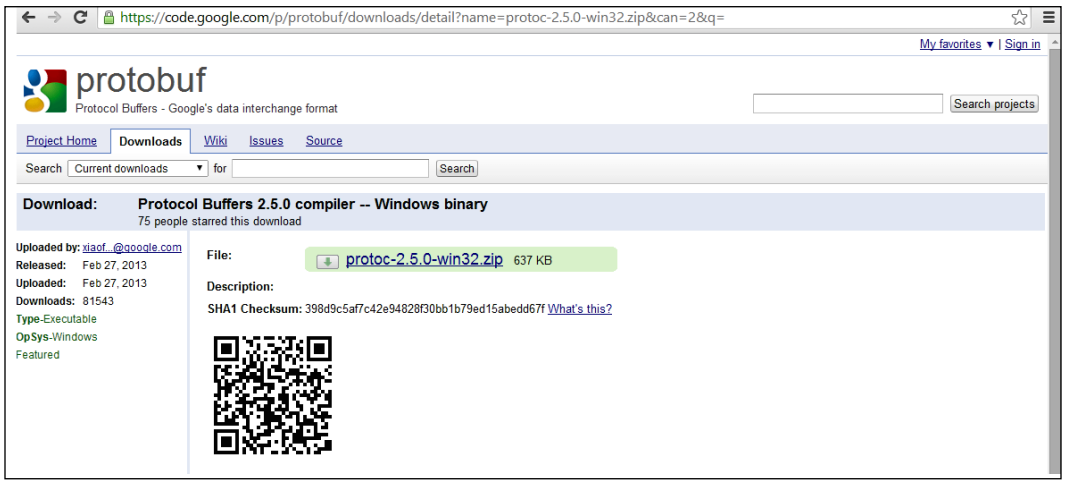


- **Downloading Hadoop sources**: We download the Hadoop sources from the nearest mirror site. It is important to download the sources, compile them, and then deploy Hadoop on Windows for native support. Using the binaries as is throws an error when deploying Hadoop on Windows. At a later point of time, there could be support to install Hadoop from binaries. We choose the latest version of Hadoop to install, Hadoop 2.5.0. As in the following screenshot, we download only the source tar file, `hadoop-2.5.0-src.tar.gz`. The sources can then be extracted into a local folder. The download is about 15 MB in size.

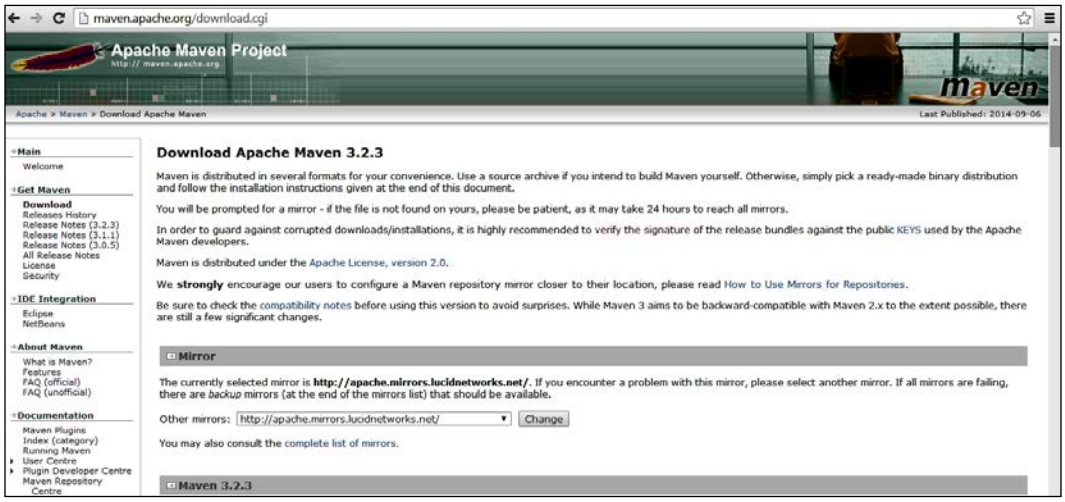
In the example, we download and extract the sources to the `C:\hdp\hdp` directory. It will save you a lot of pain if the directory names are short. Windows has some restrictions on the maximum number of characters in the directory name.



- **Protobuf compiler**: Protobuf is a serialization format and the Hadoop build requires this compiler to be available during the build process. The Windows version of the compiler binary needs to be downloaded. In this example, we choose `protoc-2.5.0-win32.zip` and download it as shown in the following screenshot. Once we download and extract it, we use the **Environment Variables** dialog to add the `bin` directory of the protobuf compiler to the `Path`.



- **Maven Build System**: Hadoop is built using the Maven build system. The build system uses specifications specified in the `pom.xml` file, which is found in the `root` directory of the Hadoop sources. To install Maven on the Windows machine, we can go to the Maven project page and download the latest Apache Maven binaries. We choose the version 3.2.3. Once downloaded, the ZIP file is extracted and the `bin` folder is again appended in the `Path` environment variable for ease of use.



Download page for Apache Maven 3.2.3

- The next important thing is to download the Windows SDK. If you are using an x86 machine, it is important to get the x86 build tools. Otherwise, the user needs to get the x64 build tools. Installing a high SKU of Visual Studio may install all the necessary tools. In this example, we will install Visual Studio Express for C++ 2010, which is a free Visual Studio download. This Visual Studio SKU does not come with the Windows SDK. We must separately install the Windows SDK. The version of the Windows SDK installed is 7.1. To verify that the SDK is installed, you can navigate to `C:\Program Files\Microsoft SDKs\Windows` on your computer. For x86 machines, it will be present in `C:\Program Files(x86)\Microsoft SDKs\Windows`. Also, it is important to include the SDKs `bin` folder in the `Path` environment so that the build process can automatically pick it up. An alternative to using the Windows build tools is installing CMake, but this requires the user to change a few configurations within the `pom.xml` files in the Hadoop sources.

Building Hadoop

Once all the pre-requisites are in place, Hadoop can be built and packaged. To do the build, you can open the Microsoft Visual Studio command prompt. It sets some of the necessary environment settings. Additionally:

1. It is important to set the `Platform` environment variable to `x64` or `Win32` depending on the Hadoop deployment desired. This can be done using the following command:

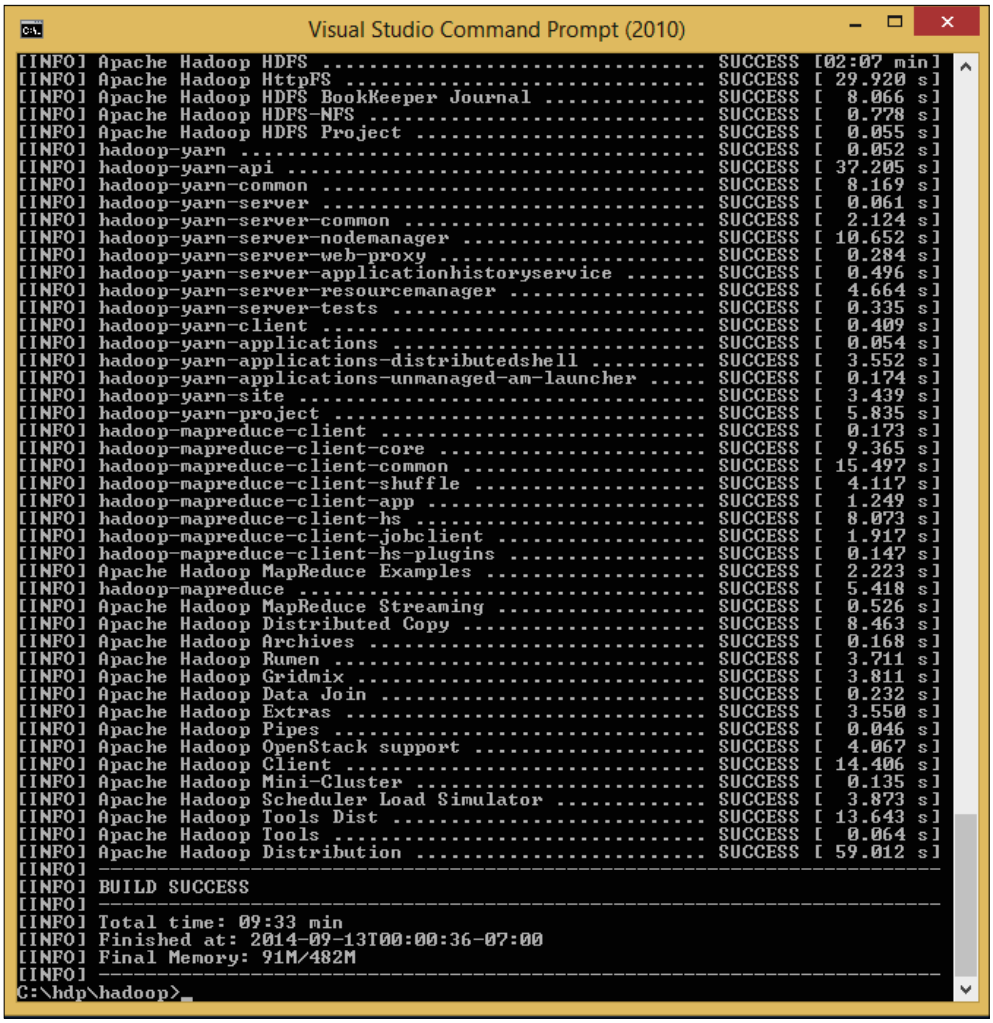
   ```
   set Platform=x64
   ```

   For Win32, use the following command:

   ```
   set Platform=Win32
   ```

2. It is very important to ensure that the environment variable has the right name. This variable is case sensitive and instructs the Visual Studio project files to use the appropriate build configuration.
3. The next step is to actually issue the Maven build command. The command `mvn package -Pdist,native-win -DskipTests –Dtar` is used to start the build. Using the newer JDK can cause some parse issues when generating Javadocs. This can be solved by either using an older JDK such as 1.7 or skipping Javadocs generation. The latter is done by adding the `-Dmaven.javadocs.skip=true` option in the Maven package command.

   The following screenshot shows the end of the build process. The summary of the standard output shows the status of each build step. Once a failure is encountered, the rest of the steps are skipped. It is also important to have the computer connected to the Internet during the build process. Maven automatically downloads dependencies from configured binary repositories during the build process:



4. The build yields a target directory. Inside the target directory, Hadoop binaries, samples, and configuration files are bundled in a zipped TAR file. In this example, a `hadoop-2.5.0.tar.gz` file is generated. We extract the contents of the file to the `C:\hdp\hdp` path.

Configuring Hadoop

In this section, we will see the different configuration settings for single-node deployment of Hadoop on Windows:

1. The `Hadoop-env.cmd` file is present in the `etc\hadoop` directory at the root of the Hadoop installation. This is the configuration directory for Hadoop. The `hadoop-env.cmd` command file needs to be modified to set the right environment to execute the Hadoop daemons correctly. The most important configuration is the setting of the `JAVA_HOME` environment variable. We also set `HADOOP_HOME` to the root of the Hadoop installation, that is, the path from where we extracted the Hadoop binaries and configuration files. The `HADOOP_CONF_DIR` and `YARN_CONF_DIR` environment variables are set to the configuration directories of Hadoop and YARN respectively. The YARN configuration directory is the same as the Hadoop configuration directory in our example. We also add the Hadoop directories to the `Path` variable. The following script snippet is a sample `hadoop-env.cmd` script file:

```
@rem The java implementation to use.  Required.
set JAVA_HOME=%JAVA_HOME%

set HADOOP_HOME=c:\hdp\hdp

@rem The jsvc implementation to use. Jsvc is required to run
secure datanodes.
@rem set JSVC_HOME=%JSVC_HOME%

set HADOOP_CONF_DIR=%HADOOP_HOME%\etc\hadoop

set YARN_CONF_DIR=%HADOOP_CONF_DIR%
set PATH=%PATH%;%HADOOP_HOME%\bin

@rem Extra Java CLASSPATH elements.  Automatically insert
capacity-scheduler.
if exist %HADOOP_HOME%\contrib\capacity-scheduler (
  if not defined HADOOP_CLASSPATH (
    set HADOOP_CLASSPATH=%HADOOP_HOME%\contrib\capacity-
      scheduler\*.jar
  ) else (
    set HADOOP_CLASSPATH=%HADOOP_CLASSPATH%;%HADOOP_HOME%
      \contrib\capacity-scheduler\*.jar
  )
```

```
)

@rem The maximum amount of heap to use, in MB. Default is
1000.
@rem set HADOOP_HEAPSIZE=
@rem set HADOOP_NAMENODE_INIT_HEAPSIZE=""

@rem Extra Java runtime options.  Empty by default.
@rem set HADOOP_OPTS=%HADOOP_OPTS% -
Djava.net.preferIPv4Stack=true

@rem Command specific options appended to HADOOP_OPTS when
specified
if not defined HADOOP_SECURITY_LOGGER (
  set HADOOP_SECURITY_LOGGER=INFO,RFAS
)
if not defined HDFS_AUDIT_LOGGER (
  set HDFS_AUDIT_LOGGER=INFO,NullAppender
)

set HADOOP_NAMENODE_OPTS=-
Dhadoop.security.logger=%HADOOP_SECURITY_LOGGER% -
Dhdfs.audit.logger=%HDFS_AUDIT_LOGGER% %HADOOP_NAMENODE_OPTS%
set HADOOP_DATANODE_OPTS=-Dhadoop.security.logger=ERROR,RFAS
  %HADOOP_DATANODE_OPTS%
set HADOOP_SECONDARYNAMENODE_OPTS=-
  Dhadoop.security.logger=%HADOOP_SECURITY_LOGGER% -
    Dhdfs.audit.logger=%HDFS_AUDIT_LOGGER%
      %HADOOP_SECONDARYNAMENODE_OPTS%

@rem The following applies to multiple commands (fs, dfs,
  fsck, distcp etc)
set HADOOP_CLIENT_OPTS=-Xmx512m %HADOOP_CLIENT_OPTS%
@rem set HADOOP_JAVA_PLATFORM_OPTS="-XX:-UsePerfData
  %HADOOP_JAVA_PLATFORM_OPTS%"

@rem On secure datanodes, user to run the datanode as after
  dropping privileges
set HADOOP_SECURE_DN_USER=%HADOOP_SECURE_DN_USER%

@rem Where log files are stored.  %HADOOP_HOME%/logs by
  default.
```

```
@rem set HADOOP_LOG_DIR=%HADOOP_LOG_DIR%\%USERNAME%


@rem Where log files are stored in the secure data environment.
set
HADOOP_SECURE_DN_LOG_DIR=%HADOOP_LOG_DIR%\%HADOOP_HDFS_USER%


@rem The directory where pid files are stored. /tmp by
  default.
@rem NOTE: this should be set to a directory that can only be
  written to by
@rem       the user that will run the hadoop daemons.
   Otherwise there is the
@rem       potential for a symlink attack.
set HADOOP_PID_DIR=%HADOOP_PID_DIR%
set HADOOP_SECURE_DN_PID_DIR=%HADOOP_PID_DIR%


@rem A string representing this instance of hadoop. %USERNAME%
  by default.
set HADOOP_IDENT_STRING=%USERNAME%
```

2. Next we configure the `core-site.xml` file. The most important configuration is setting the `fs.default.name` property to the HDFS NameNode host and port. In our case, since it is a single node deployment, it points to `localhost` on port `19000`. The following configuration snippet illustrates this setting:

```
<configuration>
    <property>
        <name>fs.default.name</name>
       <value>hdfs://0.0.0.0:19000</value>
    </property>
 </configuration>
```

3. We then configure the `hdfs-site.xml` file. Here we set the replication factor to 1 as we are doing a single-node deployment of Hadoop. The following configuration snippet illustrates this setting:

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

4. The `mapred-site.xml` file needs to be configured and pointed to YARN in Hadoop 2.X. The `%USERNAME%` element can be replaced by the username of the entity submitting the jobs. The following configuration snippet illustrates a sample `mapred-site.xml` file. If the file is not present, it can be copied from the `mapred-site.xml.template` file present in the configuration directory:

```
<configuration>
    <property>
        <name>mapreduce.job.user.name</name>
        <value>%USERNAME%</value>
    </property>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
        <name>yarn.apps.stagingDir</name>
        <value>/user/%USERNAME%/staging</value>
    </property>
    <property>
        <name>mapreduce.jobtracker.address</name>
        <value>local</value>
    </property>
</configuration>
```

5. The `yarn-site.xml` file is configured for the settings on the ResourceManager and NodeManager daemons. The configurations include setting the daemon endpoints and the log directories, and specifying the shuffle handlers. The following configuration snippet illustrates a sample configuration for the YARN daemons:

```
<configuration>
    <property>
        <name>yarn.server.resourcemanager.address</name>
        <value>0.0.0.0:8020</value>
    </property>
    <property>
        <name>yarn.server.resourcemanager.application
            .expiry.interval</name>
        <value>60000</value>
    </property>
     <property>
        <name>yarn.server.nodemanager.address</name>
        <value>0.0.0.0:45454</value>
    </property>
     <property>
        <name>yarn.nodemanager.aux-services</name>
```

```
            <value>mapreduce_shuffle</value>
        </property>
        <property>
            <name>yarn.nodemanager.aux-
                services.mapreduce.shuffle.class</name>
            <value>org.apache.hadoop.mapred
                .ShuffleHandler</value>
        </property>
        <property>
            <name>yarn.server.nodemanager.remote-app-log-
                dir</name>
         <value>/app-logs</value>
        </property>
        <property>
            <name>yarn.nodemanager.log-dirs</name>
            <value>/dep/logs/userlogs</value>
        </property>
        <property>
            <name>yarn.server.mapreduce-appmanager.attempt-
                listener.bindAddress</name>
            <value>0.0.0.0</value>
        </property>
        <property>
            <name>yarn.server.mapreduce-appmanager.client-
                service.bindAddress</name>
            <value>0.0.0.0</value>
        </property>
        <property>
            <name>yarn.log-aggregation-enable</name>
            <value>true</value>
        </property>
        <property>
            <name>yarn.log-aggregation.retain-seconds</name>
            <value>-1</value>   </property>
        <property>
            <name>yarn.application.classpath</name>
            <value>%HADOOP_CONF_DIR%,%HADOOP_COMMON_HOME%/share/
    hadoop/
    common/*,%HADOOP_COMMON_HOME%/share/hadoop/common/lib/*,%HA
    DOOP_HDFS_HOME%/share/hadoop/hdfs/*,%HADOOP_HDFS_HOME%/shar
    e/hadoop/hdfs/lib/*,%HADOOP_MAPRED_HOME%/share/hadoop/mapre
    duce/*,%HADOOP_MAPRED_HOME%/share/hadoop/mapreduce/lib/*,%H
    ADOOP_YARN_HOME%/share/hadoop/yarn/*,%HADOOP_YARN_HOME%/sha
    re/hadoop/yarn/lib/*</value>
        </property>
    </configuration>
```

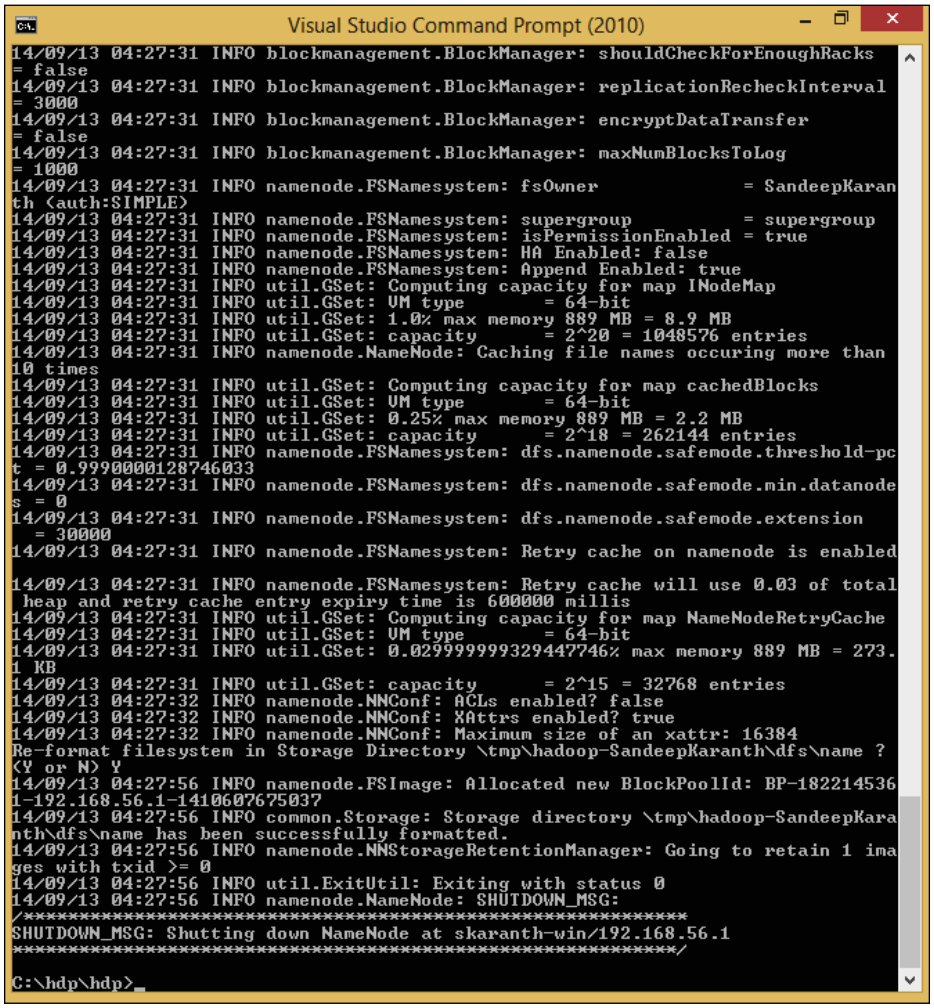
Deploying Hadoop

Once the configurations are complete, it is time to start the Hadoop daemons. This is done by performing the following steps:

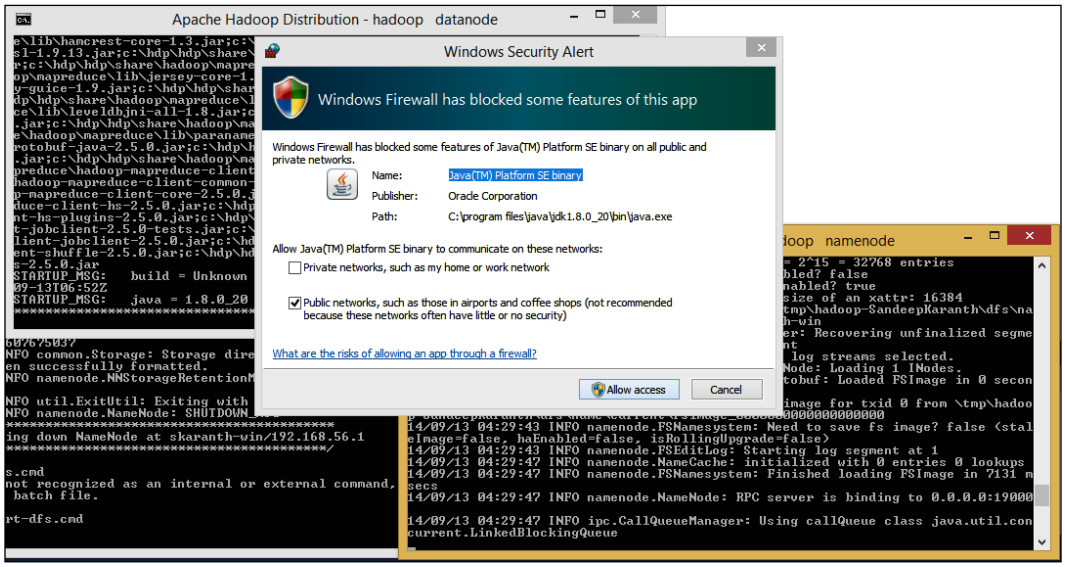
1. Before starting the daemons, we can format the NameNode by issuing the following command:

   ```
   hdfs namenode -format
   ```

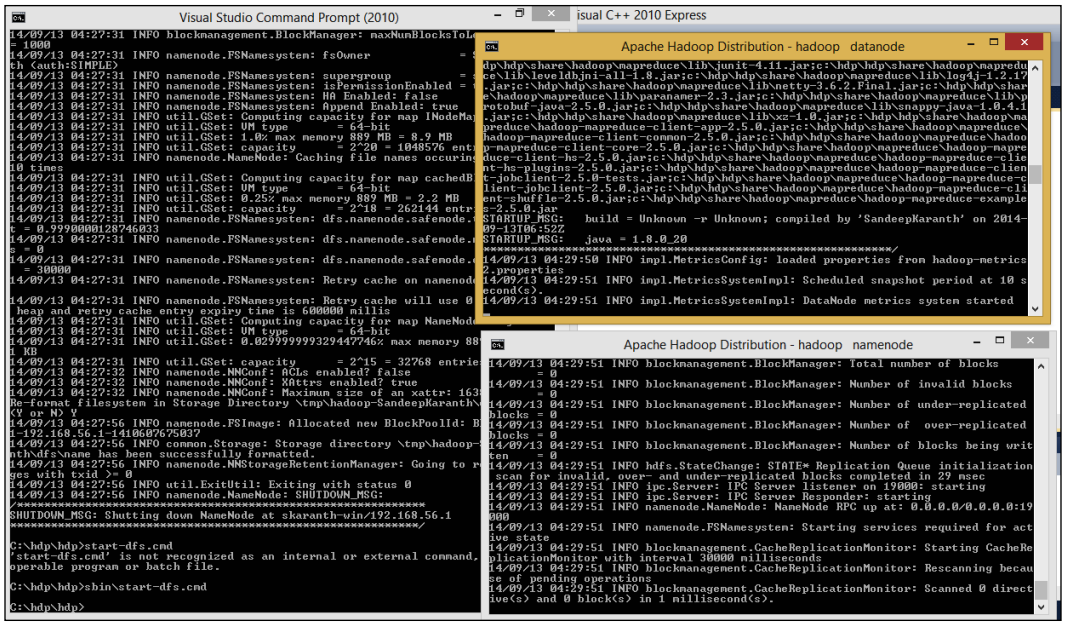
   The following screenshot shows the output of the format command. Now the HDFS is formatted and ready to use. Since we have not specified a particular directory name, the NameNode uses the `C:\tmp` directory to store all of the metadata.



2. We then start the HDFS daemons, the NameNode, and the DataNode. This is done by issuing `start-dfs.cmd`. This command script is present in the `%HADOOP_HOME%\sbin` folder. The Windows firewall may pop up a notification asking the user to allow the daemons to open a listening port in the firewall. It is important to allow the firewall to be reconfigured so that the DataNode and NameNode can communicate with each other. The following screenshot shows the Windows Firewall screen offering to allow access:

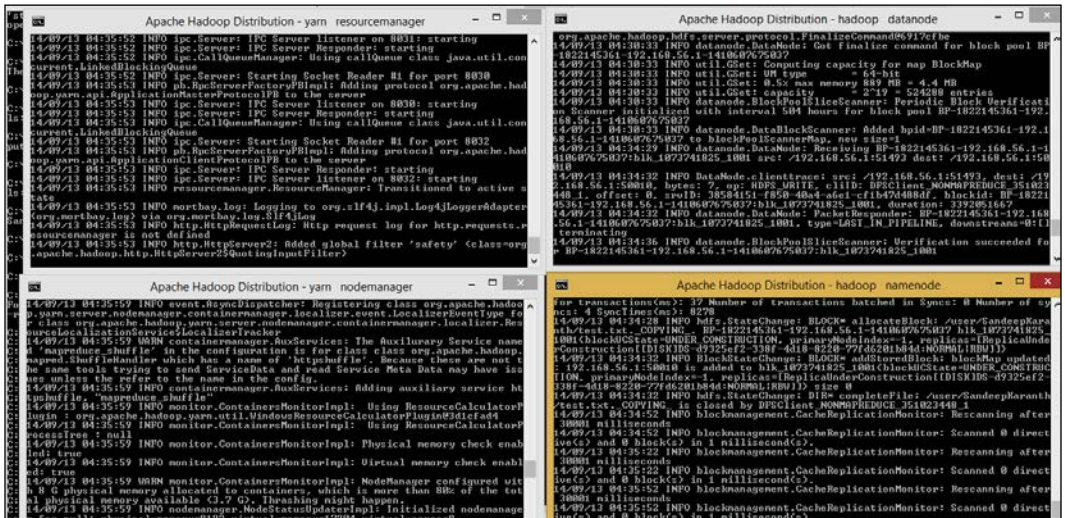


Once access has been granted, the NameNode and DataNode start in two separate command windows, as shown in the following screenshot. The standard output of each HDFS operation can be examined in these two windows. Issuing HDFS commands on the filesystem once both the daemons are up and running can conduct tests to validate HDFS. You might have to create the user directories using the `mkdir` command before starting off though.



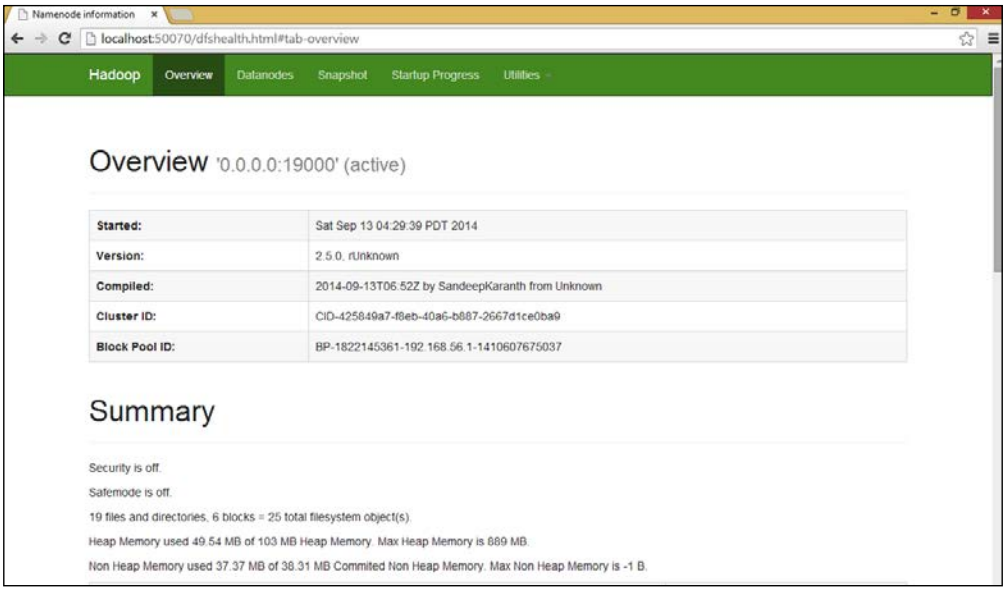
The NameNode and DataNode command windows

3. Next we need to start the YARN to run MapReduce jobs. This can be done by the `start-yarn.cmd` file present in the `sbin` folder. Again, the ResourceManager and the NodeManager start off in two separate command windows as illustrated in the following screenshot. The standard output can be examined to see the trace on the ResourceManager and NodeManager.

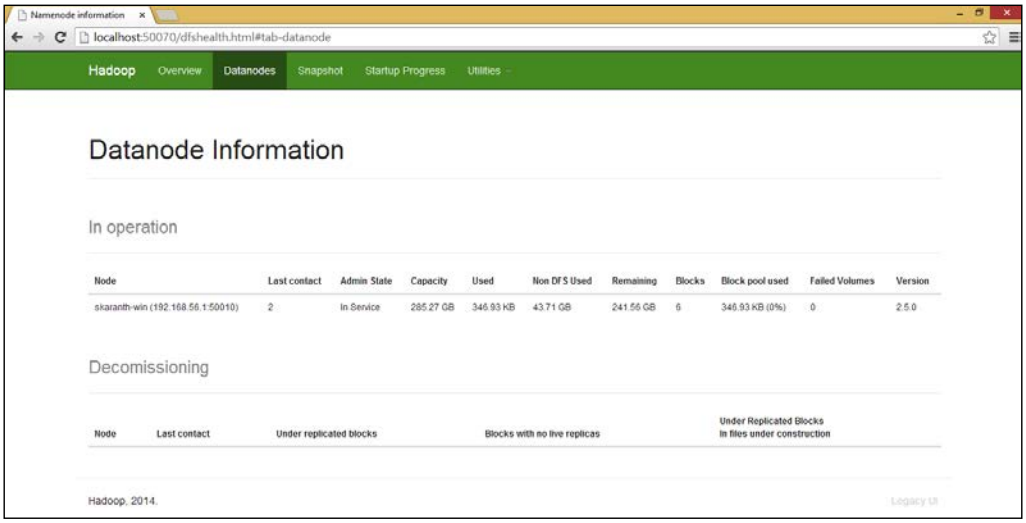


The ResourceManager and NodeManager command windows

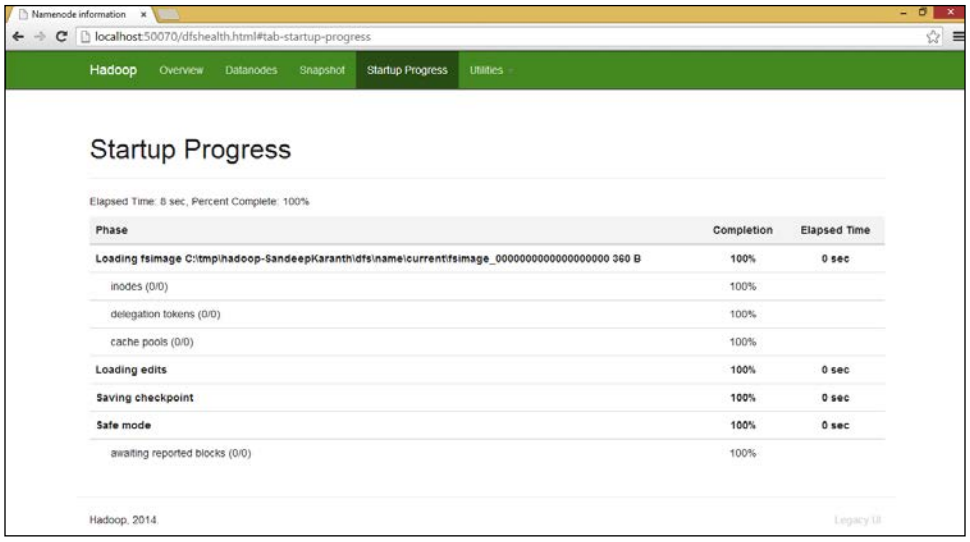
4. By navigating to `localhost:50070` on the browser, the user should now be able to see the web endpoint for HDFS. The home page is shown in the following screenshot. It gives an overview of the health of HDFS and the different parameters that were used to configure it.



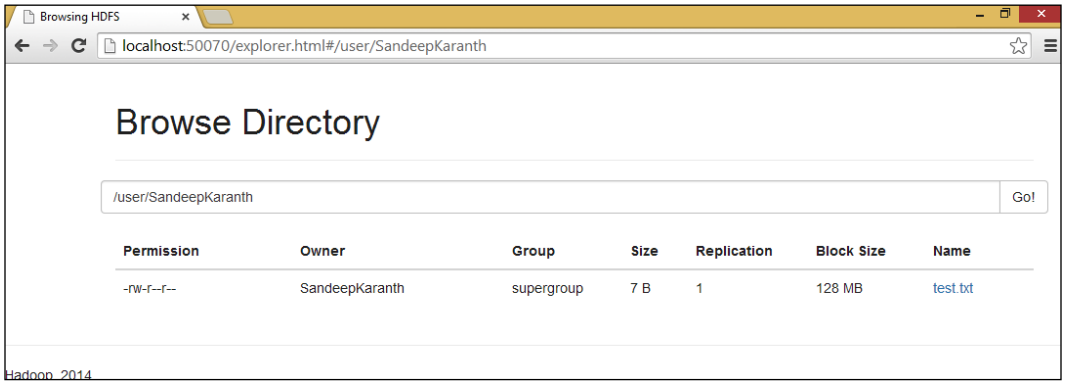
5. Selecting the **Datanodes** link on the top bar gives the different DataNodes present in HDFS and the health of each DataNode, as shown in the following screenshot:



6. The startup progress link in the top bar shows the health of HDFS during startup. This includes statistics about the `fsimage` and `edits` file during NameNode startup. It also indicates whether HDFS went into safe mode or not.



7. The utilities link gives two options: one to browse HDFS and the other to view the logfiles. The browse functionality is based on a search box that can be used to search the HDFS directory structure. Each listing for a file is similar to executing the `hdfs dfs -ls` command on the directory. It also gives statistics about the block size and a deep link to peek into the contents of the file.



Summary

With cloud computing becoming a focus for its elasticity and cost-effectiveness, Microsoft has moved into the arena to compete with existing players. To maintain parity with competition, Microsoft Azure not only offers Linux-based Virtual Machines, but has also embraced open source big data systems such as Hadoop. HDInsight offers HaaS on Microsoft Azure.

The key takeaways from this chapter are as follows:

- Hadoop is now natively available on Windows. Installing Unix emulators or Linux VMs on Windows OS is no longer necessary.
- Hadoop support on Windows natively has two missing features: Security features and short-circuit HDFS reads are not yet integrated with this system.
- Hadoop on Windows requires building the Hadoop distribution from scratch. Direct download of Hadoop binaries for Windows is not yet available.
- HDInsight, Hadoop as a service offering on Microsoft Azure, provides seamless Excel integration and integration with platforms such as the Hortonworks Data Platform.

Bibliography

This course is a blend of text and quizzes all packaged up keeping your journey in mind. It include the content from the following Packt products:

- *Hadoop Beginners Guide, Garry Turkington*
- *Hadoop Real-World Solutoins Cookbook, Second Edition, Tanmay Deshpande*
- *Mastering Hadoop, Sandeep Karnath*



Thank you for buying

Hadoop: Data Processing and Modelling

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 that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `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, then please 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.

Please check `www.PacktPub.com` for information on our titles