

Apache Hadoop 3 Quick Start Guide

Learn about big data processing and analytics

Packt>

www.packt.com

Hrishikesh Vijay Karambelkar

Apache Hadoop 3 Quick Start Guide

Learn about big data processing and analytics

Hrishikesh Vijay Karambelkar



BIRMINGHAM - MUMBAI

Apache Hadoop 3 Quick Start Guide

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book 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 book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

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

Commissioning Editor: Amey Varangaonkar

Acquisition Editor: Reshma Raman

Content Development Editor: Kirk Dsouza

Technical Editor: Jinesh Topiwala

Copy Editor: Safis Editing

Project Coordinator: Hardik Bhinde

Proofreader: Safis Editing

Indexer: Rekha Nair

Graphics: Alishon Mendonsa

Production Coordinator: Deepika Naik

First published: October 2018

Production reference: 1311018

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

ISBN 978-1-78899-983-0

www.packtpub.com

To my lovely wife, Dhanashree, for her unconditional support and endless love.

– Hrishikesh Vijay Karambelkar



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

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

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

Contributors

About the author

Hrishikesh Vijay Karambelkar is an innovator and an enterprise architect with 16 years of software design and development experience, specifically in the areas of big data, enterprise search, data analytics, text mining, and databases. He is passionate about architecting new software implementations for the next generation of software solutions for various industries, including oil and gas, chemicals, manufacturing, utilities, healthcare, and government infrastructure. In the past, he has authored three books for Packt Publishing: two editions of *Scaling Big Data with Hadoop and Solr* and one of *Scaling Apache Solr*. He has also worked with graph databases, and some of his work has been published at international conferences such as VLDB and ICDE.

Writing a book is harder than I thought and more rewarding than I could have ever imagined. None of this would have been possible without support from my wife, Dhanashree. I'm eternally grateful to my parents, who have always encouraged me to work sincerely and respect others. Special thanks to my editor, Kirk, who ensured that the book was completed within the stipulated time and to the highest quality standards. I would also like to thank all the reviewers.

About the reviewer

Dayong Du has led a career dedicated to enterprise data and analytics for more than 10 years, especially on enterprise use cases with open source big data technology, such as Hadoop, Hive, HBase, and Spark. Dayong is a big data practitioner, as well as an author and coach. He has published the first and second editions of *Apache Hive Essential* and has coached lots of people who are interested in learning about and using big data technology. In addition, he is a seasonal blogger, contributor, and adviser for big data start-ups, and a co-founder of the Toronto Big Data Professionals Association.

I would like to sincerely thank my wife and daughter for their sacrifices and encouragement during my time spent on the big data community and technology.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Hadoop 3.0 - Background and Introduction	7
How it all started	9
What Hadoop is and why it is important	11
How Apache Hadoop works	15
Resource Manager	16
Node Manager	17
YARN Timeline Service version 2	18
NameNode	18
DataNode	19
Hadoop 3.0 releases and new features	20
Choosing the right Hadoop distribution	22
Cloudera Hadoop distribution	23
Hortonworks Hadoop distribution	24
MapR Hadoop distribution	25
Summary	26
Chapter 2: Planning and Setting Up Hadoop Clusters	27
Technical requirements	28
Prerequisites for Hadoop setup	28
Preparing hardware for Hadoop	28
Readying your system	29
Installing the prerequisites	30
Working across nodes without passwords (SSH in keyless)	32
Downloading Hadoop	33
Running Hadoop in standalone mode	36
Setting up a pseudo Hadoop cluster	39
Planning and sizing clusters	44
Initial load of data	44
Organizational data growth	45
Workload and computational requirements	46
High availability and fault tolerance	46
Velocity of data and other factors	47
Setting up Hadoop in cluster mode	48
Installing and configuring HDFS in cluster mode	48
Setting up YARN in cluster mode	52
Diagnosing the Hadoop cluster	55
Working with log files	55
Cluster debugging and tuning tools	56

JPS (Java Virtual Machine Process Status)	56
JStack	57
Summary	57
Chapter 3: Deep Dive into the Hadoop Distributed File System	58
Technical requirements	59
How HDFS works	59
Key features of HDFS	61
Achieving multi tenancy in HDFS	61
Snapshots of HDFS	62
Safe mode	63
Hot swapping	64
Federation	64
Intra-DataNode balancer	65
Data flow patterns of HDFS	65
HDFS as primary storage with cache	66
HDFS as archival storage	67
HDFS as historical storage	69
HDFS as a backbone	70
HDFS configuration files	71
Hadoop filesystem CLIs	73
Working with HDFS user commands	73
Working with Hadoop shell commands	75
Working with data structures in HDFS	78
Understanding SequenceFile	78
MapFile and its variants	79
Summary	80
Chapter 4: Developing MapReduce Applications	81
Technical requirements	82
How MapReduce works	82
What is MapReduce?	83
An example of MapReduce	84
Configuring a MapReduce environment	85
Working with mapred-site.xml	86
Working with Job history server	87
RESTful APIs for Job history server	87
Understanding Hadoop APIs and packages	89
Setting up a MapReduce project	91
Setting up an Eclipse project	91
Deep diving into MapReduce APIs	96
Configuring MapReduce jobs	96
Understanding input formats	99
Understanding output formats	101
Working with Mapper APIs	103

Working with the Reducer API	105
Compiling and running MapReduce jobs	107
Triggering the job remotely	107
Using Tool and ToolRunner	108
Unit testing of MapReduce jobs	110
Failure handling in MapReduce	111
Streaming in MapReduce programming	113
Summary	114
Chapter 5: Building Rich YARN Applications	115
Technical requirements	116
Understanding YARN architecture	117
Key features of YARN	118
Resource models in YARN	118
YARN federation	119
RESTful APIs	120
Configuring the YARN environment in a cluster	121
Working with YARN distributed CLI	122
Deep dive with YARN application framework	124
Setting up YARN projects	125
Writing your YARN application with YarnClient	126
Writing a custom application master	127
Building and monitoring a YARN application on a cluster	128
Building a YARN application	128
Monitoring your application	129
Summary	132
Chapter 6: Monitoring and Administration of a Hadoop Cluster	133
Roles and responsibilities of Hadoop administrators	134
Planning your distributed cluster	135
Hadoop applications, ports, and URLs	137
Resource management in Hadoop	139
Fair Scheduler	140
Capacity Scheduler	141
High availability of Hadoop	142
High availability for NameNode	142
High availability for Resource Manager	144
Securing Hadoop clusters	146
Securing your Hadoop application	146
Securing your data in HDFS	147
Performing routine tasks	148
Working with safe mode	148
Archiving in Hadoop	149
Commissioning and decommissioning of nodes	150
Working with Hadoop Metric	151

Summary	153
Chapter 7: Demystifying Hadoop Ecosystem Components	154
Technical requirements	155
Understanding Hadoop's Ecosystem	155
Working with Apache Kafka	160
Writing Apache Pig scripts	164
Pig Latin	165
User-defined functions (UDFs)	165
Transferring data with Sqoop	167
Writing Flume jobs	169
Understanding Hive	171
Interacting with Hive – CLI, beeline, and web interface	172
Hive as a transactional system	174
Using HBase for NoSQL storage	175
Summary	177
Chapter 8: Advanced Topics in Apache Hadoop	178
Technical requirements	179
Hadoop use cases in industries	179
Healthcare	180
Oil and Gas	180
Finance	181
Government Institutions	181
Telecommunications	181
Retail	182
Insurance	182
Advanced Hadoop data storage file formats	183
Parquet	184
Apache ORC	186
Avro	187
Real-time streaming with Apache Storm	187
Data analytics with Apache Spark	192
Summary	195
Other Books You May Enjoy	197
Index	200

Preface

This book is a quick-start guide for learning Apache Hadoop version 3. It is targeted at readers with no prior knowledge of Apache Hadoop, and covers key big data concepts, such as data manipulation using MapReduce, flexible model utilization with YARN, and storing different datasets with **Hadoop Distributed File System (HDFS)**. This book will teach you about different configurations of Hadoop version 3 clusters, from a lightweight developer edition to an enterprise-ready deployment. Throughout your journey, this guide will demonstrate how parallel programming paradigms such as MapReduce can be used to solve many complex data processing problems, using case studies and code to do so. Along with development, the book will also cover the important aspects of the big data software development life cycle, such as quality assurance and control, performance, administration, and monitoring. This book serves as a starting point for those who wish to master the Apache Hadoop ecosystem.

Who this book is for

Hadoop 3 Quick Start Guide is intended for those who wish to learn about Apache Hadoop version 3 in the quickest manner, including the most important areas of it, such as MapReduce, YARN, and HDFS. This book serves as a starting point for programmers who are looking to analyze datasets of any kind with the help of big data, quality teams who are interested in evaluating MapReduce programs with respect to their functionality and performance, administrators who are setting up enterprise-ready Hadoop clusters with horizontal scaling, and individuals who wish to enhance their expertise on Apache Hadoop version 3 to solve complex problems.

What this book covers

Chapter 1, *Hadoop 3.0 – Background and Introduction*, gives you an overview of big data and Apache Hadoop. You will go through the history of Apache Hadoop's evolution, learn about what Hadoop offers today, and explore how it works. Also, you'll learn about the architecture of Apache Hadoop, as well as its new features and releases. Finally, you'll cover the commercial implementations of Hadoop.

Chapter 2, *Planning and Setting Up Hadoop Clusters*, covers the installation and setup of Apache Hadoop. We will start with learning about the prerequisites for setting up a Hadoop cluster. You will go through the different Hadoop configurations available for users, covering development mode, pseudo-distributed single nodes, and cluster setup. You'll learn how each of these configurations can be set up, and also run an example application of the configuration. Toward the end of the chapter, we will cover how you can diagnose Hadoop clusters by understanding log files and the different debugging tools available.

Chapter 3, *Deep Diving into the Hadoop Distributed File System*, goes into how HDFS works and its key features. We will look at the different data flowing patterns of HDFS, examining HDFS in different roles. Also, we'll take a look at various command-line interface commands for HDFS and the Hadoop shell. Finally, we'll look at the data structures that are used by HDFS with some examples.

Chapter 4, *Developing MapReduce Applications*, looks in depth at various topics pertaining to MapReduce. We will start by understanding the concept of MapReduce. We will take a look at the Hadoop application URL ports. Also, we'll study the different data formats needed for MapReduce. Then, we'll take a look at job compilation, remote job runs, and using utilities such as Tool. Finally, we'll learn about unit testing and failure handling.

Chapter 5, *Building Rich YARN Applications*, teaches you about the YARN architecture and the key features of YARN, such as resource models, federation, and RESTful APIs. Then, you'll configure a YARN environment in a Hadoop distributed cluster. Also, you'll study some of the additional properties of `yarn-site.xml`. You'll learn about the YARN distributed command-line interface. After this, we will delve into building YARN applications and monitoring them.

Chapter 6, *Monitoring and Administration of a Hadoop Cluster*, explores the different activities performed by Hadoop administrators for the monitoring and optimization of a Hadoop cluster. You'll learn about the roles and responsibilities of an administrator, followed by cluster planning. You'll dive deep into key management aspects of Hadoop clusters, such as resource management through job scheduling with algorithms such as Fair Scheduler and Capacity Scheduler. Also, you'll discover how to ensure high availability and security for an Apache Hadoop cluster.

Chapter 7, *Demystifying Hadoop Ecosystem Components*, covers the different components that constitute Hadoop's overall ecosystem offerings to solve complex industrial problems. We will take a brief overview of the tools and software that run on Hadoop. Also, we'll take a look at some components, such as Apache Kafka, Apache PIG, Apache Sqoop, and Apache Flume. After that, we'll cover the SQL and NoSQL Hadoop-based databases: Hive and HBase, respectively.

Chapter 8, *Advanced Topics in Apache Hadoop*, gets into advanced topics, such as the use of Hadoop for analytics using Apache Spark and processing streaming data using an Apache Storm pipeline. It will provide an overview of real-world use cases for different industries, with some sample code for you to try out independently.

To get the most out of this book

You won't need too much hardware to set up Hadoop. The minimum setup is a single machine / virtual machine, and the recommended setup is three machines.

It is better to have some hands-on experience of writing and running basic programs in Java, as well as some experience of using developer tools such as Eclipse.

Some understanding of the standard software development life cycle would be a plus.

As this is a quick-start guide, it does not provide complete coverage of all topics. Therefore, you will find links provided throughout the book to take you to the deep-dive of the given topic.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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 book is also hosted on GitHub at <https://github.com/PacktPublishing/Apache-Hadoop-3-Quick-Start-Guide>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Code in action

Visit the following link to check out videos of the code being run:

<http://bit.ly/2AznxS3>

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "You will need the `hadoop-client-<version>.jar` file to be added".

A block of code is set as follows:

```
<dependencies>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>3.1.0</version>
  </dependency>
</dependencies>
```

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

```
<!-- Put site-specific property overrides in this file. --><b><configuration>
  <b><property>
    <name>fs.default.name</name>
    <value>hdfs://<master-host>:9000</value>
  </property>
</b></configuration>
```

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

```
hrishikesh@base0:/$ df -m
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Right-click on the project and run **Maven install**, as shown in the following screenshot".



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Hadoop 3.0 - Background and Introduction

"There were 5 exabytes of information created between the dawn of civilization through 2003, but that much information is now created every two days."

– Eric Schmidt of Google, 2010

The world is evolving day by day, from automated call assistance to smart devices taking intelligent decisions, to self-driven decision-making cars to humanoid robots, all driven by processing large amount of data and analyzing it. We are rapidly approaching to the new era of data age. The IDC whitepaper (<https://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf>) on data evolution published in 2017 predicts data volumes to reach 163 zettabytes (1 zettabyte = 1 trillion terabytes) by the year 2025. This will involve digitization of all the analog data that we see between now and then. This flood of data will come from a broad variety of different device types, including IoT devices (sensor data) from industrial plants as well as home devices, smart meters, social media, wearables, mobile phones, and so on.

In our day-to-day life, we have seen ourselves participating in this evolution. For example, I started using a mobile phone in 2000 and, at that time, it had basic functions such as calls, torch, radio, and SMS. My phone could barely generate any data as such. Today, I use a 4G LTE smartphone capable of transmitting GBs of data including my photos, navigation history, and my health parameters from my smartwatch, on different devices over the internet. This data is effectively being utilized to make smart decisions.

Let's look at some real-world examples of big data:

- Companies such as Facebook and Instagram are using face recognition tools to identify photos, classify them, and bring you friend suggestions by comparison
- Companies such as Google and Amazon are looking at human behavior based on navigation patterns and location data, providing automated recommendations for shopping
- Many government organizations are analyzing information from CCTV cameras, social media feeds, network traffic, phone data, and bookings to trace criminals and predict potential threats and terrorist attacks
- Companies are using sentiment analysis from message posts and tweets to improve the quality of their products, as well as brand equities, and have targeted business growth
- Every minute, we send 204 million emails, view 20 million photos on Flickr, perform 2 million searches on Google, and generate 1.8 million likes on Facebook (Source)

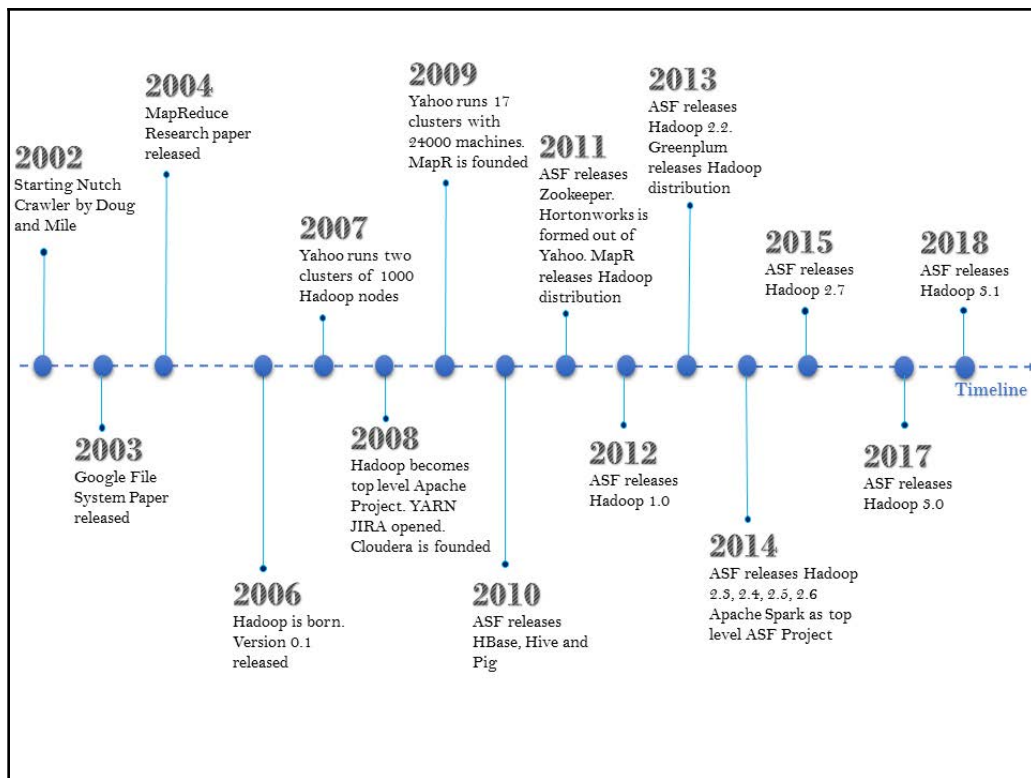
With this data growth, the demands to process, store, and analyze data in a faster and scalable manner will arise. So, the question is: are we ready to accommodate these demands? Year after year, computer systems have evolved and so has storage media in terms of capacities; however, the capability to read-write byte data is yet to catch up with these demands. Similarly, data coming from various sources and various forms needs to be correlated together to make meaningful information. For example, with a combination of my mobile phone location information, billing information, and credit card details, someone can derive my interests in food, social status, and financial strength. The good part is that we see a lot of potential of working with big data. Today, companies are barely scratching the surface; however, we are still struggling to deal with storage and processing problems unfortunately.

This chapter is intended to provide the necessary background for you to get started on Apache Hadoop. It will cover the following key topics:

- How it all started
- What Apache Hadoop is and why it is important
- How Apache Hadoop works
- Hadoop 3.0 releases and new features
- Choosing the right Hadoop distribution

How it all started

In the early 2000s, search engines on the World Wide Web were competing to bring improved and accurate results. One of the key challenges was about indexing this large data, keeping a limit over the cost factor on hardware. Doug Cutting and Mike Cafarella started development on Nutch in 2002, which would include a search engine and web crawler. However, the biggest challenge was to index billions of pages due to lack of matured cluster management systems. In 2003, Google published a research paper on **Google's distributed filesystem (GFS)** (<https://ai.google/research/pubs/pub51>). This helped them devise a distributed filesystem for Nutch called NDFS. In 2004, Google introduced MapReduce programming to the world. The concept of MapReduce was inspired from the Lisp programming language. In 2006, Hadoop was created under the Lucene umbrella. In the same year, Doug was employed by Yahoo to solve some of the most challenging issues with Yahoo Search, which was barely surviving. The following is a timeline of these and later events:



In 2007, many companies such as LinkedIn, Twitter, and Facebook started working on this platform, whereas Yahoo's production Hadoop cluster reached the 1,000-node mark. In 2008, **Apache Software Foundation (ASF)** moved Hadoop out of Lucene and graduated it as a top-level project. This was the time when the first Hadoop-based commercial system integration company, called Cloudera, was formed.

In 2009, AWS started giving MapReduce hosting capabilities, whereas Yahoo achieved the 24k nodes production cluster mark. This was the year when another **SI (System Integrator)** called MapR was founded. In 2010, ASF released HBase, Hive, and Pig to the world. In the year 2011, the road ahead for Yahoo looked difficult, so original Hadoop developers from Yahoo separated from it, and formed a company called Hortonworks. Hortonworks offers 100% open source implementation of Hadoop. The same team also become part of the Project Management Committee of ASF.

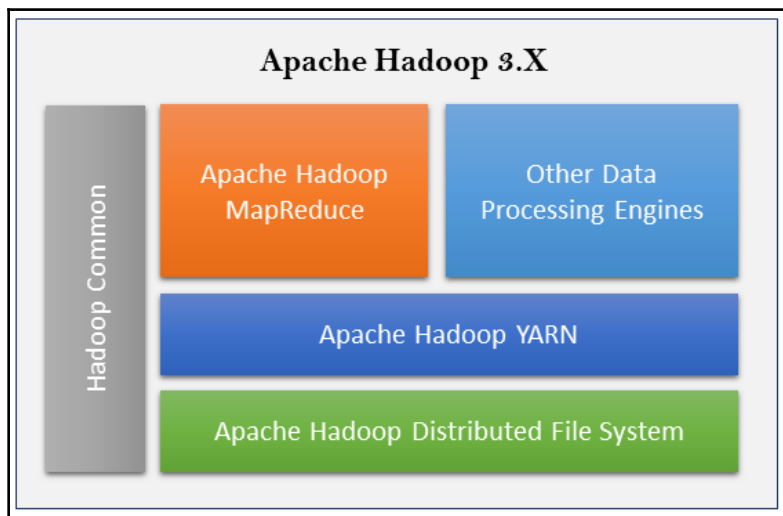
In 2012, ASF released the first major release of Hadoop 1.0, and immediately next year, it released Hadoop 2.X. In subsequent years, the Apache open source community continued with minor releases of Hadoop due to its dedicated, diverse community of developers. In 2017, ASF released Apache Hadoop version 3.0. On similar lines, companies such as Hortonworks, Cloudera, MapR, and Greenplum are also engaged in providing their own distribution of the Apache Hadoop ecosystem.

What Hadoop is and why it is important

The Apache Hadoop is a collection of open source software that enables distributed storage and processing of large datasets across a cluster of different types of computer systems. The Apache Hadoop framework consists of the following four key modules:

- Apache Hadoop Common
- Apache **Hadoop Distributed File System (HDFS)**
- Apache Hadoop MapReduce
- Apache Hadoop YARN (Yet Another Resource Manager)

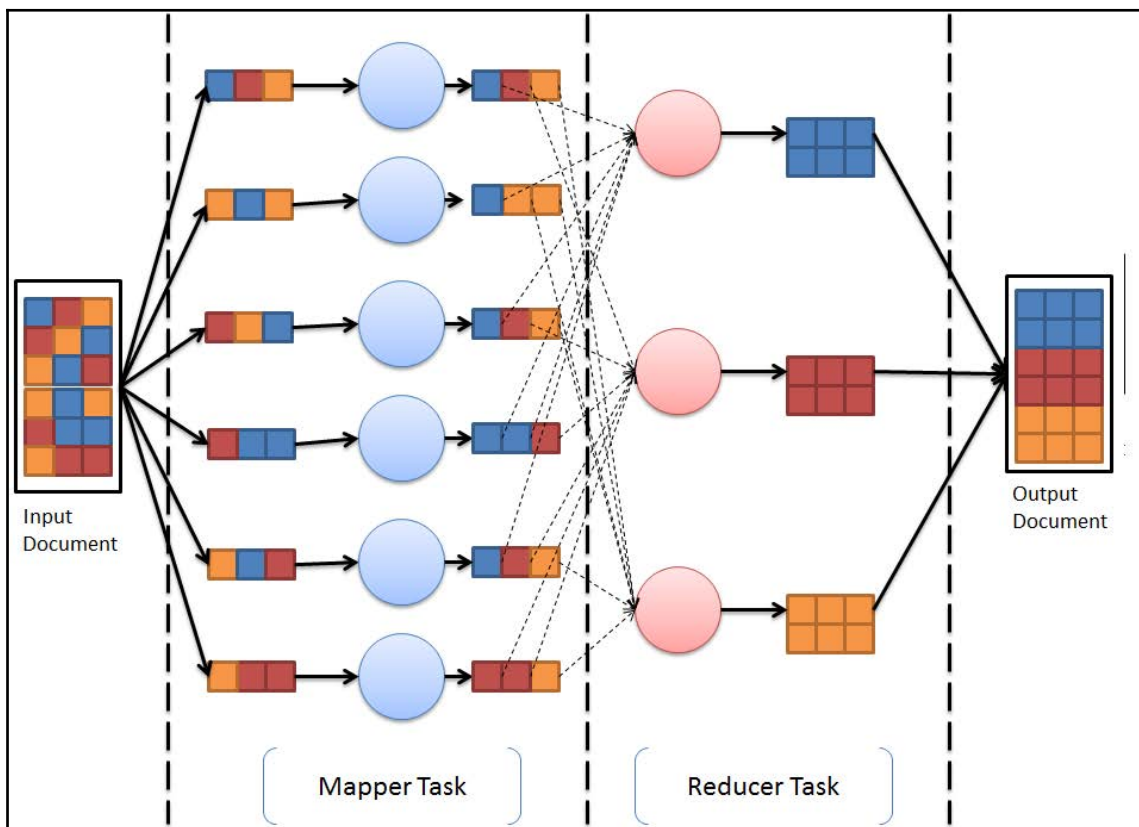
Each of these modules covers different capabilities of the Hadoop framework. The following diagram depicts their positioning in terms of applicability for Hadoop 3.X releases:



Apache Hadoop Common consists of shared libraries that are consumed across all other modules including key management, generic I/O packages, libraries for metric collection, and utilities for registry, security, and streaming. **Apache HDFS** provides highly tolerant distributed filesystem across clustered computers.

Apache Hadoop provides a distributed data processing framework for large datasets using a simple programming model called **MapReduce**. A programming task that is divided into multiple identical subtasks and that is distributed among multiple machines for processing is called a **map task**. The results of these map tasks are combined together into one or many **reduce tasks**. Overall, this approach of computing tasks is called the **MapReduce**

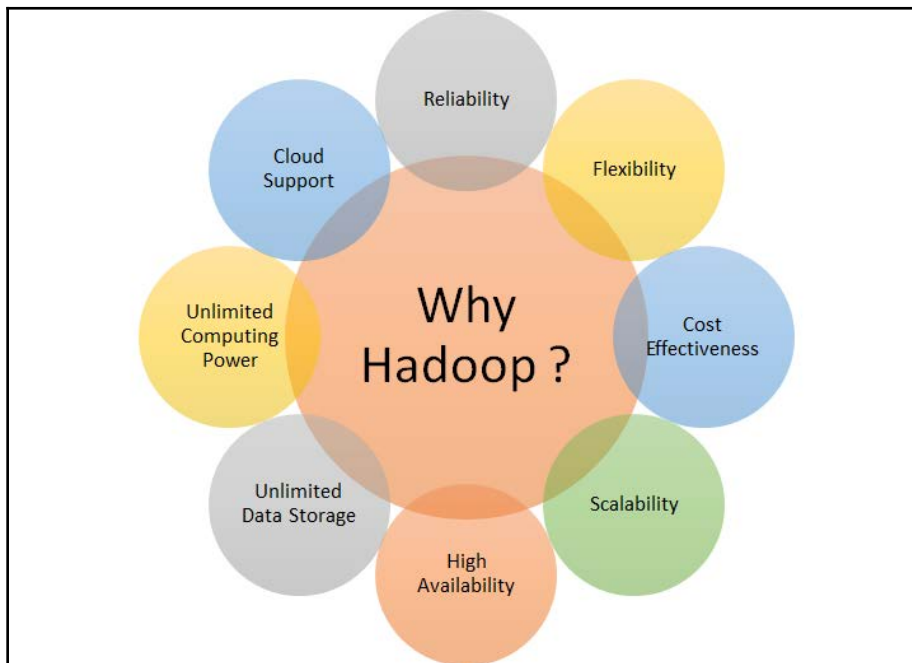
Approach. The MapReduce programming paradigm forms the heart of the Apache Hadoop framework, and any application that is deployed on this framework must comply to MapReduce programming. Each task is divided into a mapper task, followed by a reducer task. The following diagram demonstrates how MapReduce uses the divide-and-conquer methodology to solve its complex problem using a simplified methodology:



Apache Hadoop MapReduce provides a framework to write applications to process large amounts of data in parallel on Hadoop clusters in a reliable manner. The following diagram describes the placement of multiple layers of the Hadoop framework. Apache Hadoop YARN provides a new runtime for MapReduce (also called MapReduce 2) for running distributed applications across clusters. This module was introduced in Hadoop version 2 onward. We will be discussing these modules further in later chapters. Together, these components provide a base platform to build and compute applications from scratch. To speed up the overall application building experience and to provide efficient mechanisms for large data processing, storage, and analytics, the Apache Hadoop ecosystem comprises additional software. We will cover these in the last section of this chapter.

Now that we have given a quick overview of the Apache Hadoop framework, let's understand why Hadoop-based systems are needed in the real world.

Apache Hadoop was invented to solve large data problems that no existing system or commercial software could solve. With the help of Apache Hadoop, the data that used to get archived on tape backups or was lost is now being utilized in the system. This data offers immense opportunities to provide insights in history and to predict the best course of action. Hadoop is targeted to solve problems involving the four Vs (Volume, Variety, Velocity, and Veracity) of data. The following diagram shows key differentiators of why Apache Hadoop is useful for business:



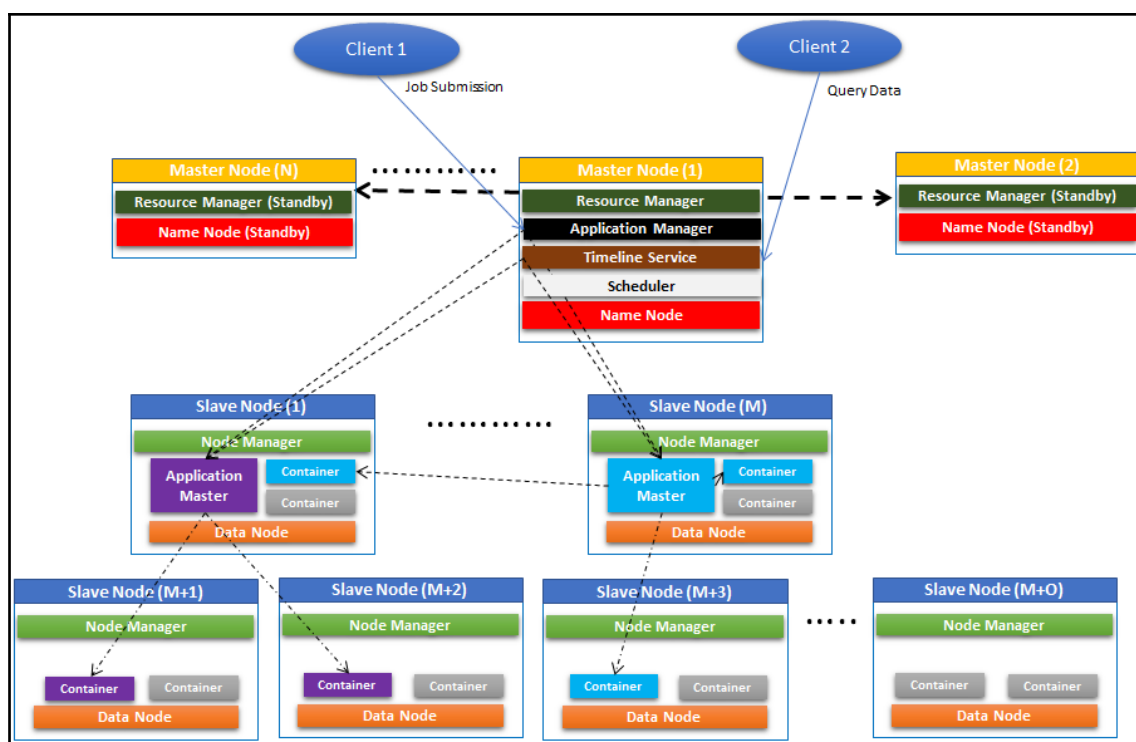
Let's go through each of the differentiators:

- **Reliability:** The Apache Hadoop distributed filesystem offers replication of data, with a default replication of 3x. This ensures that there is no data loss despite failure of cluster nodes.
- **Flexibility:** Most of the data that users today must deal with is unstructured. Traditionally, this data goes unnoticed; however, with Apache Hadoop, variety of data including structured and unstructured data can be processed, stored, and analyzed to make better future decisions. Hadoop offers complete flexibility to work across any type of data.
- **Cost effectiveness:** Apache Hadoop is completely open source; it comes for free. Unlike traditional software, it can run on any hardware or commodity systems and it does not require high-end servers; the overall investment and total cost of ownership of building a Hadoop cluster is much less than the traditional high-end system required to process data of the same scale.
- **Scalability:** Hadoop is a completely distributed system. With data growth, implementation of Hadoop clusters can add more nodes dynamically or even downsize them based on data processing and storage demands.
- **High availability:** With data replication and massively parallel computation running on multi-node commodity hardware, applications running on top of Hadoop provide high availability environment for all implementations.
- **Unlimited storage space:** Storage in Hadoop can scale up to petabytes of data storage with HDFS. HDFS can store any type of data of larger size in a completely distributed manner. This capability enables Hadoop to solve large data problems.
- **Unlimited computing power:** Hadoop 3.x onward supports more than 10,000 nodes of Hadoop clusters, whereas Hadoop 2.x supports up to 10,000 node clusters. With such a massive parallel processing capability, Apache Hadoop offers unlimited computing power to all applications.
- **Cloud support:** Today, almost all cloud providers support Hadoop directly as a service, which means a completely automated Hadoop setup is available on demand. It supports dynamic scaling too; overall it becomes an attractive model due to the reduced **Total Cost of Ownership (TCO)**.

Now is the time to do a deep dive into how Apache Hadoop works.

How Apache Hadoop works

The Apache Hadoop framework works on a cluster of nodes. These nodes can be either virtual machines or physical servers. The Hadoop framework is designed to work seamlessly on all types of these systems. The core of Apache Hadoop is based on Java. Each of the components in the Apache Hadoop framework performs different operations. Apache Hadoop is comprised of the following key modules, which work across HDFS, MapReduce, and YARN to provide a truly distributed experience to the applications. The following diagram shows the overall big picture of the Apache Hadoop cluster with key components:



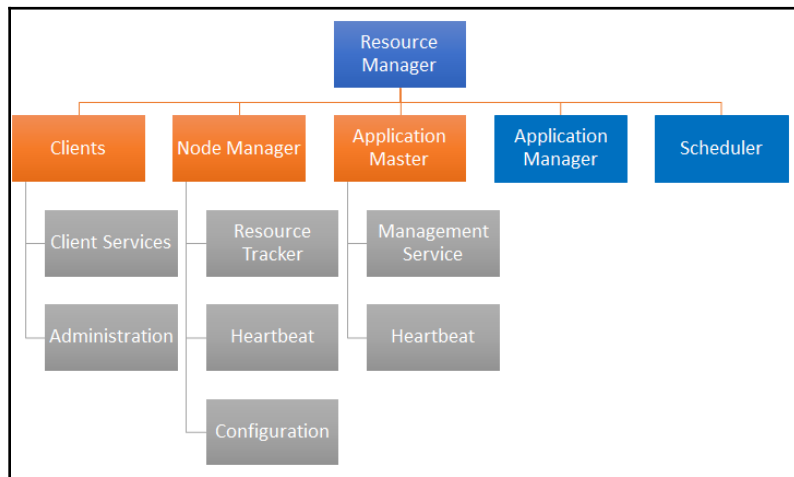
Let's go over the following key components and understand what role they play in the overall architecture:

- Resource Manager
- Node Manager
- YARN Timeline Service

- NameNode
- DataNode

Resource Manager

Resource Manager is a key component in the YARN ecosystem. It was introduced in Hadoop 2.X, replacing JobTracker (MapReduce version 1.X). There is one Resource Manager per cluster. Resource Manager knows the location of all slaves in the cluster and their resources, which includes information such as GPUs (Hadoop 3.X), CPU, and memory that is needed for execution of an application. Resource Manager acts as a proxy between the client and all other Hadoop nodes. The following diagram depicts the overall capabilities of Resource Manager:



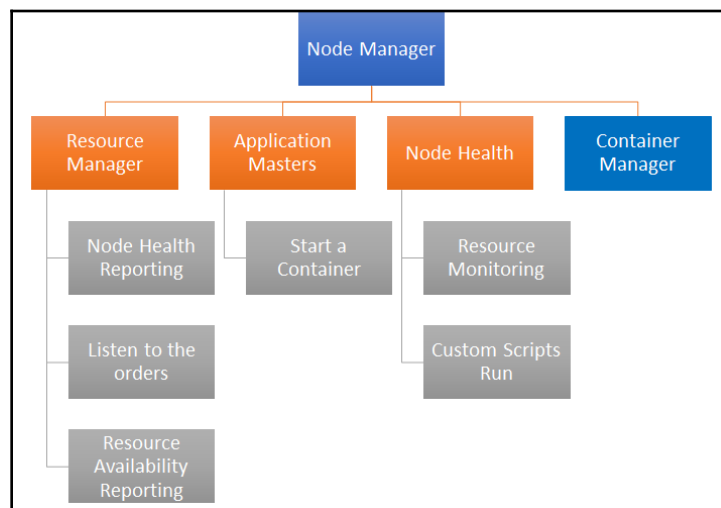
YARN resource manager handles all RPC such as services that allow clients to submit their jobs for execution and obtain information about clusters and queues and termination of jobs. In addition to regular client requests, it provides separate administration services, which get priorities over normal services. Similarly, it also keeps track of available resources and heartbeats from Hadoop nodes. Resource Manager communicates with Application Masters to manage registration/termination of an Application Master, as well as checking health. Resource Manager can be communicated through the following mechanisms:

- RESTful APIs
- User interface (New Web UI)
- Command-line interface (CLI)

These APIs provide information such as cluster health, performance index on a cluster, and application-specific information. Application Manager is the primary interacting point for managing all submitted applications. YARN Schedule is primarily used to schedule jobs with different strategies. It supports strategies such as capacity scheduling and fair scheduling for running applications. Another new feature of resource manager is to provide a fail-over with near zero downtime for all users. We will be looking at more details on resource manager in Chapter 5, *Building Rich YARN Applications* on YARN.

Node Manager

As the name suggests, Node Manager runs on each of the Hadoop slave nodes participating in the cluster. This means that there could many Node Managers present in a cluster when that cluster is running with several nodes. The following diagram depicts key functions performed by Node Manager:



Node Manager runs different services to determine and share the health of the node. If any services fail to run on a node, Node Manager marks it as unhealthy and reports it back to resource manager. In addition to managing the life cycles of nodes, it also looks at available resources, which include memory and CPU. On startup, Node Manager registers itself to resource manager and sends information about resource availability. One of the key responsibilities of Node Manager is to manage containers running on a node through its **Container Manager**. These activities involve starting a new container when a request is received from Application Master and logging the operations performed on container. It also keeps tabs on the health of the node.

Application Master is responsible for running one single application. It is initiated based on the new application submitted to a Hadoop cluster. When a request to execute an application is received, it demands container availability from resource manager to execute a specific program. Application Master is aware of execution logic and it is usually specific for frameworks. For example, Apache Hadoop MapReduce has its own implementation of Application Master.

YARN Timeline Service version 2

This service is responsible for collecting different metric data through its timeline collectors, which run in a distributed manner across Hadoop cluster. This collected information is then written back to storage. These collectors exist along with Application Masters—one per application. Similar to Application Manager, resource managers also utilize these timeline collectors to log metric information in the system. YARN Timeline Server version 2.X provides a RESTful API service to allow users to run queries for getting insights on this data. It supports aggregation of information. Timeline Server V2 utilizes Apache HBase as storage for these metrics by default, however, users can choose to change it.

NameNode

NameNode is the gatekeeper for all HDFS-related queries. It serves as a single point for all types of coordination on HDFS data, which is distributed across multiple nodes. NameNode works as a registry to maintain data blocks that are spread across Data Nodes in the cluster. Similarly, the secondary NameNodes keep a backup of active Name Node data periodically (typically every four hours). In addition to maintaining the data blocks, NameNode also maintains the health of each DataNode through the heartbeat mechanism. In any given Hadoop cluster, there can only be one active name node at a time. When an active NameNode goes down, the secondary NameNode takes up responsibility. A filesystem in HDFS is inspired from Unix-like filesystem data structures. Any request to create, edit, or delete HDFS files first gets recorded in journal nodes; journal nodes are responsible for coordinating with data nodes for propagating changes. Once the writing is complete, changes are flushed and a response is sent back to calling APIs. In case the flushing of changes in the journal files fails, the NameNode moves on to another node to record changes.



NameNode used to be single point of failure in Hadoop 1.X; however, in Hadoop 2.X, the secondary name node was introduced to handle the failure condition. In Hadoop 3.X, more than one secondary name node is supported. The same has been depicted in the overall architecture diagram.

DataNode

DataNode in the Hadoop ecosystem is primarily responsible for storing application data in distributed and replicated form. It acts as a slave in the system and is controlled by NameNode. Each disk in the Hadoop system is divided into multiple blocks, just like a traditional computer storage device. A block is a minimal unit in which the data can be read or written by the Hadoop filesystem. This ecosystem gives a natural advantage in slicing large files into these blocks and storing them across multiple nodes. The default block size of data node varies from 64 MB to 128 MB, depending upon Hadoop implementation. This can be changed through the configuration of data node. HDFS is designed to support very large file sizes and for write-once-read-many-based semantics.

Data nodes are primarily responsible for storing and retrieving these blocks when they are requested by consumers through Name Node. In Hadoop version 3.X, DataNode not only stores the data in blocks, but also the checksum or parity of the original blocks in a distributed manner. DataNodes follow the replication pipeline mechanism to store data in chunks propagating portions to other data nodes.

When a cluster starts, NameNode starts in a safe mode, until the data nodes register the data block information with NameNode. Once this is validated, it starts engaging with clients for serving the requests. When a data node starts, it first connects with Name Node, reporting all of the information about its data blocks' availability. This information is registered in NameNode, and when a client requests information about a certain block, NameNode points to the respective data not from its registry. The client then interacts with DataNode directly to read/write the data block. During the cluster processing, data node communicates with name node periodically, sending a heartbeat signal. The frequency of the heartbeat can be configured through configuration files.

We have gone through different key architecture components of the Apache Hadoop framework; we will be getting a deeper understanding in each of these areas in the next chapters.

Hadoop 3.0 releases and new features

Apache Hadoop development is happening on multiple tracks. The releases of 2.X, 3.0.X, and 3.1.X were simultaneous. Hadoop 3.X was separated from Hadoop 2.x six years ago. We will look at major improvements in the latest releases: 3.X and 2.X. In Hadoop version 3.0, each area has seen a major overhaul, as can be seen in the following quick overview:

- HDFS benefited from the following:
 - Erasure code
 - Multiple secondary Name Node support
 - Intra-Data Node Balancer
- Improvements to YARN include the following:
 - Improved support for long-running services
 - Docker support and isolation
 - Enhancements in the Scheduler
 - Application Timeline Service v.2
 - A new User Interface for YARN
 - YARN Federation
- MapReduce received the following overhaul:
 - Task-level native optimization
 - Feature to device heap-size automatically
- Overall feature enhancements include the following:
 - Migration to JDK 8
 - Changes in hosted ports
 - Classpath Isolation
 - Shell script rewrite and ShellDoc

Erasure Code (EC) is a one of the major features of the Hadoop 3.X release. It changes the way HDFS stores data blocks. In earlier implementations, the replication of data blocks was achieved by creating replicas of blocks on different node. For a file of 192 MB with a HDFS block size of 64 MB, the old HDFS would create three blocks and, if a cluster has a replication of three, it would require the cluster to store nine different blocks of data—576 MB. So the overhead becomes 200%, additional to the original 192 MB. In the case of EC, instead of replicating the data blocks, it creates parity blocks. In this case, for three blocks of data, the system would create two parity blocks, resulting in a total of 320 MB, which is approximately 66.67% overhead. Although EC achieves significant gain on data storage, it requires additional computing to recover data blocks in case of corruption, slowing down recovery with respect to the traditional way in old Hadoop versions.



A **parity drive** is a hard drive used in a RAID array to provide fault tolerance. A parity can be achieved with the Boolean XOR function to reconstruct missing data.

We have already seen **multiple secondary Name Node support** in the architecture section. **Intra-Data Node Balancer** is used to balance skewed data resulting from the addition or replacement of disks among Hadoop slave nodes. This balancer can be explicitly called from the HDFS shell asynchronously. This can be used when new nodes are added to the system.

In Hadoop v3, **YARN Scheduler** has been improved in terms of its scheduling strategies and prioritization between queues and applications. Scheduling can be performed among the most eligible nodes rather than one node at a time, driven by heartbeat reporting, as in older versions. YARN is being enhanced with abstract framework to support **long-running services**; it provides features to manage the life cycle of these services and support upgrades, resizing containers dynamically rather than statically. Another major enhancement is the release of **Application Timeline Service v2**. This service now supports multiple instances of readers and writes (compared to single instances in older Hadoop versions) with pluggable storage options. The overall metric computation can be done in real time, and it can perform aggregations on collected information. The RESTful APIs are also enhanced to support queries for metric data. **YARN User Interface** is enhanced significantly, for example, to show better statistics and more information, such as queue. We will be looking at it in Chapter 5, *Building Rich YARN Applications* and Chapter 6, *Monitoring and Administration of a Hadoop Cluster*.

Hadoop version 3 and above allows developers to define new resource types (earlier there were only two managed resources: CPU and memory). This enables applications to consider GPUs and disks as resources too. There have been new proposals to allow static resources such as hardware profiles and software versions to be part of the resourcing. Docker has been one of the most successful container applications that the world has adapted rapidly. In Hadoop version 3.0 onward, the experimental/alpha dockerization of YARN tasks is now made part of standard features. So, YARN can be deployed in **dockerized containers**, giving a complete **isolation** of tasks. Similarly, MapReduce Tasks are optimized (<https://issues.apache.org/jira/browse/MAPREDUCE-2841>) further with native implementation of Map output collector for activities such as sort and spill. This enhancement is intended to improve the performance of MapReduce tasks by two to three times.

YARN Federation is a new feature that enables YARN to scale over 100,000 of nodes. This feature allows a very large cluster to be divided into multiple sub-clusters, each running YARN Resource Manager and computations. YARN Federation will bring all these clusters together, making them appear as a single large YARN cluster to the applications. More information about YARN Federation can be obtained from [this source](#).

Another interesting enhancement is migration to newer JDK 8. Here is the supportability matrix for previous and new Hadoop versions and JDK:

Releases	Supported JDK
Hadoop 2.6.X	JDK 6 onward
Hadoop 2.7.X/2.8.X/2.9.X	JDK 7 onward
Hadoop 3.X	JDK 8 onward

Earlier, applications often had conflicts due to the single JAR file; however, the new release has two separate jar libraries: server side and client side. This achieves **isolation of classpaths between server and client jars**. The filesystem is being enhanced to support various types of storage such as Amazon S3, Azure Data Lake storage, and OpenStack Swift storage. Hadoop **Command-line interface** has been renewed and so are the daemons/processes to start, stop, and configure clusters. With older Hadoop (version 2.X), the heap size for Java and other tasks was required to be set through the `map/reduce.java.opts` and `map/reduce.memory.mb` properties. With Hadoop version 3.X, the **heap size** is derived automatically. All of the default ports used for NameNode, DataNode, and so forth are changed. We will be looking at new ports in the next chapter. In Hadoop 3, the **shell scripts** are rewritten completely to address some long-standing defects. The new enhancement allows users to add build directories to classpaths; the command to change permissions and the owner of HDFS folder structure will be done as a MapReduce job.

Choosing the right Hadoop distribution

We have seen the evolution of Hadoop from a simple lab experiment tool to one of the most famous projects of Apache Software Foundation in the previous section. When the evolution started, many commercial implementations of Hadoop spawned. Today, we see more than 10 different implementations that exist in the market ([Source](#)). There is a debate about whether to go with full open source-based Hadoop or with a commercial Hadoop implementation. Each approach has its pros and cons. Let's look at the open source approach.

Pros of open source-based Hadoop include the following:

- With a complete open source approach, you can take full advantage of community releases.
- It's easier and faster to reach customers due to software being free. It also reduces the initial cost of investment.
- Open source Hadoop supports open standards, making it easy to integrate with any system.

Cons of open source-based Hadoop include the following:

- In the complete open source Hadoop scenario, it takes longer to build implementations compared to commercial software, due to lack of handy tools that speed up implementation
- Supporting customers and fixing issues can become a tedious job due to the chaotic nature of the open source community
- The roadmap of the product cannot be controlled/influenced based on business needs

Given these challenges, many times, companies prefer to go with commercial implementations of Apache Hadoop. We will cover some of the key Hadoop distributions in this section.

Cloudera Hadoop distribution

Cloudera is well known and one of the oldest big data implementation players in the market. They have done first commercial releases of Hadoop in the past. Along with a Hadoop core distribution called CDH, Cloudera today provides many innovative tools such as proprietary **Cloudera Manager** to administer, monitor, and manage the Cloudera platform; **Cloudera Director** to easily deploy Cloudera clusters across the cloud; **Cloudera Data Science Workbench** to analyze large data and create statistical models out of it; and **Cloudera Navigator** to provide governance on the Cloudera platform. Besides ready-to-use products, it also provides services such as training and support. Cloudera follows separate versioning for its CDH; the latest CDH (5.14) uses Apache Hadoop 2.6.

Pros of Cloudera include the following:

- Cloudera comes with many tools that can help speed up the overall cluster creation process
- Cloudera-based Hadoop distribution is one of the most mature implementations of Hadoop so far

- The Cloudera User Interface and features such as the dashboard management and wizard-based deployment offer an excellent support system while implementing and monitoring Hadoop clusters
- Cloudera is focusing beyond Hadoop; it has brought in a new era of enterprise data hubs, along with many other tools that can handle much more complex business scenarios instead of just focusing on Hadoop distributions

Cons of Cloudera include the following:

- Cloudera distribution is not completely open source; there are proprietary components that require users to use commercial licenses. Cloudera offers a limited 60-day trial license.

Hortonworks Hadoop distribution

Hortonworks, although late in the game (founded in 2011), has quickly emerged as a leading vendor in the big data market. Hortonworks was started by Yahoo engineers. The biggest differentiator between Hortonworks and other Hadoop distributions is that Hortonworks is the only commercial vendor to offer its enterprise Hadoop distribution completely free and 100% open source. Unlike Cloudera, Hortonworks focuses on embedding Hadoop in existing data platforms. Hortonworks has two major product releases. **Hortonworks Data Platform (HDP)** provides an enterprise-grade open source Apache Hadoop distribution, while **Hortonworks Data Flow (HDF)** provides the only end-to-end platform that collects, curates, analyzes, and acts on data in real time and on-premises or in the cloud, with a drag-and-drop visual interface. In addition to products, Hortonworks also provides services such as training, consultancy, and support through its partner network. Now, let's look at its pros and cons.

Pros of the Hortonworks Hadoop distribution include the following:

- 100% open source-based enterprise Hadoop implementation with commercial license need
- Hortonworks provides additional open source-based tools to monitor and administer clusters

Cons of the Hortonworks Hadoop distribution include the following:

- As a business strategy, Hortonworks has focused on developing the platform layer so, for customers planning to utilize Hortonworks clusters, the cost to build capabilities is higher

MapR Hadoop distribution

MapR is one of the initial companies that started working on their own Hadoop distribution. When it comes to a Hadoop distribution, MapR has gone one step further and replaced HDFS of Hadoop with its own proprietary filesystem called MapRFS. MapRFS is a filesystem that supports enterprise-grade features such as better data management, fault tolerance, and ease of use. One key differentiator between HDFS and MapRFS is that MapRFS allows random writes on its filesystem. Additionally, unlike HDFS, it can be mounted locally through NFS to any filesystem. MapR implements POSIX (HDFS has POSIX-like implementation), so any Linux developer can apply their knowledge to run different commands seamlessly. MapR-like filesystems can be utilized for OLTP-like business requirements due to its unique features.

Pros of the MapR Hadoop distribution include the following:

- It's the only Hadoop distribution without Java dependencies (as MapR is based on C)
- Offers excellent and production-ready Hadoop clusters
- MapRFS is easy to use and it provides multi-node FS access on a local NFS mounted

Cons of the MapR Hadoop distribution include the following:

- It gets more and more proprietary instead of open source. Many companies are looking for vendor-free development, so MapR does not fit there.

Each of the distributions, including open source, that we covered have unique business strategy and features. Choosing the right Hadoop distribution for a problem is driven by multiple factors such as the following:

- What kind of application needs to be addressed by Hadoop
- The type of application—transactional or analytical—and what are the key data processing requirements
- Investments and the timeline of project implementation
- Support and training requirements of a given project

Summary

In this chapter, we started with big data problems and with an overview of big data and Apache Hadoop. We went through the history of Apache Hadoop's evolution, learned about what Hadoop offers today, and learned how it works. We also explored the architecture of Apache Hadoop, and new features and releases. Finally, we covered commercial implementations of Hadoop.

In the next chapter, we will learn about setting up an Apache Hadoop cluster in different modes.

2

Planning and Setting Up Hadoop Clusters

In the last chapter, we looked at big data problems, the history of Hadoop, along with an overview of big data, Hadoop architecture, and commercial offerings. This chapter will focus on hands-on, practical knowledge of how to set up Hadoop in different configurations. Apache Hadoop can be set up in the following three different configurations:

- **Developer mode:** Developer mode can be used to run programs in a standalone manner. This arrangement does not require any Hadoop process daemons, and jars can run directly. This mode is useful if developers wish to debug their code on MapReduce.
- **Pseudo cluster (single node Hadoop):** A pseudo cluster is a single node cluster that has similar capabilities to that of a standard cluster; it is also used for the development and testing of programs before they are deployed on a production cluster. Pseudo clusters provide an independent environment for all developers for coding and testing.
- **Cluster mode:** This mode is the real Hadoop cluster where you will set up multiple nodes of Hadoop across your production environment. You should use it to solve all of your big data problems.

This chapter will focus on setting up a new Hadoop cluster. The standard cluster is the one used in the production, as well as the staging, environment. It can also be scaled down and used for development in many cases to ensure that programs can run across clusters, handle fail-over, and so on. In this chapter, we will cover the following topics:

- Prerequisites for Hadoop
- Running Hadoop in development mode
- Setting up a pseudo Hadoop cluster
- Sizing the cluster

- Setting up Hadoop in cluster mode
- Diagnosing the Hadoop cluster

Technical requirements

You will need Eclipse development environment and Java 8 installed on your system where you can run/tweak these examples. If you prefer to use maven, then you will need maven installed to compile the code. To run the example, you also need Apache Hadoop 3.1 setup on Linux system. Finally, to use the Git repository of this book, you need to install Git.

The code files of this chapter can be found on GitHub:

<https://github.com/PacktPublishing/Apache-Hadoop-3-Quick-Start-Guide/tree/master/Chapter2>

Check out the following video to see the code in action:

<http://bit.ly/2Jofk5P>

Prerequisites for Hadoop setup

In this section, we will look at the necessary prerequisites for setting up Apache Hadoop in cluster or pseudo mode. Often, teams are forced to go through a major reinstallation of Hadoop and the data migration of their clusters due to improper planning for their cluster requirements. Hadoop can be installed on Windows as well as Linux; however, most productions that Hadoop installations run on are Unix or Linux-based platforms.

Preparing hardware for Hadoop

One important aspect of Hadoop setup is defining the hardware requirements and sizing before the start of a project. Although Apache Hadoop can run on commodity hardware, most of the implementations utilize server-class hardware for their Hadoop cluster. (Look at [powered by Hadoop](#) or go through the [Facebook Data warehouse research paper](#) in SIGMOD-2010 for more information).

There is no rule of thumb regarding the minimum hardware requirements for setting up Hadoop, but we would recommend the following configurations while running Hadoop to ensure reasonable performance:

- CPU $\geq$ 2 Core 2.5 GHz or more frequency
- Memory $\geq$ 8 GB RAM
- Storage $\geq$ 100 GB of free space, for running programs and processing data
- Good internet connection

There is an [official Cloudera blog](#) for cluster sizing information if you need more detail. If you are setting up a virtual machine, you can always opt for dynamically sized disks that can be increased based on your needs. We will look at how to size the cluster in the upcoming Hadoop cluster section.

Readying your system

Before you start with the prerequisites, you must ensure that you have sufficient space on your Hadoop nodes, and that you are using the respective directory appropriately. First, find out how much available disk space you have with the following command, also shown in the screenshot:

```
hrishikesh@base0:/$ df -m
```

```
hrishikesh@base0: /
hrishikesh@base0:/$ df -m
Filesystem      1M-blocks  Used Available Use% Mounted on
udev             1947      0      1947   0% /dev
tmpfs            395       7       389   2% /run
/dev/sda2        8944    6005     2463  71% /
tmpfs            1973      1     1973   1% /dev/shm
tmpfs             5        1        5   1% /run/lock
tmpfs            1973      0     1973   0% /sys/fs/cgroup
tmpfs            395       1       395   1% /run/user/1000
/dev/sr0         56       56        0 100% /media/hrishikesh/VBox_GAs_5.2.8
hrishikesh@base0:/$
```

The preceding command should present you with insight about the space available in MBs. Note that Apache Hadoop can be set up on a root user account or separately; it is safe to install it on a separate user account with space.

Although you need root access to these systems and Hadoop nodes, it is highly recommended that you create a user for Hadoop so that any installation impact is localized and controlled. You can create a user with a home directory with the following command:

```
hrishikesh@base0:/$ sudo adduser hadoop
```

The preceding command will prompt you for a password and will create a home directory for a given user in the default location (which is usually `/home/hadoop`). Remember the password. Now, switch the user to Hadoop for all future work using the following command:

```
hrishikesh@base0:/$ su - hadoop
```

This command will log you in as a Hadoop user. You can even add a Hadoop user in the `sudoers` list, as given [here](#).

Installing the prerequisites

In Linux, you will need to install all prerequisites through the package manager so they can be updated, removed, and managed in a much cleaner way. Overall, you will find two major flavors for Linux that each have different package management tools; they are as follows:

- RedHat Enterprise, Fedora, and CentOS primarily deal with rpm and they use `yum` and `rpm`
- Debian and Ubuntu use `.deb` for package management, and you can use `apt-get` or `dpkg`

In addition to the tools available on the command-line interface, you can also use user interface-based package management tools such as the software center or package manager, which are provided through the admin functionality of the mentioned operating systems. Before you start working on prerequisites, you must first update your local package manager database with the latest updates from source with the following command:

```
hadoop@base0:/$ sudo apt-get update
```

The update will take some time depending on the state of your OS. Once the update is complete, you may need to install an SSH client on your system. Secure Shell is used to connect Hadoop nodes with each other; this can be done with the following command:

```
hadoop@base0:/$ sudo apt-get install ssh
```

Once SSH is installed, you need to test whether you have the SSH server and client set up correctly. You can test this by simply logging in to the localhost using the SSH utility, as follows:

```
hadoop@base0:/$ ssh localhost
```

You will then be asked for the user's password that you typed earlier, and if you log in successfully, the setup has been successful. If you get a 'connection refused' error relating to port 22, you may need to install the SSH server on your system, which can be done with the following command:

```
hadoop@base0:/$ sudo apt-get install openssh-server
```

Next, you will need to install JDK on your system. Hadoop requires JDK version 1.8 and above. (Please visit this [link](#) for older compatible Java versions.) Most of the Linux installations have JDK installed by default, however, you may need to look for compatibility. You can check the current installation on your machine with the following command:

```
hadoop@base0:/$ sudo apt list | grep openjdk
```

To remove an older installation, use the following command:

```
hadoop@base0:/$ sudo apt-get remove <old-jdk>
```

To install JDK 8, use the following command:

```
hadoop@base0:/$ sudo apt-get install openjdk-8-jdk
```



All of the Hadoop installations and examples that you are seeing in this book are done on the following software: Ubuntu 16.04.3_LTS, OpenJDK 1.8.0_171 64 bit, and Apache Hadoop-3.1.0.

You need to ensure that your `JAVA_HOME` environment variable is set correctly in the Hadoop environment file, which is found in `$HADOOP_HOME/etc/hadoop/hadoop-env.sh`. Make sure that you add the following entry:

```
export JAVA_HOME= <location-of-java-home>
```

Working across nodes without passwords (SSH in keyless)

When Apache Hadoop is set up across multiple nodes, it often becomes evident that administrators and developers need to connect to different nodes to diagnose problems, run scripts, install software, and so on. Usually, these scripts are automated and are fired in a bulk manner. Similarly, master nodes often need to connect to slaves to start or stop the Hadoop processes using SSH. To allow the system to connect to a Hadoop node without any password prompt, it is important to make sure that all SSH access is keyless. Usually, this works in one direction, meaning system A can set up direct access to system B using a keyless SSH mechanism. Master nodes often hold data nodes or map-reduce jobs, so the scripts may run on the same machine using the SSH protocol. To achieve this, we first need to generate a passphrase for the SSH client on system A, as follows:

```
hadoop@base0:/$ ssh-keygen -t rsa
```

Press *Enter* when prompted for the passphrase (you do not want any passwords) or file location. This will create two keys: a private (`id_rsa`) key and a public (`id_rsa.pub`) key in your `.ssh` directory inside home (such as `/home/hadoop/.ssh`). You may choose to use a different protocol. The next step will only be necessary if you are working across two machines—for example, using a master and slave.

Now, copy the `id_rsa.pub` file of system A to system B. You can use the `scp` command to copy that, as follows:

```
hadoop@base0:/$ scp ~/.ssh/id_rsa.pub hadoop@base1:
```

The preceding command will copy the public key to a target system (for example, `base1`) under a Hadoop user's home directory. You should now be able to log in to the system to check whether the file has been copied or not.

Keyless entry is allowed by SSH only if the public key entry is part of the `authorized_key` file in the `.ssh` folder of the target system. So, to ensure that, we need to input the following command:

```
hadoop@base0:/$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

The following command can be used for different machines:

```
hadoop@base0:/$ cat ~/id_rsa.pub >> ~/.ssh/authorized_keys
```

That's it! Now it's time to test out your SSH keyless entry by logging in using SSH on your target machine. If you face any issues, you should run the SSH daemon in debug mode to see the error messages, as described [here](#). This is usually caused by a permissions issue, so make sure that all authorized keys and `id_rsa.pub` have ready access for all users, and that the private key is assigned to permission 600 (owner read/write only).

Downloading Hadoop

Once you have completed the prerequisites and SSH keyless entry with all the necessary nodes, you are good to download the Hadoop release. You can download Apache Hadoop from <http://www.apache.org/dyn/closer.cgi/hadoop/common/>. Hadoop provides two options for downloading—you can either download the source code of Apache Hadoop or you can download binaries. If you download the source code, you need to compile it and create binaries out of it. We will proceed with downloading binaries.

One important question that often arises while downloading Hadoop involves which version to choose. You will find many alpha and beta versions, as well as stable versions. Currently, the stable Hadoop version is 2.9.1, however this may change by the time you read this book. The answer to such a question depends upon usage. For example, if you are evaluating Hadoop for the first time, you may choose to go with the latest Hadoop version (3.1.0) with all-new features, so as to keep yourself updated with the latest trends and skills.

However, if you are looking to set up a production-based cluster, you may need to choose a version of Hadoop that is stable (such as 2.9.1), as well as established, to ensure peaceful project execution. In our case, we will download Hadoop 3.1.0, as shown in the following screenshot:

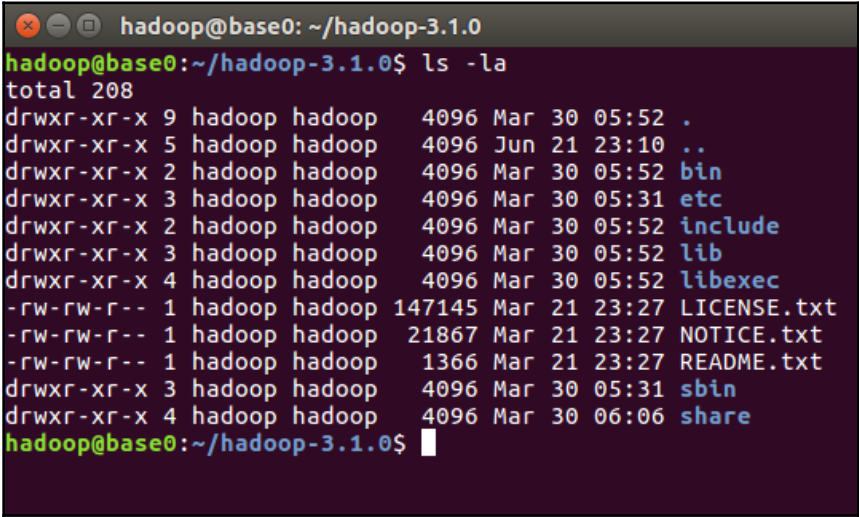


Name	Last modified	Size	Description
 Parent Directory		-	
 current/	2018-06-04 00:32	-	
 current2/	2018-06-04 00:19	-	
 hadoop-2.6.5/	2018-06-04 00:27	-	
 hadoop-2.7.6/	2018-06-04 00:29	-	
 hadoop-2.8.4/	2018-06-04 00:28	-	
 hadoop-2.9.0/	2018-06-04 00:19	-	
 hadoop-2.9.1/	2018-06-04 00:19	-	
 hadoop-3.0.2/	2018-06-04 00:38	-	
 hadoop-3.0.3/	2018-06-12 06:53	-	
 hadoop-3.1.0/	2018-06-04 00:32	-	
 stable/	2018-06-04 00:19	-	
 stable2/	2018-06-04 00:19	-	
 KEYS	2018-06-15 21:36	254K	
 readme.txt	2015-04-21 01:32	184	

You can download the binary (`tar.gz`) from Apache's website, and you can untar it with following command:

```
hadoop@base0:/$ tar xvfz <hadoop-downloaded-file>.tar.gz
```

The preceding command will extract the file in a given location. When you list the directory, you should see the following folders:

A terminal window titled 'hadoop@base0: ~/hadoop-3.1.0' shows the command 'ls -la' being executed. The output lists the contents of the directory, including folders like bin, etc, include, lib, libexec, sbin, and share, as well as files like LICENSE.txt, NOTICE.txt, and README.txt.

```
hadoop@base0:~/hadoop-3.1.0$ ls -la
total 208
drwxr-xr-x 9 hadoop hadoop 4096 Mar 30 05:52 .
drwxr-xr-x 5 hadoop hadoop 4096 Jun 21 23:10 ..
drwxr-xr-x 2 hadoop hadoop 4096 Mar 30 05:52 bin
drwxr-xr-x 3 hadoop hadoop 4096 Mar 30 05:31 etc
drwxr-xr-x 2 hadoop hadoop 4096 Mar 30 05:52 include
drwxr-xr-x 3 hadoop hadoop 4096 Mar 30 05:52 lib
drwxr-xr-x 4 hadoop hadoop 4096 Mar 30 05:52 libexec
-rw-rw-r-- 1 hadoop hadoop 147145 Mar 21 23:27 LICENSE.txt
-rw-rw-r-- 1 hadoop hadoop 21867 Mar 21 23:27 NOTICE.txt
-rw-rw-r-- 1 hadoop hadoop 1366 Mar 21 23:27 README.txt
drwxr-xr-x 3 hadoop hadoop 4096 Mar 30 05:31 sbin
drwxr-xr-x 4 hadoop hadoop 4096 Mar 30 06:06 share
hadoop@base0:~/hadoop-3.1.0$
```

- The `bin/` folder contains all executable for Hadoop
- `sbin/` contains all scripts to start or stop clusters
- `etc/` contains all configuration pertaining to Hadoop
- `share/` contains all the documentation and examples
- Other folders such as `include/`, `lib/`, and `libexec/` contain libraries and other dependencies

Running Hadoop in standalone mode

Now that you have successfully unzipped Hadoop, let's try and run a Hadoop program in standalone mode. As we mentioned in the introduction, Hadoop's standalone mode does not require any runtime; you can directly run your MapReduce program by running your compiled jar. We will look at how you can write MapReduce programs in the [Chapter 4, Developing MapReduce Applications](#). For now, it's time to run a program we have already prepared. To download, compile, and run the sample program, simply take the following steps:



Please note that this is not a mandatory requirement for setting up Apache Hadoop. You do not need a Maven or Git repository setup to compile or run Hadoop. We are doing this to run some simple examples.

1. You will need Maven and Git on your machine to proceed. Apache Maven can be set up with the following command:

```
hadoop@base0:/$ sudo apt-get install maven
```

2. This will install Maven on your local machine. Try running the `mvn` command to see if it has been installed properly. Now, install Git on your local machine with the following command:

```
hadoop@base0:/$ sudo apt-get install git
```

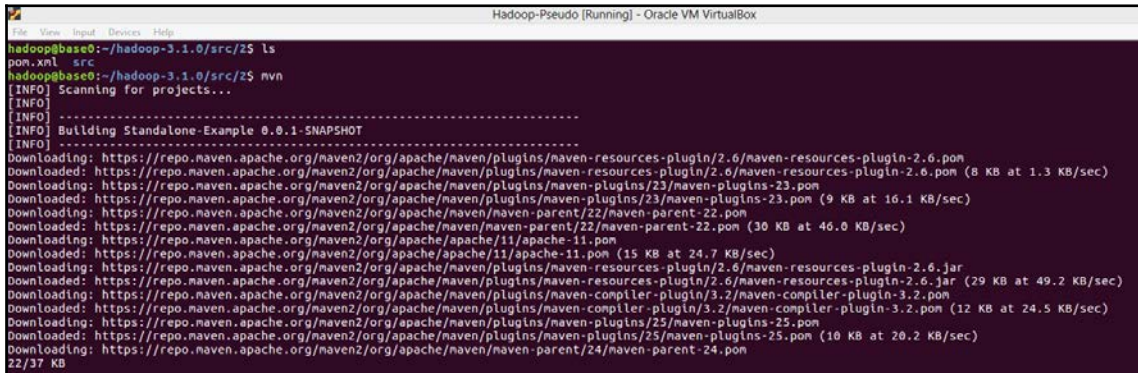
3. Now, create a folder in your home directory (such as `src/`) to keep all examples, and then run the following command to clone the Git repository locally:

```
hadoop@base0:/$ git clone https://github.com/PacktPublishing/  
Apache-Hadoop-3-Quick-Start-Guide/ src/
```

4. The preceding command will create a copy of your repository locally. Now go to folder `2/` for the relevant examples for [Chapter 2, Planning and Setting Up Hadoop Clusters](#).

- Now run the following **mvn** command from the `2/` folder. This will start downloading artifacts from the internet that have a dependency to build an example project, as shown in the next screenshot:

```
hadoop@base0: /$ mvn
```



```
hadoop@base0:~/hadoop-3.1.0/src/2$ ls
pom.xml src
hadoop@base0:~/hadoop-3.1.0/src/2$ mvn
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Standalone-Example 0.0.1-SNAPSHOT
[INFO] -----
[INFO]
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plugin/2.6/maven-resources-plugin-2.6.pom (8 KB at 1.3 KB/sec)
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plugin/2.6/maven-resources-plugin-2.6.pom (8 KB at 1.3 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/23/maven-plugins-23.pom (9 KB at 16.1 KB/sec)
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/23/maven-plugins-23.pom (9 KB at 16.1 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/22/maven-parent-22.pom (30 KB at 46.0 KB/sec)
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/22/maven-parent-22.pom (30 KB at 46.0 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/apache/11/apache-11.pom (15 KB at 24.7 KB/sec)
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/apache/11/apache-11.pom (15 KB at 24.7 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plugin/2.6/maven-resources-plugin-2.6.jar (29 KB at 49.2 KB/sec)
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plugin/2.6/maven-resources-plugin-2.6.jar (29 KB at 49.2 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-compiler-plugin/3.2/maven-compiler-plugin-3.2.pom (12 KB at 24.5 KB/sec)
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-compiler-plugin/3.2/maven-compiler-plugin-3.2.pom (12 KB at 24.5 KB/sec)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/25/maven-plugins-25.pom (10 KB at 20.2 KB/sec)
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/25/maven-plugins-25.pom (10 KB at 20.2 KB/sec)
[INFO] 22/37 KB
```

- Finally, you will get a build successful message. This means the jar, including your example, has been created and is ready to go. The next step is to use this jar to run the sample program which, in this case, provides a utility that allow users to supply a regular expression. The MapReduce program will then search across the given folder and bring up the matched content and its count.
- Let's now create an input folder and copy some documents into it. We will use a simple expression to get all the words that are separated by at least one white space. In that case, the expression will be `\\s+`. (Please refer to the standard Java documentation for information on how to create regular Java expressions for string patterns [here](#).)
- Create a folder in which you can put sample text files for expression matching. Similarly, create an output folder to save output. To run the program, run the following command:

```
hadoop@base0: /$ <hadoop-home>/bin/hadoop jar
<location-of generated-jar> ExpressionFinder "\\s+" <folder-
containing-files-for input> <new-output-folder> > stdout.txt
```

In most cases, the location of the jar will be in the target folder inside the project's home. The command will create a MapReduce job, run the program, and then produce the output in the given output folder. A successful run should end with no errors, as shown in the following screenshot:

```

Bytes Written=72872
2018-10-23 19:16:30,746 INFO mapred.LocalJobRunner: Finishing task: attempt_local420769815_0001_r_000000_0
2018-10-23 19:16:30,746 INFO mapred.LocalJobRunner: reduce task executor complete.
2018-10-23 19:16:31,072 INFO mapreduce.Job: map 100% reduce 100%
2018-10-23 19:16:31,086 INFO mapreduce.Job: Job job_local420769815_0001 completed successfully
2018-10-23 19:16:31,793 INFO mapreduce.Job: Counters: 35
  File System Counters
    FILE: Number of bytes read=1841141
    FILE: Number of bytes written=18518686
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
    HDFS: Number of bytes read=2555813
    HDFS: Number of bytes written=72872
    HDFS: Number of read operations=1155
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=34
  Map-Reduce Framework
    Map input records=2678
    Map output records=2842
    Map output bytes=111764
    Map output materialized bytes=97282
    Input split bytes=3676
    Combine input records=2842
    Combine output records=1812
    Reduce input groups=1433
    Reduce shuffle bytes=97282
    Reduce input records=1812
    Reduce output records=1433
    Spilled Records=3624
    Shuffled Maps =31
    Failed Shuffles=0
    Merged Map outputs=31
    GC time elapsed (ms)=1315
    Total committed heap usage (bytes)=36865835008
  Shuffle Errors
    BAD_ID=0
    CONNECTION=0
    IO_ERROR=0
    WRONG_LENGTH=0
    WRONG_MAP=0
    WRONG_REDUCE=0
  File Input Format Counters
    Bytes Read=101378
  File Output Format Counters
    Bytes Written=72872
hadoop@base0:~/hadoop-3.1.0$

```

Similarly, the output folder will contain the files `part-r-00000` and `_SUCCESS`. The file `part-r-00000` should contain the output of your expression run on multiple files. You can play with other regular expressions if you wish. Here, we have simply run a regular expression program that can run over masses of files in a completely distributed manner. We will move on to look at the programming aspects of MapReduce in the [Chapter 4, Developing MapReduce Applications](#).

Setting up a pseudo Hadoop cluster

In the last section, we managed run Hadoop in a standalone mode. In this section, we will create a pseudo Hadoop cluster on a single node. So, let's try and set up HDFS daemons on a system in the pseudo distributed mode. When we set up HDFS in a pseudo distributed mode, we install name nodes and data nodes on the same machine, but before we start the instances for HDFS, we need to set the configuration files correctly. We will study different configuration files in the next chapter. First, open `core-sites.xml` with the following command:

```
hadoop@base0:/$ vim etc/hadoop/core-sites.xml
```

Now, set the DFS default name for the file system using the `fs.default.name` property. The core site file is responsible for storing all of the configuration related to Hadoop Core. Replace the content of the file with the following snippet:

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

Setting the preceding property simplifies all of your command-line work, as you do not need to provide the file system location every time you use the CLI (command-line interface) of HDFS. The port `9000` is the location where name nodes are supposed to receive a heartbeat from data nodes (in this case, on the same machine). You can also provide your machine IP address as well, if you want to make your file system accessible from the outside. The file should look like the following screenshot:

```
hadoop@base0: ~/hadoop-3.1.0
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License. See accompanying LICENSE file.
-->

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

<configuration>
    <property>
        <name>fs.default.name</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
~
~
~
~
~
~
~
~
~
~
/etc/hadoop/core-site.xml" 24L, 869C written
```

Similarly, we now need to set up the `hdfs-site.xml` file with a replication property. Since we are running in a pseudo distributed mode on a single system, we will set the replication factor to 1, as follows:

```
hadoop@base0:/$ vim etc/hadoop/hdfs-sites.xml
```

Now add the following code snippet to the file:

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

The HDFS site file is responsible for storing all configuration related to HDFS (including name node, secondary name node, and data node). When setting up HDFS for the first time, the HDFS needs to be formatted. This process will create a file system and additional storage structures on name nodes (primarily the metadata part of HDFS). Type the following command on your Linux shell to format the name node:

```
hadoop@base0:/$ bin/hdfs namenode -format
```

You can now start the HDFS processes by running the following command from Hadoop's home directory:

```
hadoop@base0:/$ ./sbin/start-dfs.sh
```

The logs can be traced at `$HADOOP_HOME/logs/`. Now, access `http://localhost:9870` from your browser, and you should see the DFS health page, as shown in the following screenshot:

The screenshot shows a web browser window with the URL `localhost:9870/dfshealth.html#tab-overview`. The page has a green header with navigation tabs: Hadoop, Overview (selected), Datanodes, Datanode Volume Failures, Snapshot, Startup Progress, and Utilities. The main content area is titled "Overview 'localhost:9000' (active)".

Started:	Sat Jun 23 16:39:10 +0530 2018
Version:	3.1.0, r16b70619a24cdcfd3b0fcf4b58ca77238ccbe6d
Compiled:	Fri Mar 30 05:30:00 +0530 2018 by centos from branch-3.1.0
Cluster ID:	CID-0c2b35b4-a9df-4871-af56-eebbcc4e7e88
Block Pool ID:	BP-1107831925-127.0.1.1-1529752132627

Summary

Security is off.
 Safemode is off.

1 files and directories, 0 blocks (0 replicated blocks, 0 erasure coded block groups) = 1 total filesystem object(s).

Heap Memory used 34.96 MB of 65.17 MB Heap Memory. Max Heap Memory is 955.13 MB.

Non Heap Memory used 44.55 MB of 45.65 MB Committed Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	8.73 GB
Configured Remote Capacity:	0 B
DFS Used:	24 KB (0%)
Non DFS Used:	6.99 GB

As you can see, data node-related information can be found on `http://localhost:9864`. If you try running the same example again on the node, it will not run; this is because the input folder is defaulted to HDFS, and the system can no longer find it, thereby throwing `InvalidInputException`. To run the same example, you need to create an input folder first and copy the files into it. So, let's create an input folder on HDFS with the following code:

```
hadoop@base0:/$ ./bin/hdfs dfs -mkdir /user
hadoop@base0:/$ ./bin/hdfs dfs -mkdir /user/hadoop
hadoop@base0:/$ ./bin/hdfs dfs -mkdir /user/hadoop/input
```

Now the folders have been created, you can copy the content from the input folder present on the local machine to HDFS with the following command:

```
hadoop@base0:/$ ./bin/hdfs dfs -copyFromLocal input/* /user/hadoop/input/
```

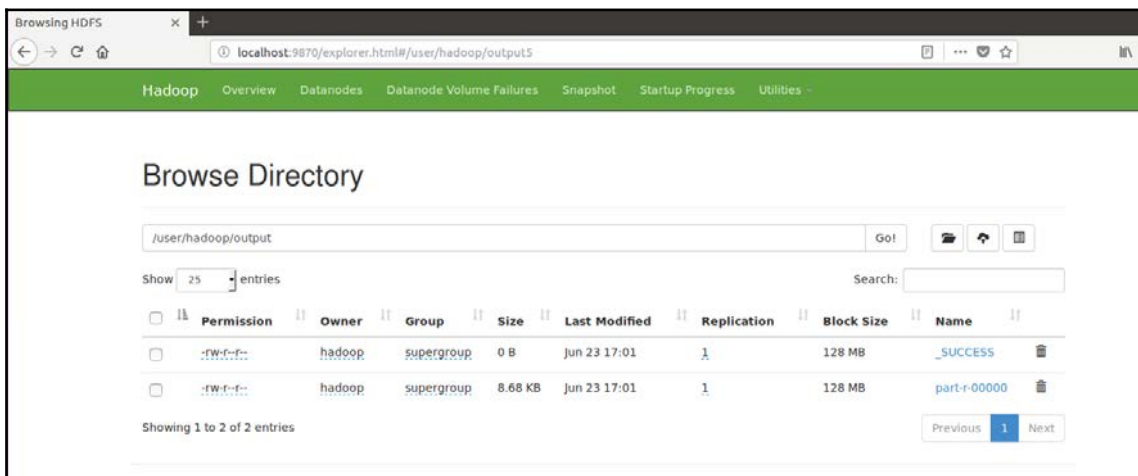
Input the following to check the content of the input folder:

```
hadoop@base0:/$ ./bin/hdfs dfs -ls input/
```

Now run your program with the input folder name, and output folder; you should be able to see the outcome on HDFS inside `/user/hadoop/<output-folder>`. You can run the following concatenated command on your folder:

```
hadoop@base0:/$ ./bin/hdfs dfs -cat <output folder path>/part-r-00000
```

Note that the output of your MapReduce program can be seen through the name node in your browser, as shown in the following screenshot:



Congratulations! You have successfully set up your pseudo distributed Hadoop node installation. We will look at setting up YARN for clusters, as well as pseudo distributed setup, in Chapter 5, *Building Rich YARN Applications*. Before we jump into the Hadoop cluster setup, let's first look at planning and sizing with Hadoop.

Planning and sizing clusters

Once you start working on problems and implementing Hadoop clusters, you'll have to deal with the issue of sizing. It's not just the sizing aspect of clusters that needs to be considered, but the SLAs associated with Hadoop runtime as well. A cluster can be categorized based on workloads as follows:

- **Lightweight:** This category is intended for low computation and fewer storage requirements, and is more useful for defined datasets with no growth
- **Balanced:** A balanced cluster can have storage and computation requirements that grow over time
- **Storage-centric:** This category is more focused towards storing data, and less towards computation; it is mostly used for archival purposes, as well as minimal processing
- **Computational-centric:** This cluster is intended for high computation which requires CPU or GPU-intensive work, such as analytics, prediction, and data mining

Before we get on to solve the sizing problem of a Hadoop cluster, however, we have to understand the following topics.

Initial load of data

The initial load of data is driven by existing content that migrates on Hadoop. The initial load can be calculated from the existing landscape. For example, if there are three applications holding different types of data (structured and unstructured), the initial storage estimation will be calculated based on the existing data size. However, the data size will change based on the Hadoop component. So, if you are moving tables from RDBMS to Hive, you need to look at the size of each table as well as the table data types to compute the size accordingly. This is instead of looking at DB files for sizing. Note that Hive data sizes are available [here](#).

Organizational data growth

Although Hadoop allows you to add and remove new nodes dynamically for on-premise cluster setup, it is never a day-to-day task. So, when you approach sizing, you must be cognizant of data growth over the years. For example, if you are building a cluster to process social media analytics, and the organization expects to add x pages a month for processing, sizing needs to be computed accordingly. You may start computing data generation for each with the following formula:

Data Generated in Year X = Data Generated in Year (X-1) X (1 * % Growth) + Data coming from additional sources in year X.

The following image shows a cluster sizing calculator, which can be used to compute the size of your cluster based on data growth (**Excel attached**). In this case, for the first year, last year's data can provide an initial size estimate:

New Data Sources Sizing						Reference Data	
Data Source	Frequency	Avg. Size	Avg. Size Unit	No. of Counts	Annual Data Generated in MB		
Data Source1	Daily	10 KB		1	3.65	Frequency	Multiplier
Data Source2	Monthly	100 KB		1	1.2	Annually	1
Data Source3	Monthly	120 KB		1	1.44	Daily	365
Data Source4	Monthly	1 MB		1	12	Monthly	12
Data Source5	Monthly	100 Bytes		1	0.0012	Weekly	52
Data Source6	Annually	1 GB		1	1000000	Hourly	105120
Data Source7	Hourly	8 Bytes		1	0.84096	Per Minute	6307200
Data Source8	Daily	1 KB		1	0.365	Per Second	378432000
Data Source9	Daily	11 KB		1	4.015		
Data Source10	Daily	21 KB		1	7.665		
Total Data Added Newly in MB					1000031		
Total Data Added Newly in TB (D)					1	Unit	Bytes
						Bytes	1
						KB	1000
						MB	1000000
						GB	1E+12
						TB	1E+24
Cluster Sizing for a given Year							
Last Year's Data in TB (A)					1.5		
% Data Growth (B)					10%		
Current Year's Data in TB (C = A*(1+B))					1.65		
Newly Added Data in TB (D)					1		
Total Data for Hadoop in TB for given year (C+D)					2.65		

While we work through storage sizing, it is worth pointing out another interesting difference between Hadoop and traditional storage systems, that is, Hadoop does not require RAID servers. This is because it does not add value primarily due to the underlying data replication of HDFS, scalability, and high-availability capability.

Workload and computational requirements

While the previous two areas cover the sizing of the cluster, the workload requirements drive the computational capabilities of the cluster. All CPU-intensive operations require a higher count of CPUs and better configuration for computing. The number of Mapper and Reducer jobs that are run as a part of Hadoop also contribute to the requirements. Mapper tasks are usually higher than Reducer tasks, for example. The ratio of Mapper and Reducer is determined by processing requirements at both ends.

There is no definitive count that one can reach regarding memory and CPU requirements, as they vary based on replicas of block, the computational processing of tasks, and data storage needs. To help with this, we have provided a calculator which considers different configurations of a Hadoop cluster, such as CPU-intensive, memory-intensive, and balanced.

High availability and fault tolerance

One of the major advantages of Hadoop is the high availability of a cluster. However, it also brings the additional burden of processing nodes based on requirements, thereby impacting sizing. The Data Replication Factor (DRF) of an HDFS node is directly proportional to the size of cluster; for example, if you have 200 GB of usable data, and you need a high replication of 5 (that means each data block will be replicated five times in the cluster), then you need to work out sizing for $200 \text{ GB} \times 5$, which equals 1 TB. The default value of DRF in Hadoop is 3. A replication value of 3 works well because:

- It offers ample avenues to recover from one of two copies, in the case of a corrupt third copy
- Additionally, even if a second copy fails during the recovery period, you still have one copy of your data to recover

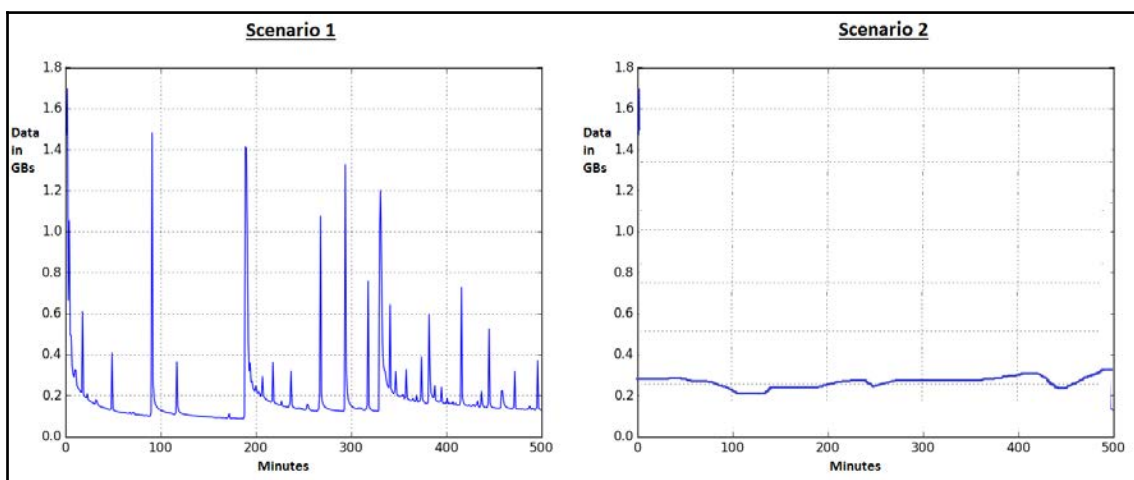
While determining the replication factor, you need to consider the following parameters:

- The network reliability of your Hadoop cluster
- The probability of failure of a node in a given network
- The cost of increasing the replication factor by one
- The number of nodes or VMs that will make up your cluster

If you are building a Hadoop cluster with three nodes, a replication factor of 4 does not make sense. Similarly, if a network is not reliable, the name node can access copy from a nearby available node. For systems with higher failure probabilities, the risk of losing data is higher, given that the probability of a second node increases.

Velocity of data and other factors

The velocity of data generated and transferred to the Hadoop cluster also impacts cluster sizing. Take two scenarios of data population, such as data generated in GBs per minute, as shown in the following diagram:



In the preceding diagram, both scenarios have generated the same data each day, but with a different velocity. In the first scenario, there are spikes of data, whereas the second sees a consistent flow of data. In scenario 1, you will need more hardware with additional CPUs or GPUs and storage over scenario 2. There are many other influencing parameters that can impact the sizing of the cluster; for example, the type of data can influence the compression factor of your cluster. Compression can be achieved with gzip, bzip, and other compression utilities. If the data is textual, the compression is usually higher. Similarly, intermediate storage requirements also add up to an additional 25% to 35%. Intermediate storage is used by MapReduce tasks to store intermediate results of processing. You can access an example Hadoop sizing calculator [here](#).

Setting up Hadoop in cluster mode

In this section, we will focus on setting up a cluster of Hadoop. We will also go over other important aspects of a Hadoop cluster, such as sizing guidelines, setup instructions, and so on. A Hadoop cluster can be set up with `Apache Ambari`, which offers a much simpler, semi-automated, and error-prone configuration of a cluster. However, the latest version of Ambari at the time of writing supports older Hadoop versions. To set up Hadoop 3.1, we must do so manually. By the time this book is out, you may be able to use a much simpler installation process. You can read about older Hadoop installations in the Ambari installation guide, available [here](#).



Before you set up a Hadoop cluster, it would be good to check the sizing of a cluster so that you can plan better, and avoid reinstallation due to incorrectly estimated cluster size. Please refer to the *Sizing the cluster* section in this chapter before you actually install and configure a Hadoop cluster.

Installing and configuring HDFS in cluster mode

First of all, for all master nodes (name node and secondary name node) and slaves, you need to enable keyless SSH entry in both directions, as described in previous sections. Similarly, you will need a Java environment on all of the available nodes, as most of Hadoop is based on Java itself.



When you add nodes to your cluster, you need to copy all of your configuration and your Hadoop folder. The same applies to all components of Hadoop, including HDFS, YARN, MapReduce, and so on.

It is a good idea to have a shared network drive with access to all hosts, as this will enable easier file sharing. Alternatively, you can write a simple shell script to make multiple copies using SCP. So, create a file (`targets.txt`) with a list of hosts (`user@system`) at each line, as follows:

```
hadoop@base0
```

```
hadoop@base1
```

```
hadoop@base2
```

```
....
```

Now create the following script in a text file and save it as `.sh` (for example, `scpall.sh`):

```
#!/bin/sh
# This is a SCP script to copy files to all folders
for dest in $(< targets.txt); do
    scp $1 ${dest}:$2
done
```

You can call the preceding script with the first parameter as the source file name, and the second parameter as the target directory location, as follows:

```
hadoop@base0:/$ ./scpall.sh etc/hadoop/mapred-conf.xml etc/hadoop/mapred-conf.xml
```

When identifying slaves or master nodes, you can choose to use the IP address or the host name. It is better to use host names for readability, but bear in mind that they require DNS entries to resolve an IP address. If you do not have access allowing you to introduce DNS entries (DNS entries are usually controlled by the IT teams of an organization), you can simply work an entry out by adding entries in the `/etc/hosts` file using a root login. The following screenshot illustrates how this file can be updated; the same file can be passed to all hosts through the SCP utility or shared folder:

```
hadoop@base0: ~/hadoop-3.1.0
# 27.0.0.1      localhost
# 127.0.1.1     base0
# 192.168.0.110 base1

# The following lines are desirable for IPv6 capable hosts
::1            ip6-localhost ip6-loopback
fe00::0        ip6-localnet
ff00::0        ip6-mcastprefix
ff02::1        ip6-allnodes
ff02::2        ip6-allrouters
~
~
~
~
~
~
~
~
~
~
"/etc/hosts" [readonly] 10L, 240C
```

Now download the Hadoop distribution as discussed. If you are working with multiple slave nodes, you can configure the folder for one slave and then simply copy it to another slave using the `scpall` utility. The slave configuration is usually similar. When we refer to slaves, we mean the nodes that do not have any master processes, such as name node, secondary name node, or YARN services.

Let's now proceed with the configuration of important files.

First, edit `etc/hadoop/core-site.xml`. It should have no metadata except an empty `<configuration>` tag, so add the following entries to it using the relevant code.

For `core-site.xml`, input:

```
<!-- Put site-specific property overrides in this file. --><configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://<master-host>:9000</value>
  </property>
</configuration>
```

Here, the `<master-host>` is the host name where your name node is configured. This configuration is to go in all of the data nodes in Hadoop. Remember to set up the Hadoop DFS replication factor as planned and add its entry in `etc/hadoop/hdfs-site.xml`.

For `hdfs-site.xml`, input:

```
<!-- Put site-specific property overrides in this file. --><configuration>
  <property>
    <name>dfs.replication</name>
    <value>3</value>
  </property>
</configuration>
```

The preceding snippet covers the configuration needed to run the HDFS. We will look at important, specific aspects of these configuration files in Chapter 3, *Deep Dive into the Hadoop Distributed File System*.

Another important configuration required is the `etc/hadoop/workers` file, which lists all of the data nodes. You will need to add the data nodes' host names and save it as follows:

```
base0
base1
base2
```

..

In this case, we are using `base*` names for all Hadoop nodes. This configuration has to happen over all of the nodes that are participating in the cluster. You may use the `scpall.sh` script to propagate the changes. Once this is done, the configuration is complete.

Let's start by formatting the name node first, as follows:

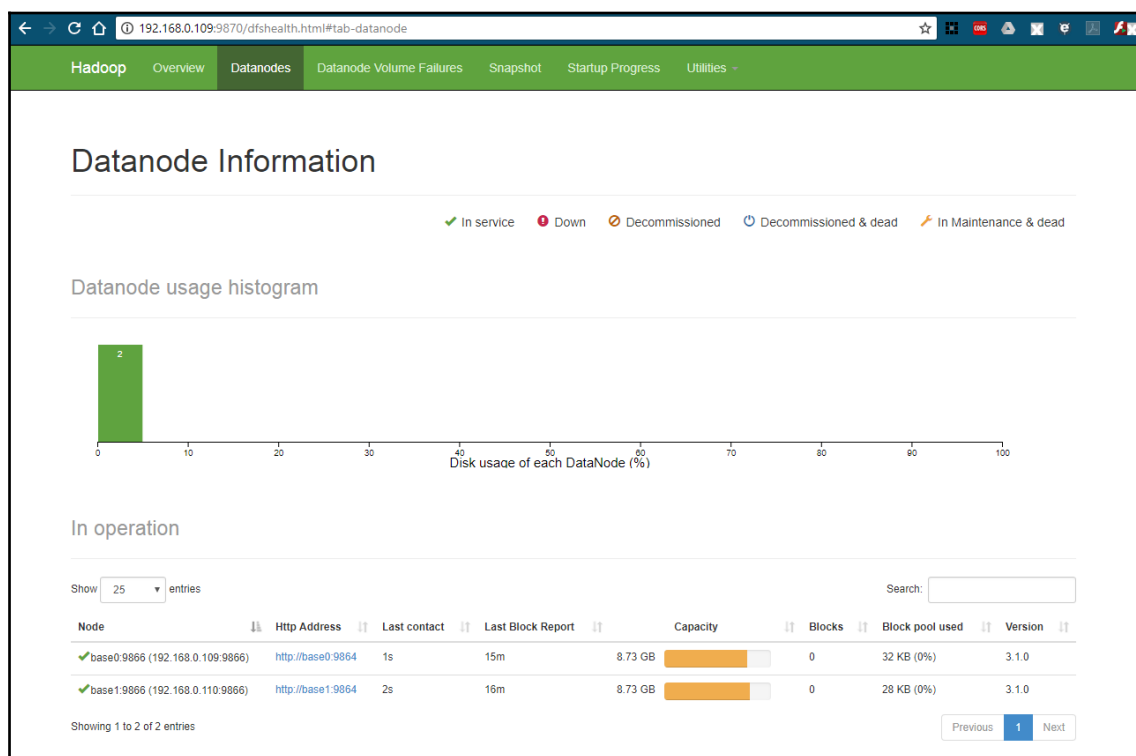
```
hadoop@base0:/$ bin/hdfs namenode -format
```

Once formatted, you can start HDFS by running the following command from any Hadoop directory:

```
hadoop@base0:/$ ./sbin/start-dfs.sh
```

Now, access the NameNode UI at `http://<master-hostname>:9870/`.

You should see an overview similar to that in the following screenshot. If you go to the **Datanodes** tab, you should see all DataNodes in the active stage:



Setting up YARN in cluster mode

YARN (Yet Another Resource Negotiator) provides a cluster-wide dynamic computing platform for different Hadoop subsystem components such as Apache Spark and MapReduce. YARN applications can be written in any language, and can now utilize the capabilities of cluster and HDFS storage without any MapReduce programming. YARN can be set up in a single node or a cluster node. We will set up YARN in a cluster node.

First, we need to inform Hadoop that the cluster will be using YARN instead of the MapReduce framework for processing; this can be done by editing `etc/hadoop/mapred-site.xml`, and adding the following entry to it:

```
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

Another configuration that is required goes in `etc/hadoop/yarn-site.xml`. Here, you can simply provide the host name for YARN's resource manager. The property `yarn.nodemanager.aux-services` tells the node manager that a MapReduce container will have to shuffle the map tasks to the reduce tasks with the following code:

```
<configuration>
<!-- Site specific YARN configuration properties -->
<property>
<name>yarn.resourcemanager.hostname</name>
<value>base0</value>
</property>
<property>
<name>yarn.nodemanager.aux-services </name>
<value> mapreduce.shuffle</value>
</property>
</configuration>
```

Alternatively, you can also provide specific resource manager properties instead of just a host name; they are as follows:

- `yarn.resourcemanager.address`: This is a Resource Manager host:port for clients to submit jobs
- `yarn.resourcemanager.scheduler.address`: This is a Resource Manager host:port for ApplicationMasters to talk to Scheduler to obtain resources
- `yarn.resourcemanager.resource-tracker.address`: This is a Resource Manager host:port for NodeManagers
- `yarn.resourcemanager.admin.address`: This is a Resource Manager host:port for administrative commands
- `yarn.resourcemanager.webapp.address`: This is a Resource Manager for the web-UI address

You can look at more specific configuration properties at Apache's website [here](#).

This completes the minimal configuration needed to run your YARN on a Hadoop cluster. Now, simply start the YARN daemons with the following command:

```
hadoop@base0: /$ ./sbin/start-yarn.sh
```

Access the Hadoop resource manager's user interface at `http://<resource-manager-host>:8088`; you should see something similar to the following screenshot:

The screenshot shows the Hadoop Resource Manager web interface at 192.168.0.109:8088/cluster. The page title is "All Applications". On the left is a navigation menu with options: Cluster, About, Nodes, Node Labels, Applications, NEW, NEW SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED, Scheduler, and Tools. The main content area displays "Cluster Metrics" and "Cluster Nodes Metrics".

Cluster Metrics									
Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total
0	0	0	0	0	0 B	16 GB	0 B	0	16

Cluster Nodes Metrics							
Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes		
2	0	0	0	0	0	2	

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation	Maximum Cluster App
Capacity Scheduler	[memory-mb (unit=Mi), vcores]	<memory: 1024, vCores: 1>	<memory: 8192, vCores: 4>	0

Show: 20 entries

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU Vcores	Allocated Memory MB	Reserved CPU Vcores	Reserved Memory MB	% of Queue	% of Cluster	Progress
No data available in table																	

Showing 0 to 0 of 0 entries

You can now browse through the **Nodes** section to see the available nodes for computation in the YARN engine, shown as follows:

Nodes of the cluster

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	V-Cores Used	V-Cores Total	V-Cores Reserved
0	0	0	0	0	0 B	16 GB	0 B	0	16	0

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes	Shutdown Nodes
2	0	0	0	0	0	2

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation	Maximum Cluster Application Priority
Capacity Scheduler	[memory-mb (unit=M), vcores]	<memory:1024, vCores:1>	<memory:8192, vCores:4>	0

Show 20 entries

Node Labels	Rack	Node State	Node Address	Node HTTP Address	Last health-update	Health-report	Containers	Allocation Tags	Mem Used	Mem Avail	V-Cores Used	V-Cores Avail	Version
/default-rack		RUNNING	base0.40415	base0.8042	Sun Jun 24 16:18:08 +0530 2018		0		0 B	8 GB	0	8	3.1.0
/default-rack		RUNNING	base1.45417	base1.8042	Sun Jun 24 16:17:36 +0530 2018		0		0 B	8 GB	0	8	3.1.0

Showing 1 to 2 of 2 entries

Now try to run an example from the `hadoop-example` list (or the one we prepared for a pseudo cluster). You can run it in the same way you ran it in the previous section, which is as follows:

```
hadoop@base0:/$ <hadoop-home>/bin/hadoop jar <location-of generated-jar>
ExpressionFinder "\\s+" <folder-containing-files-for input> <new-output-
folder> > stdout.txt
```

You can now look at the state of your program on the resource manager, as shown in the following screenshot:

All Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved
1	0	1	0	1	2 GB	16 GB	0 B

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes
2	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation
Capacity Scheduler	[memory-mb (unit=M), vcores]	<memory:1024, vCores:1>	<memory:8192, vCores:4>

Show 20 entries

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU V-Cores	Allocated Memory MB	Ret
application_1523836638830_0001	hadoop	expression finder	MAPREDUCE	default	0	Sun Jun 24 16:32:58 +0530 2018	N/A	ACCEPTED	UNDEFINED	1	1	2048	0

Showing 1 to 1 of 1 entries

As you can see, by clicking on a job, you get access to log files to see specific progress.

In addition to YARN, you can also set up a YARN history server to keep track of all the historical jobs that were run on a cluster. To do so, use the following command:

```
hadoop@base0:/$ ./bin/mapred --daemon start historyserver
```

The job history server runs on port 19888. Congratulations! You have now successfully set up your first Hadoop cluster.

Diagnosing the Hadoop cluster

As you get into deeper configuration and analysis, you will start facing new issues as you progress. This might include exceptions coming from programs, failing nodes, or even random errors. In this section, we will try to cover how they can be identified and addressed. Note that we will look at debugging MapReduce programs in [Chapter 4](#), *Developing MapReduce Applications*; this section is more focused on debugging issues pertaining to the Hadoop cluster.

Working with log files

Logging into Hadoop uses the rolling file mechanism based on First In, First Out. There are different types of log files intended for developers, administrators, and other users. You can find out the location of these log files through `log4j.properties`, which is accessible at `$HADOOP_HOME/etc/hadoop/log4j.properties`. The default log files cannot exceed 256 MB, but they can be changed in the relevant properties file. You can change the logging level in this file from `DEBUG` to `INFO`. Let's have a quick look at the different types of log files.

Job log files: The YARN UI provides details of a task whether it is successful or has failed. When you run the job, you see its status, such as failed or successful, on the resource manager UI once your job has finished. This provides a link to a log file, which you can then open and look at for a specific job. These files will be typically used by developers to diagnose the reason for job failures. Alternatively, you can also use CLIs to see the log details for a deployed job; you can look at job logs using `mapred log`, as follows:

```
hadoop@base0:/$ mapred job -logs [job_id]
```

Similarly, you can track YARN application logs with the following CLI:

```
hadoop@base0:/$ yarn logs -applicationId <application-id>
```

Daemon log files: When you run daemons of node manager, resource manager, data node, name node, and so on, you can also diagnose issues through the log files generated for those daemons. If you have access to the cluster and node, you can go to the `HADOOP_HOME` directory of the node that is failing and check the specific log files in the `logs/` folder of `HADOOP_HOME`. There are two types of files: `.log` and `.out`. The `.out` extension represents the console output of daemons, whereas log files log the outcome of these processes. The log files have the following format:

```
hadoop-<os-user-running-hadoop>-<instance>-datetime.log
```

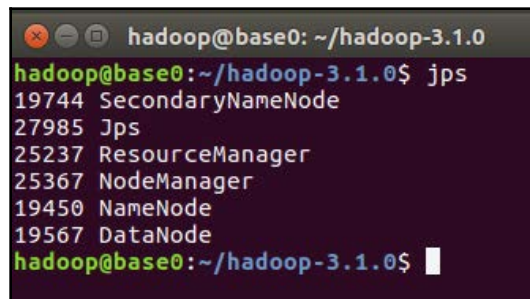
Cluster debugging and tuning tools

To analyze issues in a running cluster, you often need faster mechanisms to perform root cause analysis. In this section, we will look at a few tools that can be used by developers and administrators to debug the cluster.

JPS (Java Virtual Machine Process Status)

When you run Hadoop on any machine, you can look at the specific processes of Hadoop through one of the utilities provided by Java called the **JPS (Java Virtual Machine Process Status)** tool.

Running JPS from the command line will provide the process ID and the process name of any given JVM process, as shown in the following screenshot:



```
hadoop@base0: ~/hadoop-3.1.0
hadoop@base0:~/hadoop-3.1.0$ jps
19744 SecondaryNameNode
27985 Jps
25237 ResourceManager
25367 NodeManager
19450 NameNode
19567 DataNode
hadoop@base0:~/hadoop-3.1.0$
```

JStack

JStack is a Java tool that prints a stack trace for a given process. This tool can be used along with JPS. JStack provides insight into multiple thread dumps out of the Java process to help developers understand detail status and thread information aside from log outputs. To run JStack, you need to know the process number. Once you know it, you can simply call the following:

```
hadoop@base0:/$ jstack <pid>
```

Note that option **-F** in particular can be used for Java processes that are not responding to requests. This option will make your life a lot easier.

Summary

In this chapter, we covered the installation and setup of Apache Hadoop. We started with the prerequisites for setting up a Hadoop cluster. We also went through different Hadoop configurations available for users, covering the development mode, pseudo distributed single nodes, and the cluster setup. We learned how each of these configurations can be set up, and we also ran an example application on the configurations. Finally, we covered how one can diagnose the Hadoop cluster by understanding the log files and different debugging tools available. In the next chapter, we will start looking at the Hadoop Distributed File System in detail.

3

Deep Dive into the Hadoop Distributed File System

In the previous chapter, we saw how you can set up a Hadoop cluster in different modes, including standalone mode, pseudo-distributed cluster mode, and full cluster mode. We also covered some aspects on debugging clusters. In this chapter, we will do a deep dive into Hadoop's Distributed File System. The Apache Hadoop release comes with its own HDFS (Hadoop Distributed File System). However, Hadoop also supports other filesystems such as Local FS, WebHDFS, and Amazon S3 file system. The complete list of supported filesystems can be seen here (<https://wiki.apache.org/hadoop/HDFS>).

In this section, we will primarily focus on HDFS, and we will cover the following aspects of Hadoop's filesystems:

- How HDFS works
- Key features of HDFS
- Data flow patterns of HDFS
- Configuration for HDFS
- Filesystem CLIs
- Working with data structures in HDFS

Technical requirements

You will need Eclipse development environment and Java 8 installed on your system where you can run/tweak these examples. If you prefer to use maven, then you will need maven installed to compile the code. To run the example, you also need Apache Hadoop 3.1 setup on Linux system. Finally, to use the Git repository of this book, you need to install Git.

The code files of this chapter can be found on GitHub:

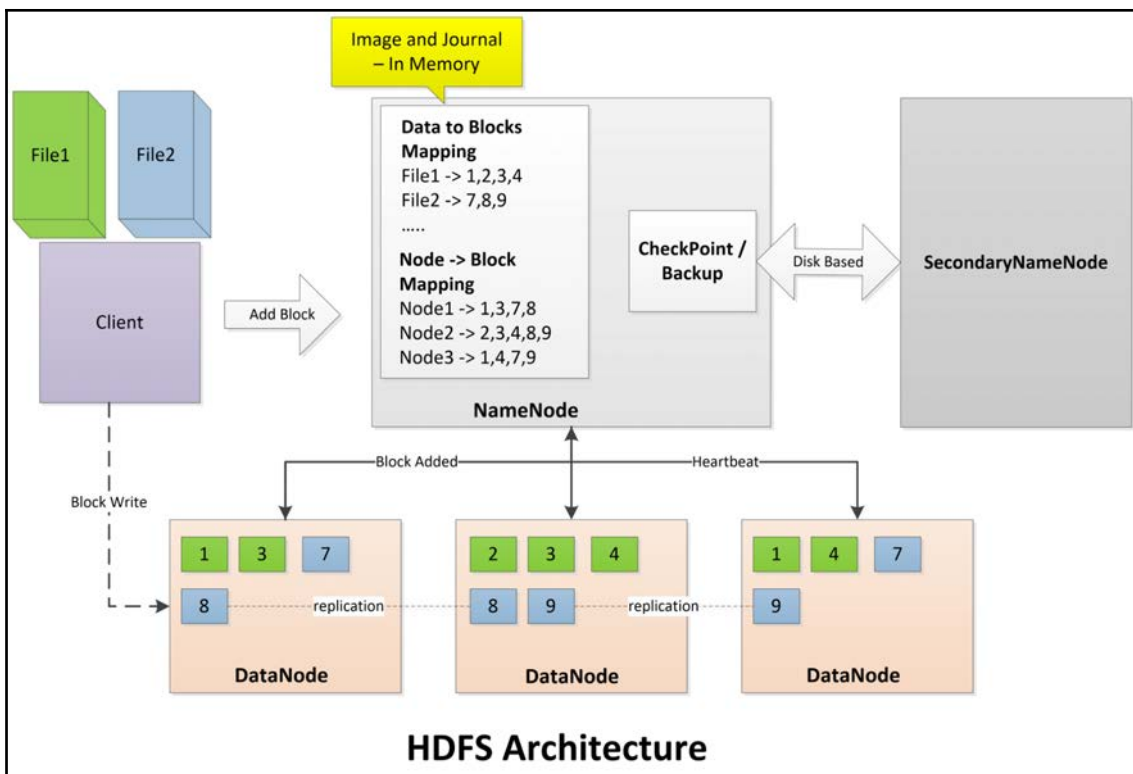
<https://github.com/PacktPublishing/Apache-Hadoop-3-Quick-Start-Guide/tree/master/Chapter3>

Check out the following video to see the code in action:

<http://bit.ly/2Jq5b8N>

How HDFS works

When we set up a Hadoop cluster, Hadoop creates a virtual layer on top of your local filesystem (such as a Windows- or Linux-based filesystem). As you might have noticed, HDFS does not map to any physical filesystem on operating system, but Hadoop offers abstraction on top of your Local FS to provide a fault-tolerant distributed filesystem service with HDFS. The overall design and access pattern in HDFS is like a Linux-based filesystem. The following diagram shows the high-level architecture of HDFS:



We have covered **NameNode**, **Secondary Name**, and **DataNode** in Chapter 1, *Hadoop 3.0 - Background and Introduction*. Each file sent to HDFS is sliced into a number of blocks that need to be distributed. The NameNode maintains the registry (or name table) of all of the nodes present in the data in the local filesystem path specified with `dfs.namenode.name.dir` in `hdfs-site.xml`, whereas the Secondary NameNode replicates this information through checkpoints. You can have many Secondary NameNodes. Typically the NameNode would store information pertaining to directory structure, permissions, mapping of files to block, and so forth.

This filesystem is persisted in two formats: FSImage and Editlogs. FSImage is a snapshot of a namenode's filesystem metadata at a given point, whereas Editlogs record all of the changes from the last snapshot that is stored in FSImage. FSImage is a data structure made efficient for reading, so HDFS captures the changes to the namespace in Editlogs to ensure durability. Hadoop provides an offline image viewer to dump FSImage data into human-readable format.

Key features of HDFS

In this section, we will go over some of the marquee features of HDFS that offer advantages for Hadoop users. We have already covered some of the features of HDFS in Chapter 1, *Hadoop 3.0 - Background and Introduction*, such as erasure coding and high availability, so we will not be covering them switch.

Achieving multi tenancy in HDFS

HDFS supports multi tenancy through its Linux-like **Access Control Lists (ACLs)** on its filesystem. The filesystem-specific commands are covered in the next section. When you are working across multiple tenants, it boils down to controlling access for different users through the HDFS command-line interface. So, the HDFS Administrator can add tenant spaces to HDFS through its namespace (or directory), for example, `hdfs://<host>:<port>/tenant/<tenant-id>`. The default namespace parameter can be specified in `hdfs-site.xml`, as described in the next section.

It is important to note that HDFS uses local filesystem's users and groups for its own, and it does not govern or validate whether the created group exists or not. Typically, for each tenant, one group can be created, and users who are part of that group can get access to all of the artifacts of that group. Alternatively, the user identity of a client process can happen through a Kerberos principal. Similarly, HDFS supports attaching LDAP servers for the groups. With local filesystem, it can be achieved with the following steps:

1. Create a group for each tenant, and add users to this group in local FS
2. Create a new namespace for each tenant, for example, `/tenant/<tenant-id>`
3. Make the tenant the complete owner of that directory through the `chown` command
4. Set access permissions on `tenant-id` of a group for the tenant
5. Set up a quota for each tenant through `dfadmin -setSpaceQuota <Size> <path>` to control the size of files created by each tenant



HDFS does not provide any control over the creation of users and groups or the processing of user tokens. Its user identity management is handled externally by third-party systems.

Snapshots of HDFS

Creating snapshots in HDFS is a feature by which one can take a snapshot of the filesystem and preserve it. These snapshots can be used as data backup and provide DR in case of any data losses. Before you take a snapshot, you need to make the directory `snapshottable`.

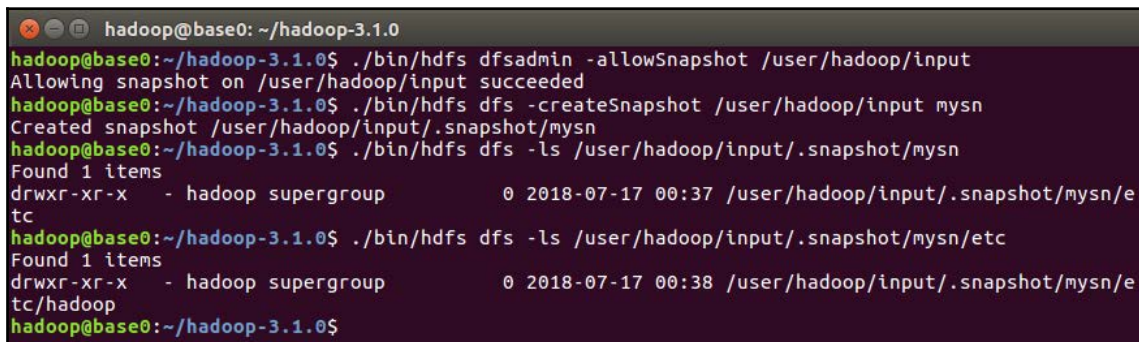
Use the following command:

```
hrishikesh@base0:/$ ./bin/hdfs dfsadmin -allowSnapshot <path>
```

Once this is run, you will get a message stating that it has succeeded. Now you are good to create a snapshot, so run the following command:

```
hrishikesh@base0:/$ ./bin/hdfs dfs -createSnapshot <path> <snapshot-name>
```

Once this is done, you will get a directory path to where this snapshot is taken. You can access the contents of your snapshot. The following screenshot shows how the overall snapshot runs:

A terminal window with a dark background and light-colored text. The prompt is 'hadoop@base0: ~/hadoop-3.1.0'. The user enters several commands: 1. './bin/hdfs dfsadmin -allowSnapshot /user/hadoop/input' which returns 'Allowing snapshot on /user/hadoop/input succeeded'. 2. './bin/hdfs dfs -createSnapshot /user/hadoop/input mysn' which returns 'Created snapshot /user/hadoop/input/.snapshot/mysn'. 3. './bin/hdfs dfs -ls /user/hadoop/input/.snapshot/mysn' which returns 'Found 1 items' followed by a long listing of the snapshot directory. 4. './bin/hdfs dfs -ls /user/hadoop/input/.snapshot/mysn/etc' which returns 'Found 1 items' followed by a long listing of the 'etc' subdirectory within the snapshot. The prompt returns to 'hadoop@base0:~/hadoop-3.1.0\$'.

You can access a full list of snapshot-related operations, such as renaming a snapshot and deleting a snapshot, here (<https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsSnapshots.html>).

Safe mode

When a NameNode starts, it looks for FSImage and loads it in memory, then it looks for past edit logs and applies them on FSImage, creating a new FSImage. After this process is complete, the NameNode starts service requests over HTTP and other protocols. Usually, DataNodes hold the information pertaining to the location of blocks; when a NameNode loads up, DataNodes provide this information to the NameNode. This is the time when the system runs in safe mode. Safe Mode is exited when the `dfs.replication.min` value for each block is met.

HDFS provides a command to check if a given filesystem is running in safe mode or not:

```
hrishikesh@base0:/$ ./bin/hadoop dfsadmin -safemode get
```

This should provide you the information of whether safe mode is on. In that case, the filesystem only provides read access to its repository. Similarly, the Administrator can choose to enter in safe mode with the following command:

```
hrishikesh@base0:/$ ./bin/hadoop dfsadmin -safemode enter
```

Similarly, the `safemode leave` option is provided.

Hot swapping

HDFS allows users to hot swap its DataNode in the live fashion. The associated Hadoop JIRA issue is listed here (<https://issues.apache.org/jira/browse/HDFS-664>). Please note that hot swapping has to be supported by the underlying hardware system. If this is not supported, you may have to restart the affected DataNode, after replacing its storage device. However, before Hadoop gets into replication mode, you would need to provide the new corrected DataNode volume storage. The new volume should be formatted and, once it's done, the user should update `dfs.datanode.data.dir` in the configuration. After this, the user should run the reconfiguration using the `dfsadmin` command as listed here:

```
hrishikesh@base0:/$ ./bin/hdfs dfsadmin -reconfig datanode HOST:PORT start
```

Once this activity is complete, the user can take out the problematic data storage from the datanode.

Federation

HDFS provides federation capabilities for its various users. This also adds up in multi tenancy. Previously, each deployment cluster of HDFS used to work with a single namespace, thereby limiting horizontal scalability. With HDFS Federation, the Hadoop cluster can scale horizontally.

A block pool represents a single namespace containing a group of blocks. Each NameNode in the cluster is directly correlated to one block pool. Since DataNodes are agnostic to namespaces, the responsibility of managing blocks pertaining to any namespace stays with the NameNode. Even if the NameNode for any federated tenant goes down, the remaining NameNodes and DataNodes can function without any failures. The document here (<https://hadoop.apache.org/docs/r3.1.0/hadoop-project-dist/hadoop-hdfs/Federation.html>) covers the configuration for HDFS Federation.

Intra-DataNode balancer

The need for a DataNode balancer arose for various reasons. The first is because, when a disk is replaced, the DataNodes need to be re-balanced based on available space. Secondly, with default round-robin scheduling available in Hadoop, mass file deletion from certain DataNodes leads to unbalanced DataNode storage. This was raised as JIRA issue HDFS-1312 (<https://issues.apache.org/jira/browse/HDFS-1312>), and it was fixed in Hadoop 3.0-alpha1. The new disk balancer supports reporting and balancing functions. The following table describes all available commands:

Command	Parameters	Description
diskbalancer	-plan <datanode>	This command allows the user to create a plan (before/after) for a given DataNode.
diskbalancer	-execute <plan.json>	The plan generated from -plan is passed to execute on the disk balancer.
diskbalancer	-query <datanode>	This gets the current status of the disk balancer.
diskbalancer	-cancel <plan.json>	This cancels a running plan.
diskbalancer	-fs <path> -report <params>	This command provides a report to a few candidates or the namespace URI.

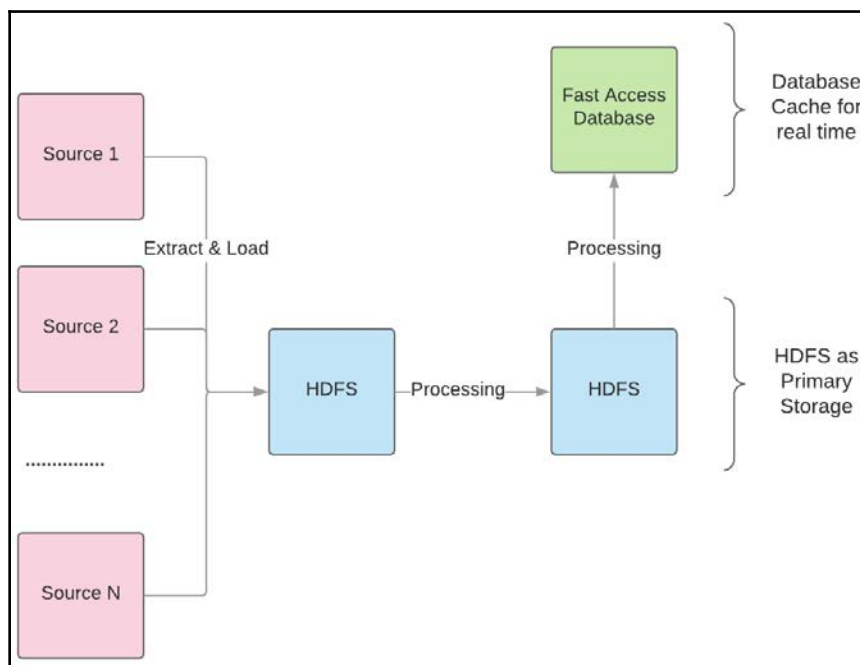
Today, the system supports round-robin-based disk balancing and free space, the percentage of which is based on load distribution scheduling algorithms.

Data flow patterns of HDFS

In this section, we will look at the different types of data flow patterns in HDFS. HDFS serves as storage for all processed data. The data may arrive with different velocity and variety; it may require extensive processing before it is ready for consumption by an application. Apache Hadoop provides frameworks such as MapReduce and YARN to process the data. We will be covering the data variety and velocity aspect in a later part of this chapter. Let's look at the different data flow patterns that are possible with HDFS.

HDFS as primary storage with cache

HDFS can be used as a primary data storage. In fact, in many implementations of Hadoop, that has been the case. The data is usually supplied by many source systems, which may include social media information, application log data, or data coming from various sensors. The following data flow diagram depicts the overall pattern:



This data is first extracted and stored in HDFS to ensure minimal data loss. Then, the data is picked up for transformation; this where the data is cleansed and transformed and information is extracted and stored in HDFS. This transformation can be multi-stage processing, and it may require intermediate HDFS storage. Once the data is ready, it can be moved to the consuming application through a cache, which can again be another traditional database.

Having a cache ensures that the application can provide a request-response-based communication, without any latency or wait. This is because HDFS response is slower compared to the traditional database and/or cache. So, only the information that is needed by the consuming application is moved periodically to the fast access database.

The **pros** of this pattern are as follows:

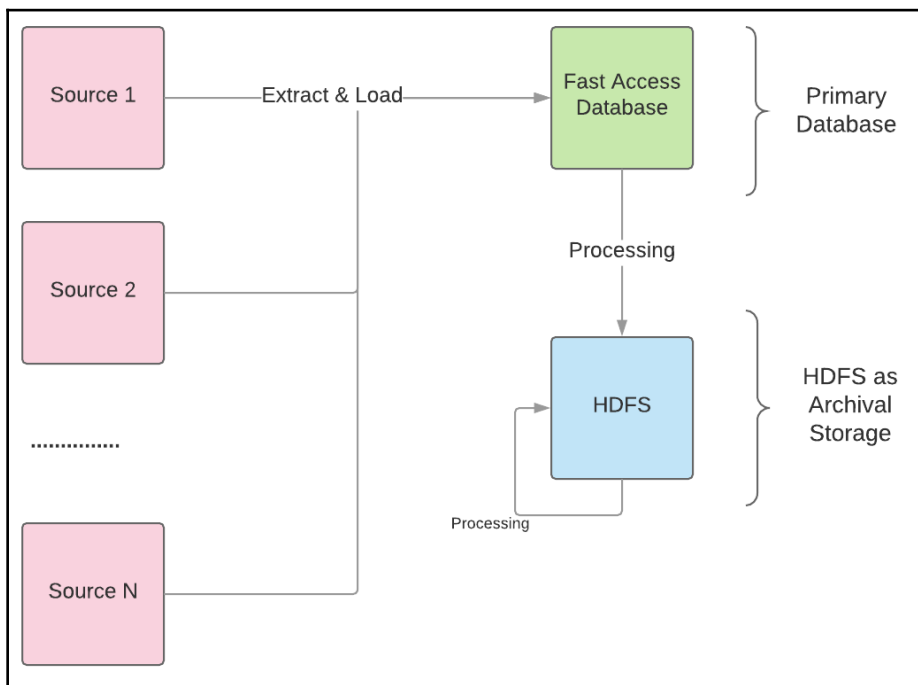
- It provides seamless data processing achieved using Hadoop
- Applications can work the way they do with traditional databases, as it supports request-response
- It's suitable for historical trend analysis, user behavioral pattern analysis, and so on

The **cons** of this pattern are as follows:

- Usually, there is a huge latency between the data being picked for processing and it reaching the consuming application
- It's not suitable for real-time or near-real-time processing

HDFS as archival storage

HDFS offers unlimited storage with scalability, so it can be used as an archival storage system. The following **Data Flow Diagram (DFD)** depicts the pattern of HDFS as an archive store:



All of the sources supply data in real time to the **Primary Database**, which provides faster access. This data, once it is stored and utilized, is periodically moved to archival storage in HDFS for data recovery and change logging. HDFS can also process this data and provide analytics over time, whereas the primary database continues to serve the requests that demand real time data.

The pros of this pattern are as follows:

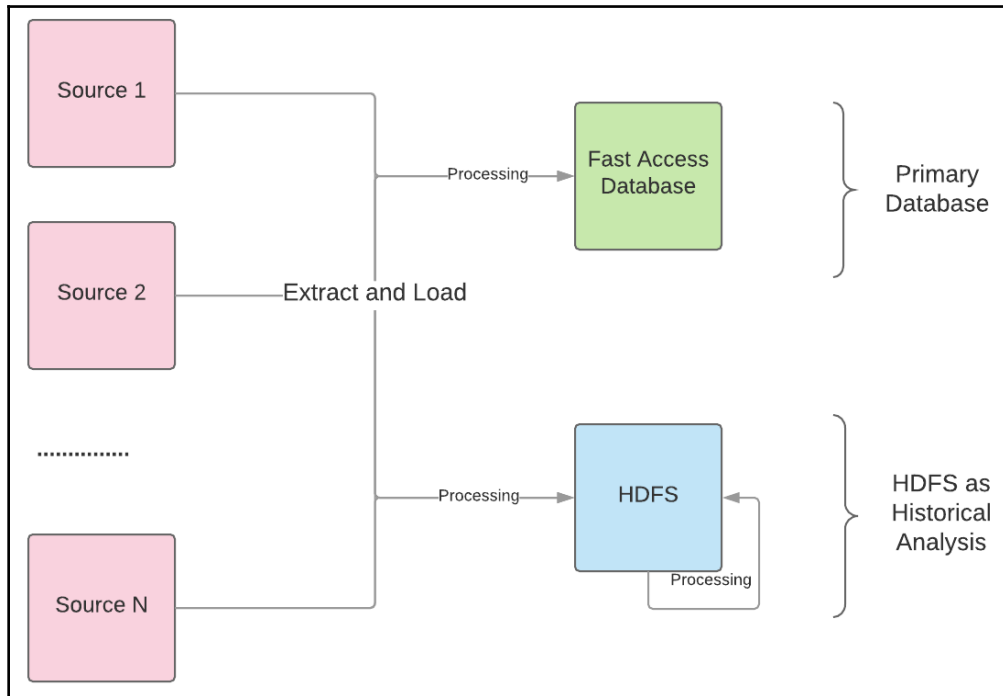
- It's suitable for real-time and near-real-time streaming data and processing
- It can also be used for event-based processing
- It may support microbatches

The **cons** of this pattern are as follows:

- It cannot be used for large data processing or batch processing that requires huge storage and processing capabilities

HDFS as historical storage

Many times, when data is retrieved, processed, and stored in a high-speed database, the same data is periodically passed to HDFS for historical storage in batch mode. The following new DFD provides a different way of storing the data directly with HDFS instead of using the two-stage processing that is typically seen:



The data from multiple sources is processed in the processing pipeline, which then sinks the data to two different storage systems: the primary database, to provide real-time data access rapidly, and HDFS, to provide historical data analysis across large data over time. This model provides a way to pass only limited parts of processed data (for example, key attributes of social media tweets, such as tweet name and author), whereas the complete data (in this example, tweets, account details, URL links, metadata, retweet count, and other information about the post) can be persisted in HDFS.

The **pros** of of this pattern this are as follows:

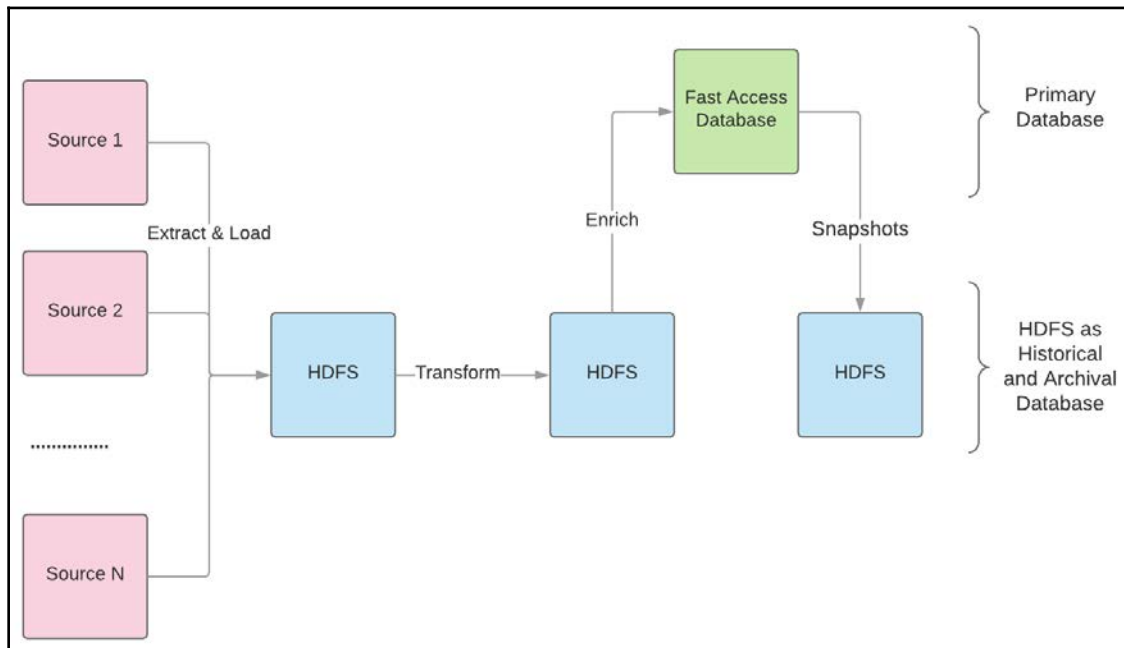
- The processing is single-staged, rather than two-staged
- It provides real-time storage on HDFS, which means there is no or minimal data latency
- It ensures that the primary database storage (such as in-memory) is efficiently utilized

The cons of this pattern are as follows:

- For large data, the process pipeline requires MapReduce-like processing, which may impact the performance and make it difficult for real time
- As the write latency in HDFS is higher than most of the in-memory/disk-based primary database, it may impact data processing performance

HDFS as a backbone

This data flow pattern provides the best utilization of a combination of the various patterns we have just seen. The following DFD shows the overall flow:



HDFS, in this case, can be used in multiple roles: it can be used as historical analytics storage, as well as archival storage for your application. The sources are processed with multi-stage pipelines with HDFS as intermediate storage for large data. Once the information is processed, only the content that is needed for application consumption is passed to the primary database for faster access, whereas the rest of the information is made accessible through HDFS. Additionally, the snapshots of enriched data, which was passed to the primary database, can also be archived back to HDFS in a separate namespace. This pattern is primarily useful for applications, such as warehousing, which need large data processing as well as data archiving.

The **pros** of this pattern are as follows:

- Utilization of HDFS for different purposes
- It's suitable for batch data, ETL data, and large data processing

The **cons** of this pattern are as follows:

- Lots of data processing in different stages can bring extensive latency between the data received from sources and its visibility through the primary database

HDFS configuration files

Unlike lots of software, Apache Hadoop provides few configuration files that give you flexibility when configuring your Hadoop cluster. Among them are two primary files that influence the overall functioning of HDFS:

- **core-site.xml**: This file is primarily used to configure Hadoop IOs; primarily, all of the common settings of HDFS and MapReduce would go here.
- **hdfs-site.xml**: This file is the main file for all HDFS configuration. Anything pertaining to NameNode, SecondaryNameNode, or DataNode can be found here.

The `core-site` file has more than 315 parameters that can be set. We will look at different configurations in the administration section. The full list can be seen here (<https://hadoop.apache.org/docs/r3.1.0/hadoop-project-dist/hadoop-common/core-default.xml>). We will cover some important parameters that you may need for configuration:

Property Name	Default Value	Description
<code>hadoop.tmp.dir</code>	<code>/tmp/hadoop-\${user.name}</code>	This is a temporary file location base for all Hadoop-related activities.

hadoop.security.authentication	simple	Choose between no authentication (simple) or Kerberos authentication
io.file.buffer.size	4096	The default size of the Hadoop IO buffer used for a sequence file. The default will be 4 KB.
file.blocksize	67108864	The block size for each file.
file.replication	1	The replication factor for each file.
fs.defaultFS	hdfs://localhost:9000	The URL of the default filesystem, in the form of <code>pdfs://host:port</code>

Similarly, HDFS Site offers 470+ different properties that can be set up in the configuration file. Please look at the default values of all the configuration here (<https://hadoop.apache.org/docs/r3.1.0/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml>). Let's go through the important properties in this case:

Property Name	Default Value	Description
dfs.namenode.secondary.http-address	0.0.0.0:9868	The secondary namenode HTTP server address and port.
dfs.namenode.secondary.https-address	0.0.0.0:9869	The secondary namenode HTTPS server address and port.
dfs.datanode.address	0.0.0.0:9866	The datanode server address and port for data transfer.
dfs.namenode.http-address	0.0.0.0:9870	The address and the base port where the dfs namenode web UI will listen on.
dfs.http.policy	HTTP_ONLY	HTTP_ONLY, HTTPS_ONLY, and HTTP_AND_HTTPS.
dfs.namenode.name.dir	file://\${hadoop.tmp.dir}/dfs/name	Comma-separated list of the directory to store the name table. The table is replicated across the list for redundancy management.
dfs.replication	3	Default replication factor for each file block.

Hadoop filesystem CLIs

Hadoop provides a command-line shell for its filesystem, which could be HDFS or any other filesystem supported by Hadoop. There are different ways through which the commands can be called:

```
hrishikesh@base0:/$ hadoop fs -<command> <parameter>
```

```
hrishikesh@base0:/$ hadoop dfs -<command> <parameter>
```

```
hrishikesh@base0:/$ hdfs dfs -<command> <parameter>
```

Although all commands can be used on HDFS, the first command listed is for Hadoop FS, which can be either HDFS or any other filesystem used by Hadoop. The second and third commands are specific to HDFS; however, the second command is deprecated, and it is replaced by the third command. Most filesystem commands are inspired by Linux shell commands, except for minor differences in syntax. The HDFS CLI follows a POSIX-like filesystem interface.

Working with HDFS user commands

HDFS provides a command-line interface for users as well as administrators. They can perform different actions pertaining to the filesystem or to play with clusters.

Administrative commands are covered in [Chapter 6, Monitoring and Administration of a Hadoop Cluster](#), targeted for administration. In this section, we will go over HDFS user commands:

Command	Parameters	Description	Important Parameters
classpath	--jar <file>	Prints the classpath for Hadoop as a JAR file.	
dfs	<command> <params>	Runs filesystem commands. Please refer to the next section for specific commands.	
envvars		Displays Hadoop environment variables.	

fetchdt	<token-file>	Fetches the delegation token needed to connect a secure server from a non-secure client.	
fsck	<path> <params>	Just like the Linux system, this is a filesystem check utility.	Use -list-corruptfileblocks to list corrupt blocks.
getconf	-<param>	Gets configuration information based on the parameter.	Use -namenode to get Namenode-related configuration.
groups	<username>	Provides group information for the given user.	
httpfs		Runs a HTTP server for HDFS.	
lsSnapshottableDir		Provides a list of user directories that are "snapshottable" for a given user. If a user is super-user , it provides all directories.	
jmxget	<params>	Gets JMX-related information from a service. You can supply additional information such as URL and connection information.	Use -service <servicename> .

oew	<code><params> -I <input-file> -o <output-file></code>	Parses a Hadoop Editlog file and saves it. Covered in the <i>Monitoring and administration</i> section.	
oiv	<code><params> -I <input-file> -o <output-file></code>	Dumps the content of HDFS FSImage to readable format and provides the WebHDFS API.	
oiv_legacy	<code><params> -I <input-file> -o <output-file></code>	This is the same as <code>ioiv</code> but for older versions of Hadoop.	
version		Prints the version of the current HDFS.	

Working with Hadoop shell commands

We have seen HDFS-specific commands in the previous section. Now, let's go over all of the filesystem-specific commands. These can be called with `hadoop fs <command>` or `hdfs dfs <command>` directly. Hadoop provides a generic shell command that can be used across different filesystems. The following table describes the list of commands, the different parameters that need to be passed, and their description. I have also covered important parameters in a day-to-day context that you would need. Apache also provides FS shell command guide (<https://hadoop.apache.org/docs/r3.1.0/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html>), where you can see more specific details with examples:

Command	Parameters	Description	Important Parameters
appendToFile	<code><localsrc> ... <hdfs-file-path></code>	Appends the local source file (or files) to the given HDFS file path.	
cat	<code><hdfs-file-path></code>	Reads the file and prints its content on the screen.	
checksum	<code><hdfs-file-path></code>	Returns the checksum of the file.	

chgrp	<param> Group <hdfs-filpath>	Allows the user to change the group association of a given file or path. Of course, the given user should be the owner of these files.	Use -R for recursive alternative.
chmod	<param> <Mode> <hdfs-filpath>	Allows the user to change the permission of a given file or path. Of course, the given user should be the owner of these files.	Use -R for recursive alternative.
chown	<param> <Owner>:<Group> <hdfs-filpath>	Allows users to change the owner, as well as group for a given HDFS file path.	Use -R for recursive alternative.
copyFromLocal/put	<param> <local-files> <hdfs-path>	Copies file from the local source to the given HDFS destination.	-p to preserve date time and -f for overwrite.
copyToLocal/get	<param> <hdfs-path> <local-file>	Copies file from HDFS to the local target.	
count	<param> <hdfs-path>	The count gets the count of number of directories and files in the given folder path(s).	
cp	<params> <source> <destination>	Copies file from source to destination. In this case, the source can be any source including the HDFS data path.	-p to preserve date time and -f for overwrite.
df	<param> <hdfs-paths>	Displays the available space.	Use -h for better readability.
du	<param> <hdfs-paths>	Displays the file size or length in the given path.	Use -s for summary and -h for better readability.
expunge		Removes the files in the checkpoint that are older than the retention threshold.	
find	<hdfs-path> <expression>	Just like Unix find, it finds all of the files in the given path that match the expression.	
getfacl	<param> <hdfs-path>	Displays the Access Control List for a given path.	Use -R for recursive alternative.
getfattr	<param> <hdfs-path>	Displays extended attribute names and values for a given path.	Use -R for recursive alternative.
getmerge	<param> <localsrc> <hdfs-file-path>	Merges all of the sources file from the local filesystem in the given HDFS file path.	-nl to put newline between two files and -skip-empty-file to skip empty files.

head	<hdfs-file-path>	Displays the first few characters of files.	
help		Provides help text.	
ls	<param> <hdfs-path>	Lists the content of a given path—the file and directories.	Use -R for recursive alternative.
lsr	<param> <hdfs-path>	Recursive display of the given path.	
mkdir	<param> <hdfs-path>	Creates an HDFS directory. Usually, the last path name is the one that is created.	Use -p to create the full path—even the parents.
moveFromLocal	<param> <local-file> <hdfs-path>	Similar to <code>copyFromLocal</code> but, post-movement, the original local copy is deleted.	-p to preserve date time and -f for overwrite.
mv	<param> <src-file-paths> <dest-file-path>	Moves files from multiple sources to one destination in one filesystem.	
rm	<param> <hdfs-paths>	Deletes files listed in the path; you may use wildcards.	Use -R or -r for recursive, -f to force it, and -skipTrash to not store it in trash.
rmdir	<param> <hdfs-paths>	Deletes the directory; you may use wildcards.	Use -ignore-fail-non-empty for not deleting directories that are not empty.
rmr	<param> <hdfs-paths>	Delete recursively.	-skipTrash to not store it in trash.
setfacl	<param> <acl> <hdfs-paths>	Sets ACLs for a given directory/regular expression. Typically the ACL specification is <user>:<group>:<ACL> . <ACL> is rwx.	--set to fully replace and -R for recursive alternative.
setfattr	-n <name> (-v <value>) <hdfs-path>	Set an extended attribute for a given file or directory.	-x <name> to remove the extended attribute.
setrep	<replica-count> <hdfs-path>	Allows users to change replication factor for a file.	Use -w to wait for the replica to complete.
stat	<format> <hdfs-path>	Provides statistics about the given file/directory as per the format listed.	
tail	<param> <hdfs-file-path>	Displays the last KB of a given file.	-f provides continuous additions to a given file in loop.

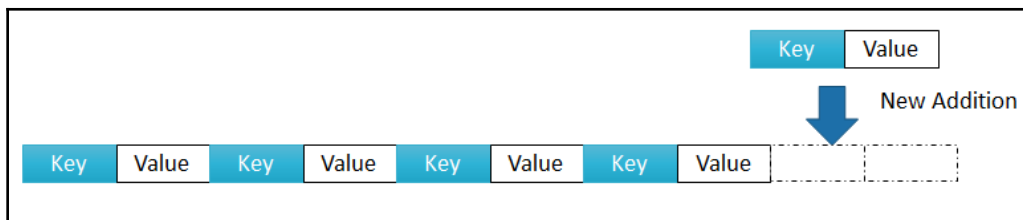
test	<param> <hdfs-path>	Checks whether the given directory or file exists or not. Returns 0 if successful.	Use -d to check if it's a directory and -f to check if it's a file.
text	<hdfs-file-path>	Prints the given file in text format.	
touchz	<hdfs-file-path>	Similar to Linux touch. Creates a file of zero characters.	
truncate	<param> <number> <hdfs-file-path>	Truncates all files that match the specified file pattern to the specified length.	Use -w to wait for the replica to complete.
usage	<command-name>	Provides help text for a given command.	

Working with data structures in HDFS

When you work with Apache Hadoop, one of the key design decisions that you take is to identify the most appropriate data structures for storing your data in HDFS. In general, Apache Hadoop provides different data storage for any kind of data, which could be text data, image data, or any other binary data format. We will be looking at different data structures supported by HDFS, as well as other ecosystems, in this section.

Understanding SequenceFile

Hadoop `SequenceFile` is one of the most commonly used file formats for all HDFS storage. `SequenceFile` is a binary file format that persists all of the data that is passed to Hadoop in `<key, value>` pairs in a serialized form, depicted in the following diagram:



The `SequenceFile` format is primarily used by MapReduce as default input and output parameters. `SequenceFile` provides a single long file, which can accommodate multiple files together to create a single large Hadoop distributed file.

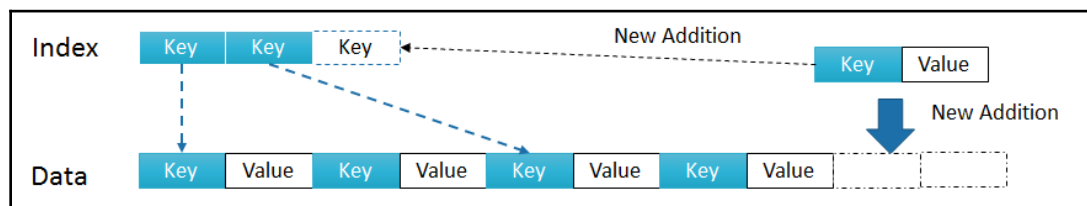
When the Hadoop cluster has to deal with multiple files that are of small nature (such as images, scanned PDF documents, tweets from social media, email data, or office documents), it cannot be imported as is, primarily due to efficiency challenges while storing these files. Given that the minimum HDFS block size is higher than that of most files, it results in fragmentation of storage.

The `SequenceFile` format can be used when multiple small files are to be loaded in HDFS combined. They can all go in one `SequenceFile` format. The `SequenceFile` class provides a reader, writer, and sorter to perform operations. `SequenceFile` supports the compression of values or keys and values together through compression codecs. The JavaDoc for `SequenceFile` can be accessed here (<https://hadoop.apache.org/docs/r3.1.0/api/index.html?org/apache/hadoop/io/SequenceFile.html>) for more details about compression. I have provided some examples of `SequenceFile` reading and writing in code repository, for practice. The following topics are covered:

- Creating a new `SequenceFile` class
- Displaying `SequenceFile`
- Sorting `SequenceFile`
- Merging `SequenceFile`

MapFile and its variants

While the `SequenceFile` class offers `<key, value>` to store any data elements, `MapFile` provides `<Key, Value>`, as well as an index file of keys. The index file is used for faster access to the keys of each Map. The following diagram shows the storage pattern of `MapFile`:



`SequenceFile` provides a sequential pattern for reading and writing data, as HDFS supports an append-only mechanism, whereas `MapFile` can provide random access capability. The index file contains the fractions of the keys; this is determined by the `MapFile.Writer.getIndexInterval()` method. The index file is loaded in memory for faster access. You can read more about `MapFile` in the Java API documentation here (<https://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/io/MapFile.html>).

`SetFile` and `ArrayFile` are extended from the `MapFile` class. `SetFile` stores the keys in the set and provides all set operations on its index, whereas `ArrayFile` stores all values in array format without keys. The documentation for `SetFile` can be accessed here (<https://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/io/SetFile.html>) and, for `ArrayFile`, here (<https://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/io/ArrayFile.html>).

`BloomMapFile` offers `MapFile`-like functionalities; however, the Map index is created with the help of the dynamic bloom filter. You may go through the bloom filter data structure here (<https://ieeexplore.ieee.org/document/4796196/>). The dynamic bloom filter provides an additional wrapper to test the membership of the key in the actual index file, thereby avoiding an unnecessary search of the index. This implementation provides a rapid `get()` call for sparsely populated index files. I have provided some examples of `MapFile` reading and writing in <https://github.com/PacktPublishing/Apache-Hadoop-3-Quick-Start-Guide/tree/master/Chapter3>; these cover the following:

- Reading from `MapFile`
- Writing to `MapFile`

Summary

In this chapter, we have took a deep dive into HDFS. We tried to figure out how HDFS works and its key features. We looked at different data flow patterns of HDFS, where we can see HDFS in different roles. This was supported with various configuration files of HDFS and key attributes. We also looked at various command-line interface commands for HDFS and the Hadoop shell. Finally, we looked at the data structures that are used by HDFS with some examples.

In the next chapter, we will study the creation of a new MapReduce application with Apache Hadoop MapReduce.

4

Developing MapReduce Applications

"Programs must be written for people to read, and only incidentally for machines to execute."

– Harold Abelson, Structure and Interpretation of Computer Programs, 1984

When Apache Hadoop was designed, it was intended for large-scale processing of humongous data, where traditional programming techniques could not be applied. This was at a time when MapReduce was considered a part of Apache Hadoop. Earlier, MapReduce was the only programming option available in Hadoop; however, with new Hadoop releases, it was enhanced with YARN. It's also called MRv2 and older MapReduce is usually referred to as MRv1. In the previous chapter, we saw how HDFS can be configured and used for various application usages. In this chapter, we will do a deep dive into MapReduce programming to learn the different facets of how you can effectively use MapReduce programming to solve various complex problems.

This chapter assumes that you are well-versed in Java programming, as most of the MapReduce programs are based on Java. I am using Hadoop version 3.1 with Java 8 for all examples and work.

We will cover the following topics:

- How MapReduce works
- Configuring a MapReduce environment
- Understanding Hadoop APIs and packages
- Setting up a MapReduce project

- Deep diving into MapReduce APIs
- Compiling and running MapReduce jobs
- Streaming in MapReduce programming

Technical requirements

You will need Eclipse development environment and Java 8 installed on your system where you can run/tweak these examples. If you prefer to use maven, then you will need maven installed to compile the code. To run the example, you also need Apache Hadoop 3.1 setup on Linux system. Finally, to use the Git repository of this book, you need to install Git.

The code files of this chapter can be found on GitHub:

<https://github.com/PacktPublishing/Apache-Hadoop-3-Quick-Start-Guide/tree/master/Chapter4>

Check out the following video to see the code in action:

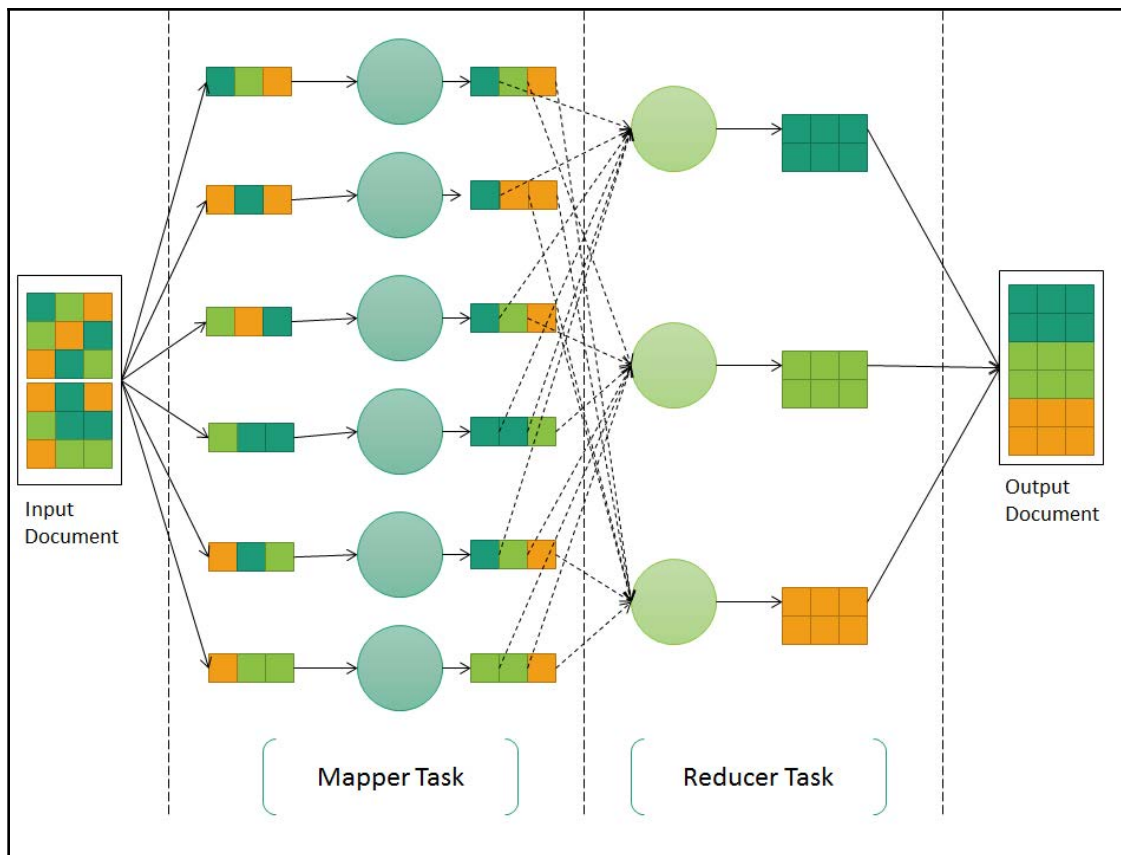
<http://bit.ly/2znViEb>

How MapReduce works

MapReduce is a programming methodology used for writing programs on Apache Hadoop. It allows the programs to run on a large scalable cluster of servers. MapReduce was inspired by functional programming (https://en.wikipedia.org/wiki/Functional_programming). **Functional Programming (FP)** offers amazing unique features when compared to today's popular programming paradigms such as object-oriented (Java and JavaScript), declarative (SQL and CSS), or procedural (C, PHP, and Python). You can look at a comparison between multiple programming paradigms [here](#). While we see a lot of interest in functional programming in academics, we rarely see equivalent enthusiasm from the developer community. Many developers and mentors claim that MapReduce is not actually a functional programming paradigm. Higher order functions in FP are functions that can take a function as a parameter or return a function (https://en.wikipedia.org/wiki/Higher-order_function). Map and Reduce are among the most widely used higher-order functions of functional programming. In this section, we will try to understand how MapReduce works in Hadoop.

What is MapReduce?

MapReduce programming provides a simpler framework to write complex processing on cluster applications. Although the programming model is simple, it is difficult to implement or convert any standard programs. Any job in MapReduce is seen as a combination of the `map` function and the `reduce` function. All of the activities are broken into these two phases. Each phase communicates with the other phase through standard input and output, comprising keys and their values. The following data flow diagram shows how MapReduce programming resolves different problems with its methodology. The color denotes similar entities, the circle denotes the processing units (either map or reduce), and the square boxes denote the data elements or data chunks:



In the Map phase, the map function collects data in the form of `<key, value>` pairs from HDFS and converts it into another set of `<key, value>` pairs, whereas in the Reduce phase, the `<key, value>` pair generated from the Map function is passed as input to the reduce function, which eventually produces another set of `<key, value>` pairs as output. This output gets stored in HDFS by default.

An example of MapReduce

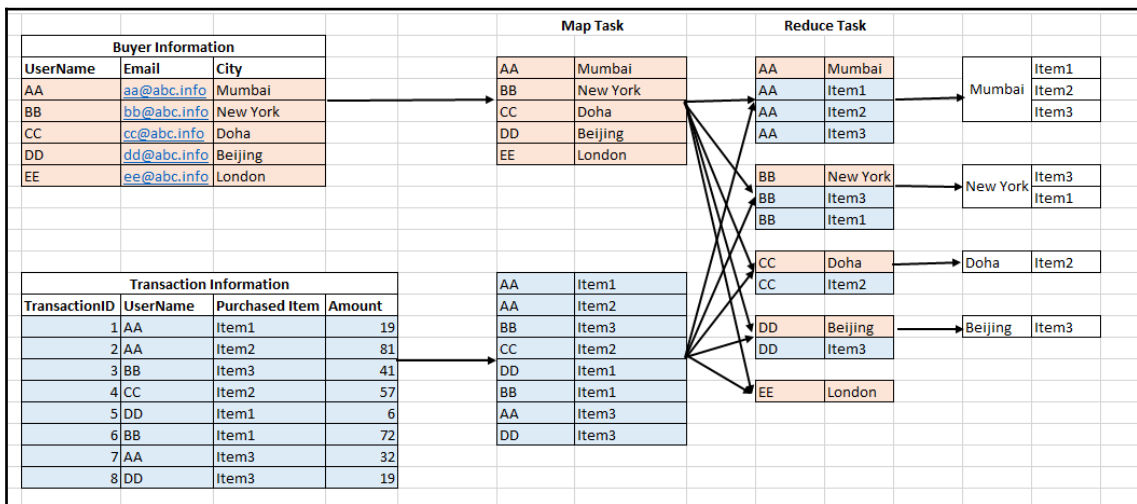
Let's understand the MapReduce concept with a simple example:

- **Problem:** There is an e-commerce company that offers different products for purchase through online sale. The task is to find out the items that are sold in each of the cities. The following is the available information:

Buyer Information			Transaction Information			
UserName	Email	City	TransactionID	UserName	Purchased Item	Amount
AA	aa@abc.info	Mumbai	1	AA	Item1	19
BB	bb@abc.info	New York	2	AA	Item2	81
CC	cc@abc.info	Doha	3	BB	Item3	41
DD	dd@abc.info	Beijing	4	CC	Item2	57
EE	ee@abc.info	London	5	DD	Item1	6
			6	BB	Item1	72
			7	AA	Item3	32
			8	DD	Item3	19

- **Solution:** As you can see, we need to perform the right outer join across these tables to get the city-wise item sale report. I am sure database experts who are reading this book can simply write a SQL query, to do this join using database. It works well in general. When we look at high-volume data processing, this can be alternatively performed using MapReduce and with massively parallel processing. The overall processing happens in two phases:
 - **Map phase:** In this phase, the Mapper job is relatively simple—it cleanses all of the input and creates key-value pairs for further processing. User will supply the information pertaining to user in `<key, value>` form for the **Map Task**. So, a Map Task will only pick relevant attributes in this case, which would matter for further processing, such as **UserName** and **City**.

- Reduce phase:** This is the second stage, where the processed `<key, value>` pair is reduced to a smaller set. The Reducer will receive information directly from Map Task. As you can see in the following screenshot, the reduce task performs the majority of operations; in this case, it reads the tuples and creates intermediate files process. Once the processing is complete, the output gets persisted in HDFS. In this activity, the actual merging takes place between multiple tuples based on `UserName` as a shared key. The Reducer produces a group of collated information per city, as follows:



Configuring a MapReduce environment

When you install the Hadoop environment, the default environment is set up with MapReduce. You do not need to make any major changes in configuration. However, if you wish to run MapReduce program in an environment that is already set up, please ensure that the following property is set to `local` or `classic` in `mapred-site.xml`:

```
<property>
  <name>mapreduce.framework.name</name>
  <value>local</value>
</property>
```

I have elaborated on this property in detail in the next section.

Working with mapred-site.xml

We have seen `core-site.xml` and `hdfs-site.xml` files in previous files. To configure MapReduce, primarily Hadoop provides `mapred-site.xml`. In addition to `mapred-site.xml`, Hadoop also provides a default read-only configuration for references called `mapred-default.xml`. The location of `mapred-site.xml` can be found in the `$HADOOP_HOME/etc/Hadoop` directory. Now, let's look at all of the other important parameters that are needed for MapReduce to run without any hurdles:

Property Name	Default Value	Description
<code>mapreduce.cluster.local.dir</code>	<code>\${Hadoop.tmp.dir}/mapred/local</code>	A local directory for keeping all MapReduce-related intermediate data. You need to ensure that you have sufficient space.
<code>mapreduce.framework.name</code>	Local	<code>local</code> : This is to run MR jobs. <code>classic</code> : This is to run MR jobs in cluster as well as pseudo-distributed mode (MRv1). <code>yarn</code> : This is to run MR jobs as YARN (MRv2).
<code>mapreduce.map.memory.mb</code>	1024	The memory to be requested for each map task from the scheduler. For large jobs that require intensive processing in the Map phase, set this number high.
<code>mapreduce.map.java.opts</code>	None	You can specify <code>Xmx</code> , <code>verbose</code> , and <code>gc</code> strategy through this parameter, which can take place during Map task execution.
<code>mapreduce.reduce.memory.mb</code>	1024	The memory to be requested for each map task from the scheduler. For large jobs that require intensive processing in the Reduce phase, set this number high.
<code>mapreduce.reduce.java.opts</code>	No Defaults	You can specify <code>Xmx</code> , <code>verbose</code> , and <code>gc</code> strategy through this parameter, which can take place during Reduce task execution.
<code>mapreduce.jobhistory.address</code>	<code>0.0.0.0:10020</code>	This is for Job history server and IPC port.
<code>mapreduce.jobhistory.webapp.address</code>	<code>0.0.0.0:19888</code>	This is again for Job history server but to host its web application. Once this is set, you will be able to access the Job history server UI at 19888.

You will find list of all the different configuration properties for `mapred-site.xml` here.

Working with Job history server

Apache Hadoop is blessed with the daemon of Job history server. As the name indicates, the responsibility of Job history server is to keep track of all of the jobs that are run in the past, as well as those currently running. Job history server provides a user interface through its web application to system users for accessing this information. In addition to job-related information, it also provides statistics and log data after the job is completed. The logs can be used during debugging phase; you do not need physical server access, as it is all available over the web.

Job history server can be set up independently, as well as with part of the cluster. If you did not set up Job history server, you can do it quickly. Hadoop provides a script, `mr-jobhistory-daemon.sh`, in the `$HADOOP_HOME/sbin` folder to run Job history daemon from the command line. You can run the following command:

```
Hadoop@base0:/$ $HADOOP_HOME/sbin/mr-jobhistory-daemon.sh -conf ig
$HADOOP_HOME/etc/Hadoop/ start historyserver
```

Alternatively, you can run the following:

```
Hadoop@base0:/$ $HADOOP_HOME/bin/mapred --daemon start historyserver
```

Now, try accessing the Job history server User Interface from your browser by typing the `http://<job-history-server-host>:19888` URL.



Job history server will only start working when you run your Hadoop environment in cluster or pseudo-distributed mode.

RESTful APIs for Job history server

In addition to the HTTP Web URL to get the status of jobs, you can also use APIs to get job history information. It primarily provides two types of APIs through RESTful service:

- APIs to provide information about Job history server (the application)
- APIs to provide information about the jobs

Please read more about REST here (https://en.wikipedia.org/wiki/Representational_state_transfer). You can test Job History RESTful APIs with simple browser plugins for Firefox, Google Chrome, and IE/Edge. You can also get an XML response if you try accessing it directly through the web. Now, try accessing the information API by typing the following URL in your browser: `http://<job-history-host>:19888/ws/v1/history`; you should see something like the following screenshot:



Let's quickly glance through all of the APIs that are available:

API	URL	Description	API Details
Get information about Job history server.	<code>http://<job-history-host>:19888/ws/v1/history</code> <code>http://<job-history-host>:19888/ws/v1/history/info</code>	This API returns information about Job history server. The same is available when you access the URL: <code>http://<job-history-host>:19888/jobhistory/about</code> .	Link
Get a list of finished MapReduce job information.	<code>http://:19888/ws/v1/history/mapreduce/jobs</code>	This API supports query parameters such as Username, status, and job timings and it returns an array of job objects, each of which contains information such as job name, timings, map task and reduce task count, job ID, and name.	Link
Get information about a specific job.	<code>http://:19888/ws/v1/history/mapreduce/jobs/{jobid}</code>	This API provides information about specific jobs. This response is more detailed, so you can get the list of jobs and the job ID can be passed as a parameter to get this information.	Link
Get information about job attempts.	<code>http://:19888/ws/v1/history/mapreduce/jobs/{jobid}/jobattempts</code>	This API provides information about attempts taken to run the jobs in MapReduce. It returns information such as the node where the attempt was performed and links to log information. This API is useful primarily in debugging.	Link
Get counter information about jobs.	<code>http://:19888/ws/v1/history/mapreduce/jobs/{jobid}/counters</code>	This API provides information about counters for Map Tasks and Reduce Tasks. The counters will typically include counts of bytes read/written, memory-related counts, and record information.	Link

Get information about job configuration.	http://:19888/ws/v1/history/mapreduce/jobs/{jobid}/conf	This API provides information about a given job configuration, in terms of name value pairs.	Link
Get information about tasks.	http://:19888/ws/v1/history/mapreduce/jobs/{jobid}/tasks	This API gets information about tasks in your job, for example, Map Task, Reduce Task, or any other tasks. This information typically contains status, timing information, and ID.	Link
Get detailed information about single task.	http://:19888/ws/v1/history/mapreduce/jobs/{jobid}/tasks/{taskid}	This API returns information about specific tasks; you have to pass the task ID to this API.	Link
Get counter information about the task.	http://:19888/ws/v1/history/mapreduce/jobs/{jobid}/tasks/{taskid}	This API is similar to the job counter, except that it returns counters for specific tasks.	Link
Get information about attempts of tasks.	http://:19888/ws/v1/history/mapreduce/jobs/{jobid}/tasks/{taskid}/attempts	Similar to job attempts.	Link
Get detailed information about attempts of single tasks.	http://:19888/ws/v1/history/mapreduce/jobs/{jobid}/tasks/{taskid}/attempts/{attemptid}	This API gets detailed information about task attempts. The difference between previous API is that it is specific to one attempt, and one has to pass it as a parameter.	Link
Get counter information for task attempts.	http://:19888/ws/v1/history/mapreduce/jobs/{jobid}/tasks/{taskid}/attempts/{attemptid}/counters	For a given attempt, the history server will return counter information.	Link

Understanding Hadoop APIs and packages

Now let's go through some of the key APIs that you will be using while you program in MapReduce. First, let's understand the important packages that are part of Apache Hadoop MapReduce APIs and their capabilities:

Java API Packages	Description
<code>org.apache.Hadoop.mapred</code>	Primarily provides interfaces for MapReduce, input/output formats, and job-related classes. This is an older API.
<code>org.apache.Hadoop.mapred.lib</code>	Contains libraries for Mapper, Reducer, partitioners, and so on. To be avoided—use <code>mapreduce.lib</code> .
<code>org.apache.Hadoop.mapred.pipes</code>	Job submitter-related classes.
<code>org.apache.Hadoop.mapred.tools</code>	Command-line tools associated with MapReduce.

<code>org.apache.Hadoop.mapred.uploader</code>	The <code>org.apache.Hadoop.mapred.uploader</code> package contains classes related to the MapReduce framework upload tool.
<code>org.apache.Hadoop.mapreduce</code>	New APIs pertaining to MapReduce; these provide a lot of convenience for end users.
<code>org.apache.Hadoop.mapreduce.counters</code>	This package contains the implementations of different types of MapReduce counters.
<code>org.apache.Hadoop.mapreduce.lib</code>	This package contains multiple libraries pertaining to various Mappers, Reduces, and Partitioners.
<code>org.apache.Hadoop.mapreduce.lib.aggregate</code>	Provides classes related to aggregation of value.
<code>org.apache.Hadoop.mapreduce.lib.chain</code>	Allows multiple chains of Mapper and Reducer classes within a single Map/Reduce task.
<code>org.apache.Hadoop.mapreduce.lib.db</code>	Package that provides classes to connect to databases, such as MySQL and Oracle, and read/write information.
<code>org.apache.Hadoop.mapreduce.lib.fieldsel</code>	This package implements a Mapper/Reducer class that can be used to perform field selections in a manner similar to Unix <code>cut</code> .
<code>org.apache.Hadoop.mapreduce.lib.input</code>	Contains all the classes pertaining to input of various formats.
<code>org.apache.Hadoop.mapreduce.lib.jobcontrol</code>	Provides helper classes to consolidate the jobs with all of their dependencies.
<code>org.apache.Hadoop.mapreduce.lib.map</code>	Provides ready-made mappers such as RegEx, Swapper, multi threaded, and so on.
<code>org.apache.Hadoop.mapreduce.lib.output</code>	Provides library of classes for output format.
<code>org.apache.Hadoop.mapreduce.lib.partition</code>	Provides classes related to data partitioning such as binary partitioning and hash partitioning.
<code>org.apache.Hadoop.mapreduce.lib.reduce</code>	Provides ready-made reusable reduce functions.
<code>org.apache.Hadoop.mapreduce.tools</code>	Command-line tools associated with MapReduce.

Setting up a MapReduce project

In this section, we will learn how to create the environment to start writing applications for MapReduce programming. The programming is typically done in Java. The development of a MapReduce application follows standard Java development principles as follows:

1. Usually, developers write the programs in a development environment such as Eclipse or NetBeans.
2. Developers do unit testing usually with a small subset of data. In case of failure, they can run an IDE Debugger to do fault identification.
3. It is then packaged in JAR files and is tested in a standalone fashion for functionality.
4. Developers should ideally write unit test cases to test each functionality.
5. Once it is tested in standalone mode, developers should test it in a cluster or pseudo-distributed environment with full datasets. This will expose more problems, and they can be fixed. Here debugging can pose a challenge, so one may need to rely on logging and remote debugging.
6. When it all works well, the compiled artifacts can move into the staging environment for system and integration testing by testers.
7. At the same time, you may also look at tuning the jobs for performance. Once a job is certified for performance and all other acceptance testing, it can move into the production environment.

When you write programs in MapReduce, usually you focus more on writing Map and Reduce functions of it.

Setting up an Eclipse project

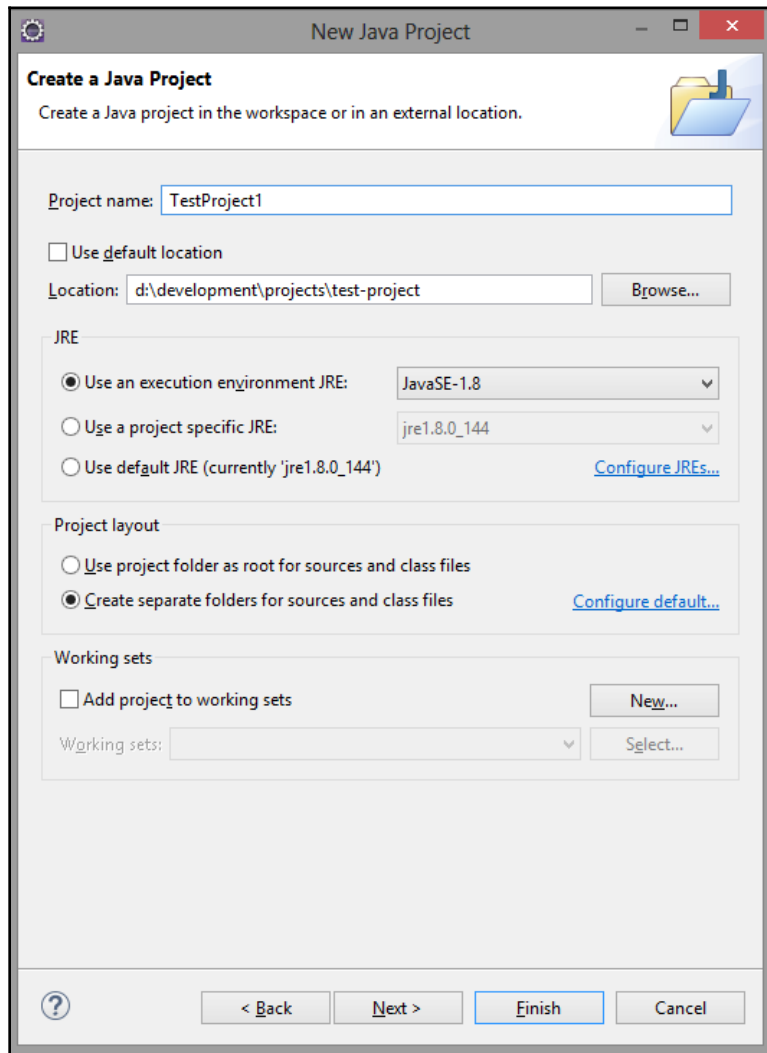
When you need to write new programs for Hadoop, you need a development environment for coding. There are multiple Java IDEs available, and Eclipse is the most widely used open source IDE for your development. You can download the latest version of Eclipse from <http://www.eclipse.org>.

In addition to Eclipse, you also need JDK 8 for compiling and running your programs. When you write your program in an IDE such as Eclipse or NetBeans, you need to create a Java or Maven project. Now, once you have downloaded Eclipse on your local machine, follow these steps:

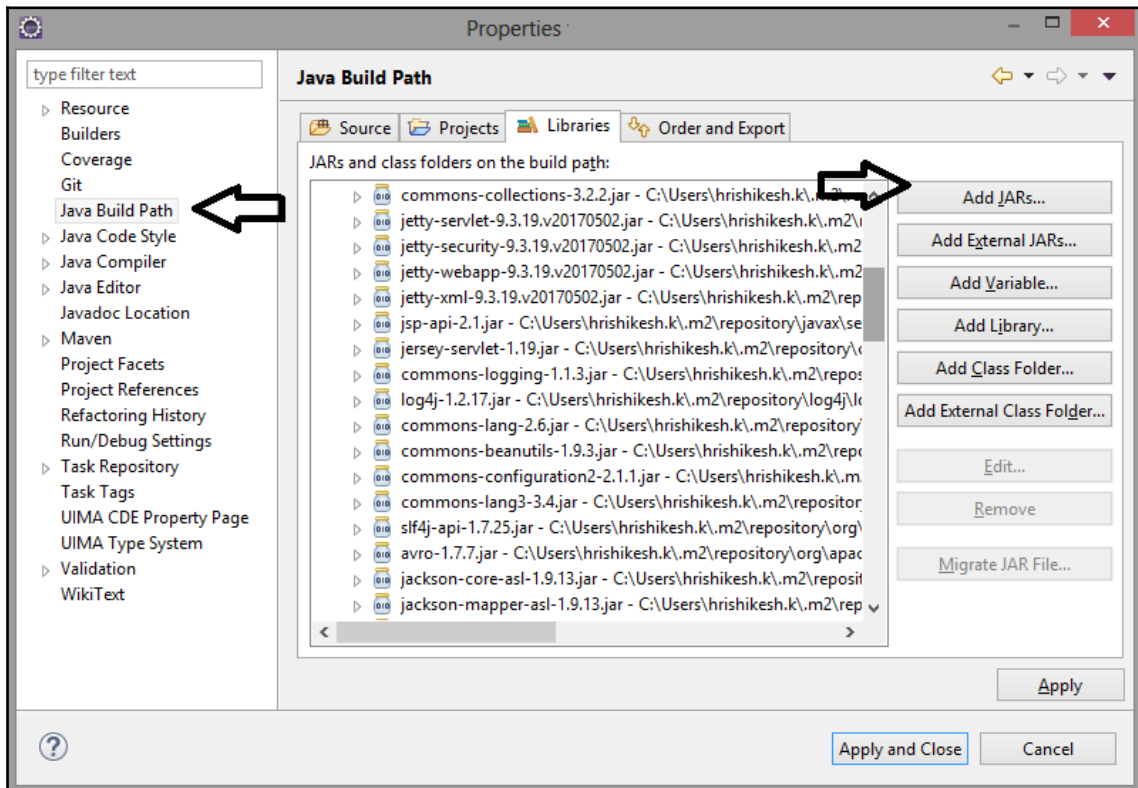
1. Open Eclipse and create a new Java Project:

File | New | Java Project

See the following screenshot:



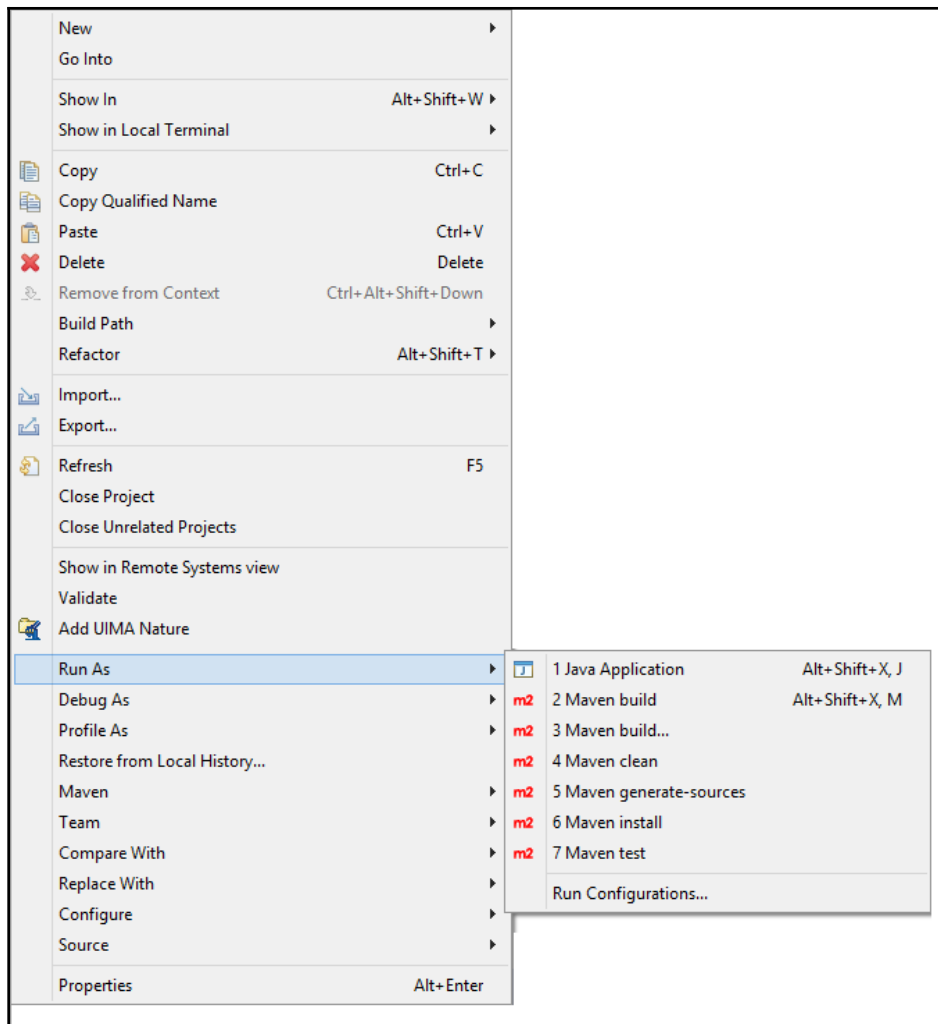
2. Once a project is created, you will need to add Hadoop libraries and other relevant libraries for this project. You can do that by right-clicking on your project in **package explorer/project explorer** and then by clicking on **Properties**. Now go to **Java Build Path** and add the Hadoop client libraries, as shown in in the following screenshot:



3. You will need the `Hadoop-client-<version>.jar` file to be added. Additionally, you may also need the `Hadoop-common-<version>.jar` file. You can get these files from `$HADOOP_HOME/share/Hadoop`. There are sub directories for each area such as `client`, `common`, `MapReduce`, `hdfs`, and `yarn`.
4. Now, you can write your program and compile it. To create a JAR file for Hadoop, please follow the standard process of JAR creation in Eclipse as listed [here](#).
5. You can alternatively create a Maven project, and use a Maven dependency, as follows:

```
<dependencies>
  <dependency>
    <groupId>org.apache.Hadoop</groupId>
    <artifactId>Hadoop-client</artifactId>
    <version>3.1.0</version>
  </dependency>
</dependencies>
```

6. Now run **mvn install** from the command-line interface or, from Eclipse, right-click on the project, directly through Eclipse, and run **Maven install**, as shown in the following screenshot:



The Apache Hadoop Development Tools (<http://hdt.incubator.apache.org/>) project provides Eclipse IDE plugins for Hadoop 1.x and 2.x; these tools provide ready-made wizards for Hadoop project creation and management, features for launching MapReduce from Eclipse, and monitoring jobs. However, the latest Hadoop version is not supported in the plugin (<http://hdt.incubator.apache.org/>).

Deep diving into MapReduce APIs

Let's start looking at different types of data structures and classes that you will be using while writing MapReduce programs. We will be looking data structures of input and output to MapReduce, and different classes that you can use for Mapper, Combiner, Shuffle, and Reducer.

Configuring MapReduce jobs

Usually, when you write programs in MapReduce, you start with configuration APIs first. In our programs that we have run in previous chapters, the following code represents the configuration part:

```
37 public class SequenceFileCreator {
38     /**
39      * This is a mapper class for reading the file information and put it in sequences
40      *
41      * @author hrishikesh.k
42      */
43     public static class ReadFileMapper extends Mapper<LongWritable, Text, LongWritable, Text> {
44         public void map(LongWritable key, Text value, Context context)
45             throws IOException, InterruptedException {
46             Configuration conf = context.getConfiguration();
47             //simply write the text as it is
48             context.write(key, value);
49         }
50     }
51 }
52
53 public static void main(String[] args)
54     throws IOException, ClassNotFoundException, InterruptedException {
55     Configuration conf = new Configuration();
56
57     if (args.length < 2) {
58         System.err.println("Usage: SequenceFileCreator <input-path> <output-file-path>");
59         System.exit(2);
60     }
61
62     Job job = Job.getInstance(conf, "Sequence File Creator");
63     job.setJarByClass(SequenceFileCreator.class);
64     job.setMapperClass(ReadFileMapper.class);
65     job.setOutputFormatClass(SequenceFileOutputFormat.class);
66
67     FileInputFormat.addInputPath(job, new Path(args[0]));
68     FileOutputFormat.setOutputPath(job, new Path(args[1]));
69
70     System.exit(job.waitForCompletion(true) ? 0 : 1);
71 }
72
73 }
```

The `Configuration` class (part of the `org.apache.hadoop.conf` package) provides access to different configuration parameters. The API reads properties from the supplied file. The configuration file for a given job can be provided through the `Path` class (<https://Hadoop.apache.org/docs/r3.1.0/api/org/apache/Hadoop/fs/Path.html>) or through `InputStream` (<http://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html?is-external=true>) using `Configuration.addResource()` (<https://Hadoop.apache.org/docs/r3.1.0/api/org/apache/Hadoop/conf/Configuration.html#addResource-java.io.InputStream->).

`Configuration` is a collection of properties with a key (usually `String`) and value (can be `String`, `Int`, `Long`, or `Boolean`). The following code snippet shows how you can instantiate the `Configuration` object and add resources such as a configuration file to it:

```
Configuration conf = new Configuration();
conf.addResource("configurationfile.xml");
```

The `Configuration` class is useful while switching between different configurations. It is common that, when you develop Hadoop applications, you switch between your local, pseudo-distributed, and cluster environments; the files can change according to the environment without any impact to your program. The `Configuration` filename can be passed as an argument to your program to make it dynamic. The following is an example configuration for a pseudo-distributed node:

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost/</value>
  </property>
</configuration>
```



The `fs.default.name` property may change; for the local filesystem, it could be `file:///`, and for a cluster, it could be `hdfs://<host>:9000`.

The `Job` class (part of the `org.apache.hadoop.mapreduce` package) allows users to specify different parameters for your application, which typically would include configuration, classes for input and output, and so forth. The functionality is not just limited to configuration, but the `Job` class allows users to submit the job, wait till it finishes off, get the status of `Job`, and so forth. The `Job` class can be instantiated with the `Job.getInstance()` method call:

- `getInstance(Configuration conf)`
- `getInstance(Configuration conf, String jobName)`
- `getInstance(JobStatus status, Configuration conf)`

Once initialized, you can set different parameters of the class. When you are writing a MapReduce job, you need to set the following parameters at minimum:

- Name of `Job`
- Input format and output formats (files or key-values)
- Mapper and Reducer classes to run; Combiner is an optional parameter
- If your MapReduce application is part of a separate JAR, you may have to set it as well

We will look at the details of these classes in next section. There are other optional configuration parameters that can be passed to `Job`; they are listed in MapReduce Job API documentation [here](https://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/mapreduce/Job.html#setArchiveSharedCacheUploadPolicies-org.apache.hadoop.conf.Configuration-java.util.Map-) (<https://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/mapreduce/Job.html#setArchiveSharedCacheUploadPolicies-org.apache.hadoop.conf.Configuration-java.util.Map->). When the required parameters are set, you can submit `Job` for execution to MapReduce Engine. You can do it with two options—you can either have an asynchronous submission through `Job.submit()`, where the call returns immediately; or have a synchronous submission through the `Job.waitForSubmission(boolean verbose)` call, where the control waits for `Job` to finish. If it's asynchronous, you can keep checking the status of your job through the `Job.getStatus()` call. There are five different statuses:

- **PREP:** `Job` is getting prepared
- **RUNNING:** `Job` is running
- **FAILED:** `Job` has failed to complete
- **KILLED:** `Job` is killed by some user/process
- **SUCCEEDED:** `Job` has completed successfully

Understanding input formats

Before you consider writing your MapReduce programs, you first you need to identify the input and output formats of your job. We have seen some formats in our last chapter about HDFS (different file formats). The `InputFormat<K, V>` interface (found in the `org.apache.Hadoop.mapreduce` package) and `OutputFormat<K, V>` interface (found in the `org.apache.Hadoop.mapreduce` package) describe the specifications for the input and output of your job respectively.

In the case of the `InputFormat` class, the MapReduce framework verifies the specification with actual input passed to the job, then it splits the input into a set of records for different Map Tasks using the `InputSplit` class and then uses an implementation of the `RecordReader` class to extract key-value pairs that are supplied to the Map task. Luckily, as the application writer, you do not have to worry about writing `InputSplit` directly; in many cases, you would be looking at the `InputFormat` interface.

Let's look at the different implementations that are available:

InputFormat SubClass	Sub-SubClass	Description
<code>ComposableInputFormat</code>		Provides an enhanced <code>RecordReader</code> interface for joins.
<code>CompositeInputFormat</code>		It's useful to join different data sources together when sorted and partitioned in a similar way. It allows you to extend the default comparator for joining based on keys.
<code>DBInputFormat</code>		Designed to work with SQLs, it can read tables directly. It produces the <code>LongWritable</code> class as a key and <code>DBWritable</code> class as a value. It uses <code>LIMIT</code> and <code>OFFSET</code> to separate data.
<code>DBInputFormat</code>	<code>DataDrivenDBInputFormat</code>	Similar to <code>DBInputFormat</code> , but it uses the <code>WHERE</code> clause for splitting the data.
<code>DBInputFormat</code>	<code>DBInputFormat</code>	This is a pointer to old package: <code>org.apache.Hadoop.mapred.lib.db</code> .
<code>FileInputFormat</code>		Widely used for file-based operations, it allows extending logic to split files with <code>getSplit()</code> and prevent them by overriding the <code>isSplittable()</code> methods.

FileInputFormat	CombineFileInputFormat	This interface is used when you want to combine multiple small files together and create splits based on file sizes. Typically, small file refers to a file that is smaller than the HDFS block size.
FileInputFormat	FixedLengthInputFormat	This is used primarily to read fixed-length records, which could be binary, text, or any other form. They must set the length of the record by calling <code>FixedLengthInputFormat.setRecordLength()</code> , or set it in the Configuration class through <code>Configuration.setInt(FixedLengthInputFormat.FIXED_RECORD_LENGTH, recordLength)</code> .
FileInputFormat	KeyValueTextInputFormat	This format is primarily for well-formatted files such as CSVs. The file should have the <code>key<separator>value</code> form. The separator can be provided as the Configuration class attribute: <code>mapreduce.input.keyvaluelinerecordreader.key.value.separator</code> .
FileInputFormat	NLineInputFormat	This format is useful when you have one or more large files and you need to process different file blocks separately. The file can be split with the <code>N</code> line.
FileInputFormat	SequenceFileInputFormat	In the previous chapter, we saw Sequence Files; this format is used to work with those files directly.
FileInputFormat	TextInputFormat	This format is primarily used to process text files. The key is the location of the text, and the value is the line itself in your files. Line feed or carriage return is used as a record separator.

Many times, applications may require each file to be processed by one Map Task rather than the default behavior. In that case, you can prevent this splitting with `isSplittable()`. Each `InputFormat` has the `isSplittable()` method which determines whether the file can be split or not, so simply overriding it as shown in the following example should address your concerns:

```
import org.apache.Hadoop.fs.Path;
import org.apache.Hadoop.mapreduce.JobContext;
import org.apache.Hadoop.mapreduce.lib.input.KeyValueTextInputFormat;
import org.apache.Hadoop.mapreduce.lib.input.TextInputFormat;

public class SampleKeyValueInputFormat extends KeyValueTextInputFormat {

    @Override
    protected boolean isSplittable(JobContext context, Path file) {
        return false;
    }
}
```

Based on your requirements, you can also extend the `InputFormat` class and create your own implementation. Interested readers can read this blog, which provides some examples of a custom `InputFormat` class: <https://iamssoftwareengineer.wordpress.com/2017/02/14/custom-input-format-in-mapreduce/>.

Understanding output formats

Similar to `InputFormat`, the `OutputFormat<K, V>` interface is responsible to represent the output of `Job`. When MapReduce job activity finishes, the output format specification is validated against the class definition, and the system provides the `RecordWriter` class to write the record to the underlying filesystem.

Now let's look at class hierarchy of the `OutputFormat` class (found in `org.apache.Hadoop.mapred.lib.db`):

OutputFormat SubClass	Sub-SubClass	Description
DBOutputFormat		This class is useful when you wish to write your output to a relational database. Please go through the following information box to understand the risks of using this format with traditional RDBMS.
FileOutputFormat		This class is a base class for writing file output from your MapReduce jobs. The files that are produced can be stored on HDFS. Additionally, you can compress the output files with <code>FileOutputFormat.setCompressOutput(JobConf, true)</code> ; you can also provide custom compression with your own class by setting <code>FileOutputFormat.setOutputCompressorClass(JobConf, CustomClass<extending CompressionCodec>, codecClass)</code> . This class creates <code>part-r-nnnnn</code> files as output.
FileOutputFormat	MapFileOutputFormat	In the previous chapter, we saw map files. This class produces Map files as output. The responsibility of producing sorted keys lies with the Reducer class.
FileOutputFormat	MultipleOutputFormat	As the name suggests, this class can produce more than one file as output. There is one file produced per Reducer, and they are named by the partition number (<code>part-r-00000</code> , <code>part-r-00001</code> and so on).
FileOutputFormat	MultipleOutputFormat > MultipleSequenceFileOutputFormat	This class allows you to write data to different <code>SequenceFile</code> formats.
FileOutputFormat	MultipleOutputFormat > MultipleTextOutputFormat	This class allows you to write your data to multiple files in text format.
FileOutputFormat	SequenceFileOutputFormat	This class can write sequence files as output, as shown in Chapter 3 code repository. You will need <code>SequenceFile</code> output only when your MapReduce program is part of a larger project where there is a need to continue the processing of jobs.
FileOutputFormat	SequenceFileOutputFormat > SequenceFileAsBinaryOutputFormat	This class is responsible for creating <code>SequenceFile</code> output in binary form. It writes key-value pairs in its raw form.
FileOutputFormat	TextOutputFormat	This is a default <code>OutputFormat</code> ; it produces text output; each key-value pair is separated with the <code>mapreduce.output.textoutputformat.separator</code> attribute. All <code>Key</code> , <code>Value</code> classes are converted into string with <code>toString()</code> and then written to files.
FilterOutputFormat	LazyOutputFormat	This class produces output in lazy fashion. So, in many cases when you wish to avoid proliferation of empty files that have no records, you can wrap <code>OutputFormat</code> with this class, so only when a record is produced will the file be created.
NullOutputFormat		This class does not produce any output; instead it consumes all output produced out of the MapReduce job and passes it to <code>/dev/null</code> (https://en.wikipedia.org/wiki/Null_device). This is useful when you are explicitly producing output in your Reducer job and do not wish to proliferate any more output files.

The `MultipleOutputs` class is a helper class that allows you to write data to multiple files. This class enables `map()` and `reduce()` functions to create data into multiple files. Filenames are of the `-r-nnnnn,part-r-nnnn(n+1)` part. I have provided a sample test code for `MultipleOutputFormat` (please look at `SuperStoreAnalyzer.java`); the dataset can be downloaded from <https://opendata.socrata.com/Business/Sample-Superstore-Subset-Excel-/2dgv-cxpb/data>.



When you use `DBInputFormat` or `DBOutputFormat`, you need to take into account the amount of Mapper tasks that will be connecting to the traditional relational database for read operation or reducers that will be sending output to the database in parallel. The classes do not have any data slicing or sharding capabilities, so this may impact the database performance. It is recommended that large data reads and writes with the database should be handled through export/import rather these formats. These formats are useful for processing smaller datasets. Alternatively, you can control the map-tasks and reduce task count through configuration as well. However, HBase provides its own `TableInputFormat` and `TableOutputFormat`, which can scale well for large datasets.

Working with Mapper APIs

Map and Reduce functions are designed to input list of (key,value) pairs and produce list of (key, value) pairs. The `Mapper` class provides three different methods that users can override to get the mapping activity complete:

- `setup`: This is called once in the beginning of map call. You can initialize your variables here or get the context for Map tasks here.
- `map`: This is called for each (key,value) in the input that is split.
- `cleanup`: This is called again once at the end of tasks. This should close all allocations, connections, and so on.

The extended class API for Mapper is as follows:

```
public class <YourClassName>
    extends
    Mapper<InputKeyClass, InputValueClass, OutputKeyClass, OutputValueClass> {
    protected void setup(Context context) {
        //setup related code goes here
    }
}
```

```

    }

    protected void map(InputKeyClass key, InputValueClass value, Context
context)
    throws IOException {
        // your code goes here
    }

    protected void cleanup(Context context) {
        //clean up related code goes here
    }
}

```

Each API passes context information that was created when you created jobs. You can use the context to pass your information to Map Task; there is no other direct way of passing your parameters.

Let's now look at a different implementation of pre defined Mapper in the map class. I have provided a link to each mapper's JavaDoc for a quick example and reference:

Mapper Class	Description
ChainMapper	As the name suggests, it allows the multiple Mapper classes in one map task. The tasks are piped or chained together. Chainedmapper: input (k1,v1) -> map() -> intermediate(k2,v2) -> map() -> intermediate(k3,v3) -> map() -> output (k4,v4)
FieldSelectionMapper	This mapper allows multiple fields to be passed in a single (key,value) pair. The fields can have a separator (the default is \t). This separator can be changed by setting <code>mapreduce.fieldsel.data.field.separator</code> , for example, <code>firstname,lastname,middlename:Hrishikesh,Karambelkar,Vijay</code> can be one of the input specifications for this mapper.
InverseMapper	Provides inverse function by swapping with keys and values.
MultiThreadedMapper	Runs the map function in multi threaded mode; you can use <code>MultiThreadedMapper. getNumberOfThreads(JobContext job)</code> (https://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/mapreduce/lib/map/MultithreadedMapper.html#getNumberOfThreads-org.apache.hadoop.mapreduce.JobContext-) to know the number of threads from the thread pool that are active.
RegexMapper	This mapper extracts the text that is matching the given regular expression. You can set its pattern by setting <code>RegexMapper.PATTERN</code> .
TokenCounterMapper	Provides tokenizing capabilities for input values; in addition to tokenizer, it also publishes the count of each token.
ValueAggregatorMapper	Provides generic mapper for aggregate functions.
WrappedMapper	Enables a wrap context across mapper.



When you need to share large amounts of information across multiple maps or reduce tasks, you cannot use traditional ways such as a filesystem or local cache, which you would otherwise prefer. Since there is no control over which node the given Map task and Reduce task will run, it is better to have a database or standard third-party service layer to store your larger context across MapReduce tasks. However, you must be careful, because for each (key, value) pair in the Map task, the control will try to read it from the database, impacting performance; hence, you can utilize the `setup()` method to set the context at once for all map tasks.

Working with the Reducer API

Just like `map()`, the `reducer()` function reduces the input list of (key, value) pairs to the output list of (key, value) pairs. A Reducer function goes through three major phases all in one function:

- **Shuffle:** The relevant portion of each output of Mapper is passed to reducer for shuffle through HTTP
- **Sort:** Reducer performs sorting on a group of keys
- **Reduce:** Merges or reduces the sorted keys

Similar to Mapper, Reducer provides `setup()` and `cleanup` methods. Overall class structure of Reducer implementation may look like the following:

```
public class <YourClassName>
    extends
    Reducer<InputKeyClass, InputValueClass, OutputKeyClass, OutputValueClass> {
    protected void setup(Context context) {
        //setup related code goes here
    }

    protected void reduce(InputKeyClass key, Iterator<InputValueClass>
    values, Context context)
    throws IOException {
        // your code goes here
    }

    protected void cleanup(Context context) {
        //clean up related code goes here
    }
}
```

The three phases that I described are part of the `reduce` function of the Reducer class.

Now let's look at different predefined reducer classes that are provided by the Hadoop framework:

Reducer Class	Description	Links
ChainReducer	Similar to ChainMapper, this provides a chain of reducers.	https://Hadoop.apache.org/docs/r3.1.0/api/org/apache/Hadoop/mapreduce/lib/chain/ChainReducer.html
FieldSelectionReducer	This is similar to FieldSelectionMapper.	https://Hadoop.apache.org/docs/r3.1.0/api/org/apache/Hadoop/mapreduce/lib/fieldsel/FieldSelectionReducer.html
IntSumReducer	This reducer is intended to get the sum of integer values when performed Group by on keys.	https://Hadoop.apache.org/docs/r3.1.0/api/org/apache/Hadoop/mapreduce/lib/reduce/IntSumReducer.html
LongSumReducer	Similar to IntSumReducer, this class performs sum on long values instead of integer values.	https://Hadoop.apache.org/docs/r3.1.0/api/org/apache/Hadoop/mapreduce/lib/reduce/LongSumReducer.html
ValueAggregatorCombiner	Similar to ValueAggregatorMapper, just that the class provides the combiner function in addition to reducer.	https://Hadoop.apache.org/docs/r3.1.0/api/org/apache/Hadoop/mapreduce/lib/aggregate/ValueAggregatorCombiner.html
ValueAggregatorReducer	This is similar to ValueAggregatorMapper.	https://Hadoop.apache.org/docs/r3.1.0/api/org/apache/Hadoop/mapreduce/lib/aggregate/ValueAggregatorReducer.html
WrappedReducer	This is similar to WrappedMapper with custom reducer context implementation.	https://Hadoop.apache.org/docs/r3.1.0/api/org/apache/Hadoop/mapreduce/lib/reduce/WrappedReducer.html

When you have multiple Reducers, a Partitioner instance is created to control the partitioning of keys in intermediate state of processing. Typically there is a direct proportion of number of partitions with number of reduce tasks.



Serialization is a process to transform Java objects into byte stream, and through de-serialization you can revert it back. This is useful in a Hadoop environment to transfer objects from one node to another or to persist the state on disk, and so forth. However, most of the Hadoop applications avoid using Java serialization; instead, it creates its own writable types such as BooleanWritable (<https://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/io/BooleanWritable.html>) and BytesWritable (<https://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/io/BytesWritable.html>). This is primarily due to the overhead associated with general-purpose Java serialization process. Additionally, Hadoop's framework avoids creating new instances of objects and looks at reuse aspects more. This becomes a big differentiator when you deal with thousands of such objects.

Compiling and running MapReduce jobs

In this section, we will cover compiling and running MapReduce jobs. We have already seen examples of how jobs can be run on standalone, pseudo-development, and cluster environments. You need to remember that, when you compile the classes, you must do it with same versions of your libraries and Java that you will otherwise run in production, otherwise you may get major-minor version mismatch errors in your run-time (read the description [here](#)). In almost all cases, the JAR for programs is created and run directly through the following command:

```
Hadoop jar <jarfile> <parameters>
```

Now let's look at different alternatives available for running the jobs.

Triggering the job remotely

So far, we have seen how one can run the MapReduce program directly on the server. It is possible to send the program to a remote Hadoop cluster for running it. All you need to ensure is that you have set the resource manager address, `fs.defaultFS`, library files, and `mapreduce.framework.name` correctly before running the actual job. So, your program snippet would look something like this:

```
Configuration conf = new Configuration();
conf.set("yarn.resourcemanager.address", "<your-hostname>:<port>");
conf.set("mapreduce.framework.name", "mapreduce");
conf.set("fs.defaultFS", "hdfs://<your-hostname>/");
conf.set("yarn.application.classpath", "<client-jar-libraries>");
conf.set("HADOOP_USER_NAME", "<pass-username>");
conf.set("mapreduce.job.jar", "myjobfile.jar");
//you can also set jar file in job configuration
Job job = Job.getInstance(conf);
//now run your regular flow from here
```

Using Tool and ToolRunner

Any MapReduce job will have your mapper logic, a reducer, and a driver class. We have already gone through Mapper and Reducer in a previous chapter. The driver class is the one that is responsible for running the MapReduce job. Apache Hadoop provides helper classes for its developers to make life easy. In previous examples, we have seen direct calls to MapReduce APIs through job configuration with synchronous and asynchronous calling. The following example shows one such `Driver` class construct:

```
public class MapReduceDriver {
    public static void main(String[] args) throws Exception {
        Job job = new Job();
        job.setMapperClass(MyMapper.class);
        job.setReducerClass(MyReducer.class);

        //set other variables
        job.set....
        //run and wait for completion
        job.waitForCompletion(true);
    }
}
```

Now let's look at some interesting options available out of the box. An interface called `Tool` provides a mechanism to run your programs with generic standard command-line options. The beauty of `ToolRunner` is that the effort of extracting parameters that are passed from the command line get handled by themselves. When you have to pass parameters to Mapper or Reducer from the command line, you would typically do something like the following:

```
//in main method
Configuration conf = new Configuration();
//first set it
conf.set("property1", args[0]);
conf.set("property2", args[1]);

//whereever you use it
conf.get("property1");
conf.get("property2");
```

And then you call them through the following parameter:

```
Hadoop jar NoToolRunner.jar com.Main property1 property2
```

With `ToolRunner`, you can save that effort, as follows:

```
public int run(String args[]) {  
    Configuration conf = getConf();  
    //whereever you get it  
    conf.get("property1");  
    conf.get("property2");  
}
```

And a command line can pass parameters through in the following way:

```
hadoop jar ToolRunner.jar com.Main -D property1=value1 -D property2=value2
```

Please note that these properties are different from standard JVM properties, which cannot have spaces between `-D` and the property names. Also, note the difference in terms of their position after main class name specification. The `Tool` interface provides the `run()` function where you can put your code for calling your code for setting configuration and job parameters:

```
public class ToolBasedDriver extends Configured implements Tool {  
  
    public static void main(String[] args) throws Exception {  
        int myRunner = ToolRunner.run(new Configuration(), new ToolBasedDriver(),  
args);  
        System.exit(myRunner);  
    }  
  
    @Override  
    public int run(String[] args) throws Exception {  
        // When implementing tool  
        Configuration conf = this.getConf();  
  
        Job job = new Job(conf, "MyConfig");  
  
        job.setMapperClass(MyMapper.class);  
        job.setReducerClass(MyReducer.class);  
  
        job.set.....  
  
        // Execute job and return status  
        return job.waitForCompletion(true) ? 0 : 1;  
    }  
}
```

Unit testing of MapReduce jobs

As a part of application building, you must provide unit test cases for your MapReduce program. Unit testing is a software testing capability that can be used to test individual parts/units of your application. In our case, the focus on unit testing will be on Mapper and Reducer functions. The testing done during the development stage can prevent large amount of losses of time, efforts, and money, which may be incurred due to issues found in the production environment. As a good practice for testing, refer to the following guidelines:

- Use automation tools to test your program with less/no human intervention
- Unit testing should happen primarily on the development environment in an isolated manner
- You must create a subset of data as test data for your testing
- If you get any defects, enhance your test to check the defect first
- Test cases should be independent of each other; the focus should be on key functionalities—in this case, it will be `map()` and `reduce()`
- Every time code changes are done, the tests should be run

Luckily, all MapReduce frameworks follow specific practice of development; that makes our life easy for testing. There are many tools available in the market for testing your MapReduce programs, such as `Apache MRUnit`, `Mockito`, and `PowerMock`. Among them, `Apache MRUnit` was under development; however, in 2016, it was retired by Apache. `Mockito` and `PowerMock` are used today.

Both Map and Reduce functions require `Context` to be passed as a parameter; we can provide a mock `Context` parameter to these classes and write test cases with `Mockito's mock()` method. The following code snippet shows how unit testing can be performed on Mapper directly:

```
import static org.mockito.Mockito.*;

public class TestMapper {
    @Mock
    Mapper.Context context;

    @Rule public MockitoRule mockitoRule = MockitoJUnit.rule();

    @Test
    public void testMapper() {
        //set Key and Value
        //Text key = ...;
        //Text value = ...;
```

```
CustomMapper m = new CustomMapper(keyin,valuein,context);  
//now check if the context produced expected output text  
verify(context).write(new Text("<passoutputvalue>"), new  
Text("<passoutputvalue>"));  
}  
}
```

You can pass expected input to your mapper, and get the expected output from `Context`. The same can be verified with the `verify()` call of Mockito. You can apply the same principles to test reduce calls as well.

Failure handling in MapReduce

Many times, when you run your MapReduce application, it becomes imperative to handle errors that can occur when your complex processing of data is in progress. If it is not handled aggressively, it may cause failure and take your output into inconsistent state. Such situations may require a lot of human intervention to cleanse the data and re-run it. So, handling expected failures much in advance in the code and configuration helps a lot. There could be different types of error; let's look at common errors:

- **Run-time errors:**
 - Errors due to failure of tasks—child tasks
 - Issues pertaining to resources
- **Data errors:**
 - Errors due to bad input records
 - Malformed data errors
- **Other errors:**
 - System issues
 - Cluster issues
 - Network issues

The first two errors can be handled by your program (in fact run-time errors can be handled only partially). Errors pertaining to the system, network, and cluster will get handled automatically thanks to Apache Hadoop's distributed multi-node High Availability cluster.

Let's look at the first two errors, which are the most common. The child task fails at times, for unforeseen reasons such as user-written code through `RuntimeException` or processing resource timeout. These errors get logged into the user logging file for Hadoop. For both map and reduce functions, the Hadoop configuration provides `mapreduce.map.maxattempts` for Map tasks and `mapreduce.reduce.maxattempts` with the default value 4. This means if a task fails a maximum of four times and it fails again, the job will be marked as failed.

When it comes down to handling bad records, you need to have conditions to detect such records, log them, and ignore them. One such example is the use of a counter to keep track of such records. Apache provides a way to keep track of different entities, through its `counter` mechanism. There are system-provided counters, such as bytes read and number of map tasks; we have seen some of them in Job History APIs. In addition to that, users can also define their own counters for tracking. So, your mapper can be enriched to keep track of these counts; look at the following example:

```
if (color not red condition true){
    context.getCounter(COLOR.NOT_RED).increment(1);
}
```

Or, you can handle your exception, as follows:

```
catch (NullPointerException npe){
    context.getCounter(EXCEPTION.NPE).increment(1);
}
```

You can then get the final count through job history APIs or from the `Job` instance directly, as follows:

```
....
job.waitForCompletion(true);
Counters counters = job.getCounters();
Counter cl = counters.findCounter(COLOR.NOT_RED);
System.out.println("Errors" + cl.getDisplayName() + ":" + cl.getValue());
```

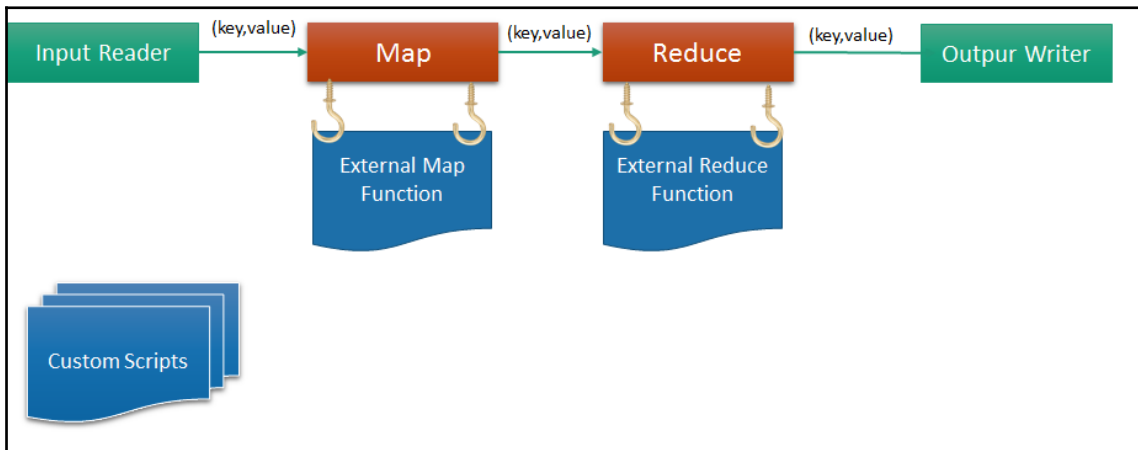
If a Mapper or Reducer terminates for any reason, the counters will be reset to zero, so you need to be careful. Similarly, you may connect to a database and pass on the status or alternatively log it in the logger. It all depends upon how you are planning to act on the output of failures. For example, if you are planning to process the failed records later, then you cannot keep the failure records in the log file, as it would require script or human intervention to extract it.

Well-formed data cannot be guaranteed when you work with very large datasets so, in such cases, your mapper and reducer need to handle even the key and value fields. For example, text data needs to have a maximum length of line, to ensure that no junk is getting in. Typically, such data is ignored by Hadoop programs, as most of the applications of Hadoop look at analytics over large-scale data, unlike any other transaction system, which requires each data element and its dependencies.

Streaming in MapReduce programming

The traditional MapReduce programming requires users to write map and reduction functions as per the specifications of the defined API. However, what if I already have a processing function written, and I want to federate the processing to my own function, still using the MapReduce concept over Hadoop's distributed File System? There is a possibility to solve this with the streaming and pipes functions of Apache Hadoop.

Hadoop streaming allows user to code their logic in any programming language such as C, C++, and Python, and it provides a hook for the custom logic to integrate with traditional MapReduce framework with no or minimal lines of Java code. The Hadoop streaming APIs allow users to run any scripts or executables outside of the traditional Java platform. This capability is similar to Unix's Pipe function ([https://en.wikipedia.org/wiki/Pipeline_\(Unix\)](https://en.wikipedia.org/wiki/Pipeline_(Unix))), as shown in the following diagram:



Please note that, in the case of streaming, it is okay not to have any reducer, so in that case, you can pass `-Dmapred.reduce.task=0`; you may also set map tasks through the `mapred.map.task` parameter. Here is what the streaming command looks like:

```
$HADOOP_HOME/bin/Hadoop jar contrib/streaming/Hadoop-streaming-  
<version>.jar \  
    -input input_dirs <directory> \  
    -output output_dir <directory>\  
    -mapper <script> \  
    -reducer <script>
```

Let's look at important parameters for streaming APIs now:

Important Parameters	Description
<code>-input directory/file-name</code>	Input location for mapper (Required)
<code>-output directory-name</code>	Output location for reducer (Required)
<code>-mapper executable or script</code>	Executable for Mapper (Required)
<code>-reducer executable or script</code>	Executable for Reducer (Required)

For more details regarding MapReduce Streaming, you may refer to (<https://Hadoop.apache.org/docs/r3.1.0/Hadoop-streaming/HadoopStreaming.html>).

Summary

In this chapter, we have gone through various topics pertaining to MapReduce with a deeper walk through. We started with understanding the concept of MapReduce and an example of how it works. We started configuring the config files for a MapReduce environment; we also configured Job history server. We then looked at Hadoop application URLs, ports, and so on. Post-configuration, we focused on some hands-on work of setting up a MapReduce project and going through Hadoop packages, and then we did a deeper dive into writing MapReduce programs. We also studied different data formats needed for MapReduce. Later, we looked at job compilation, remote job run, and using utilities such as `Tool` for a simple life. We then studied unit testing and failure handling.

Now that you are able to write applications in MapReduce, in the next chapter, we will start looking at building applications in Apache YARN, a new MapReduce (also called MapReduce v2).

5 Building Rich YARN Applications

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

– Martin Golding

YARN or (Yet **A**nother **R**esource **N**egotiator) was introduced in Hadoop version 2 to open distributed programming for all of the problems that may not necessarily be addressed using the MapReduce programming technique. Let's look at the key reasons behind introducing YARN in Hadoop:

- The older Hadoop used Job Tracker to coordinate running jobs whereas Task Tracker was used to run assigned jobs. This eventually became a bottleneck due to a single Job Tracker when working with a high number of Hadoop nodes.
- With traditional MapReduce, the nodes were assigned fixed numbers of Map and Reduce slots. Due to this nature, the utilization of the cluster resources was not optimal due to inflexibility between Map and Reduce slots.
- Mapping every problem that requires distributed computing to classic MapReduce was becoming a tedious activity for developers.
- Earlier MapReduce was mostly Java-driven; all of the programs needed to be coded in Java. With YARN in place, writing a YARN application can be done beyond the Java language.

The work for YARN started around 2009-2010 in Yahoo. The cluster manager in Hadoop 1.X was replaced with Resource Manager; similarly, JobTracker was replaced with ApplicationMaster and TaskTracker was replaced with Node Manager. Please note that the responsibilities for each of the YARN components are a bit different from Hadoop 1.X. Previously, we have gone through the details of Hadoop 3.X and 2.X components. We will be covering the job scheduler as a part of the *Chapter 6, Monitoring and Administration of Hadoop Cluster*.

Today, YARN is getting popularity primarily due to the clear advantages of scalability and flexibility it offers over traditional MapReduce. Additionally, it can be utilized over commodity hardware, making it low cost distributed application framework. Today, YARN is successfully implemented in production by many companies including eBay, Facebook, Spotify, Xing, Yahoo, and so on. Many applications such as Apache Storm and Apache Spark provide YARN-based services, which utilize the YARN framework in a continuous manner. Many applications provide support to YARN-based framework components. We will be looking at these applications in *Chapters 7, Demystifying Hadoop Ecosystem Components* and *Chapter 8, Advanced Topics in Apache Hadoop*.

In this chapter, we will be doing a deep dive into YARN with focus on the following topics:

- Understanding YARN architecture
- Configuring the YARN environment
- Using the Apache YARN distributed CLI
- Setting up a YARN project
- Developing a YARN application

Technical requirements

You will need Eclipse development environment and Java 8 installed on your system where you can run/tweak these examples. If you prefer to use maven, then you will need maven installed to compile the code. To run the example, you also need Apache Hadoop 3.1 setup on Linux system. Finally, to use the Git repository of this book, you need to install Git.

The code files of this chapter can be found on GitHub:

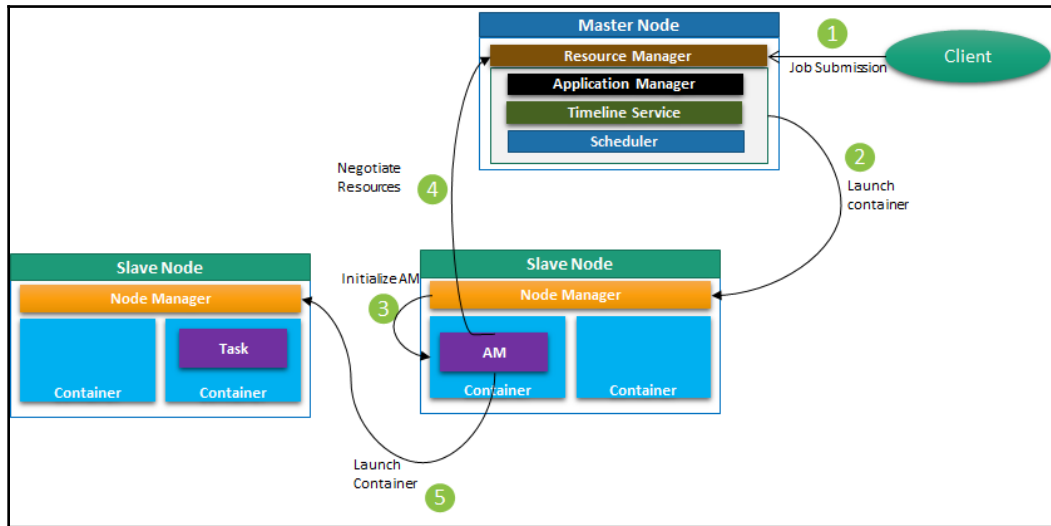
<https://github.com/PacktPublishing/Apache-Hadoop-3-Quick-Start-Guide/tree/master/Chapter5>

Check out the following video to see the code in action:

<http://bit.ly/2CRSq5P>

Understanding YARN architecture

YARN separates the role of Job Tracker into two separate entities. A Resource Manager is a central authority and is responsible for allocation and management of cluster resources, and an application master to manage the life cycle of applications that are running on the cluster. The following diagram depicts YARN architecture and the flow of requests-response:



YARN provides the basic units of applications such as memory, CPU, and GPU. The units of an application are utilized by containers. All containers are managed by respective Node Managers running on the Hadoop cluster. The **Application master (AM)** negotiates with the **Resource Manager (RM)** for container availability along with the resource manager. The AM container is initialized by client through resource manager as shown in step 2. Once AM is initialized, it demands container availability, and then requests that Node Manager initializes an application container for the running job. Additionally, AM responsibilities include monitoring tasks, restarting failed tasks, and calculating different metric application counters. Unlike the Job Tracker, each application running on YARN has a dedicated application master.

The Resource Manager additionally keeps track of live **Node Managers (NMs)** and available resources. The RM has two main components:

- **Scheduler:** Responsible for allocating resources to jobs as per configured scheduler policy; we will be looking at this in detail in the *Chapter 6, Monitoring and Administration of a Hadoop Cluster*
- **Application manager:** Front face module to accept jobs, identify Application Master, and negotiate the availability of containers

Now, the interesting part is that application master can run any jobs. We will study more about this in the *YARN application development* section. YARN also provides a web-based proxy as a part of RM to avoid direct access to RM. This can prevent attack on RM directly. You can read more about the proxy server here (<https://hadoop.apache.org/docs/r3.1.0/hadoop-yarn/hadoop-yarn-site/WebApplicationProxy.html>).

Key features of YARN

YARN offers significant gain over traditional MapReduce programming that comes from older versions of Apache Hadoop. With YARN, you can write any custom applications that can utilize the power of commodity hardware and Apache Hadoop's HDFS filesystem to scale and perform. Let's go through some of the key features of YARN that brings major additions. We have already covered the new features of YARN 3.0, such as the intra-disk balancer in *Chapter 1, Hadoop 3.0 - Background and Introduction*.

Resource models in YARN

YARN supports an extensible resource model. This means that the definition of resources can be extended from its default values (such as CPU and memory) to any types of resources that can be consumed when the tasks run in container. You can also enable profiling of resources through `yarn-site.xml`, which offers a group of multiple resources request through a single profile. To enable the resource configuration in `yarn-site.xml`, please set the `yarn.resourcemanager.resource-profiles.enabled` property to `true`. Create two additional configuration files, `resource-type.xml` and `node-resources.xml`, in the same directory where `yarn-site.xml` is placed. A sample of the resource profile (`resource-profiles.json`) is shown in the following snippet:

```
{
  "small": {
    "memory-mb" : 1024,
```

```

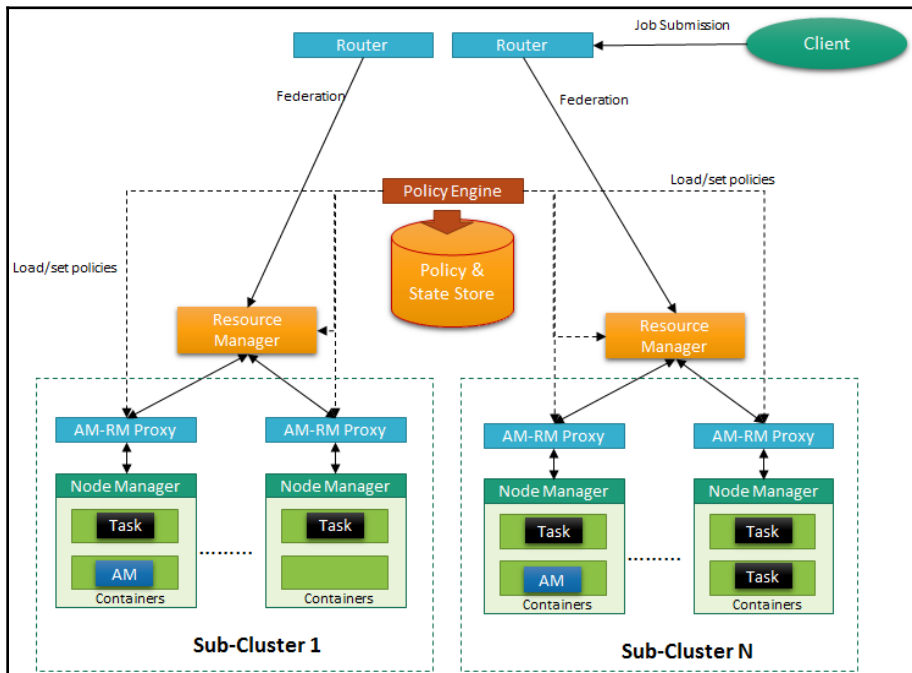
    "vcores" : 1
  },
  "large" : {
    "memory-mb": 4096,
    "vcores" : 4
    "gpu" : 1
  },
},

```

You can read more details about resource profiling [here](#).

YARN federation

When you work across large numbers of Hadoop nodes, the possible limitation of resource manager being a single standalone instance dealing with multiple nodes becomes evident. Although it supports high availability, it is still impacted by performance due to various interactions between Hadoop nodes and resource manager. YARN federation is a feature in which Hadoop nodes can be classified into multiple clusters, all of which work together through federation giving applications a single view of one massive YARN cluster. The following architecture shows how YARN federation works:



With Federation, YARN brings in the routers which are responsible for applying routing as per the routing policy set by the **Policy Engine** to all incoming job applications. Routers identify the sub-cluster that will execute the given job and work with resource manager for further execution, hiding Resource Manager's visibility to the outside world. **AM-RM Proxy** is a sub-component that hides the Resource Managers and allows Application Masters to work across multiple clusters. It is also useful to protect the resource and prevent DDOS attacks. The **Policy and State Store** is responsible for storing the states of clusters and policies such as routing patterns and prioritization. You can activate Federation by setting `true` the `yarn.federation.enabled` property in `yarn-site.xml`, as seen previously. For the **Router**, there are additional properties to be set, as covered in the previous section. You may need to set up multiple Hadoop clusters and then bring them together through YARN Federation. Apache documentation for YARN Federation covers setup and properties [here](#).

RESTful APIs

Apache YARN provides RESTful APIs to give client applications access to different metric data pertaining to clusters, nodes, resource managers, applications, and so on. So, consumers can use these RESTful services in their own monitoring applications to keep tab of YARN applications, as well as system context, remotely. Today, the following components support RESTful information:

- Resource Manager
- Application Master
- History Server
- Node Manager

The system supports both JSON and XML format (the default is XML); you have to specify the format as a parameter to header. The access pattern to the RESTful service is as follows:

```
http://<host>:<port>/ws/<version>/<resource-path>
```

`host` is typically Node Manager, Resource Manager, and Application Master, and `version` usually is 1 (unless you have deployed updated versions). The Resource Manager RESTful API provides information about cluster metrics, schedulers, nodes, application states, priorities and other parameters, scheduler configuration, and other statistical information. You can read more about these [here](#). Similarly, the Node Manager RESTful APIs provide information and statistics about the NM instance, application statistics, and container statistics. You can look at the API specification [here](#).

Configuring the YARN environment in a cluster

We have seen the configuration of MapReduce and HDFS. To enable YARN, first you need to inform Hadoop that you are using YARN as your framework, so you need to add the following entries in `mapred-site.xml`:

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

Please refer to Chapter 2, *Planning and Setting Up Hadoop Clusters*, for additional properties and steps for configuring YARN. Now, let's look at key configuration elements in `yarn-site.xml` that you would be looking at day by day:

Property Name	Default Value	Description
yarn.resourcemanager.hostname	0.0.0.0	Specify the hostname of resource manager.
yarn.resourcemanager.address		IP address and port. The default will pick up 8032 port and hostname.
yarn.resourcemanager.scheduler.address		The IP address and port of scheduler. Default port is 8030.
yarn.http.policy	HTTP_ONLY	Endpoints: HTTP, HTTPS.
yarn.resourcemanager.webapp.address		Web App Address, default is 8088.
yarn.resourcemanager.webapp.https.address		HTTP address default is 8090.
yarn.acl.enable	FALSE	Whether ACLs should be enabled on YARN or not.
yarn.scheduler.minimum-allocation-mb	1024	Minimum memory allocation for every container in MB.
yarn.scheduler.maximum-allocation-mb	8192	Maximum allocation in MB. Any requests higher than this value can result in exception.
yarn.scheduler.minimum-allocation-vcores	1	Minimum Virtual CPU Core allocations.
yarn.scheduler.maximum-allocation-vcores	4	Maximum Virtual CPU Core allocations.
yarn.resourcemanager.ha.enabled	FALSE	Whether High availability of resource manager is enabled or not (Active-Standby).
yarn.resourcemanager.ha.automatic-failover.enabled	TRUE	Enable automatic failover. By default, it is enabled only when HA is enabled.
yarn.resourcemanager.resource-profiles.enabled	FALSE	Flag to enable/disable resource profiles.
yarn.resourcemanager.resource-profiles.source-file	resource-profiles.json	Filename for resource profile. More details follow the table.
yarn.web-proxy.address		Web Proxy IP and Port if enabled.
yarn.federation.enabled	FALSE	Whether federation is enabled for RM or not.
yarn.router.bind-host		Router will bind to given address (useful for federation).

yarn.router.clientrm .interceptorclass .pipeline	org.apache.hadoop .yarn.server.router.clientrm .FederationClientInterceptor	Routing strategies in a comma-separated manner. Finally, it should end with org.apache.hadoop.yarn.server .router.clientrm.FederationClientInterceptor.
--	---	--

You can access a list of all properties here (<http://hadoop.apache.org/docs/r3.1.0/hadoop-yarn/hadoop-yarn-common/yarn-default.xml>).

Working with YARN distributed CLI

YARN CLI provides three types of commands. The first type is for users who wish to use YARN infrastructure for developing applications. The second type are administrative commands, which provide monitoring and administrative capabilities of all components of YARN including resource manager, application master, and timeline server. The third type are daemon commands, which are used for maintenance purposes covering stopping, starting, and restarting of daemons. Now, let's look at user commands for YARN:

Command	Parameters	Description	Important Parameters
application	yarn application <command> <parameters>	All actions pertaining to applications such as print and kill.	- appID <applicationID> - kill <applicationID> - list - status <applicationID>
applicationattempt	yarn applicationattempt <parameter>	Prints an application attempt(s) report.	
classpath	yarn classpath -- jar <path>	Prints the classpath needed for the given JAR or prints the current classpath set when passed without a parameter.	
container	yarn container <parameters>	Prints a container report.	-status <containerID> -list <applicationattemptID>

jar	<code>yarn jar <jar file> <mainClassName></code>	Runs the given JAR file in YARN. The main class name is needed.	
logs	<code>yarn logs <command> <parameter></code>	Dumps the log for a given application, container, or owner.	<code>-applicationId <applicationID> - containerId <containerID></code>
node	<code>yarn node <command> <parameter></code>	Prints node-related reports.	<code>-all prints it for all nodes - list - lists all nodes</code>
queue	<code>yarn queue <options></code>	Prints queue information.	<code>-status <queueName></code>
version		Prints current Hadoop version.	
envvars		Displays current environment variables.	

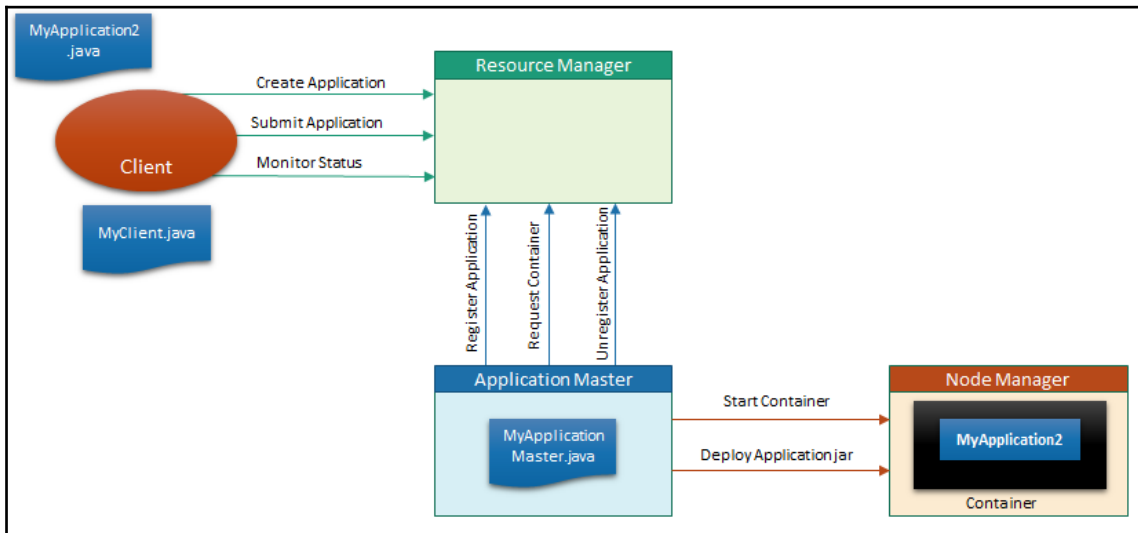
The following screenshot shows how a command is fired on YARN:

```
hadoop@base0:~/hadoop-3.1.0$ ./bin/yarn node -list
2018-09-26 10:39:11,281 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
Total Nodes:1
  Node-Id          Node-State Node-Http-Address  Number-of-Running-Containers
localhost:40279    RUNNING   localhost:8042      0
```

When a command is run, the YARN client connects to the Resource Manager default port to get the details—in this case, node listing. More details about administrative and daemon commands can be read [here](#).

Deep dive with YARN application framework

In this section, we will do a deep dive into YARN application development. YARN offers flexibility to developers to write applications that can run on Hadoop clusters in different programming languages. In this section, we will focus on setting up a YARN project, we will write a sample client and application master, and we will see how it runs on a YARN cluster. The following block diagram shows typical interaction patterns between various components of Apache Hadoop when a YARN application is developed and deployed:



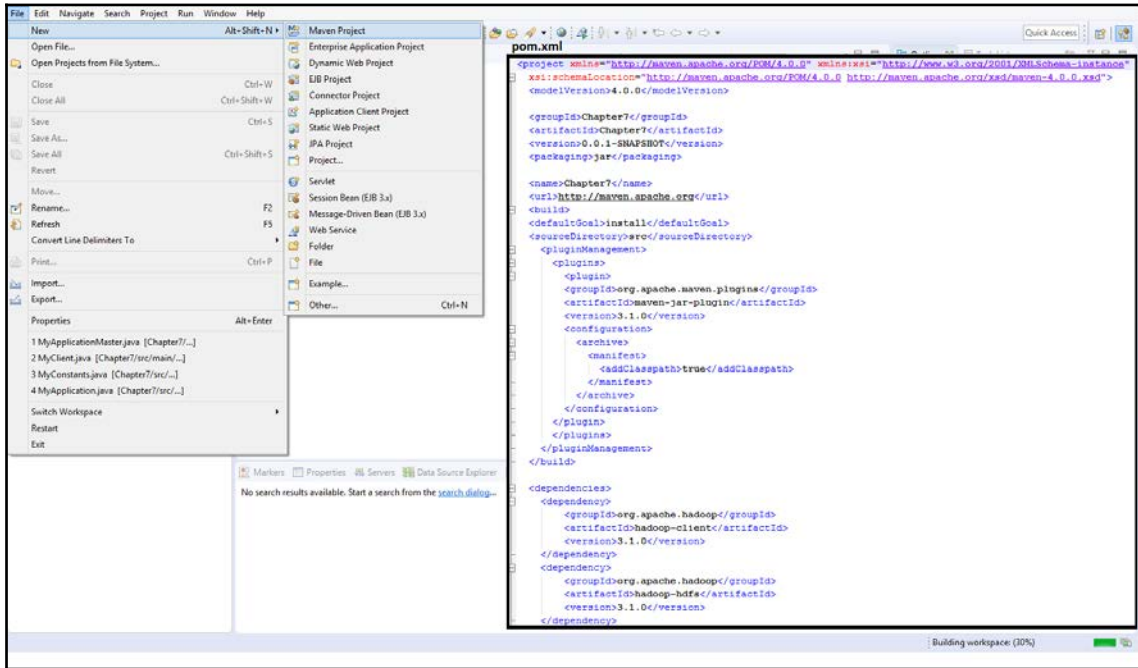
Primarily, there are three major components involved: Resource Manager, Application Master, and Node Manager. We will be creating a custom client application, a custom application master, and a YARN client app. As you can see, there are three different interactions that take place between different components:

- Client and Resource Manager through `ClientRMProtocol`
- ApplicationMaster and Resource Manager through `AMRMProtocol`
- ApplicationMaster and Node Manager through the `ContainerManager` mechanism

Let's look at each of them in detail.

Setting up YARN projects

Now let's start with setting up a YARN project for your development. A YARN project can be set up as a Maven application over Eclipse or any other development environment. Now simply create a new Maven project as shown in the following screenshot:



Creating an Eclipse project

Now, open `pom.xml` and add the dependency for the Apache Hadoop client:

```

<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-client</artifactId>
  <version>3.1.0</version>
</dependency>

```

Now try compiling the project and create a JAR out of it. You may consider adding a manifest to your JAR where you can put an executable class name to the path.

Writing your YARN application with YarnClient

When you write your custom YARN application, you need to use the `YarnClient` API (<https://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/yarn/client/api/YarnClient.html>). You need to write a YARN client initially to create a client object, which you will be using for further calling. First, you create a new instance of `YarnClient` by calling static `createYarnClient()`. `YarnClient` requires a configuration object to initialize:

```
YarnClient yarnClient = YarnClient.createYarnClient();
Configuration conf = new YarnConfiguration();
//add your configuration here
yarnClient.init(conf);
```

A call to `init()` initializes the `YarnClient` service. Once a service is initialized, you need to start the `YarnClient` service by calling `yarnClient.start()`. Once a client is started, you can create a YARN application through the YARN client application class, as follows:

```
YarnClientApplication app = yarnClient.createApplication();
GetNewApplicationResponse appResponse = app.getNewApplicationResponse();
```

I have provided a sample code for the same. Please refer to the `MyClient.java` file. Before you submit the application, you must first get all of the relevant metrics pertaining to memory and core from your YARN cluster to ensure that you have sufficient resources. Now, the next thing is to set the application name; you can do it with the following code snippet:

```
ApplicationSubmissionContext appContext =
app.getApplicationSubmissionContext();
ApplicationId appId = appContext.getApplicationId();
appContext.setApplicationName(appName);
```

Once you set this up, you need get the queue requirements, as well as set the priority for your application. You may also request ACL information for a given user to run the application to ensure that user can run the application. Once this is all done, you may need to set the container specification needed by Node Manager to initialize by calling `appContext.setAMContainerSpec()`, which is set through `ContainerLaunchContext` (<https://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/yarn/api/records/ContainerLaunchContext.html>). This will typically be your application master JAR file with parameters such as cores, memory, number of containers, priority, and minimum/maximum memory. Now you can submit this application with `YarnClient.submitApplication(appContext)` to initialize the container and run it

Writing a custom application master

Now that you have written a client to initiate or trigger the resource manager with the application and monitor it, we need to write a custom application master that can interact with Resource Manager and Node Manager to ensure that the application is executed successfully. First, you need to establish a client that can connect to Resource Manager through `AMRMClient`, through the following snippet:

```
AMRMClient<ContainerRequest> amRMClient = AMRMClient.createAMRMClient()
amRMClient.init(conf);
```

Initialization can happen over through standard configuration, which can be either `yarn-default.xml` or `yarn-site.xml`. Now you can start the container with `amRMClient.start()`. The next step is to register the current `ApplicationMaster`; this should be called before any other interaction steps:

```
amRMClient.registerApplicationMaster(host, port, trackingURL);
```

You need to pass `host`, `port`, and `trackingURL`; when left empty, it will consider default values. Once the registration is successful, to run our program, we need to request a container from Resource Manager. This can be requested with priority passed, as shown in the following code snippet:

```
ContainerRequest containerAsk = new ContainerRequest(capability, null,
null, priority);
amRMClient.addContainerRequest(containerAsk);
```

You may request additional containers through the `allocate()` call to `ResourceManager`. While `ResourceManager` is set up, the application master needs to talk with `Node Manager`, to ensure that the container is allocated and the application is getting executed successfully. So, first you need to initialize `NMClient` (<https://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/yarn/client/api/NMClient.html>) with the configuration, and start the `NMClient` service, as follows:

```
NMClient nmClient = NMClient.createNMClient();
nmClient.init(conf);
nmClient.start();
```

Now that the client is established, the next step for you is to start the container on `Node Manager` for you to deploy and run the application. You can do that by calling the following API:

```
nmClient.startContainer(container, appContainer);
```

When you start the container, you need to pass the application context, which includes the JAR file you wish to run on the container. The container gets initialized and starts running the JAR file. You can allocate one or more containers to your process through the `AMRMClient.allocate()` method. While the application runs on your container, you need to check the status of your container through the `AllocateResponse` class. Once it is complete, you can unregister the application master from status by calling `AMRMClient.unregisterApplicationMaster()`. This completes all of your coding work. In the next section, we will look at how you can compile, run, and monitor a YARN application on a Hadoop cluster.

Building and monitoring a YARN application on a cluster

YARN is a completely rewritten architecture of a Hadoop cluster. Once you are done with your development of the YARN application framework, the next step is to create your own custom application that you wish to run on YARN across a Hadoop cluster. Let's write a small application. In my example code, I have provided two applications:

- `MyApplication.java`: This prints *Hello World*
- `MyApplication2.java`: This calculates the value of PI to the 1,000th level

These simple applications would be run on the YARN environment through the client we have created. Let's look at how you can build a YARN application.

Building a YARN application

There are different approaches to building a YARN application. You can use your development environment to compile and create a JAR file out of it. In Eclipse, you can go to **File | Export | Jar File**, then you can choose the required classes and other artifacts and create the JAR file to be deployed. If you are using a Maven project, simply right-click on `pom.xml` | **Run as | Maven install**. You can also use the command line to run `mvn install` to generate the JAR file in your project target location.

Alternatively, you can use the `yarn jar` CLI to pass your compiled JAR file as input to the cluster. So, first create and package your project in Java Archive form. Once it is done, you can run it with the following YARN CLI:

```
yarn jar <jarlocation> <runnable-class> -jar <jar filename> <additional-parameters>
```

For example, you can compile and run sample code provided with this book with the following command:

```
yarn jar ~/copy/Chapter5-0.0.1-SNAPSHOT.jar
org.hk.book.hadoop3.examples.MyClient -jar ~/copy/Chapter5-0.0.1-
SNAPSHOT.jar -num_containers=1 -
apppath=org.hk.book.hadoop3.examples.MyApplication2
```

This command runs the given job on your YARN cluster. You should see the output of your CLI run:

```
18/09/27 20:06:03 INFO impl.TimelineClientImpl: Timeline service address: http://base3.rolta.com:8188/ws/v1/timeline/
18/09/27 20:06:03 INFO examples.MyClient: Initializing Client..
18/09/27 20:06:03 INFO examples.MyClient: Running Client
18/09/27 20:06:03 INFO client.RMProxy: Connecting to ResourceManager at base0.rolta.com/172.16.168.101:8050
18/09/27 20:06:04 INFO client.AHSProxy: Connecting to Application History server at base3.rolta.com/172.16.168.75:10200
18/09/27 20:06:04 INFO examples.MyClient: Max mem capability of resources in this cluster 11264
18/09/27 20:06:04 INFO examples.MyClient: Max virtual cores capability of resources in this cluster 6
18/09/27 20:06:04 INFO examples.MyClient: Copy App Master jar from local filesystem and add to local environment
18/09/27 20:06:05 INFO examples.MyClient: Set the environment for the application master
18/09/27 20:06:05 INFO examples.MyClient: Setting up app master command
18/09/27 20:06:05 INFO examples.MyClient: Completed setting up app master command ({JAVA_HOME})/bin/java -Xmx10m org.hk.book.hadoop3.examples.MyApplication2 --container_memory 10
--container_vcores 1 --num_containers 1 --priority 0 1><LOG_DIR>/AppMaster.stdout 2><LOG_DIR>/AppMaster.stderr
18/09/27 20:06:05 INFO examples.MyClient: Submitting application to ASM
18/09/27 20:06:05 INFO impl.YarnClientImpl: Submitted application application_1533042341514_0007
18/09/27 20:06:10 INFO examples.MyClient: Application completed successfully. YarnState=SUCCEEDED, DSFinalStatus=SUCCEEDED. Breaking monitoring loop : ApplicationId:7
```

Monitoring your application

Once the application is submitted, you can start monitoring the application by requesting the `ApplicationReport` object from `YarnClient` for a given app ID. From this report, you can extract the YARN application state and the application status directly through available methods, as shown in the following code snippet:

```
ApplicationReport report = yarnClient.getApplicationReport(appId);
YarnApplicationState state = report.getYarnApplicationState();
FinalApplicationStatus dsStatus = report.getFinalApplicationStatus();
```

The request for an application report can be done periodically to find the latest state of the application. The status should return different types of status for you to verify. For your application to be successful, your Yarn application state object should be `YarnApplicationState.FINISHED` and `FinalApplicationStatus` should be `FinalApplicationStatus.SUCCEEDED`. If you are not getting the SUCCESS status, you can kill the application from `YarnClient` by calling `yarnClient.killApplication(appId)`. Alternatively, you can track the status on the resource manager UI, as follows:

FINISHED Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decon
5	0	0	5	0	0 B	11 GB	0 B	0	6	0	1	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation
Capacity Scheduler	[MEMORY]	<memory 1024, vCores 1>

Show 20 entries

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCoers	Allocated Memory MB
application_1533042341514_0004	labuser	MyApplication	YARN	default	0	Wed Sep 26 13:51:24 +0550 2018	Wed Sep 26 13:51:27 +0550 2018	FINISHED	SUCCEEDED	N/A	N/A	N/A
application_1533042341514_0003	labuser	Hello	YARN	default	0	Wed Sep 26 13:30:41 +0550 2018	Wed Sep 26 13:30:44 +0550 2018	FINISHED	SUCCEEDED	N/A	N/A	N/A

Showing 1 to 2 of 2 entries

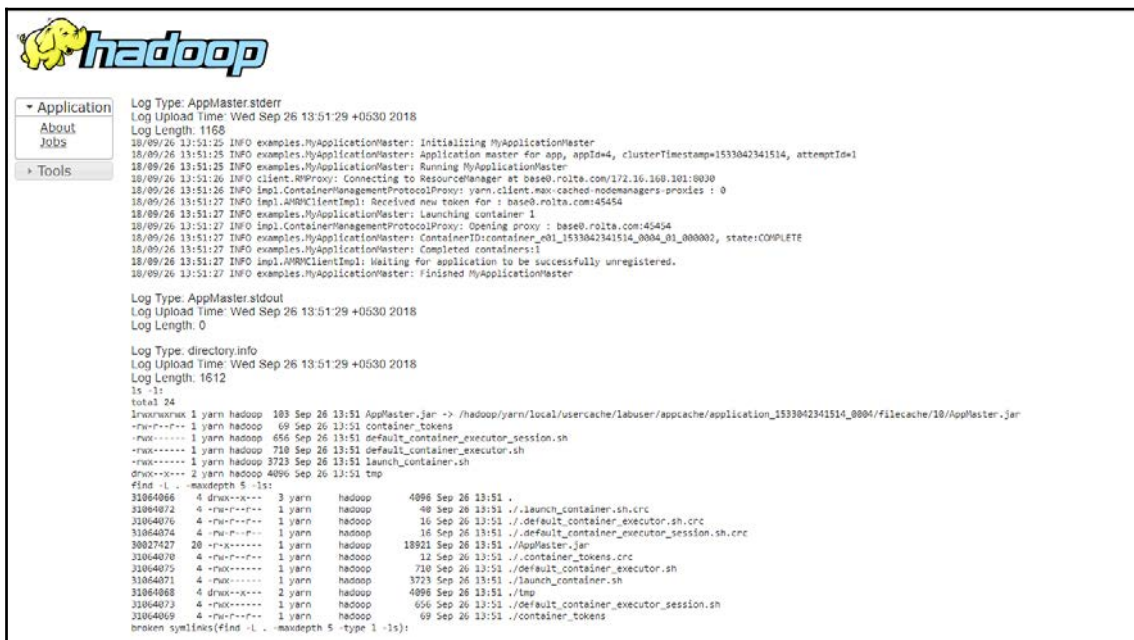
We have already seen this screen in a previous chapter. You can go inside the application and, if you click on Node Manager records, you should see node manager details in a new window, as shown in the following screenshot:

Logged in as: drwho

NodeManager information

Total Vmem allocated for Containers	23.10 GB
Vmem enforcement enabled	false
Total Pmem allocated for Container	11 GB
Pmem enforcement enabled	true
Total VCoers allocated for Containers	6
NodeHealthyStatus	true
LastNodeHealthTime	Thu Sep 27 17:23:46 IST 2018
NodeManager Version:	2.7.3.2.5.3.0-37 from 9828acfddec41a121f0121f556b09e2d112259e92 by jenkins source checksum b69b9e79b6035edc7c02dd322da024 on 2016-11-29T18:46Z
Hadoop Version:	2.7.3.2.5.3.0-37 from 9828acfddec41a121f0121f556b09e2d112259e92 by jenkins source checksum a6c8c64bfe734b01a46c7c480ebc7ff on 2016-11-29T18:37Z

The node manager UI provides details of cores, memory, and other resource allocations done for a given node. From your resource manager home, you can go inside your application and you can look through specific log comments that you might have recorded by going into details of a given application and accessing logs of it. The logs would show the `stderr` and `stdout` log file output. The following screenshot shows the output of the PI calculation example (`MyApplication2.java`):



The screenshot shows the Hadoop Node Manager UI. On the left, there is a sidebar with a yellow elephant icon and the word "hadoop" in blue. Below the icon, there are three tabs: "Application" (selected), "About", and "Jobs". The main content area displays logs for an application. The first log entry is for "AppMaster.stderr" with a log upload time of "Wed Sep 26 13:51:29 +0530 2018" and a log length of 1168. The log content shows the initialization and running of the application master. The second log entry is for "AppMaster.stdout" with the same upload time and a log length of 0. The third log entry is for "directory.info" with the same upload time and a log length of 1612. The log content shows the directory structure of the application.

```

Log Type: AppMaster.stderr
Log Upload Time: Wed Sep 26 13:51:29 +0530 2018
Log Length: 1168
18/09/26 13:51:25 INFO examples.MyApplicationMaster: Initializing MyApplicationMaster
18/09/26 13:51:25 INFO examples.MyApplicationMaster: Application master for app, appId=4, clusterTimestamp=1533042341514, attemptId=1
18/09/26 13:51:25 INFO examples.MyApplicationMaster: Running MyApplicationMaster
18/09/26 13:51:26 INFO client.WPProxy: Connecting to ResourceManager at base0.rota.com/172.16.168.101:8030
18/09/26 13:51:26 INFO Impl.ContainerManagementProtocolProxy: yarn.client.max.cached.nodemanager.proxies : 0
18/09/26 13:51:27 INFO Impl.AMRClientImpl: Received new token for : base0.rota.com:45454
18/09/26 13:51:27 INFO examples.MyApplicationMaster: Launching container: 1
18/09/26 13:51:27 INFO Impl.ContainerManagementProtocolProxy: Opening proxy : base0.rota.com:45454
18/09/26 13:51:27 INFO examples.MyApplicationMaster: ContainerID:container_rm1_1533042341514_0004_01_0000002, state:CONTAINER
18/09/26 13:51:27 INFO examples.MyApplicationMaster: Completed containers:1
18/09/26 13:51:27 INFO Impl.AMRClientImpl: Waiting for application to be successfully unregistered.
18/09/26 13:51:27 INFO examples.MyApplicationMaster: Finished MyApplicationMaster

Log Type: AppMaster.stdout
Log Upload Time: Wed Sep 26 13:51:29 +0530 2018
Log Length: 0

Log Type: directory.info
Log Upload Time: Wed Sep 26 13:51:29 +0530 2018
Log Length: 1612
ls -ls
total 24
drwxr-xr-x 1 yarn hadoop 103 Sep 26 13:51 AppMaster.jar -> /hadoop/yarn/local/usercache/labuser/appcache/application_1533042341514_0004/filecache/10/AppMaster.jar
-rw-r--r-- 1 yarn hadoop 69 Sep 26 13:51 container_tokens
-rwx----- 1 yarn hadoop 656 Sep 26 13:51 default_container_executor_session.sh
-rwx----- 1 yarn hadoop 710 Sep 26 13:51 default_container_executor.sh
-rwx----- 1 yarn hadoop 3723 Sep 26 13:51 launch_container.sh
drwx-xr-x 2 yarn hadoop 4096 Sep 26 13:51 tmp
find -L -maxdepth 5 -ls:
31064066 4 drwx-xr-x 3 yarn hadoop 4096 Sep 26 13:51 .
31064072 4 -rwx-r--r-- 1 yarn hadoop 40 Sep 26 13:51 ./launch_container.sh.crc
31064076 4 -rwx-r--r-- 1 yarn hadoop 16 Sep 26 13:51 ./default_container_executor.sh.crc
31064074 4 -rwx-r--r-- 1 yarn hadoop 16 Sep 26 13:51 ./default_container_executor_session.sh.crc
30027427 20 -rwx----- 1 yarn hadoop 18921 Sep 26 13:51 ./AppMaster.jar
31064070 4 -rwx-r--r-- 1 yarn hadoop 12 Sep 26 13:51 ./container_tokens.crc
31064075 4 -rwx----- 1 yarn hadoop 710 Sep 26 13:51 ./default_container_executor.sh
31064071 4 -rwx----- 1 yarn hadoop 3723 Sep 26 13:51 ./launch_container.sh
31064068 4 drwx-xr-x 2 yarn hadoop 4096 Sep 26 13:51 ./tmp
31064073 4 -rwx----- 1 yarn hadoop 656 Sep 26 13:51 ./default_container_executor_session.sh
31064069 4 -rwx-r--r-- 1 yarn hadoop 69 Sep 26 13:51 ./container_tokens
broken symlinks(find -L -maxdepth 5 -type l -ls):

```

Alternatively, YARN also provides JMX beans for you to track the status of your application. You can access `http://<host>:8088/jmx` to get the JMX beans response in JSON format. You can also access logs of your YARN cluster over the web by accessing `http://<host>:8088/logs`. The logs would provide logs and console output for node manager and resource manager. The example creation has been detailed out at Apache's official site about writing YARN applications, [here](#).

Summary

In this chapter, we have done a deep dive into YARN. We understood YARN architecture and key features of YARN such as resource models, federation, and RESTful APIs. We then configured a YARN environment in a Hadoop distributed cluster. We also studied some of the additional properties of `yarn-site.xml`. We then looked at the YARN distributed command-line interface. After this, we dived deep into building a YARN application, where we first created a framework needed for the application to run, then we created a sample application. We also covered building YARN applications and monitoring them.

In the next chapter, we will look at monitoring and administration of a Hadoop cluster.

6

Monitoring and Administration of a Hadoop Cluster

Previously, we have seen YARN and gained a deeper understanding of its capabilities. This chapter is focused on introducing you to the process-oriented approach to managing, monitoring, and optimizing your Hadoop cluster. We have already covered part of administration, when we set up a single node, a pseudo-distributed node, and a fully fledged distributed Hadoop cluster. We covered sizing the cluster, which is needed as part of the planning activity. We have also gone through some developer and system CLIs in the respective chapters on HDFS, MapReduce, and YARN. Hadoop administration is a vast topic; you will find lot of books dedicated to this activity in the market. I will be touching on key points of monitoring, managing, and optimizing your cluster.

We will cover the following topics:

- Roles and responsibilities of Hadoop administrators
- Planning your distributed cluster
- Resource management in Hadoop
- High availability of clusters
- Securing Hadoop clusters
- Performing routine tasks

Now, let's start understanding the roles and responsibilities of a Hadoop administrator.

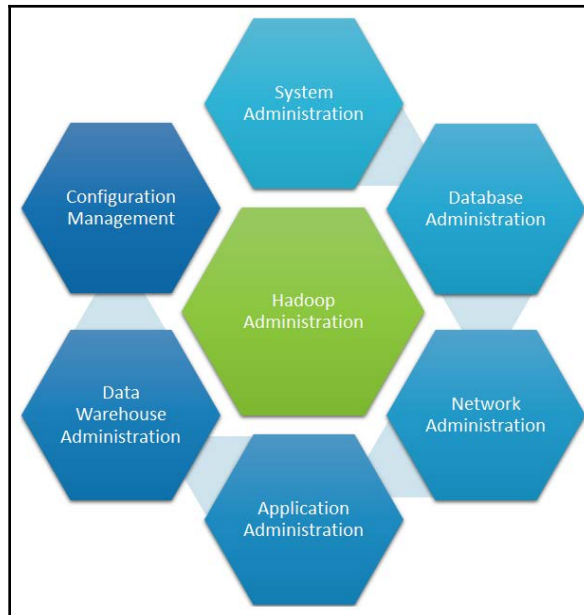
Roles and responsibilities of Hadoop administrators

Hadoop administration is highly technical work, where professionals need to have deeper understanding of the concepts of Hadoop, how it functions, and how it can be managed. The challenges faced by Hadoop administrators differ from other similar roles such as database or network administrators. For example, if you are a DBA, you typically get proactive alerts from the underlying database system when you run into tablespace threshold alerts when the disk space is not available for allocation, and you need to act on it, or else the operations will fail. In the case of Hadoop, the appropriate action is to move the job to another node in case it fails on one node due to sizing.

The following are the different responsibilities of a Hadoop administrator:

- Installation and upgrades of clusters
- Backup and disaster recovery
- Application management on Hadoop
- Assisting Hadoop teams
- Tuning cluster performance
- Monitoring and troubleshooting
- Log file management

We will be studying these in depth in this chapter. The installation and upgrades of clusters deals with installing new Hadoop ecosystem components, such as Hive or Spark, across clusters, upgrading them, and so on. The following diagram shows the 360 degrees of coverage Hadoop administration should be capable of:



Typically, administrators work with different teams and provide assistance to troubleshoot their jobs, tune the performance of clusters, deploy and schedule their jobs, and so on. The role requires a strong understanding of different technologies, such as Java and Scala, but, in addition to that, experience in sizing and capacity planning. This role also demands strong Unix shell scripting and DBA skills.

Planning your distributed cluster

In this section, we will cover the planning of your distributed cluster. We have already studied the sizing of clusters and estimation and data load aspects of clusters. When you explore different hardware alternatives, it is found that rack servers are the most suitable option available. Although Hadoop claims to support commodity hardware, the nodes still require server-class machines, and you should not consider setting up desktop-class machines. However, unlike high-end databases, Hadoop does not require high-end server configuration; it can easily work on Intel-based processors, along with standard hard drives. This is where you save the cost.

Reliability is a major aspect to consider while working with any production system. Disk drives use **Mean Time Between Failure (MTBF)**. It varies based on disk type. Hadoop is designed to work with hardware failures, so with the replication factor of HDFS, data is replicated by Hadoop across three nodes by default. So, you can work with SATA drives for your data nodes. You do not require high-end RAID for storing your HDFS data. Please visit this (<https://hadoopoopadoop.com/2015/09/22/hadoop-hardware/>) interesting blog, which covers SSDs, SATA, RAID, and other disk comparison.



Although RAID is not recommended for data nodes, it is useful for the master node where you are setting up NameNode and Filesystem image. With RAID, in the case of failure, it would be easy for you to recover data, block information, FS image information, and so on.

The amount of memory needed for Hadoop can vary from 26 GB to 128 GB. I have already provided pointers from the Cloudera guideline for a Hadoop cluster. When you do sizing for memory, you need to keep aside memory requirement for JVM and the underlying operating system, which is typically 1-2 GB. The same holds true while deciding on CPU or cores. You need to keep two cores aside in general for handling routine functions, talking with other nodes, NameNode, and so forth. There are some interesting references you may wish to study before taking the call on hardware:

- Hortonworks Cluster Planning Guide (https://docs.hortonworks.com/HDPDocuments/HDP1/HDP-1.3.3/bk_cluster-planning-guide/content/conclusion.html)
- Best practices for selecting Apache Hadoop hardware (<http://hortonworks.com/blog/best-practices-for-selecting-apache-hadoop-hardware/>)
- Cloudera Guide: how to select the right hardware for your new hadoop cluster (<http://blog.cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster/>)



Many times, people do have concerns over whether to go with a few large nodes or many small nodes in a Hadoop cluster. It's a trade-off, and it depends upon various parameters. For example, commercial Cloudera or Hortonworks clusters charge licenses per node. The cost of hardware of a high-end server will be relatively more than having small but many nodes.

Hadoop applications, ports, and URLs

We have gone through various configuration files in Chapter 2, *Planning and Setting Up Hadoop Clusters*, Chapter 3, *Deep Dive into the Hadoop Distributed File System* and Chapter 4, *Developing MapReduce Applications*. When Hadoop is set up, it uses different ports for communication between multiple nodes. It is important to understand which ports are used for what purposes and their default values. In the following table, I have tried to capture this information for all different services that are run as a part of HDFS and MapReduce with old ports (primarily for Hadoop 1.X and 2.X), new port names (for Hadoop 3.x), and protocols for communication. Please note that I am not covering YARN ports; I will cover them in the chapter focused primarily on YARN:

Service	Protocol	Hadoop 1.X, 2.X default ports	Hadoop 3.X default ports	Hadoop 3.x URL
NameNode User Interface	HTTP	50070	9870	http://:9870/
NameNode secured User Interface	HTTPS	50470	9871	https://:9870/
DataNode User Interface	HTTP	50075	9864	http://:9864
DataNode secured User Interface	HTTPS	50475	9865	https://:9865
Resource Manager User Interface	HTTP	8032	8088	http://:8088/
Secondary NameNode User Interface	HTTP	50090	9868	
MapReduce Job History Server UI	HTTP	51111	19888	http://:19888

MapReduce Job History Server secured UI	HTTPS	51112	19890	https://:19890
MapReduce Job History administration IPC port	IPC	NA	10033	http://:10033
NameNode metadata service	IPC	8020	9820	
Secondary NameNode	IPC	50091	9869	
DataNode metadata service	IPC	50020	9867	
DataNode data transfer service	IPC	50010	9866	
KMS service	kms	16000	9600	
MapReduce Job History service	IPC	NA	10020	

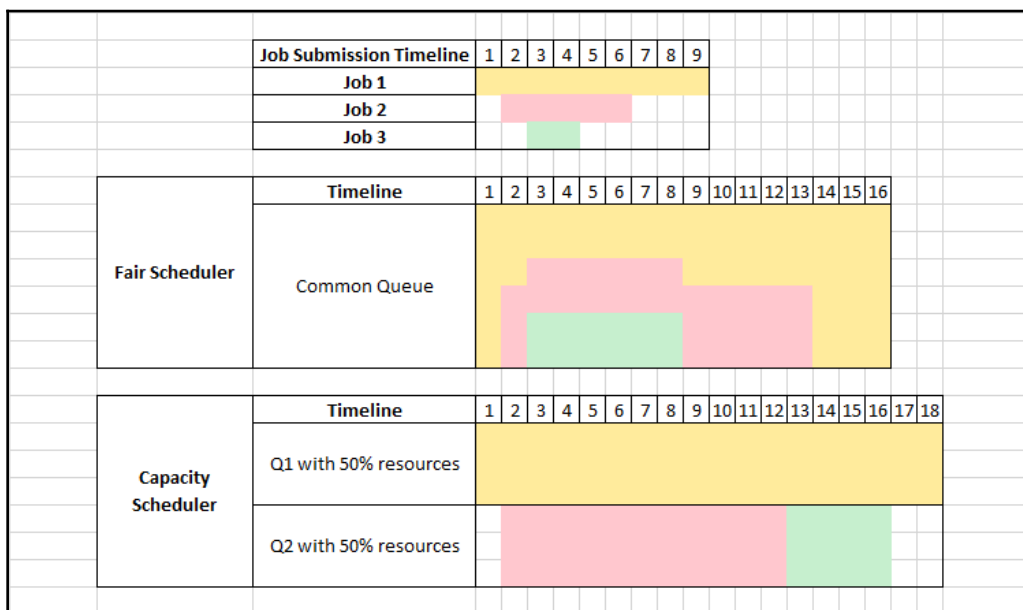
Apache Hadoop provides **Key Management Service (KMS)** for securing interaction with Hadoop RESTful APIs. KMS enables client to communicate over HTTPS and Kerberos to ensure a secured communication channel between client and server.

Resource management in Hadoop

As a Hadoop administrator, one important activity that you need to do is to ensure that all of the resources are used in the most optimal manner inside the cluster. When I refer to a resource, I mean the CPU time, the memory allocated to jobs, the network bandwidth utilization, and storage space consumed. Administrators can achieve that by balancing workloads on the jobs that are running in the cluster environment. When a cluster is set up, it may run different types of jobs, requiring different levels of time- and complexity-based SLAs. Fortunately, Apache Hadoop provides a built-in scheduler for scheduling jobs to allow administrators to prioritize different jobs as per the SLAs defined. So, overall resources can be managed by resource scheduling. All schedulers used in Hadoop use job queues to line up the jobs for prioritization. Among all, the following types of job scheduler are mostly used by Hadoop implementations:

- Fair Scheduler
- Capacity Scheduler

Let's look at an example now to understand these scheduler is better. Let's assume that there are three jobs, with **Job 1** requiring nine units of dedicated time to complete, **Job 2** requiring five units, and **Job 3** requiring two units. Let's say Job 1 arrived at the time **T1**, **Job 2** arrived at **T2**, and **Job 3** arrived at **T3**. The following diagram shows the work distribution done by both of the schedulers:



Now let's understand these in more detail.

Fair Scheduler

As the name suggests, Fair Scheduler is designed to provide each user with an equal share of all of the cluster resources. In this context, a resource is CPU time, GPU time, or memory required for a job to run. So, each job submitted to this Scheduler makes progress periodically with an equal share or average resource sharing. The sharing of resources is not based on the number of jobs, but on the number of users. So, if User A has submitted 20 jobs and User B has submitted two jobs, the probability of User B finishing their jobs is higher, because of the fair distribution of resources done at user level. Fair Scheduler allows the creation of queues, which can have resource allocation. Now, each queue applies the FIFO policy and resources are shared among all of the applications submitted to that queue.

To enable Fair Scheduler, you need to add the following lines to `yarn-site.xml`:

```
<property>
  <name>yarn.resourcemanager.scheduler.class</name>
  <value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler</value></property>
```

Once this is added, you can set various properties to configure your Scheduler to meet your needs. The following are some of the key properties:

Property	Description
<code>yarn.scheduler.fair.preemption</code>	Preemption allows the Scheduler to kill the tasks of the pool that is running over capability to give a fair share to the pool that is running under capability. The default is <code>false</code> .
<code>yarn.scheduler.fair.allocation.file</code>	A pointer to a file where the queue and its specification is described. The default is <code>fair-scheduler.xml</code> .

You can find out more details about Fair Scheduler such as configuration and files [here](#).

The **benefits** of Fair Scheduler are as follows:

- It's good for cases where you do not have any predictability of a job, as it allocates a fair share of resources as and when a job is received
- You do not run into a problem of starvation, due to fairness in scheduling

Capacity Scheduler

Given that organizations can run multiple clusters, Capacity Scheduler uses a different approach. Instead of a fair distribution of resources across users, it allows administrators to allocate resources to queues, which can then be distributed among tenants of the queues. The objective here is to enable multiple users of the organization to share the resources among each other in a predictable manner. This means that bad resource allocation for a queue can result in an imbalance of resources, where some users are starving for resources, while others are enjoying excessive resource allocation. The scheduler then offers elasticity, where it automatically transfers resources across queues to ensure a balance. Capacity Scheduler supports a hierarchical queue structure.

The following is a screenshot of Hadoop administration Capacity Scheduler, which you can access at <http://<host>:8088/cluster/scheduler>:

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
0	0	0	0	0	0 B	11 GB	0 B	0	6	0	1	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation
Capacity Scheduler	[MEMORY]	<memory 1024, vCores 1>	<memory 11264, vCores 6>

Dump scheduler logs 10 min

Application Queues

Legend: Capacity Used Used (over capacity) Max Capacity

Queue: root 0.0% used

Queue: default 0.0% used

'default' Queue Status

Queue State: RUNNING

Used Capacity: 0.0%

Configured Capacity: 100.0%

Configured Max Capacity: 100.0%

Absolute Used Capacity: 0.0%

Absolute Configured Capacity: 100.0%

Absolute Configured Max Capacity: 100.0%

Used Resources: <memory:0, vCores:0>

Configured Max Application Master Limit: 20.0

Max Application Master Resources: <memory:3072, vCores:1>

Used Application Master Resources: <memory:0, vCores:0>

Max Application Master Resources Per User: <memory:3072, vCores:1>

Num Schedulable Applications: 0

Num Non-Schedulable Applications: 0

Num Containers: 0

Max Applications: 10000

As you can see, on top of all queues, there is a default queue, and then users can have their queues below as a subset of the default queue. Capacity Scheduler has a predefined queue called **root**. All queues in the system are children of the root queue.

To enable Capacity Scheduler, you need to add following lines to `yarn-site.xml`:

```
<property>
  <name>yarn.resourcemanager.scheduler.class</name>
  <value>
    org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacitySc
    heduler</value>
</property>
```

You can specify the queue-related information at `$HADOOP_HOME/etc/hadoop/capacity-scheduler.xml`, which is the configuration file for Capacity Scheduler. For more information about configuring this queue, please refer to the Apache documentation [here](#) about Capacity Scheduler.

One of the **benefits** of Capacity Scheduler is that it's useful when you have planned jobs, with more predictability over resource requirements. This can give a better optimization of the cluster.

High availability of Hadoop

We have seen the architecture of Apache Hadoop in a [Chapter 1, Hadoop 3.0 - Background and Introduction](#). In this section, we will go through the High Availability (HA) feature of Apache Hadoop, given the fact that HDFS supports high availability through its replication factor. However, in earlier Apache Hadoop 1.X, NameNode was the single point of failure due to it being a central gateway for accessing data blocks. Similarly, Resource Manager is responsible for managing resources for MapReduce and YARN applications. We will study both of these points with respect to high availability.

High availability for NameNode

We have understood the challenges faced with Hadoop 1.x, so now let's understand the challenges we see today with respect to Hadoop 2.0 or 3.0 for high availability. The presence of secondary NameNode being present or multiple name nodes in a hadoop cluster does not ensure high availability. That is because, when a name node goes down, the next candidate name node needs to become active from its passive mode.

This may require a significant downtime when a cluster size is large. In Hadoop 2.x onward, the new feature of high availability of name node was introduced. So, in this case, multiple name nodes can work in active-standby mode instead of active-passive mode. So, when a primary name node goes down, the other candidate can quickly assume its role. To enable HA, you need to have the following configuration snippet in `hdfs-site.xml`:

```
<property>
  <name>dfs.nameservices</name>
  <value>hkcluster</value>
</property>
```

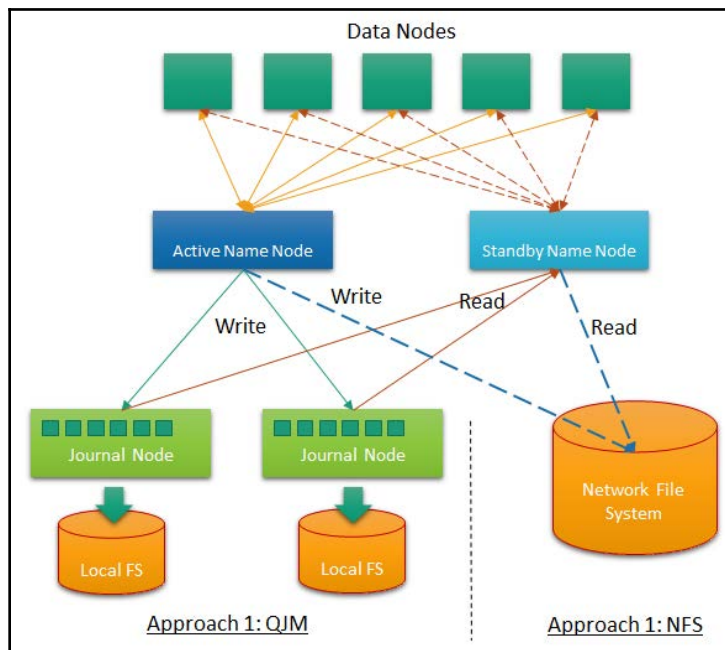
In a typical HA environment, there are at least three nodes participating in high availability and durability. The first node is NameNode in active state; the second is secondary name node, which remains in a passive state; and the third name node is in standby phase. This ensures high availability along with data consistency. You can support multiple name nodes by adding the following XML snippet in `hdfs-site.xml`:

```
<property>
  <name>dfs.ha.namenodes.mycluster</name>
  <value>nn1,nn2, nn3</value>
</property>
```

To have a shared data structure between active and standby name nodes, we have the following approaches:

- Quorum Journal Manager
- Network Filesystem

Both approaches can be seen in the following architecture:



There is an interesting article about how a name node failover process happens [here](#). In the case of the **Query Journal Manager (QJM)**, the name node communicates with process daemons called journal nodes. The active name node performs sends write commands to these journal nodes where the logs of the edit are pushed. At the same time, the standby node performs the read to keep its `fsimage` and edit logs in sync with the primary name node. There must be at least three journal node daemons available for name nodes to write the logs. Apache Hadoop provides a CLI for managing name node transitions and complete HA for QJM; you can read more about it [here](#).

Network Filesystem (NFS) is a standard Unix file sharing mechanism. The first activity that you need to do is set up an NFS, and mount it on a shared folder where the active and standby NameNodes can share data. You can do NFS setup by following the standard Linux guide—one example is [here](#). Through NFS, the need to sync the logs between both name nodes goes away. You can read more about NFS-based high availability [here](#).

High availability for Resource Manager

Just like NameNode being a single point of failure, Resource Manager is also a crucial part of Apache Hadoop. Resource Manager is responsible for keeping track of all resources in the system and scheduling of the application. We have seen resource management and different scheduling algorithms in previous sections. Resource Manager is a critical application in terms of day-to-day process execution, and it used to be a single point of failure before the hadoop 2.4 release.

With newer hadoop, Resource Manager supports the high availability function through the active-standby state. The resource metadata sync is achieved through Apache Zookeeper, which acts as a shared metadata store for all of Resource Manager's database. At any point, only one Resource Manager is active in the cluster and the rest all work in standby mode. The active Resource Manager has a responsibility to push its state, and other related information, to Zookeeper, which other Resource Managers read through.

Resource Manager supports automatic transition to the standby Resource Manager through its automatic failure feature. You can enable high availability of Resource Manager by setting the following property to `true` in `yarn-site.xml`:

```
<property>
  <name>yarn.resourcemanager.ha.enabled</name>
  <value>true</value>
</property>
```

Additionally, you need to specify the order for active and standby Resource Managers by passing comma-separated IDs to the `yarn.resourcemanager.ha.rm-ids` property. However, do remember to set the right hostname through the `yarn.resourcemanager.hostname.rm1` property. You also need to point to Zookeeper Quorum in the `yarn.resourcemanager.zk-address` property. In addition to configuration, the Resource Manager CLI also provides some commands for HA. You can read more about them here (<https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html>).

Securing Hadoop clusters

Since Apache Hadoop works with lots of information, it brings in the important aspect of data governance and security of information. Usually, the cluster is not visible directly and is used primarily for computation and historical data storage, hence the urge for security implementation is relatively less than with applications that are running over the web, which demand the highest level of security requirements to be addressed. However, should there be any need, Hadoop deployments can be extremely secure. The security in hadoop works in the following key areas:

- **Data at Rest:** How data stored can be encrypted so that no one can read it
- **Data in Motion:** How the data transferred over the wire can be encrypted
- Secured System access/APIs
- **Data Confidentiality:** to control data access across different users

The good part is, Apache Hadoop ecosystem components such as YARN, HDFS, and MapReduce can be separated and set up by different users/groups, which ensures separation of concerns.

Securing your Hadoop application

Data in motion and API access can be secured with SSL-based security over a digital certification. The Hadoop SSL Keystore Factory manages SSL for core services that communicate with other cluster services over HTTP, such as MapReduce, YARN, and HDFS. Hadoop provides its own built-in Key Management Server (KMS) to manage keys in Hadoop.

The following services support SSL configuration:

- Web HDFS
- TaskTracker
- Resource Manager
- Job History

The digital certificates can be managed using the standard Java key store or by the Hadoop Key Store Management Factory. You need to either create a certificate first or obtain it from a third-party vendor such as CA. Once you have the certificate, you need to upload it to the key store you intend to use for storing the keys. SSL can be enabled one-way or two-way. One-way is when a client validates the server identity, whereas in two-way, both parties validate each other. Please note that with two-way SSL, the performance may get impacted. To enable SSL, you need to modify the config files to start using the new certificate. You can read more about the HTTPS configuration in the Apache documentation here (<https://hadoop.apache.org/docs/r3.1.0/hadoop-hdfs-httpfs/ServerSetup.html>). In addition to digital signature, Apache Hadoop also switches in completely secured mode and all users connecting to the system must be authenticated using Kerberos. A secured mode can be achieved with authentication and authorization. You can read more about securing Hadoop through the standard documentation here (<http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/SecureMode.html>).

Securing your data in HDFS

With older Hadoop, the security in HDFS followed Linux-/Unix-style security, using permissions to files. However, with ACLs, access to the files is provided to three classes of users: Owner, Group, and Others, as well as three classes of permissions: read, write, and execute. When you wish to give access to a certain folder to a group that is not an owner's group, you cannot specifically do that in a traditional Linux system. You will end up creating a dummy user and group and so forth. HDFS has solved this problem through ACLs. So, it allows you to grant access to another group with the following command:

```
hrishikesh@base0:/$ hdfs dfs -setfacl -m group:departmentabcgroup:rwX /user/hrishi/departmentabc
```

Please note that, before you start using ACLs, you need to enable the functionality by setting the `dfs.namenode.acls.enabled` property in `hdfs-site.xml` to `true`. Similarly, you can get ACL information about any folder/file by calling the following command:

```
hrishikesh@base0:/$ hdfs dfs -getfacl /user/hrishi/departmentabc
# file: /user/hrishi/departmentabc
# owner: hrishi
# group: mygroup
user::rwgroup::r--
group:departmentabcgroup:rwX
mask::r--
other::---
```

To know more about ACLs in Hadoop, please visit Apache's documentation on ACLs [here](#).

Performing routine tasks

As a Hadoop administrator, you must work on your routine activities. Let's go through some of the most common routine tasks that you would perform with Hadoop administration.

Working with safe mode

When any client performs a write operation on HDFS, the changes get recorded in the edit log. This edit log is flushed at the end of write operations and the information is synced across nodes. Once this operation is complete, the system returns a success flag to the client. This ensures consistency of data and cleaner operation execution. Similarly, name node maintains a `fsimage` file, which is a data structure that name node uses to keep track of what goes where. This is a checkpoint copy which is preserved on a disk. If name node crashes or fails, the disk image can be used to recover name node back to a given checkpoint. Similarly, when name node starts, it loads `fsimage` in memory for quick access. Since `fsimage` is a checkpoint, it applies `editlog` changes to get the recent state back and, when it has reconstructed a new `fsimage` file, it again persists it back to disk. During this time, Hadoop runs in safe mode. Safe mode is exited when the minimal replication condition is reached, plus an extension time of 30 seconds. You can check whether a system is in safe mode or not with the following command:

```
hrishikesh@base0:/$ hdfs dfsadmin -safemode get
```

Similarly, the administrator can decide to put HDFS in safe mode by explicitly calling it, as follows:

```
hrishikesh@base0:/$ hdfs dfsadmin -safemode enter
```

This is useful when you wish to do maintenance or upgrade your cluster. Once the activities are complete, you can leave the safe mode by calling the following:

```
hrishikesh@base0:/$ hdfs dfsadmin -safemode leave
```



You can prevent accidental deletion of files on HDFS by enabling the trash feature of HDFS. In `core-site.xml`, you can specify the `hadoop.shell.safely.delete.limit.num.files` property to some number. When users run `hdfs dfs rm -r` or any other command, the system will check if the number of files exceeds the value set in the `hadoop.shell.safely.delete.limit.num.files` property. If it does, it will introduce an additional prompt.

Archiving in Hadoop

In Chapter 3, *Deep Dive into the Hadoop Distributed File System* we already studied how we can solve the problem of storing multiple small files that are less than the HDFS block size. In addition to the sequential file approach, you can also use the Hadoop Archives (HAR) mechanism to store multiple small files together. Hadoop archive files will always have the `.har` extension. Each hadoop archive holds index information and multiple parts of that file. HDFS provides the `HarFileSystem` class to work on HAR files. Hadoop Archive can be created with the archiving tool from the command-line interface of hadoop. To create an archive across multiple files, use the following command:

```
hrishikesh@base0:/$ hadoop archive -archiveName myfile.har -p /user/hrishi
foo.doc foo1.doc foo2.xls /user/hrishi/data/
```

The format for the archive is as follows:

```
hadoop archive -archiveName name -p <parent> <src>* <dest>
```

The tool uses MapReduce efficiently to split the job and create metadata and archive parts. Similarly, you can perform a lookup by calling the following command:

```
hdfs dfs -ls har:///user/hrishi/data/myfile.har/
```

It returns the list of files/folders that are part of your archive, as follows:

```
har:///user/zoo/foo.har/foo.doc
har:///user/zoo/foo.har/foo1.doc
har:///user/zoo/foo.har/foo2.xls
```

Commissioning and decommissioning of nodes

As an administrator, the commission and decommission of hadoop nodes becomes a usual practice, for example, if your organization is growing, you need to add more nodes to your cluster to meet the SLAs or, sometimes due to maintenance activity, you may need to take down a certain node. One important aspect is to govern this activity across your cluster, which may be running hundreds of nodes. This can be achieved through a single file, which can maintain the list of hadoop nodes that are actively participating in the cluster.

Before you commission a node, you will need to copy the `hadoop` folder to ensure all configuration is reflected in the new node. Now, the next step is to let your existing cluster recognize the new node as an addition. To achieve that, first, you will be required to add a governance property to explicitly state the inclusion of nodes through files for HDFS and YARN. So simply edit `hdfs-site.xml` and add the following file property:

```
<property>
  <name>dfs.hosts</name>
  <value><hadoop-home>/etc/hadoop/conf/includes</value>
</property>
```

Similarly, you need to edit `yarn-site.xml` and point to the that which will maintain the list of nodes that are participating in the given cluster:

```
<property>
  <name>yarn.resourcemanager.nodes.include-path</name>
  <value><hadoop-home>/etc/hadoop/conf/includes</value>
</property>
```

Once this is complete, you may need to restart the cluster once. Now, you can edit the `<hadoop-home>/etc/hadoop/conf/includes` file and add the nodes you wish to be part of the hadoop cluster. You need to add the IP address of these nodes. Now, run the following refresh command to let it take effect:

```
hrishikesh@base0:/$ hdfs dfsadmin -refreshNodes
Refresh nodes successful
```

And for YARN, run the following:

```
hrishikesh@base0:/$ yarn rmadmin -refreshNodes
18/09/12 00:00:58 INFO client.RMPProxy: Connecting to ResourceManager at
/0.0.0.0:8033
```

Please note that, similar to include files, Hadoop also gives the exclude mechanism. The `dfs.hosts.exclude` property in `hdfs-site.xml` and `yarn.resourcemanager.nodes.exclude-path` in `yarn-site.xml` can be set for exclusion or decommissioning. These properties can point to excludes file.

Apache Hadoop also provides a balancer utility to ensure that no node is over-utilized. When you run the balancer, the utility will work on your data nodes to ensure uniform distribution of your data blocks across HDFS data nodes. Since this utility does migration of data blocks across different nodes, it can impact day-to-day work, hence it is recommended to run this utility during off hours. You can simply run it with the following command:

```
hrishikesh@base0:/$ hadoop balancer
```

Working with Hadoop Metric

Regular monitoring activity for Apache Hadoop requires sufficient data points to be made available to the administrator to identify the potential risks or challenges to the cluster. Fortunately, Apache Hadoop has done a phenomenal job by introducing Metric to various processes and flows of the Apache Hadoop ecosystem. Metric provides real-time, as well as statistical, information about various performance indices of your cluster. This can serve as activity monitoring capability to your administration tools such as Nagios, Ganglia, or Apache Ambari. The latest version of Hadoop uses the newer version of Metrics called 2.0. This can be compared with counters provided by MapReduce application. However, one key difference to note here is that Metric is designed to provide assistance to administrators whereas counters provide specific information to MapReduce developers. The following are the areas where Metric is provided:

Area	Description
Java Virtual Machine	All of Hadoop runs on JVM. This Metric provides important information such as heap size, thread state, and GC.
Remote Procedure Calls	Provides information such as processes tracking, RPC connections, and queues for processing.
NameNode cache	As the name suggests, it provide retrycache information. It's useful for name node failover.
DFS.namenode	Provides all of the information on namenode operations.
DFS.FSNamesystem	Provides information on high availability, snapshots, edit logs, and so on.
DFS.JournalNode	Provides statistics about journal node operations.

DFS.datanode	Statistics about all data node operations.
DFS.FSVolume	Provides statistics about volume information, I/O rates, flush rates, write rates, and so on.
DFS.RouterRPCMetric	Provides various statistical information about router operations, requests, and failed status.
DFS.StateStoreMetric	Provides statistics about transaction information on the state store (GET, PUT, and REMOVE transactions).
YARN.ClusterMetrics	Statistics pertaining to node managers, heartbeats, application managers, and so on.
YARN.QueueMetrics	Statistics pertaining to application states and resources such as CPU and memory.
YARN.NodeManagerMetrics	As the name suggests, it provides statistics pertaining to the containers and cores of node managers.
YARN.ContainerMetrics	Provides statistics about memory usage, container states, CPU, and core usages.
UGI.ugiMetrics	Provides statistics pertaining to users and groups, failed logins, and so on.
MetricsSystem	Provides statistics about the Metrics system itself.
StartupProgress	Provides statistics about name node startup.

The Metric system works on producer consumer logic. The producer registers with the Metric as source, as shown in the following Java code:

```
class TestSource implements MetricsSource {
    @Override
    public void getMetrics(MetricsCollector collector, boolean all) {
        collector.addRecord("TestSource")
            .setContext("TestContext")
            .addGauge(info("CustomMetric", "Description"), 1);
    }
}
```

Similarly, consumers too can register for a sink, where it can be passed on to a third-party analytical tool for analytics (in this case I am simply printing it):

```
public class TestSink implements MetricsSink {
    public void putMetrics(MetricsRecord record) {
        //print the output
        System.out.print(record);
    }
    public void init(SubsetConfiguration conf) {}
    public void flush() {}
}
```

This can be achieved through Java annotations too. Now you can register your Metrics with the Metric system, as shown in the following Java code:

```
DefaultMetricsSystem.initialize("datanode1");
MetricsSystem.register(source1, mysource description", new TestSource());
MetricsSystem.register(sink2, mysink description", new TestSink())
```

Once you are done with it, you can specify the sink information in the config file for Metric: `hadoop-metrics2-test.properties`. You are good to track Metric information now.

You can go to the Hadoop Metric API documentation [here](http://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/metrics2/package-summary.html) to read through more information (<http://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/metrics2/package-summary.html>).

Summary

In this chapter, we have gone through different activities performed by Hadoop administrators for monitoring and optimizing the Hadoop cluster. We looked at the roles and responsibilities of an administrator, followed by cluster planning. We did a deep dive into key management aspects of the hadoop cluster, such as resource management through job scheduling with algorithms such as Fair Scheduler and Capacity Scheduler. We also looked at ensuring high availability and security for the Apache hadoop cluster. This was followed by the day-to-day activities of Hadoop administrators, covering adding new nodes, archiving, hadoop Metric, and so on.

In the next chapter, we will look at Hadoop ecosystem components, which help the business develop big data applications rapidly.

7

Demystifying Hadoop Ecosystem Components

We have gone through the Apache Hadoop subsystem in detail in previous chapters. Although Hadoop is extensively known for its core components such as HDFS, MapReduce and YARN, it also offers a whole ecosystem that is supported by various components to ensure all your business needs are addressed end-to-end. One key reason behind this evolution is because Hadoop's core components offer processing and storage in a raw form, which requires an extensive amount of investment when building software from a grass-roots level.

The ecosystem components on top of Hadoop can therefore provide the rapid development of applications, ensuring better fault-tolerance, security, and performance over custom development done on Hadoop.

In this chapter, we cover the following topics:

- Understanding Hadoop's Ecosystem
- Working with Apache Kafka
- Writing Apache Pig scripts
- Transferring data with Sqoop
- Writing Flume jobs
- Understanding Hive as big data RDBMS
- Using HBase as NoSQL storage

Technical requirements

You will need Eclipse development environment and Java 8 installed on your system where you can run/tweak these examples. If you prefer to use maven, then you will need maven installed to compile the code. To run the example, you also need Apache Hadoop 3.1 setup on Linux system. Finally, to use the Git repository of this book, you need to install Git.

The code files of this chapter can be found on GitHub:

<https://github.com/PacktPublishing/Apache-Hadoop-3-Quick-Start-Guide/tree/master/Chapter7>

Check out the following video to see the code in action:

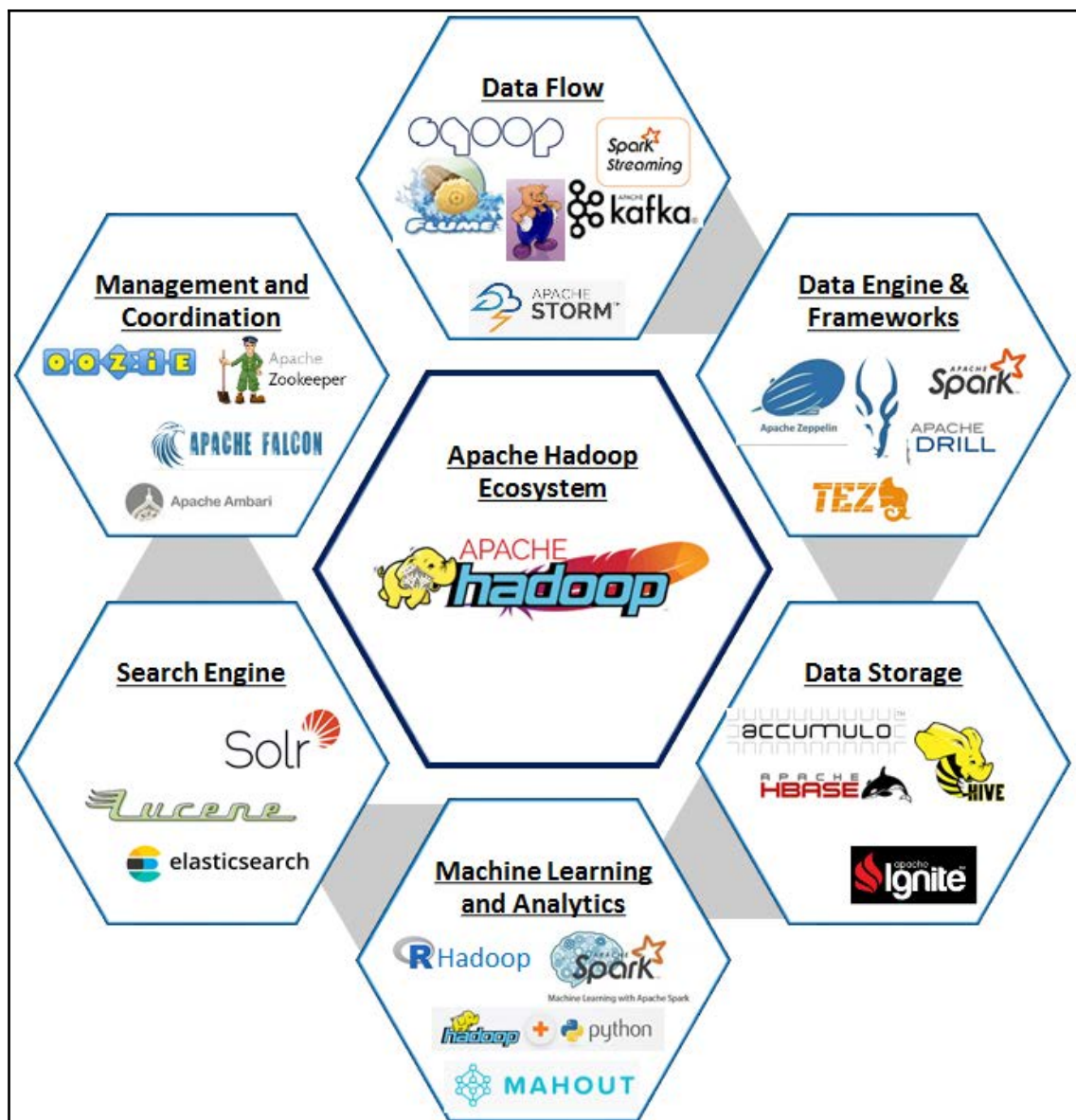
<http://bit.ly/2SBdnr4>

Understanding Hadoop's Ecosystem

Hadoop is often used for historical data analytics, although a new trend is emerging where it is used for real-time data streaming as well. Considering the offerings of Hadoop's ecosystem, we have broadly categorized them into the following categories:

- **Data flow:** This includes components that can transfer data to and from different subsystems to and from Hadoop including real-time, batch, micro-batching, and event-driven data processing.
- **Data engine and frameworks:** This provides programming capabilities on top of Hadoop YARN or MapReduce.
- **Data storage:** This category covers all types of data storage on top of HDFS.
- **Machine learning and analytics:** This category covers big data analytics and machine learning on top of Apache Hadoop.
- **Search engine:** This category covers search engines in both structured and unstructured Hadoop data.
- **Management and coordination:** This category covers all the tools and software used to manage and monitor your Hadoop cluster, and ensures coordination among the multiple nodes of your cluster.

The following diagram lists software for each of the previously discussed categories. Please note that, in keeping with the scope of this book, we have primarily considered the most commonly used open source software initiatives as depicted in the following graphic:



As you can see, in each area, there are different alternatives available; however, the features of each piece of software differ and so do their applicability. For example, in Data Flow, Sqoop is more focused towards RDBMS data transfer, whereas Flume is intended for log data transfer.

Let's walk through these components briefly with the following table:

Component	Description	Link to software
Apache Ignite	Apache Ignite is an in-memory-based database and caching platform.	https://ignite.apache.org/
Apache Tez	Apache Tez provides a flexible programming framework on YARN for users to run their jobs into multiple, directed acyclic, graph-driven tasks. It offers power and flexibility to end users, and better performance overall compared to the traditional MapReduce.	https://tez.apache.org/
Apache Kafka	Kafka offers a distributed streaming mechanism through its queues for Hadoop and non-Hadoop systems.	https://kafka.apache.org/
Apache Sqoop	Apache Sqoop is an ETL tool designed to efficiently transfer RDBMS bulk data to and from Hadoop.	https://sqoop.apache.org/
Apache Flume	Flume offers a mechanism to collect, aggregate, and transfer large amounts of unstructured data to and from Hadoop (usually log files).	https://flume.apache.org/
Apache Spark	Apache Spark provides two key aspects: analytics through Spark ML and streaming capabilities for data through Spark Streaming. Additionally, it also provides programming capabilities on top of YARN.	https://spark.apache.org/
Apache Storm	Apache Storm provides a streaming pipeline on top of YARN for all real-time data processing on Hadoop.	https://storm.apache.org/
Apache Pig	Apache Pig provides expression language for analyzing large amounts of data across Hadoop.	https://pig.apache.org/
Apache Hive	Apache Hive offers RDBMS capabilities on top of HDFS.	https://hive.apache.org/
Apache HBase	Apache HBase is a distributed key-value-based NoSQL storage mechanism on HDFS.	https://hbase.apache.org/
Apache Drill	Apache Drill offers schema-free SQL engine capabilities on top of Hadoop and other subsystems.	https://drill.apache.org/

Apache Impala	Apache Impala is an open source and parallel-processing SQL engine used across a Hadoop cluster.	https://impala.apache.org/
Apache Mahout	Apache Mahout offers a framework to build and run algorithms from ML and linear algebra on a Hadoop cluster.	https://mahout.apache.org/
Apache Zeppelin	Apache Zeppelin provides a framework for developers to write data analytics programs through its notebook and then run them.	https://zeppelin.apache.org/
Apache Oozie	Apache Oozie provides a workflow scheduler on top of Hadoop for running and controlling jobs.	http://oozie.apache.org/
Apache Ambari	Apache Ambari provides the capability to completely manage and monitor the Apache Hadoop cluster.	https://ambari.apache.org
Apache Zookeeper	Apache Zookeeper offers a distributed coordination system across multiple nodes of Hadoop; it also offers metadata sharing storage.	https://zookeeper.apache.org/
Apache Falcon	Apache Falcon provides a data-processing platform for extracting, correlating, and analyzing data on top of Hadoop.	https://falcon.apache.org/
Apache Accumulo	Accumulo is a distributed key-value store based on Google's big table design built on top of Apache Hadoop.	https://accumulo.apache.org
Lucene-Solr	Apache Lucene and Apache Solr provide search engine APIs and applications for large data processing. Although they do not run on Apache Hadoop, they are aligned with the overall ecosystem to provide search support.	http://lucene.apache.org/solr/

There are three pieces of software that are not listed in the preceding table; they are R Hadoop, Python Hadoop/Spark, and Elastic Search. Although they do not belong to the Apache Software Foundation, R and Python are well-known in the data analytics world. Elastic Search (now Elastic) is a well-known search engine that can run on HDFS-based data sources.

In addition to the listed Hadoop ecosystem components, we have also shortlisted another set of Hadoop ecosystems that are part of the Apache Software Foundation in the following table. Some of them are still incubating in Apache Lab, but it is still useful to understand the new capabilities and features they can offer:

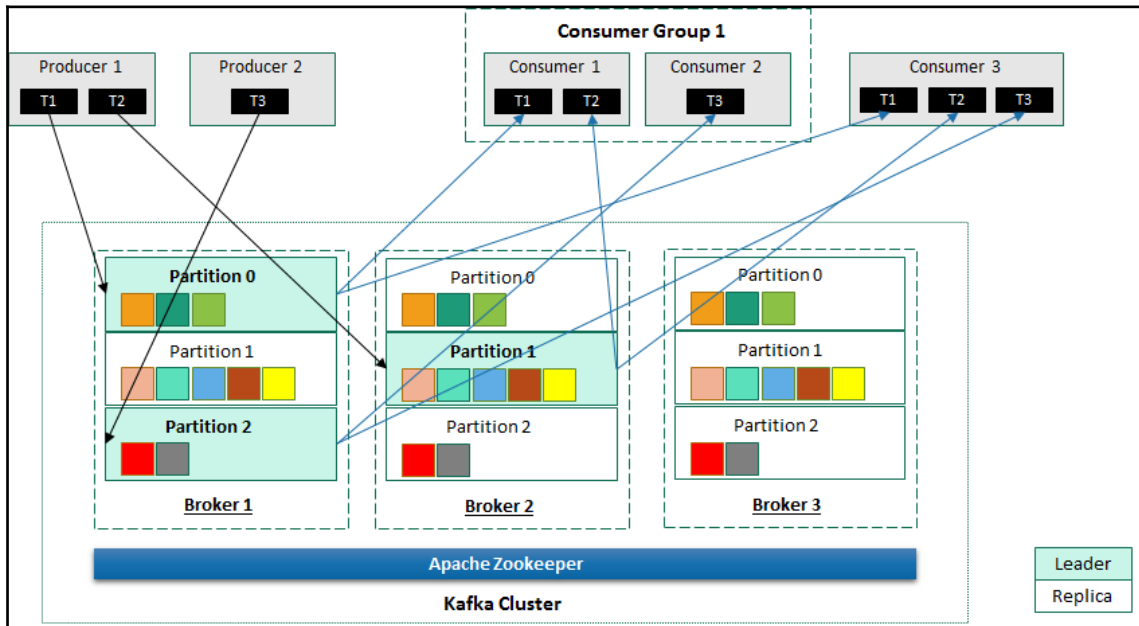
Component	Description	Link to software
Apache Parquet	Apache Parquet is a file storage format on top of HDFS that we will see in next chapter. It provides columnar storage.	http://parquet.apache.org/
Apache ORC	Apache ORC provides columnar storage on Hadoop. We will study ORC files in next chapter.	https://orc.apache.org/
Apache Crunch	Apache Crunch provides a Java library framework to code MapReduce-based pipelines, which can be efficiently written through user-defined functions.	http://crunch.apache.org/
Apache Kudu	Kudu provides a common storage layer on top of HDFS to enable applications to perform faster inserts and updates, as well as analytics on continuously changing data.	https://kudu.apache.org/
Apache Metamodel	MetaModel provides an abstraction of metadata on top of various databases through a standard mechanism. It also enables the discovery of metadata along with querying capabilities.	http://metamodel.apache.org/
Apache BigTop	Apache BigTop provides a common packaging mechanism across different components of Hadoop. It also provides the testing and configuration of these components.	http://bigtop.apache.org/
Apache Apex	Apache Apex provides streaming and batch processing support on top of YARN for data-in-motion form. It is designed to support fault-tolerance and works across a secure distributed platform.	http://apex.apache.org/
Apache Lens	Apache Lens provides OLAP-like query capabilities through its unified common analytics interface on top of Hadoop and a traditional database.	http://lens.apache.org/
Apache Fluo	Apache Fluo provides a workflow-management capability on top of Apache Accumulo for the processing of large data across multiple systems.	https://fluo.apache.org/
Apache Phoenix	Apache Phoenix provides OLTP-based analytical capabilities on Hadoop, using Apache Hbase as storage. It has RDBMS on Hbase.	http://phoenix.apache.org/

Apache Tajo	Apache Tajo provides a data warehouse on top of Hadoop and also supports SQL capabilities for interactive and batch queries.	http://tajo.apache.org/
Apache Flink	Apache Flink is an in-memory distributed processing framework on unbounded and bounded data streams.	https://flink.apache.org/
Apache Drill	Apache Drill provides an SQL query wrapper on top of NoSQL databases of Hadoop (such as Hbase).	http://drill.apache.org/
Apache Knox	Apache Knox provides a common REST API gateway to interact with the Hadoop cluster.	http://knox.apache.org/
Apache Trafodion	Apache Trafodion provides transactional SQL database capabilities on top of Hadoop. It is built on top of Apache Hive-Hcatalog.	http://trafodion.apache.org
Apache REEF	Apache REEF provides a framework library for building portable applications across Apache YARN.	http://reef.apache.org/

Working with Apache Kafka

Apache Kafka provides a data streaming pipeline across the cluster through its message service. It ensures a high degree of fault tolerance and message reliability through its architecture, and it also guarantees to maintain message ordering from a producer. A record in Kafka is a (key-value) pair along with a timestamp and it usually contains a topic name. A topic is a category of records on which the communication takes place.

Kafka supports producer-consumer-based messaging, which means producers can produce messages that can be sent to consumers. It maintains a queue of messages, where there is also an offset that represents its position or index. Kafka can be deployed on a multi-node cluster, as shown in the following diagram, where two producers and three consumers have been used as an example:



Producers produce multiple topics through producer APIs (<http://kafka.apache.org/documentation.html#producerapi>). When you configure Kafka, you need to set the replication factor, which ensures data loss is minimal. Each topic is then allocated to a partition, as shown in the preceding diagram. The partitions are replicated across brokers to ensure message reliability. There is a leader among partitions, which works as a primary partition, whereas all other partitions are replicated. A new leader will be selected when the existing leader goes down. Unlike other messaging, all Kafka messages are written on disk to ensure high durability, and are only made accessible or shared with consumers once recorded.

Kafka supports both queuing and publish-subscribe. In the queuing technique, consumers continuously listen to queues, whereas during publish-subscribe, records are published to various consumers. Kafka also supports consumer groups where one or more consumers can be combined, thereby reducing unnecessary data transfer.

You can run Kafka server by calling the following command:

```
$KAFKA_HOME/ bin/kafka-server-start.sh config/server.properties
```

The `server.properties` file contains information such as the broker name, listener port, and so on. Apache Kafka provides a utility named `kafka-topic`, which is located in `$KAFKA_HOME/bin`. This utility can be used for all Kafka-topic-related work.

First, you need to create a new topic so that messages between producers and consumers can be exchanged; in the following snippet, we are creating a topic with the name `my_topic` on Kafka and with a replication factor of 3.

```
$KAFKA_HOME/bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic  
my_topic --replication-factor 3
```

Please note that a Zookeeper port is required, as Zookeeper is a primary coordinator for the Kafka cluster. You can also list all topics on Kafka by calling the following command:

```
$KAFKA_HOME /bin/kafka-topics.sh --list --zookeeper localhost:2181 .
```

Let's now write a simple Java code to produce and consume the Kafka queue on a given host. First, let's add a Maven dependency to the client APIs with the following:

```
<dependency>  
  <groupId>org.apache.kafka</groupId>  
  <artifactId>kafka-clients</artifactId>  
  <version>2.0.0</version>  
</dependency>
```

Now let's write a Java code to produce some text, for example a key and a value. The producer requires that properties are set ahead of the client connecting to the server, and include the client ID, as follows:

```
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class
.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
Producer<String, String> producer = new KafkaProducer<String,
String>(props);
producer.send(
    new ProducerRecord<String, String>("my_topic", "myKey", "myValue"));
producer.close();
```

In this case, `BOOTSTRAP_SERVERS_CONFIG` is a list of URLs that is needed to establish a connection to the Kafka cluster. Now let's look at the following consumer code:

```
Properties consumerConfig = new Properties();
consumerConfig.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
consumerConfig.put(ConsumerConfig.GROUP_ID_CONFIG, "my-group");
consumerConfig.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
consumerConfig.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

KafkaConsumer<byte[], byte[]> consumer = new
KafkaConsumer<>(consumerConfig);
consumer.subscribe(Collections.singletonList("my_topic"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        //your logic to process the record
        break;
    }
}
consumer.close();
```

In the preceding code, the consumer performs polling every 100 milliseconds to check if any messages has been produced. The record returns an offset, key, and value along with other attributes that can be used for analyzing. Kafka code can be written in various languages; check out the client code here (<https://cwiki.apache.org/confluence/display/KAFKA/Clients>).

The following table lists the Hadoop components discussed in this book and the key aspects of each one, including the latest release, pre-requisites, supported operating systems, documentation links, install links, and so on.

Software name	Apache Kafka
Latest release	2.0.0
Prerequisites	Zookeeper
Supported OSs	Linux, Windows
Installation instructions	https://kafka.apache.org/quickstart
Overall documentation	https://kafka.apache.org/documentation/
API documentation	http://kafka.apache.org/20/javadoc/index.html?overview-summary.html

Writing Apache Pig scripts

Apache Pig allows users to write custom scripts on top of the MapReduce framework. Pig was founded to offer flexibility in terms of data programming over large data sets and non-Java programmers. Pig can apply multiple transformations on input data in order to produce output on top of a Java virtual machine or an Apache Hadoop multi-node cluster. Pig can be used as a part of ETL (**Extract Transform Load**) implementations for any big data project.

Setting up Apache Pig in your Hadoop environment is relatively easy compared to other software; all you need to do is download the Pig source and build it to a `pig.jar` file, which can be used for your programs. Pig-generated compiled artifacts can be deployed on a standalone JVM, Apache Spark, Apache Tez, and MapReduce, and Pig supports six different execution environments (both local and distributed). The respective environments can be passed as a parameter to Pig using the following command:

```
$PIG_HOME/bin/pig -x spark_local pigfile
```

The preceding command will run the Pig script in the local Spark mode. You can also pass additional parameters such as your script file to run in batch mode.

Scripts can also be run interactively with the Grunt shell, which can be called with the same script, excluding parameters, shown as follows:

```
$ pig -x mapreduce
... - Connecting to ...
grunt>
```

Pig Latin

Pig uses its own language to write data flows called Pig Latin. Pig Latin is a feature-rich expression language that enables developers to perform complex operations such as joins, sorts, and filtering across different types of datasets loaded on Pig. Developers can write scripts in Pig Latin, which then passes through the Pig Latin Compiler to produce a MapReduce job. This is then run on the traditional MapReduce framework across a Hadoop cluster, where the output file is stored in HDFS.

Let's now write a small script for batch processing with the following simple sample of students' grades:

```
2018, John, A
2017, Patrick, C
...
```

Save the file as *student-grades.csv*. You can create a Pig script for a batch run, or you can directly run the file via the Grunt CLI. First, load the file in Pig within a `records` object with the following command:

```
grunt> records = LOAD 'student-grades.csv' USING PigStorage(',')
>> AS (year:int,name:chararray,grade:chararray);
```

Now select all students of the current year who have A grades using the following command:

```
grunt> filtered_records = FILTER records BY year == 2018 AND (grade matches 'A*');
```

Now dump the filtered records to `stdout` with the following command:

```
grunt> DUMP filtered_records;
```

The preceding code should print the filtered records to you. DUMP is a diagnostic tool, so it would fire an execution. There is a nice cheat sheet available for Apache Pig scripts here (<https://www.qubole.com/resources/pig-function-cheat-sheet/>).

User-defined functions (UDFs)

Pig allows users to write custom functions using **User-Defined Functions (UDF)** support. You can write UDF in any language, so looking at a previous example, let's try to create a filter UDF for the following expression:

```
filtered_records = FILTER records BY year == 2018 AND (grade matches 'A*');
```

Remember that when you create a filter UDF, you need to extend the `FilterFunc` class. The code for this custom function can be written as follows:

```
public class CurrentYearMatch extends FilterFunc {
    @Override
    public Boolean exec(Tuple Ftuple) throws IOException {
        if (tuple == null || tuple.size() == 0) {
            return false;
        }
        try {
            Object object = tuple.get(0);
            if (object == null) {
                return false;
            }
            int currentYear = (Integer) object;
            return currentYear == 2018;
        } catch (ExecException e) {
            throw new IOException(e);
        }
    }
}
```

In the preceding code, we first checked if the tuple was valid. (A tuple in Apache Pig is a field.) A record was then formed by an ordered set of fields. We then checked if the value of the tuple matched with the year 2018.

As you can see, Pig's UDFs allow you to run User-Defined Functions for filters, custom evaluations, and custom loading functions. You can read more about UDFs here (<https://pig.apache.org/docs/latest/udf.html>).

The details of Apache Pig are as follows:

Software name	Apache Pig
Latest release	0.17.0
Prerequisites	Hadoop
Supported OSs	Linux
Installation instructions	http://pig.apache.org/docs/r0.17.0/start.html#Pig+Setup
Overall documentation	http://pig.apache.org/docs/r0.17.0/start.html
API documentation	http://pig.apache.org/docs/r0.17.0/func.html http://pig.apache.org/docs/r0.17.0/udf.html http://pig.apache.org/docs/r0.17.0/cmds.html

Transferring data with Sqoop

The beauty of Apache Hadoop lies in its ability to work with multiple data formats. HDFS can reliably store information flowing from a variety of data sources, whereas Hadoop requires external interfaces to interact with storage repositories outside of HDFS. Sqoop helps you to address part of this problem by allowing users to extract structured data from a relational database to Apache Hadoop. Similarly, raw data can be processed in Hadoop, and the final results can be shared with traditional databases thanks to Sqoop's bidirectional interfacing capabilities.

Sqoop can be downloaded from the Apache site directly, and it supports client-server-based architecture. A server can be installed on one of the nodes, which then acts as a gateway for all Sqoop activities. A client can be installed on any machine, which will eventually connect with the server. A server requires all Hadoop client libraries to be present on the system so that it can connect with the Apache Hadoop Framework; this also means that the Hadoop configuration files are made available.

The Sqoop server can be configured using the `$SQOOP_HOME/conf/sqoop_bootstrap.properties` file, which also provides the `sqoop.properties` file, where you can change its daemon port (the default is 12000). Once you have installed Sqoop, you can run it using the following code:

```
$ sqoop help
usage: sqoop COMMAND [ARGS]
```

Available commands:

```
codegen Generate code to interact with database records
create-hive-table Import a table definition into Hive
eval Evaluate a SQL statement and display the results
export Export an HDFS directory to a database table
help List available commands
import Import a table from a database to HDFS
import-all-tables Import tables from a database to HDFS
import-mainframe Import mainframe datasets to HDFS
list-databases List available databases on a server
list-tables List available tables in a database
version Display version information
```

See 'sqoop help COMMAND' for information on a specific command.

You can connect to any database and start importing the table of your interest directly into HDFS with the following command in Sqoop:

```
$ sqoop import --connect jdbc:oracle://localhost/db --username hrishi --
table MYTABLE
```

The preceding command creates multiple map tasks (unless controlled through `-m <map-task-count>`) to connect to the given database, and then downloads the table, which will be stored in HDFS with the same name. You can check this out by running the following HDFS command:

```
$ hdfs dfs -cat MYTABLE/part-m-00000
```

By default, Sqoop generates a comma-delimited text file in HDFS, and it also supports free-form query imports where you can slice and run table imports in parallel based on the relevant conditions. You can use the `-split-by` argument to control it, as shown in the following example using students' departmental data:

```
$ sqoop import \
--query 'SELECT students.*, departments.* FROM students JOIN departments on
(students.dept_id == departments.id) WHERE $CONDITIONS' \
--split-by students.dept_id --target-dir /user/hrishi/myresults
```

The data from Sqoop can also be imported in Hive, HBase, Accumulo, and other subsystems. Sqoop supports incremental imports where it will only import new rows from the source database; this is only possible when your table has a unique identifier, so make sure Sqoop can keep track of the last updated value. Please refer to this link for more detail on incremental updates (http://sqoop.apache.org/docs/1.4.7/SqoopUserGuide.html#_incremental_imports).

Sqoop also supports the exportation of data from HDFS to any target data source. The only condition to adhere to is that the target table should exist before the Sqoop `export` command has run:

```
$ sqoop export --connect jdbc:oracle://localhost/db --table MYTABLE --
export-dir /user/hrishi/mynewresults --input-fields-terminated-by '\0001'
```

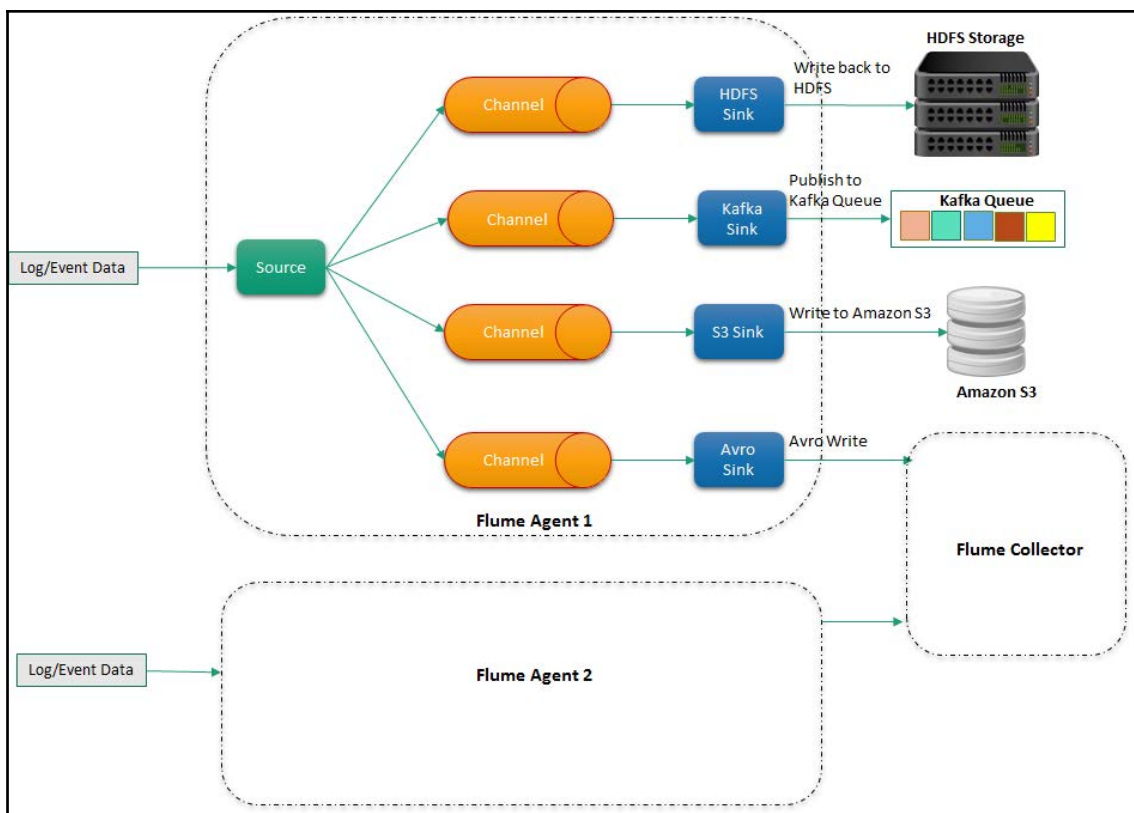
The details of Sqoop are as follows:

Software Name	Apache Sqoop
Latest release	1.99.7 / 1.4.7 is stable
Prerequisites	Hadoop, RDBMS
Supported OSs	Linux
Installation instructions	http://sqoop.apache.org/docs/1.99.7/admin/Installation.html
Overall documentation	http://sqoop.apache.org/docs/1.99.7/index.html
API documentation (1.4.7)	https://sqoop.apache.org/docs/1.4.7/api/

Writing Flume jobs

Apache Flume offers the service to feed logs containing unstructured information back to Hadoop. Flume works across any type of data source. Flume can receive both log data or continuous event data, and it consumes events, incremental logs from sources such as the application server, and social media events.

The following diagram illustrates how Flume works. When flume receives an event, it is persisted in a channel (or data store), such as a local file system, before it is removed and pushed to the target by Sink. In the case of Flume, a target can be HDFS storage, Amazon S3, or another custom application:



Flume also supports multiple Flume agents, as shown in the preceding data flow. Data can be collected, aggregated together, and then processed through a multi-agent complex workflow that is completely customizable by the end user. Flume provides message reliability by ensuring there is no loss of data in transit.

You can start one or more agents on a Hadoop node. To install Flume, download the tarball from the source, untar it, and then simply run the following command:

```
$ bin/flume-ng agent -n myagent -c conf -f conf/flume-conf.properties
```

This command will start an agent with the given name and configuration. In this case, Flume configuration has provided us with a way to specify a source, channel, and sink. The following example is nothing but a properties file but demonstrates Flume's workflow:

```
a1.sources = src1
a1.sinks = tgt1
a1.channels = cnl1
a1.sources.src1.type = netcat
a1.sources.src1.bind = localhost
a1.sources.src1.port = 9999
a1.sinks.tgt1.type = logger
a1.channels.cnl1.type = memory
a1.channels.cnl1.capacity = 1000
a1.channels.cnl1.transactionCapacity = 100
a1.sources.src1.channels = cnl1
a1.sinks.cnl1.channel = cnl1
```

As you can see in the preceding script, an instance of Netcat is set to listen on port 9999, the sink will be performed in the logger, and the channel will be in-memory. Note that the source and sinks are associated with a common channel.

The preceding example will take input from the user console and print it in a logger file. To run it, start Flume with the following command:

```
$ bin/flume-ng agent --conf conf --conf-file example.conf --name myagent -
Dflume.root.logger=INFO,console
```

Now, connect through telnet to port 9999 and type a message, a copy of which should appear in your log file.

Flume supports Avro, Thrift, Unix Commands, the Java Message queue, Tail Command, Twitter, Netcat, SysLogs, HTTP, JSON, and Scribe as sources by default, but it can be extended to support custom sources. It supports HDFS, Hive, Logger, Avro, Thrift, IRC, Rolling Files, HBase, Solr, Elasticsearch, Kite, Kafka, and HTTP as support sinks. Users can write custom sink plugins for Flume. Apache Flume also provides channel support for in-memory, JDBC (Database), Kafka, and the local file system.

The details of Apache Flume are as follows:

Software name	Apache Flume
Latest release	1.8.0
Prerequisites	Java, Hadoop is optional in case of HDFS Sink
Supported OSs	Linux, Windows
Installation instructions	https://flume.apache.org/download.html
Overall documentation	https://flume.apache.org/FlumeDeveloperGuide.html https://flume.apache.org/FlumeUserGuide.html
API documentation (1.7.0)	https://flume.apache.org/releases/content/1.7.0/apidocs/index.html

Understanding Hive

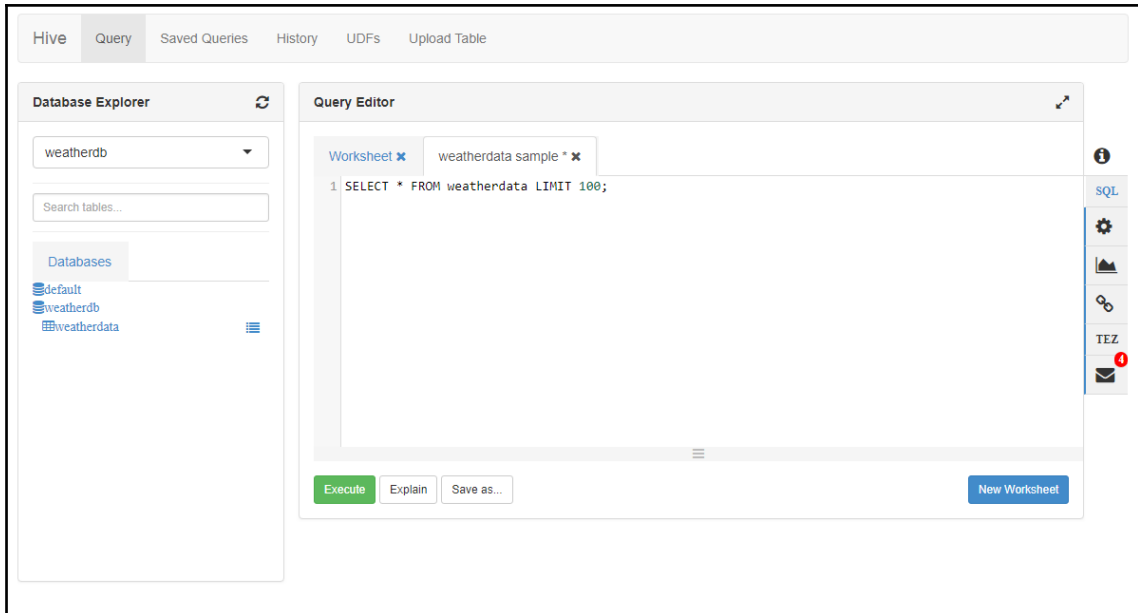
Apache Hive was developed at Facebook to primarily address the data warehousing requirements of the Hadoop platform. It was created to utilize analysts with strong SQL capabilities to run queries on the Hadoop cluster for data analytics. Although we often talk about going unstructured and using NoSQL, Apache Hive still fits in with today's information landscape regarding big data.

Apache Hive provides an SQL-like query language called HiveQL. Hive queries can be deployed on MapReduce, Apache Tez, and Apache Spark as jobs, which in turn can utilize the YARN engine to run programs. Just like RDBMS, Apache Hive provides indexing support with different index types, such as bitmap, on your HDFS data storage. Data can be stored in different formats, such as ORC, Parquet, Textfile, SequenceFile, and so on.

Hive querying also supports extended User Defined Functions, or UDFs, to extend semantics way beyond standard SQL. Please refer to [this link](https://cwiki.apache.org/confluence/display/Hive/HCatalog+UsingHCat) to see the different types of DDLs supported in Hive, and [here](https://cwiki.apache.org/confluence/display/Hive/HCatalog+UsingHCat) for DMLs. Hive also supports an abstraction layer called HCatalog on top of different file formats such as SequenceFile, ORC, and CSV that can abstract. HCatalog abstracts out all types of different forms of storage and provides users with a relational view of their data. You can read more about HCatalog here (<https://cwiki.apache.org/confluence/display/Hive/HCatalog+UsingHCat>). HCatalog also exposes a REST API, allled WebHCat (<https://cwiki.apache.org/confluence/display/Hive/WebHCat>), for users who want to read and write information remotely (<https://cwiki.apache.org/confluence/display/Hive/WebHCat>).

Interacting with Hive – CLI, beeline, and web interface

Apache Hive uses a separate metadata store (Derby, by default) to store all of its metadata. When you set up Hive, you need to provide these details. There are multiple ways through which one can connect to Apache Hive. One well-known interface is through the Apache Ambari Web Interface for Hive, as shown in the following screenshot:



Apache Hive provides a Hive shell, which you can use to run your commands, just like any other SQL shell. Hive's shell commands are heavily influenced by the MySQL command line interface. You can start Hive's CLI by running Hive from the command line and listing all of its databases with the following command :

```
hive> show databases;
OK
default
experiments
weatherdb
Time taken: 0.018 seconds, Fetched: 3 row(s)
```

To run your custom SQL script, call the Hive CLI with the following code:

```
$ hive -f myscript.sql
```

When you are using Hive shell, you can run a number of different commands, which are listed here (<https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Commands>).

In addition to Hive CLI, a new CLI called Beeline was introduced in Apache Hive 0.11, as per JIRA's HIVE-10511 (<https://issues.apache.org/jira/browse/HIVE-10511>). Beeline is based on SQLLine (<http://sqlline.sourceforge.net/>) and works on HiveServer2, using JDBC to connect to Hive remotely.

The following snippet shows a simple example of how to list tables using Beeline:

```
hrishi@base0:~$ $HIVE_HOME/bin/beeline
Beeline version 1.2.1000.2.5.3.0-37 by Apache Hive
beeline> !connect jdbc:hive2://localhost:10000 hive hive
Connecting to jdbc:hive2://localhost:10000
Connected to: Apache Hive (version 1.2.1000.2.5.3.0-37)
Driver: Hive JDBC (version 1.2.1000.2.5.3.0-37)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://localhost:10000> show tables;
+-----+---+
-----+---+
| tab_name |
+-----+---+
-----+---+
| mytest_table |
| student |
+-----+---+
-----+---+
2 rows selected (0.081 seconds)
0: jdbc:hive2://localhost:10000>
```

Now try calling all of the files with the following command:

```
$ hive -f runscript.sql
```

Once complete, you should see MapReduce run, as shown in the following screenshot:

```
$ hive -f runsript.sql

Logging initialized using configuration in file:/etc/hive/2.5.3.0-37/0/hive-log4j.properties
Loading data to table hadoop3book.students
Table hadoop3book.students stats: [numFiles=1, numRows=0, totalSize=357, rawDataSize=0]
OK
Time taken: 0.911 seconds
OK
Time taken: 0.544 seconds
Loading data to table hadoop3book.departments
Table hadoop3book.departments stats: [numFiles=1, numRows=0, totalSize=75, rawDataSize=0]
OK
Time taken: 0.844 seconds
Query ID = labuser_20181010203233_ea2a9611-1668-45fa-b33d-c7caaf1a45a9
Total jobs = 1
Launching Job 1 out of 1

Status: Running (Executing on YARN cluster with App id application_1538721105735_0021)
-----
VERTICES      STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED  KILLED
-----
Map 1 .....  SUCCEEDED    1         1         0         0         0         0
Map 2 .....  SUCCEEDED    1         1         0         0         0         0
-----
VERTICES: 02/02  [=====>>] 100%  ELAPSED TIME: 4.09 s
-----
OK
John   Male   Electrical Engineering
Patrick Male   Electrical Engineering
Rajat  Male   Electrical Engineering
Hrishikesh Male   Computer Science
Jack   Male   Computer Science
Patricia Female  Computer Science
Chang  Male   Mechanical Engineering
Sandy  Male   Mechanical Engineering
```

Hive as a transactional system

Apache Hive can be connected through the standard JDBC, ODBC, and Thrift. Hive 3 supports database **ACID** (Atomicity, Consistency, Isolation, and Durability) at row-level, making it suitable for big data in a transactional system. Data can be populated to Hive with tools such as Apache Flume, Apache Storm, and the Apache Kafka pipeline. Although Hive supports transactions, explicit calls to commit and rollback are not possible as everything is auto-committed.

Apache Hive supports ORC (Optimized Row Columnar) file formats for transactional requirements. The ORC format supports updates and deletes, whereas HDFS does not support in-place file changes. This format therefore provides an efficient way to store data in Hive tables, as it provides lightweight index and multiple reads on a file. When creating a table in Hive, you can provide the following format:

```
CREATE TABLE ... STORED AS ORC
```

You can read more about the ORC format in Hive in the next chapter.

Another condition worth mentioning is that tables that support ACID should be bucketed, as mentioned here (<https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL+BucketedTables>). Note also that Apache Hive provides specific commands for a transactional system, such as `SHOW TRANSACTIONS` for displaying transactions that have been finished or canceled.

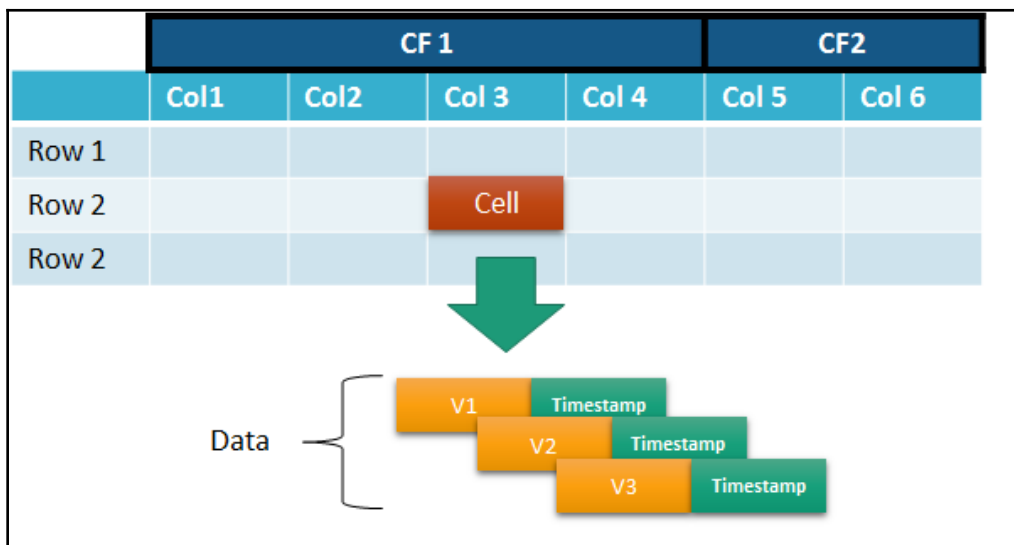
The details of Apache Hive are as follows:

Software name	Apache Hive
Latest release	3.1.0
Prerequisites	Hadoop
Supported OSs	Linux
Installation instructions	https://cwiki.apache.org/confluence/display/Hive/GettingStarted#GettingStarted-InstallingHivefromaStableRelease
Overall documentation	https://cwiki.apache.org/confluence/display/Hive/GettingStarted
API documentation	https://hive.apache.org/javadoc.html

Using HBase for NoSQL storage

Apache HBase provides a distributed, columnar key-value-based storage on Apache Hadoop. It is best suited when you need to perform read-writes randomly on large and varying data stores. HBase is capable of distributing and sharding its data across multiple nodes of Apache Hadoop, and it also provides high availability through its automatic failover from one region server to another. Apache HBase can be run in two modes: standalone and distributed. In the standalone mode, HBase does not use HDFS and instead uses a local directory by default, whereas the distributed mode works on HDFS.

Apache HBase stores its data across multiple rows and columns, where each row consists of a row key and a column containing one or more values. A value can be one or more attributes. Column families are sets of columns that are collocated together for performance reasons. The format of HBase cells is shown in the following diagram:



As you can see in the preceding diagram, each cell can contain versioned data along with a timestamp. A column qualifier provides indexing capabilities to data stored in HBase, and tables are automatically partitioned horizontally by HBase into regions. Each region comprises a subset of a table's rows. Initially, a table comprises one region, but as data grows it splits into multiple regions. Updates in the row are atomic in the HBase. Apache HBase does not guarantee ACID properties, although it ensures that all mutations in the row are atomic and consistent.

Apache HBase provides a shell that can be used to run your commands; it can be called with the following code:

```
$ ./bin/hbase shell <optional script file>
```

The HBase shell provides various commands for managing HBase tables, manipulating data in tables, auditing and analyzing HBase, managing and replicating clusters, and security capabilities. You can look at the commands we have consolidated here (<https://learnhbase.wordpress.com/2013/03/02/hbase-shell-commands/>).

To review a certain row in HBase, call the following:

```
hbase(main):001:0> get 'students', 'Tara'  
COLUMN CELL  
cf:gender timestamp=2407130286968, value=Female  
cf:department timestamp=2407130287015, value=Computer Science
```

Alternatively, you can look at HBase's user interface by going to <http://localhost:16010> once you have installed the region server on your machine. Note that the localhost should be in the same location as the HBase region server. Apache HBase supports different types of clients in various languages, such as C, Java, Scala, Ruby, and so on. HBase is primarily utilized for all NoSQL-based storage requirements and for storing information of different forms together.

The details of Apache HBase are as follows:

Software name	Apache HBase
Latest release	2.1.0
Pre-requisites	Hadoop
Supported OSs	Linux
Installation instructions	https://hbase.apache.org/book.html#quickstart
Overall documentation	https://hbase.apache.org/book.html
API documentation	https://hbase.apache.org/apidocs/index.html

Summary

In this chapter, we studied the different components of Hadoop's overall ecosystem and their tools for solving many complex industrial problems. We went through a brief overview of the tools and software that run on Hadoop, specifically Apache Kafka, Apache Pig, Apache Sqoop, and Apache Flume. We also covered SQL and NoSQL-based databases on Hadoop, which included Hive and HBase respectively.

In the next chapter, we will take a look at some analytics components along with more advanced topics in Hadoop.

8

Advanced Topics in Apache Hadoop

Previously, we have seen some of Apache Hadoop's ecosystem components. In this chapter, we will be looking at advanced topics on Apache Hadoop, which also involves use of some of the Apache Hadoop components that are not covered in previous chapters. Apache Hadoop has started solving the complex problems of large data, but it is important for developers to understand that not all data problems are really big data problems or Apache Hadoop problems. At times, Apache Hadoop may not be the suitable technology for your data problems.

The decision whether to assess a given problem is usually driven by the famous **3Vs** (Volume, Variety, and Veracity) of data. In fact, many organizations that use Apache Hadoop often face challenges in terms of efficiency and performance of solutions due to lack of good Hadoop architecture. A good example of it is a survey done by McKinsey across 273 global telecom companies listed here (<https://www.datameer.com/blog/8-big-data-telecommunication-use-case-resources/>), where it was observed that big data had sizable impact on profits both positive and negative, as shown in the graph in the link.

In this chapter, we will study the following topics:

- Apache Hadoop use cases in various industries
- Advanced HDFS file formats
- Real-time streaming with Apache Storm
- Data analytics with Apache Spark

Technical requirements

You will need Eclipse development environment and Java 8 installed on your system where you can run/tweak these examples. If you prefer to use maven, then you will need maven installed to compile the code. To run the example, you also need Apache Hadoop 3.1 setup on Linux system. Finally, to use the Git repository of this book, you need to install Git.

The code files of this chapter can be found on GitHub:

<https://github.com/PacktPublishing/Apache-Hadoop-3-Quick-Start-Guide/tree/master/Chapter8>

Check out the following video to see the code in action:

<http://bit.ly/2qiETfO>

Hadoop use cases in industries

Today, the industry is growing at a faster pace. With modernization, more and more data is getting generated out of different industries, which requires large data processing. Most of the software used in big data ecosystems is based on open source, with limited paid support for commercial implementations. So, selection of the right technology that can address your problems is important. Additionally, when you choose a technology for solving your big data problem, you should evaluate it based on the following points, at least:

- Evolution of technology with the number of years
- The release's maturity (alpha, beta, or 1.x)
- The frequency of product releases
- The number of committers, which denotes the activeness of the project
- Commercial Support from Companies like Hortonworks and Cloudera
- List of JIRA tickets
- Future roadmap for new releases

Many good Apache projects have retired due to lack of open community and industry support. At times, it has been observed that commercial implementations of these products offer more advanced features and support instead of open source ones. Let us start with understanding different use cases of Apache Hadoop in various industries. An industry that generates large amounts of data often needs an Apache Hadoop-like solution to address its big data needs. Let us look at some industries where we see growth potential of big data-based solutions.

Healthcare

The healthcare industry deals with large data flowing from different areas such as medicine and pharma, patient records, and clinical trials. US Healthcare alone reached 150 exabytes of data in 2011 ([reference here](#)) and, with this growth, it will soon touch zettabytes (10^{21} GBs) of data. Among the dataset, nearly 80% of the data is unstructured. The possible areas of the healthcare industry where Apache Hadoop can be utilized covers patient monitoring, evidence-based medical research and Electronic Health Records (EHRs), and assisted diagnosis. Recently, a lot of new health monitoring wearable devices, such as Fitbit and Garmin, have emerged in the market, which monitor your health parameters. Imagine the amount of data they require for processing. Recently, IBM and Apple started collaborating in a big data health platform, where iPhone and Apple watch users will share data with IBM Watson Cloud to do real-time monitoring of users' data and devise new medical insights. Clinical trials is another area where Hadoop can provide insight over the next best course of treatment, based on a historical analysis of data.

Oil and Gas

Apache Hadoop can store machine and human generated data in different formats. Oil and gas is an industry where you will find 90% of the data is being generated by machines, which can be tapped by the Hadoop system. Starting with upstream, where oil exploration and discovery requires large amounts of data processing and storage to identify potential drilling sites, Apache Hadoop can be used. Similarly, in the downstream, where oil is refined, there are multiple processes involving a large number of sensors and equipment. Apache Hadoop can be utilized to do preventive maintenance and optimize the yield based on historical data. Other areas include the safety and security of oil fields, as well as operational systems.

Finance

The financial and banking industry has been using Apache Hadoop to effectively deal with large amounts of data and bring business insights out of it. Companies such as Morgan Stanley are using Apache Hadoop-based infrastructure to make critical investment decisions. JP Morgan Chase has a humongous amount of structured and unstructured data out of millions of transactions and credit card information and leverages big data-based analytics using Hadoop to make critical financial decisions for its customers. The company is dealing with 150 petabytes of data spread over 3.5 billion user accounts stored in various forms using Apache Hadoop. Big data analytics is used for areas such as fraud detection, US economy statistical analysis, credit market analysis, effective cash management, and better customer experience.

Government Institutions

Government institutions such as municipal corporations and government offices work across lots and lots of data coming from different sources, such as citizen data, financial information, government schemes, and machine data. Their function includes the safety of their citizens. The system can be used to monitor social media pages, water and sanitation, and analyze feedback by citizens on policies. Apache Hadoop can also be used in the area of roads and other public infrastructure, waste management, and sanitation and to analyze accusations/feedback. There has been cases in government organizations where head count the auditors for revenue services have been reduced due to lack of sufficient funds, and they were replaced by automated hadoop driven analytical systems, to help find tax evaders from social media and internet by hunting for their digital footprint, this information was eventually provided to revenue investigators for further proceedings. This was the case of United States Internal Revenue Service department, and you may read about it [here](#).

Telecommunications

The telecom industry has been a high volume, high velocity data generator for all of its application. Over the last couple of years, the industry has evolved from a traditional voice call-based industry towards data-driven businesses. Some of the key areas where we see lot of large data problems is in handling **Call Data Records (CDRs)**, pitching new schemes and products in the market, analyzing the network for strength and weaknesses, and analytics for users. Another area where Hadoop has been effective in the telecom industry is in fraud detection and analysis. Many companies such as Ufone are using big data analytics to capitalize on human behavior.

Retail

The big data revolution has brought a major impact in the retail industry. In fact, Hadoop-like systems have given the industry a strong push to perform market-based analysis on large data; this is also accompanied by social media analysis to get the current trends and feedback on products, or even enabling potential customers to provide a path to purchase retail merchandise. The retail industry has also worked extensively to optimize the price of their products by analyzing market competition electronically and optimizing it automatically with minimal or no human interaction. The industry has not only optimized prices, but companies have also optimized on their workforce along with inventory. Many companies such as Amazon use big data to provide automated recommendation and targeted promotions, based on user behavior and historical data, to increase their sales.

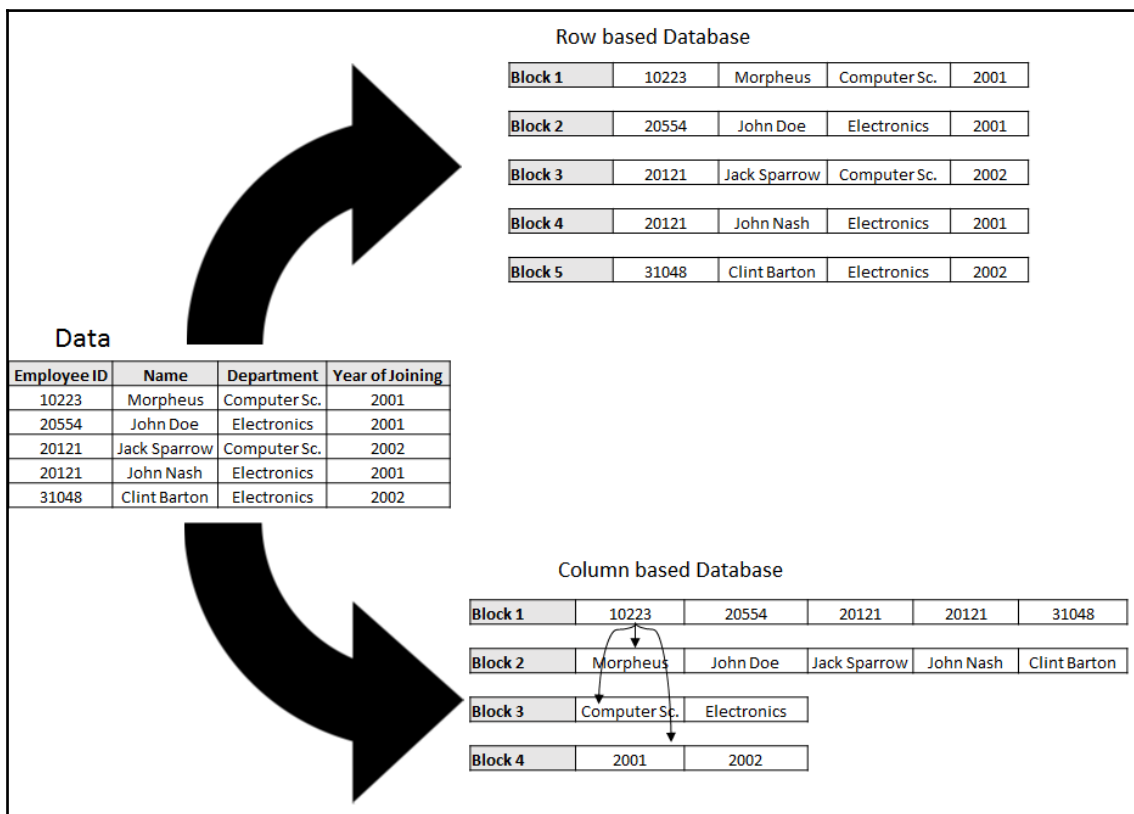
Insurance

The insurance sector is driven primarily by huge statistics and calculations. For the insurance industry, it is important to collect the necessary information about insurers from heterogeneous data sources, to assess risks and to calculate the policy premium, which may require large data processing on a Hadoop platform. Just like the retail industry, this industry can also use Apache Hadoop to gain insight about prospects and recommend suitable insurance schemes. Similarly, Apache Hadoop can be used to process large transactional data to assess the possibility of fraud. In addition to functional objectives, Apache Hadoop-based systems can be used to optimize the cost of labor and workforce and manage finances in a better way.

I have covered some industry sectors, however, the use cases of Hadoop cover other industries such as manufacturing, media and entertainment, chemicals, and utilities. Now that you have clarity over how different sectors can use Apache Hadoop to solve their complex big data problems, let us start with advanced topics of Apache Hadoop.

Advanced Hadoop data storage file formats

We have looked at different formats supported by HDFS in Chapter 3, *Deep Dive into the Hadoop Distributed File System*. We covered many formats including SequenceFile, Map File, and the Hadoop Archive format. We will look at more formats now. The reason why they are covered in this section is because these formats are not used by Apache Hadoop or HDFS directly; they are used by the ecosystem components. Before we get into the format, we must understand the difference between row-based and columnar-based databases because ORC and Parquet formats are columnar data storage formats. The difference is in the way the data gets stored in the storage device. A row-based database stores data in row format, whereas a columnar database stores it column by column. The following screenshot shows how the storage patterns differ between these types:



Please note that the block representation is for indicative purposes only—in reality, it may differ on a case to case basis. I have shown how the columns are linked in columnar storage. Traditionally, most of the relational databases have been row-based storage including the most famous Oracle, Sybase, and DB2. Recently, the importance of columnar storage has grown, and many new columnar storage databases are being introduced, such as SAP HANA and Oracle 12C.

Columnar databases offer efficient read and write data capabilities over row-based databases for certain cases. For example, if I request employee names from both storage types, a row-based store requires multiple block reads, whereas the columnar requires a single block read operation. But when I run a query with `select * from <table>`, then a row-based storage can return an entire row in one shot, whereas the columnar will require multiple reads.

Now, let us start with the Parquet format first.

Parquet

Apache Parquet offers columnar data storage on Apache Hadoop. Parquet was developed by Twitter and Cloudera together to handle the problem of storing large data with high columns. We have already seen the advantages of columnar storage over row-based storage. Parquet offers advantages in performance and storage requirements with respect to traditional storage. The Parquet format is supported by Apache Hive, Apache Pig, Apache Spark, and Impala. Parquet achieves compression of data by keeping similar values of data together.

Now, let us try and create a Parquet-based table in Apache Hive:

```
create table if not exists students_p (  
    student_id int,  
    name String,  
    gender String,  
    dept_id int) stored as parquet;
```

Now, let us try and load the same `students.csv` that we have seen in Chapter 7, *Demystifying Hadoop Ecosystem Components*, in this format. Since you have created a Parquet table, you cannot directly load a CSV file in this table, so we need create a staging table that can transform CSV to Parquet. So, let us create a text file-based table with similar attributes:

```
create table if not exists students (  
    student_id int,  
    name String,  
    gender String,  
    dept_id int) row format delimited fields terminated by ',' stored as  
    textfile;
```

Now you can load the data with the following:

```
load data local inpath '/home/labuser/hiveqry/students.csv' overwrite into  
table students;
```

Check the table out and transfer the data to Parquet format with the following SQL:

```
insert into students_p select * from students;
```

Now, run a select query on the `students_p` table; you should see the data. You can read more about the data structures, feature and storage representation at Apache's website here: <http://parquet.apache.org/documentation/latest/>.

The **pros** of the Parquet format are as follows:

- Being columnar and having efficient storage due to better compression
- Reduce I/O for select a,b,c type of queries
- Suitable for large column-based tables

The **cons** of the Parquet format are as follows:

- Performance degrades over `select * from queries`
- Not suitable for OLTP transactions
- Expensive to deal in conditions where schema is changing
- Write performance is no better than read performance

Apache ORC

Just like Parquet, which was released by Cloudera, a competitor, Hortonworks, also developed a format on top of the traditional RC file format called **ORC (Optimized Record Columnar)**. This was launched during a similar time frame with Apache Hive. ORC offers advantages such as high compression of data, predictive push down feature, and faster performance. Hortonworks performed a comparison of ORC, Parquet, RC, and traditional CSV files over compression on the TPC-DS Scale dataset, and it was published that ORC achieves the highest compression (78% smaller) using Hive, as compared to Parquet, which compressed the data to 62% using Impala. Predictive push down is a feature where ORC tries to perform analytics right at the data storage instead of bringing in the data and filtering it out. For example, you can follow the same steps you followed for Parquet, except the Parquet table creation step should be replaced with ORC. So, you can run following DDL for ORC:

```
create table if not exists students_o (  
    student_id int,  
    name String,  
    gender String,  
    dept_id int) stored as orc;
```

Given that user data is changing continuously, the ORC format ensures reliability of transactions by supporting ACID properties. Despite this, the ORC format is not recommended by the OLTP kind of systems due to high level of transactions per unit time. As HDFS is write-only, ORC performs edit and delete through its delta files. You can read more information about ORC here (<https://orc.apache.org/>).

The **pros** of the ORC format are as follows:

- Similar to the previously mentioned pros of the Parquet format, except that ORC offers additional features such as predictive push down
- Supports complex data structures and basic statistics, such as sum and count, by default

The **cons** of the ORC format are as follows:

- Similar to the Parquet format

Avro

Apache Avro offers data serialization capabilities in big data-based systems; additionally, it provides data exchange services for different Hadoop-based applications. Avro is primarily a schema-driven storage format that uses JSON to serialize the data coming from different forms. Avro's format persists the data schema along with the actual data. The benefit for storing the data structure definition along with data, is that the Avro can enable faster data writes, as well as allow the data to be stored with size optimized. For example, our case of student information can be represented in Avro as per the following JSON:

```
{ "type": "record", "name": "studentinfo",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "department", "type": "string" },
  ]
}
```

When Avro is used in the RPC format, the schema is shared with each other during the handshaking of client and server. In addition to records and numeric types, Avro stores data row-based storage. Avro includes support for arrays, maps, enums, variables, and fixed-length binary data and strings. Avro schemas are defined in JSON, and the beauty is that the schemas can evolve over time.

The **pros** of Avro are as follows:

- Suitable for data where you have less columns and `select *` queries
- Files support block compression and they can be split
- Avro is faster in data retrieval, can handle schema evolution

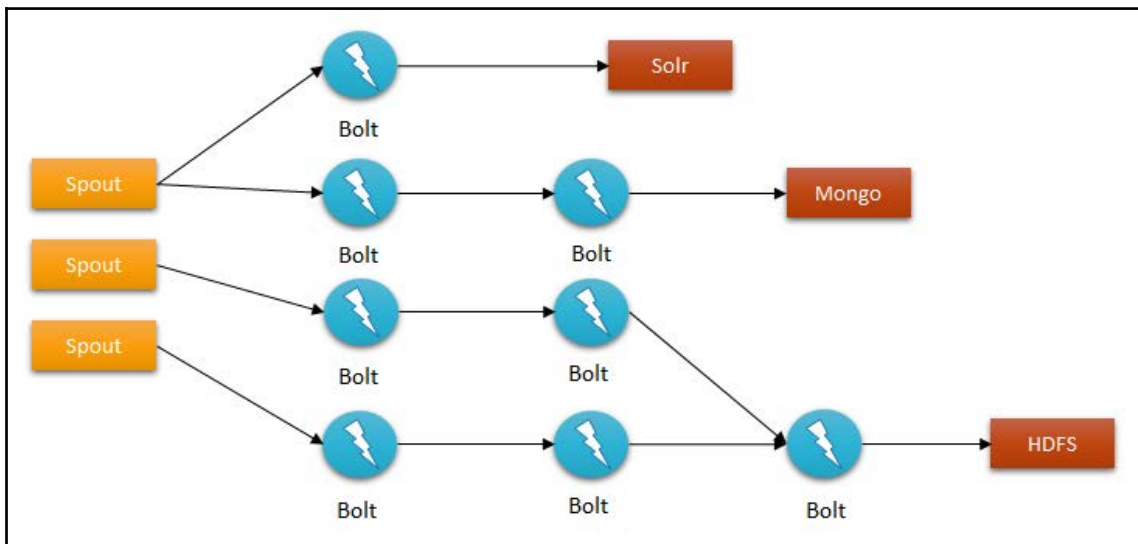
The **cons** of Avro are as follows:

- Not best suited for large tables with multiple columns

Real-time streaming with Apache Storm

Apache Storm provides a distributed real-time computational capability for processing large amounts of data with high velocity. This is one of the reasons why it is being used primarily for real-time streaming data for rapid analytics. Storm is capable of processing over thousands of data records per second on a distributed cluster. Apache Storm runs on YARN framework and can connect with queues such as JMS and Kafka or to any type of database or it can listen to streaming APIs feeding information continuously, such as Twitter-streaming APIs and RSS feeds.

Apache Storm uses networks of spouts, bolts, and sinks called **topology** to address any kind of complex problems. **Spouts** represents a source where Storm is collecting information such as APIs, databases, or message queues. **Bolts** provide computation logic for an input stream and they produce output streams. A bolt could be a `map()` function or a `reduction()` function or it could be a custom function written by a user. Spouts work as the initial source of the data stream. Bolts receive the stream from either one or more spouts or some other bolts. Part of defining a topology is specifying which streams each bolt should receive as input. The following diagram shows a sample topology in Storm:



The streams are a sequence of tuples, which flow from one spout to a bolt. Storm users define topologies for how to process the data when it comes to streaming in from the spout. When the data comes in, it is processed and the results are passed into Hadoop. Apache Storm runs on a Hadoop cluster. Each Storm cluster has four categories of nodes. **Nimbus** is responsible for managing Storm activities such as uploading a topology for running across nodes, launching workers, monitoring the units of executions, and shuffling the computations if needed. Apache Zookeeper coordinates among various nodes across a Storm cluster. **Supervisor** communicates with Nimbus to control the execution done by workers as per information received from Nimbus. **Worker** nodes are responsible for the execution of activities. Storm Nimbus uses a **scheduler** to schedule multiple topologies across multiple supervisors. Storm provides four types of schedulers to ensure fairness of resources allocation to different topologies.

You can write Storm topologies in multiple languages; we will look at a Java-based Storm example now. The example code is available in the code base of this book. First, you need to start creating a source spout. You can create your spout by extended `BaseRichSpout` (<http://storm.apache.org/releases/2.0.0-SNAPSHOT/javadocs/org/apache/storm/topology/base/BaseRichSpout.html>) or the interface, `IRichSpout` (<http://storm.apache.org/releases/2.0.0-SNAPSHOT/javadocs/org/apache/storm/topology/IRichSpout.html>). `BaseRichSpout` provides helper methods for you to simplify your coding efforts, which you may otherwise need to write using `IRichSpout`:

```
public class MySourceSpout extends BaseRichSpout {
    public void open(Map conf, TopologyContext context,
        SpoutOutputCollector collector);
    public void nextTuple();
    public void declareOutputFields(OutputFieldsDeclarer declarer);
    public void close();
}
```

The `open` method is called when a task for the component is initialized within a worker in the cluster. The method `nextTuple` is responsible to emit a new tuple in the topology, all this happens in same thread. Apache Storm Spouts can emit the output tuples to more than one stream. You can declare multiple streams using the `declareStream()` method of the `OutputFieldsDeclarer` (<http://storm.apache.org/releases/2.0.0-SNAPSHOT/javadocs/org/apache/storm/topology/OutputFieldsDeclarer.html>) and specify the stream to emit to when using the `emit` method on `SpoutOutputCollector` (<http://storm.apache.org/releases/2.0.0-SNAPSHOT/javadocs/org/apache/storm/spout/SpoutOutputCollector.html>). In `BaseRichSpout`, you can use the `declareOutputFields()` method.

Now, let us look at the computational unit—the bolt definition. You can create a bolt by extending `IRichBolt` (<http://storm.apache.org/releases/2.0.0-SNAPSHOT/javadocs/org/apache/storm/topology/IRichBolt.html>) or `IBasicBolt`. `IRichBolt` is the general interface for bolts, whereas `IBasicBolt` (<http://storm.apache.org/releases/2.0.0-SNAPSHOT/javadocs/org/apache/storm/topology/IBasicBolt.html>) is a convenient interface for defining bolts that do filtering or simple functions. The only difference between these two is `IBasicBolt` provides automation over execute processes to make life simple (such as sending acknowledgement for the input type at the end of execution) for the bolt object created on the client machine.

These interfaces are serialized and submitted to the master i.e. Nimbus. Nimbus launches the worker nodes, which deserialize the object of below class, and then call `prepare()` method on it. Post that, the worker starts processing the tuples.

```
public class MyProcessingBolt implements IRichBolt {
    public void prepare(Map conf, TopologyContext context, OutputCollector
collector);
    public void execute(Tuple tuple);
    public void cleanup();
    public void declareOutputFields (OutputFieldsDeclarer declarer);
}
```

The main method in bolts is the `execute` method, which takes in as input a new tuple. Bolts emit new tuples using the `OutputCollector` object. `prepare` is called when a task for this component is initialized within a worker on the cluster. It provides the bolt with the environment in which the bolt executes. `cleanup` is called when the bolt is shutting down; there is no guarantee that `cleanup` will be called, because the supervisor forcibly kills worker processes on the cluster.

You can create multiple bolts, which are units of processing. This provides a step-by-step refinement capability for your input data. For example, if you are parsing Twitter data, you may create bolts in the following order:

- **Bolt1:** Cleaning of tweets received
- **Bolt2:** Removal of unnecessary content from your tweets
- **Bolt3:** Identifying entities from Twitter and creating Twitter-parsed data
- **Bolt3:** Storing tweets in database or NOSQL storage

Now, initialize the topology builder with `TopologyBuilder` (<http://storm.apache.org/releases/2.0.0-SNAPSHOT/javadocs/org/apache/storm/topology/TopologyBuilder.html>). `TopologyBuilder` exposes the Java API for specifying a topology for Storm to execute. It can be initialized with the following code:

```
TopologyBuilder builder = new TopologyBuilder();
```

Part of defining a topology is specifying which streams each bolt should receive as input. A stream grouping defines how that stream should be partitioned among the bolt's tasks. There are multiple stream grouping available such as randomly distributing tuples (shuffle grouping):

```
builder.setSpout("tweetreader", new MySourceSpout ());
builder.setBolt("bolt1", new CleanseDataBolt()).shuffleGrouping("group1");
builder.setBolt("bolt2", new RemoveJunkBolt()).shuffleGrouping("group2");
builder.setBolt("bolt3", new
EntityIdentifyBolt()).shuffleGrouping("group3");
builder.setBolt("bolt4", new StoreTweetBolt()).shuffleGrouping("group4");
```

In this case, the bolts are set for sequential processing.

You can submit the topology to a cluster:

```
public class MyTopology extends ConfigurableTopology {
    protected int run(String[] args) throws Exception {
        //initialize topology, set spouts and bolts
        return submit("mytopology", conf, builder);
    }
}
```

Now, compile and create a deployable jar:

```
storm jar <jarfile> -c <cluster>
```

Once you deploy, the topology will run and start listening to streaming of data from source system. The Stream API is an alternative interface to Storm. It provides a typed API for expressing streaming computations and supports functional style operations:

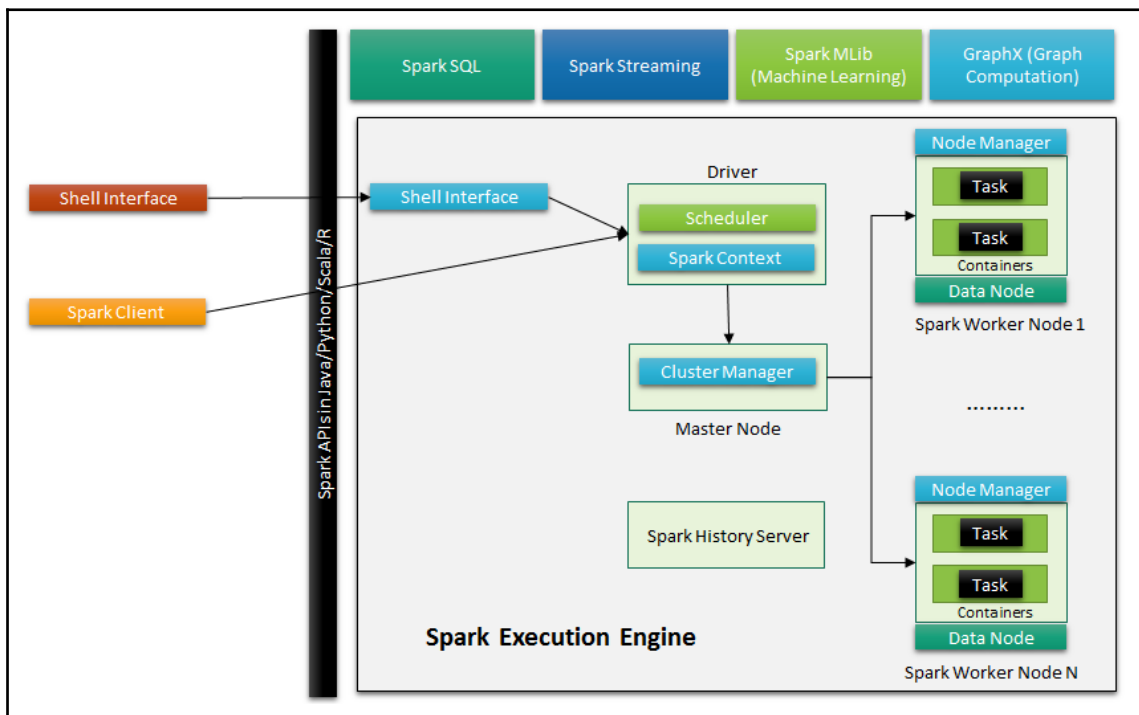
Software Name	Apache Storm
Latest Release	1.2.2
Pre-requisites	Hadoop
Supported OS	Linux
Installation Instructions	http://storm.apache.org/releases/2.0.0-SNAPSHOT/Setting-up-a-Storm-cluster.html
Overall Documentation	http://storm.apache.org/releases/2.0.0-SNAPSHOT/index.html
API Documentation	http://storm.apache.org/releases/2.0.0-SNAPSHOT/javadocs/index.html

Data analytics with Apache Spark

Apache Spark offers a blazing fast processing engine based out of Apache Hadoop. It provides in-memory cluster processing of the data, thereby providing analytics at high speeds. Apache Spark evolved in AMPLab (U. C. Berkeley) in 2009 and it was made open source through the Apache Software Foundation. Apache Spark is based out of YARN. Following are key features of Apache Spark:

- **Fast:** Due to in-memory processing capability, Spark is fast in processing
- **Multiple language support:** You can write Spark programs in Java, Scala, R, and Python
- **Deep analytics:** It provides truly distributed analytics, which includes machine learning, streaming data processing, and data querying
- **Rich API support:** It provides a rich API library for interaction in multiple languages
- **Multi-processing engine support:** Apache Spark can be deployed on MapReduce, YARN, and Mesos

The system architecture along with Spark components are shown in the following:



Apache Spark uses master-slave architecture. Spark Driver is the main component of the Spark ecosystem as it runs with a `main()` of Spark applications. To run a Spark application on a cluster, `SparkContext` can connect to several types of cluster managers include YARN, MapReduce, or Mesos. The Spark cluster manager assigns resources to the application, which gets its allocation of resources from the cluster manager, then the application can send its application code to the respective executors allocated (executors are execution units). Then, `SparkContext` sends tasks to these executors.

Spark ensures computational isolation of applications by allocated resources in a dedicated manner. You can submit your application to Apache Spark by following the simple command-line `spark-submit` script, as shown [here](#). Since the resources are dedicate assigned, it is important to have their maximum utilization. To ensure utilization, Spark provides static and dynamic resource allocation.

Additionally, following are some of Apache Spark's key components and their capabilities:

- **Core:** It provides a generic execution engine on top of big data computational platform.
- **Spark SQL:** This provides an SQL capability on top of heterogeneous data through its `SchemaRDD`.
- **Spark streaming:** It provides fast scheduling and data streaming capabilities; streaming can be performed in micro batches.
- **Spark MLlib:** This provides a distributed machine learning capability on top of the Apache Spark engine.
- **Spark GraphX:** This provides distributed graph processing capability using Apache Spark.
- **APIs:** Apache Spark provides the above capabilities through its multi-language APIs. Many times, it is considered to be part of the Apache Spark core.

Apache Spark provides a data abstraction through its own implementation of `DataFrame` or a matrix of actual data. It's also called Spark RDDs (Resilient Distributed Datasets). RDD is formed out of a collection of distributed data across multiple nodes of Hadoop. RDDs can be created from simple text files, SQL databases, and NoSQL stores. The concept of RDD came from data frames in R. In addition to RDDs, Spark provides an SQL SQL 2003 standard compliant to load the data in its RDDs, which can later be used for analysis. GraphX provides distributed implementation of Google's `PageRank`. Since Spark is an in-memory, fast cluster solution, technical use cases require Spark on real-time streaming requirements. This can be achieved through either Spark streaming APIs or other software such as Apache Storm.

Now, let us understand some code for Spark ML. First, you need Spark Context. You can get it by following code snippet in Java:

```
JavaSparkContext sc = new JavaSparkContext(new  
SparkConf().setAppName("MyTest").setMaster("local"));  
JavaSparkContext sparkContext = new JavaSparkContext(sparkConf);
```

Once you initialize the context, you can use it for any application requirements:

```
JavaRDD<String> inputFile =  
sparkContext.textFile("hdfs://host1/user/testdata.txt");
```

Now you can do processing on your RDD, as in the following example:

```
JavaRDD<String> myWords = inputFile.flatMap(content ->  
Arrays.asList(content.split(" ")));
```

This will get all of the words from the file separated into arrays in `myWords`. You can do further processing and save the RDD as a file on HDFS with following command:

```
myWords.saveAsTextFile("MyWordsFile");
```

Please look at the detailed example provided in the code base for this chapter. Similarly, you can process SQL queries through the Dataset API. In addition to the programmatic way, Apache Spark also provides a Spark shell for you to run your programs and monitor their status.



Apache Spark Release 2.X has been a major milestone release. In this release, Spark brought in SparkSQL support with 2003 SQL compliance, rich machine learning capabilities through the `spark.ml` package. This is going to replace Spark Mlib with new support models such as k-mean, linear models, and Naïve Bayes, along with streaming API support.

For data scientists, Spark is a rich analytical data processing tool. It offers built-in support for machine learning algorithms and provides exhaustive APIs for transforming or iterating over datasets. For analytics requirements, you may use notebooks such as Apache Zeppelin or Jupyter notebook:

Software Name	Apache Spark (Mlib, GraphX, and Streaming)
Latest Release	2.3.2 – Sept 24, 2018
Pre-requisites	Apache Hadoop and other libraries specific to each component
Supported OS	Linux
Installation Instructions	https://spark.apache.org/docs/latest/quick-start.html
Overall Documentation	https://spark.apache.org/docs/latest/
API Documentation	Scala : https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package Java : https://spark.apache.org/docs/latest/api/java/index.html Python : https://spark.apache.org/docs/latest/api/python/index.html R : https://spark.apache.org/docs/latest/api/R/index.html SQL : https://spark.apache.org/docs/latest/api/sql/index.html

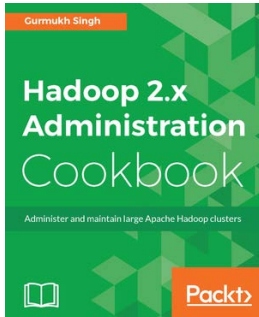
Summary

In this last chapter, we have covered advanced topics for Apache Hadoop. We started with business use cases for Apache Hadoop in different industries, covering healthcare, oil and gas, finance and banking, government, telecommunications, retail, and insurance. We then looked at advanced Hadoop storage formats, which are used today by many of Apache Hadoop's ecosystem software; we covered Parquet, ORC, and Avro. We looked at the real-time streaming capabilities of Apache Storm, which can be used on a Hadoop cluster. Finally, we looked at Apache Spark when we tried to understand the different components of Apache Spark including streaming, SQL, and analytical capabilities. We also looked at its architecture.

We started this book with history of Apache Hadoop, its architecture, and open source v/s commercial hadoop implementations. We looked at new Hadoop 3.X features. We proceeded with Apache hadoop installation with different configurations such as developer, pseudo-cluster and distributed setup. Post installation, we dived deep in core hadoop components such as HDFS, Map Reduce and YARN with component architecture, code examples, APIs. We also studied big data development lifecycle covering development, unit testing, deployment etc. Post development lifecycle, we looked at monitoring and administrative aspects of Apache Hadoop, where we studied key features of Hadoop, monitoring tools, hadoop security etc. Finally, we studied key hadoop ecosystem components for different areas such as data engine, data processing, storage and analytics. We also looked at some of the open source hadoop projects that are happening in Apache community.

Other Books You May Enjoy

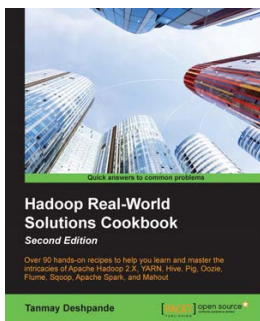
If you enjoyed this book, you may be interested in these other books by Packt:



Hadoop 2.x Administration Cookbook Gurmukh Singh

ISBN: 9781787126732

- Set up the Hadoop architecture to run a Hadoop cluster smoothly
- Maintain a Hadoop cluster on HDFS, YARN, and MapReduce
- Understand High Availability with Zookeeper and Journal Node
- Configure Flume for data ingestion and Oozie to run various workflows
- Tune the Hadoop cluster for optimal performance
- Schedule jobs on a Hadoop cluster using the Fair and Capacity scheduler
- Secure your cluster and troubleshoot it for various common pain points



Hadoop Real-World Solutions Cookbook - Second Edition

Tanmay Deshpande

ISBN: 9781784395506

- Installing and maintaining Hadoop 2.X cluster and its ecosystem.
- Write advanced Map Reduce programs and understand design patterns.
- Advanced Data Analysis using the Hive, Pig, and Map Reduce programs.
- Import and export data from various sources using Sqoop and Flume.
- Data storage in various file formats such as Text, Sequential, Parquet, ORC, and RC Files.
- Machine learning principles with libraries such as Mahout
- Batch and Stream data processing using Apache Spark

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

- Access Control Lists (ACLs) 61
- Apache Hadoop 3.0
 - features 20
 - releases 20
- Apache Hadoop Common 12
- Apache Hadoop Development Tools
 - reference 95
- Apache Hadoop
 - about 11
 - DataNode 19
 - features 11, 14
 - high availability 142
 - high availability, for NameNode 142, 144
 - high availability, for Resource Manager 145
 - NameNode 18
 - overview 9
 - reference 33
 - Resource Manager 16
 - setting, up in cluster mode 48
 - setup, prerequisites 28
 - working 15
 - YARN Timeline Service version 2 18
- Apache HDFS 11
- Apache Kafka
 - working with 160, 163
- Apache Pig scripts
 - Pig Latin 165
 - User-Defined Function (UDF) 165
 - writing 164
- architecture, YARN
 - Application master (AM) 117
 - Node Managers (NM) 117
 - Resource Manager (RM) 117

C

- Capacity Scheduler
 - about 141
 - benefits 142
- cheat sheet, Apache Pig scripts
 - reference 165
- Cloudera Hadoop distribution
 - about 23
 - cons 24
 - pros 23
- cluster mode
 - about 27
 - YARN, setting up 52, 55
- clusters
 - balanced 44
 - computational-centric 44
 - fault tolerance 46
 - high availability 46
 - initial load of data 44
 - lightweight 44
 - organizational data growth 45
 - planning 44
 - sizing 44
 - storage-centric 44
 - velocity of data 47
 - workload and computational requirements 46

D

- Data Flow Diagram (DFD) 67
- data structures, HDFS
 - MapFile 79
 - MapFile, variants 79
 - SequenceFile 78
- data
 - transferring, with Sqoop 167
- developer mode 27

- distributed cluster
 - Hadoop applications 137
 - Hadoop ports 137
 - Hadoop URLs 137
 - planning 135

E

- environment configuration, MapReduce
 - Job history server, working with 87
 - mapred-site.xml, working with 86
- Erasure Code (EC) 20
- Extract Transform Load (ETL) 164

F

- Fair Scheduler 140
- Flume jobs
 - writing 169, 170

H

- Hadoop administrators
 - roles and responsibilities 134
- Hadoop APIs and packages 89
- Hadoop applications 137
- Hadoop cluster
 - application, securing 146
 - daemon log files 56
 - data confidentiality 146
 - data, in Motion 146
 - data, in Rest 146
 - data, securing in HDFS 147
 - debugging 56
 - diagnosing 55
 - job log files 55
 - JPS (Java Virtual Machine Process Status) 56
 - JStack 57
 - log files, working with 55
 - securing 146
 - tuning tools 56
- Hadoop distribution
 - Cloudera Hadoop distribution 23
 - Hortonworks Hadoop distribution 24
 - MapR Hadoop distribution 25
 - open source-based Hadoop, cons 23
 - open source-based Hadoop, pros 23
 - selecting 22

- Hadoop ports 137
- Hadoop URLs 137
- Hadoop's Ecosystem
 - about 155, 159
 - data engine 155
 - data flow 155
 - data frameworks 155
 - data storage 155
 - machine learning and analytics 155
 - management and coordination 155
 - search engine 155

- Hadoop
 - Capacity Scheduler 141
 - downloading 33, 35
 - executing, in standalone mode 36, 39
 - Fair Scheduler 140
 - file system CLIs 73
 - resource management 139
 - shell commands, working with 75

- HBase
 - used, for NoSQL storage 175

- HDFS
 - configuration files 71
 - configuring, in cluster mode 48
 - data flow patterns 65
 - Data Node, hot swapping 64
 - data structures, working with 78
 - features 61
 - federation capabilities 64
 - importance 70
 - installing, in cluster mode 48
 - Intra-DataNode balancer 65
 - multi tenancy, achieving 62
 - safe mode 63
 - snapshots 62
 - user commands, working with 73
 - using, as archival storage 67
 - using, as historical storage 69
 - using, as primary storage with cache 66
 - working 59
- heap size 22
- Hive
 - about 171
 - as transnational system 174
 - interacting 172

- Hortonworks Data Flow (HDF) 24
- Hortonworks Data Platform (HDP) 24
- Hortonworks Hadoop distribution
 - about 24
 - cons 24
 - pros 24

I

- incremental import, Sqoop
 - reference 168
- Intra-Data Node Balancer 21

J

- Job history server
 - reference 87
- RESTful APIs 87, 89
- working with 87

K

- Key Management Service (KMS) 138

M

- Map Reduce Streaming
 - reference 114
- map task 12
- MapR Hadoop distribution
 - about 25
 - cons 25
 - pros 25
- MapReduce APIs
 - exploring 96
 - input formats 99
 - Mapper APIs 103
 - MapReduce jobs, configuring 96
 - output formats 101
 - working with 105
- MapReduce jobs
 - compiling 107
 - failure handling 111
 - job, remote triggering 107
 - Tool, using 108
 - ToolRunner, using 108
 - unit testing 110
- MapReduce programming

- streaming 113
- MapReduce project
 - Eclipse project, setting up 91, 95
 - setting up 91
- MapReduce
 - about 12, 83
 - environment, configuring 85
 - example 84
 - map phase 84
 - reduce phase 85
 - working 82

N

- NameNode UI
 - reference 51
- Network Filesystem (NFS) 144
- Node Manager
 - Application Master 18
 - Container Manager 17

P

- parity drive 20
- Pig Latin 165
- pseudo cluster (single node Hadoop) 27
- pseudo Hadoop cluster
 - setting up 39, 43

Q

- Query Journal Manager (QJM) 144

R

- resource management
 - in Hadoop 139
- Resource Manager (RM)
 - about 16
 - application manager 118
 - Node Manager 17
 - scheduler 118
- routine tasks
 - archiving, in Hadoop 149
 - Hadoop Metric, working with 151
 - nodes, commissioning 150
 - nodes, decommissioning 150
 - performing 148

safe node, working with 148

S

setup prerequisites, Apache Hadoop

about 28

hardware, preparing 28

installing 30

nodes, working without passwords 32

space, checking on Hadoop nodes 29

shell scripts 22

T

Total Cost of Ownership (TCO) 14

U

Unix's Pipe function

reference 113

User-Defined Functions (UDFs)

about 165

reference 166

Y

YARN (Yet Another Resource Negotiator)

application framework 124

architecture 117

custom application master, writing 127

distributed CLI, working with 122

environment, configuring, in cluster 121

features 118

Federation 119

resource models 118

RESTful APIs 120

setting up, in cluster mode 52

YARN application

building 128

framework, exploring 124

monitoring 129, 131

monitoring, on cluster 128

project, setting up 125

writing, with YarnClient 126

YARN Federation 22

YARN Scheduler 21

YARN User Interface 21

YarnClient

used, for writing YARN application 126