Raúl Estrada

Fast Data Processing Systems with SMACK Stack

Combine the incredible powers of Spark, Mesos, Akka, Cassandra, and Kafka to build data processing platforms that can take on even the hardest of your data troubles!

Packt>

Fast Data Processing Systems with SMACK Stack

Combine the incredible powers of Spark, Mesos, Akka, Cassandra, and Kafka to build data processing platforms that can take on even the hardest of your data troubles!

Raúl Estrada

BIRMINGHAM - MUMBAI

Fast Data Processing Systems with SMACK Stack

Production reference: 1151216

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78646-720-1

www.packtpub.com

Credits

Author

Raúl Estrada

Reviewers

Anton Kirillov
Sumit Pal

Commissioning Editor

Veena Pagare

Acquisition Editor

Divya Poojari

Content Development Editor

Amrita Noronha

Technical Editor

Sneha Hanchate

Copy Editor

Safis Editing

Project Coordinator

Shweta H Birwatkar

Proofreader

Safis Editing

Indexer

Mariammal Chettiyar

Graphics

Disha Haria

Production Coordinator

Nilesh Mohite

About the Author

Raúl Estrada is a programmer since 1996 and Java Developer since 2001. He loves functional languages such as Scala, Elixir, Clojure, and Haskell. He also loves all the topics related to Computer Science. With more than 12 years of experience in High Availability and Enterprise Software, he has designed and implemented architectures since 2003.

His specialization is in systems integration and has participated in projects mainly related to the financial sector. He has been an enterprise architect for BEA Systems and Oracle Inc., but he also enjoys Mobile Programming and Game Development. He considers himself a programmer before an architect, engineer, or developer.

He is also a Crossfitter in San Francisco, Bay Area, now focused on Open Source projects related to Data Pipelining such as Apache Flink, Apache Kafka, and Apache Beam. Raul is a supporter of free software, and enjoys to experiment with new technologies, frameworks, languages, and methods.

I want to thank my family, especially my mom for her patience and dedication.

I would like to thank Master Gerardo Borbolla and his family for the support and feedback they provided on this book writing.

I want to say thanks to the acquisition editor, Divya Poojari, who believed in this project since the beginning.

I also thank my editors Deepti Thore and Amrita Noronha. Without their effort and patience, it would not have been possible to write this book.

And finally, I want to thank all the heroes who contribute (often anonymously and without profit) with the Open Source projects specifically: Spark, Mesos, Akka, Cassandra, and Kafka; an honorable mention for those who build the connectors of these technologies.

About the Reviewers

Anton Kirillov started his career as a Java developer in 2007, working on his PhD thesis in the Semantic Search domain at the same time. After finishing and defending his thesis, he switched to Scala ecosystem and distributed systems development. He worked for and consulted startups focused on Big Data analytics in various domains (real-time bidding, telecom, B2B advertising, and social networks) in which his main responsibilities were focused on designing data platform architectures and further performance and stability validation. Besides helping startups, he has worked in the bank industry building Hadoop/Spark data analytics solutions and in a mobile games company where he has designed and implemented several reporting systems and a backend for a massive parallel online game.

The main technologies that Anton has been using for the recent years include Scala, Hadoop, Spark, Mesos, Akka, Cassandra, and Kafka and there are a number of systems he's built from scratch and successfully released using these technologies. Currently, Anton is working as a Staff Engineer in Ooyala Data Team with focus on fault-tolerant fast analytical solutions for the ad serving/reporting domain.

Sumit Pal has more than 24 years of experience in the Software Industry, spanning companies from startups to enterprises. He is a big data architect, visualization and data science consultant, and builds end-to-end data-driven analytic systems. Sumit has worked for Microsoft (SQLServer), Oracle (OLAP), and Verizon (Big Data Analytics). Currently, he works for multiple clients building their data architectures and big data solutions and works with Spark, Scala, Java, and Python. He has extensive experience in building scalable systems in middletier, datatier to visualization for analytics applications, using BigData and NoSQL DB. Sumit has expertise in DataBase Internals, Data Warehouses, Dimensional Modeling, As an Associate Director for Big Data at Verizon, Sumit, strategized, managed, architected and developed analytic platforms for machine learning applications. Sumit was the Chief Architect at ModelN/LeapfrogRX (2006-2013), where he architected the core Analytics Platform.

Sumit has recently authored a book with Apress - called - "SQL On Big Data - Technology, Architecture and Roadmap". Sumit regularly speaks on the above topic in Big Data Conferences across USA.

Sumit has hiked to Mt. Everest Base Camp at 18.2K feet in Oct, 2016. Sumit is also an avid Badminton player and has won a bronze medal in 2015 in Connecticut Open in USA in the men's single category.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.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.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

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

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

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

Table of Contents

Preface

The SMACK stack is a generalized web-scale data pipeline. It was popularized in the San Francisco Bay Area data engineering meet ups and conferences and spread around the world. SMACK stands for:

- S = Spark: This involves data in-memory distributed computing. Think in Apache Flink, Apache Ignite, Google Millwheel, and so on.
- M = Mesos: This involves Cluster OS, distributed system management, scheduling and scaling. Think in Apache YARN, Kubernetes, Docker, and so on.
- A = Akka: This is the API. It is an implementation of the actor's model. Think in Scala, Erlang, Elixir, GoLang and so on.
- C = Cassandra: This is a persistence layer, noSQL database. Think in Apache HBase, Riak, Google BigTable, MongoDB, and so on.
- K = Kafka: This is a distributed streaming platform, the message broker. Think in Apache Storm, ActiveMQ, RabbitMQ, Kestrel, JMS, and so on.

During the years 2014, 2015, and 2016, surveys show that among all software developers, those with higher wages are the data engineers, the data scientists, and the data architects. This is because there is a huge demand for technical professionals in data and unfortunately for large organizations and fortunately for developers, there is a very low offer.

If you are reading this book, it is for two reasons: either you want to belong to best paid IT professionals, or you already belong and you want to learn how today's trends in the not too distant future will become requirements.

This book explains how to dominate the SMACK stack, which is also called the Spark++, because it seems to be the open stack that will succeed in the near future.

What this book covers

`Chapter 1`, *Introducing SMACK*, speaks about the fundamental SMACK architecture. We review the differences between the technologies in SMACK and the traditional data technologies. We also reviewed every technology in the SMACK and briefly expose each tool's potential.

`Chapter 2`, *The Model - Scala and Akka*, makes it easy by dividing the text into two parts: Scala (the language) and Akka (the actor model implementation for the JVM). It is a mini Scala Akka cookbook to learn through several exercises. The first half is for the fundamental parts of Scala, the second half is focused on the Akka actor model.

`Chapter 3`, *The Engine - Apache Spark*, reviews the key points of Apache Spark from scratch. It shows how to download, install, and test Apache Spark. We will learn how to run Spark applications. We will review the Spark core concepts such as RDD and the RDD operations: Transformations and Actions. We will see how to run Apache Spark in the cluster mode, how to run the Driver Program, and how to achieve High Availability. This chapter dives into Spark Streaming, the stateless and stateful Transformations, the Output Operations, how to enable high availability, and how to improve the Spark Streaming performance.

`Chapter 4`, *The Storage - Apache Cassandra*, speaks about Storage. The C in the SMACK stack refers to Cassandra, the database that propels some giants such as Walmart, CERN, Cisco, Facebook, Netflix, and Twitter. Spark uses a lot of Cassandra's power. The SMACK application's efficiency is greatly increased by the use of the Spark Cassandra Connector.

`Chapter 5`, *The Broker - Apache Kafka,* shows the reasons why Kafka was developed, reviews the Kafka installation, and its support for different types of clusters. It also explores the Kafka's design approach, and writes a few basic producers and consumers. Learn how to set up a Kafka cluster with single and multiple brokers on a single node, run producers and consumers from the command line, and exchange some messages. We also discussed some important settings about the Broker. Finally, we discussed Kafka's integration with technologies such as Spark.

`Chapter 6`, *The Manager - Apache Mesos,* reviews the Mesos Architecture, also how Mesos makes the resource allocation and the DRF algorithm. It shows how to run Mesos on AWS and on a private datacenter. We reviewed the most important Mesos frameworks: Marathon, Chronos, Aurora, and Singularity; also the Frameworks API. It shows how to run Spark, Cassandra, and Kafka on Apache Mesos, and also covers topics such as the setup, configuration, and management of these frameworks on a distributed infrastructure using Mesos.

`Chapter 7`, *Study case 1 - Spark and Cassandra*, discusses the relation between Spark and Cassandra. It shows the Spark Cassandra connector and how to make the Cassandra and Spark Context setup, Cassandra and Spark streaming, streaming context creation, reading and writing a stream from Cassandra, saving datasets, collections and Tuples to Cassandra, modifying collections, and saving UDTs and RDDs as tables. It also reviews the Calliope project: installing Calliope, reading and writing from Cassandra with CQL3, and writing and reading from Cassandra with Thrift.

`Chapter 8`, *Study case 2 - Connectors*, analyzes the Connectors, that is, the software pieces that make SMACK stack technologies can communicate with each other. The relationship between Spark and Kafka, is covered in the Kafka chapter; also, the relationship between Spark and Cassandra is so important that has its own chapter.

`Chapter 9`, *Study case 3 - Mesos and Docker*, explains how to develop Mesos Frameworks and how to use Mesos Containerizers and Docker Containerizers. This chapter also covers the Mesos API for Framework development. As a plus, it shows how to use Mesos Containerizers and Docker Containerizers.

What you need for this book

The reader should have some experience in programming (Java or Scala), some experience in Linux/Unix operating systems and the basics of databases:

- For Scala, the reader should know the basics about programming
- For Spark, the reader should know the fundamentals of Scala Programming Language
- For Mesos, the reader should know the basics of the Operating Systems administration
- For Cassandra, the reader should know the fundamentals of Databases
- For Kafka, the reader should have basic knowledge about Scala

Who this book is for

This book is for software developers, data architects, and data engineers looking for how to integrate the most successful Open Source Data stack architecture and how to choose the correct technology in every layer and also what are the practical benefits in every case.

There are a lot of books that talk about each technology separately. This book is for people looking for alternative technologies and practical examples on how to connect the entire stack.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "In the case of HDFS, we should change the `mesos.hdfs.role` in the file `mesos-site.xml` to the value of `role1.`"

A block of code is set as follows:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
     /etc/asterisk/cdr_mysql.conf
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.
To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.
If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

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

Downloading the example code

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

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.

4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

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

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

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Fast-Data-Processing-Systems-with-SMACK-Stack`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/FastDataProcessingSystemswithSMACKStack_ColorImages.pdf`.

Errata

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

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

Piracy

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

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

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

Questions

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

1
An Introduction to SMACK

The goal of this chapter is to present data problems and scenarios solved by architecture. This chapter explains how every technology contributes to the SMACK stack. It also explains how this modern pipeline architecture solves many of the modern problems related to data-processing environments. Here we will know when to use SMACK and when it is not suitable. We will also touch on the new professional profiles created in the new data management era.

In this chapter we will cover the following topics:

- Modern data-processing challenges
- The data-processing pipeline architecture
- SMACK technologies
- Changing the data center operations
- Data expert profiles
- Is SMACK for me?

Modern data-processing challenges

We can enumerate four modern data-processing problems as follows:

- **Size matters**: In modern times, data is getting bigger or, more accurately, the number of available data sources is increasing. In the previous decade, we could precisely identify our company's internal data sources: **Customer Relationship Management** (**CRM**), **Point of Sale** (**POS**), **Enterprise Resource Planning** (**ERP**), **Supply Chain Management** (**SCM**), and all our databases and legacy systems. Easy, a system that is not internal is external. Today, it is exactly the same, except not do the data sources multiply over time, the amount of information flowing from external systems is also growing at almost logarithmic rates. New data sources include social networks, banking systems, stock systems, tracking and geolocation systems, monitoring systems, sensors, and the Internet of Things; if a company's architecture is incapable of handling these use cases, then it can't respond to upcoming challenges.
- **Sample data**: Obtaining a sample of production data is becoming more difficult. In the past, data analysts could have a fresh copy of production data on their desks almost daily. Today, it becomes increasingly more difficult, either because of the amount of data to be moved or by the expiration date; in many modern business models data from an hour ago is practically obsolete.
- **Data validity**: The validity of an analysis becomes obsolete faster. Assuming that the fresh-copy problem is solved, how often is new data needed? Looking for a trend in the last year is different from looking for one in the last few hours. If samples from a year ago are needed, what is the frequency of these samples? Many modern businesses don't even have this information, or worse, they have it but it is only stored.
- **Data Return on Investment (ROI)**: Data analysis becomes too slow to get any return on investment from the info. Now, suppose you have solved the problems of sample data and data validity. The challenge is to be able to analyze information in a timely manner so that the return on investment of all our efforts is profitable. Many companies invest in data, but never get the analysis to increase their income.

We can enumerate modern data needs which are as follows:

- **Scalable infrastructure**: Companies, every time, have to weigh the time and money spent. Scalability in a data center means the center should grow in proportion to the business growth. Vertical scalability involves adding more layers of processing. Horizontal scalability means that once a layer has more demands and requires more infrastructures, hardware can be added so that processing needs are met. One modern requirement is to have horizontal scaling with low-cost hardware.
- **Geographically dispersed data centers**: Geographically centralized data centers are being displaced. This is because companies need to have multiple data centers in multiple locations for several reasons: cost, ease of administration, or access to users. This implies a huge challenge for data center management. On the other hand, data center unification is a complex task.
- **Allow data volumes to be scaled as the business needs**: The volume of data must scale dynamically according to business demands. So, as you can have a lot of demand at a certain time of day, you can have high demand in certain geographic regions. Scaling should be dynamically possible in time and space especially horizontally.
- **Faster processing**: Today, being able to work in real time is fundamental. We live in an age where data freshness matters many times more than the amount or size of data. If the data is not processed fast enough, it becomes stale quickly. Fresh information not only needs to be obtained in a fast way, it has to be processed quickly.
- **Complex processing**: In the past, the data was smaller and simpler. Raw data doesn't help us much. The information must be processed by several layers, efficiently. The first layers are usually purely technical and the last layers mainly business-oriented. Processing complexity can kill of the best business ideas.

- **Constant data flow**: For cost reasons, the number of data warehouses is decreasing. The era when data warehouses served just to store data is dying. Today, no one can afford data warehouses just to store information. Today, data warehouses are becoming very expensive and meaningless. The better business trend is towards flows or streams of data. Data no longer stagnates, it moves like large rivers. Make data analysis on big information torrents one of the objectives of modern businesses.
- **Visible, reproducible analysis**: If we cannot reproduce phenomena, we cannot call ourselves scientists. Modern science data requires making reports and graphs in real time to take timely decisions. The aim of science data is to make effective predictions based on observation. The process should be visible and reproducible.

The data-processing pipeline architecture

If you ask several people from the information technology world, we agree on few things, except that we are always looking for a new acronym, and the year 2015 was no exception.

As this book title says, SMACK stands for **Spark**, **Mesos**, **Akka**, **Cassandra,** and **Kafka**. All these technologies are open source. And with the exception of Akka, all are Apache Software projects. This acronym was coined by Mesosphere, a company that bundles these technologies together in a product called Infinity, designed in collaboration with Cisco to solve some pipeline data challenges where the speed of response is fundamental, such as in fraud detection engines.

SMACK exists because one technology doesn't make an architecture. SMACK is a pipelined architecture model for data processing. A data pipeline is software that consolidates data from multiple sources and makes it available to be used strategically.

It is called a pipeline because each technology contributes with its characteristics to a processing line similar to a traditional industrial assembly line. In this context, our canonical reference architecture has four parts: storage, the message broker, the engine, and the hardware abstraction.

For example, Apache Cassandra alone solves some problems that a modern database can solve but, given its characteristics, leads the storage task in our reference architecture.

Similarly, Apache Kafka was designed to be a message broker, and by itself solves many problems in specific businesses; however, its integration with other tools deserves a special place in our reference architecture over its competitors.

The NoETL manifesto

The acronym **ETL** stands for **Extract**, **Transform**, **Load**. In the database data warehousing guide, Oracle says:

Designing and maintaining the ETL process is often considered one of the most difficult and resource intensive portions of a data warehouse project.

For more information, refer to `http://docs.oracle.com/cd/B19306_01/server.102/b14223/ettover.htm`.

Contrary to many companies' daily operations, ETL is not a goal, it is a step, a series of unnecessary steps:

- Each ETL step can introduce errors and risk
- It can duplicate data after failover
- Tools can cost millions of dollars
- It decreases throughput
- It increases complexity
- It writes intermediary files
- It parses and re-parses plain text
- It duplicates the pattern over all our data centers

No ETL pipelines fit on the SMACK stack: Spark, Mesos, Akka, Cassandra, and Kafka. And if you use SMACK, make sure it's highly-available, resilient, and distributed.

A good sign you're having *Etlitis* is writing intermediary files. Files are useful in day to day work, but as data types they are difficult to handle. Some programmers advocate replacing a file system with a better API.

Removing the *E* in ETL: Instead of text dumps that you need to parse over multiple systems, Scala and Parquet technologies, for example, can work with binary data that remains strongly typed and represent a return to strong typing in the data ecosystem.

Removing the *L* in ETL: If data collection is backed by a distributed messaging system (Kafka, for example) you can do a real-time fan-out of the ingested data to all customers. No need to batch-*load.*

The *T* in ETL: From this architecture, each consumer can do their own transformations.

So, the modern tendency is: no more Greek letter architectures, no more ETL.

Lambda architecture

The academic definition is a data-processing architecture designed to handle massive quantities of data by taking advantage of both batch and stream processing methods. The problem arises when we need to process data streams in real time.

Here, a special mention for two open source projects that allow batch processing and real-time stream processing in the same application: Apache Spark and Apache Flink. There is a battle between these two: Apache Spark is the solution led by Databricks, and Apache Flink is a solution led by data artisans.

For example, Apache Spark and Apache Cassandra meets two modern requirements described previously:

- It handles a massive data stream in real time
- It handles multiple and different data models from multiple data sources

Most lambda solutions, as mentioned, cannot meet these two needs at the same time. As a demonstration of power, using an architecture based only on these two technologies, Apache Spark is responsible for real-time analysis of both historical data and recent data obtained from the massive information torrent. All such information and analysis results are persisted in Apache Cassandra. So, in the case of failure we can recover real-time data from any point of time. With lambda architecture it's not always possible.

Hadoop

Hadoop was designed to transfer processing closer to the data to minimize the amount of data shuffled across the network. It was designed with data warehouse and batch problems in mind; it fits into the *slow data* category, where size, scope, and completeness of data are more important than the speed of response.

The analogy is the sea versus the waterfall. In a sea of information you have a huge amount of data, but it is a static, contained, motionless sea, perfect to do Batch processing without time pressures. In a waterfall you have a huge amount of data, dynamic, not contained, and in motion. In this context your data often has an expiration date; after time passes it is useless.

Some Hadoop adopters have been left questioning the true return on investment of their projects after running for a while; this is not a technological fault itself, but a case of whether it is the right application. SMACK has to be analyzed in the same way.

SMACK technologies

SMACK is about a full stack for pipeline data architecture--it's Spark, Mesos, Akka, Cassandra, and Kafka. Further on in the book, we will also talk about the most important factor: the integration of these technologies.

Pipeline data architecture is required for online data stream processing, but there are a lot of books talking about each technology separately. This book talks about the entire full stack and how to perform integration.

This book is a compendium of how to integrate these technologies in a pipeline data architecture.

We talk about the five main concepts of pipeline data architecture and how to integrate, replace, and reinforce every layer:

- **The engine**: Apache Spark
- **The actor model**: Akka
- **The storage**: Apache Cassandra
- **The message broker**: Apache Kafka
- **The hardware scheduler**: Apache Mesos:

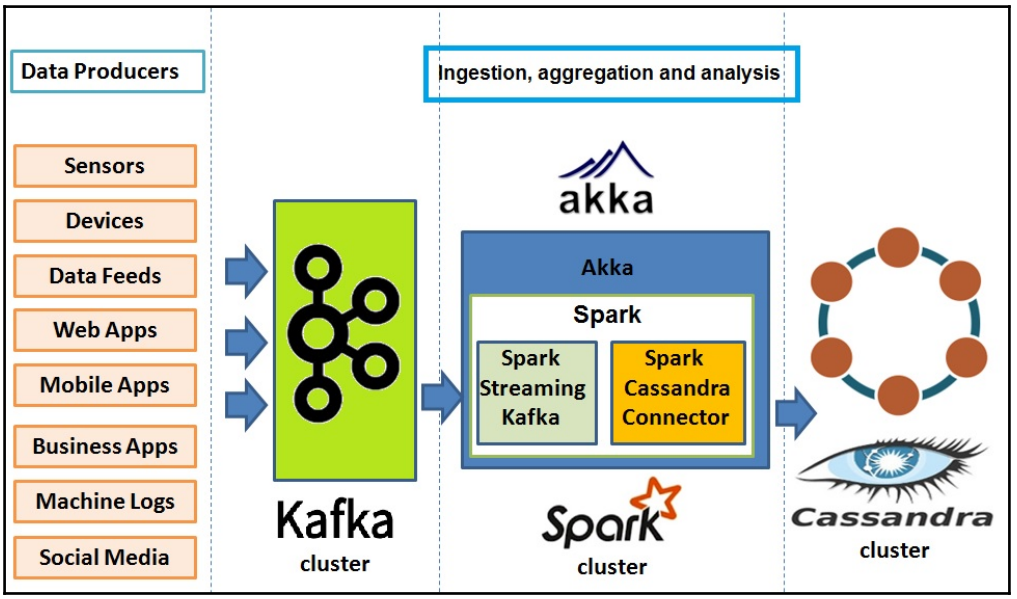

Figure 1.1 The SMACK pipeline architecture

Apache Spark

Spark is a fast and general engine for data processing on a large scale.

The Spark goals are:

- Fast data processing
- Ease of use
- Supporting multiple languages
- Supporting sophisticated analytics
- Real-time stream processing
- The ability to integrate with existing Hadoop data
- An active and expanding community

Here is some chronology:

- **2009**: Spark was initially started by Matei Zaharia at UC Berkeley AMPLab
- **2010**: Spark is open-sourced under a BSD license
- **2013**: Spark was donated to the Apache Software Foundation and its license to Apache 2.0
- **2014**: Spark became a top-level Apache Project
- **2014**: The engineering team at Databricks used Spark and set a new world record in large-scale sorting

As you are reading this book, you probably know all the Spark advantages. But here, we mention the most important:

- **Spark is faster than Hadoop**: Spark makes efficient use of memory and it is able to execute equivalent jobs 10 to 100 times faster than Hadoop's MapReduce.
- **Spark is easier to use than Hadoop**: You can develop in four languages: Scala, Java, Python, and recently R. Spark is implemented in Scala and Akka. When you work with collections in Spark it feels as if you are working with local Java, Scala, or Python collections. For practical reasons, in this book we only provide examples on Scala.
- **Spark scales differently than Hadoop**: In Hadoop, you require experts in specialized Hardware to run monolithic Software. In Spark, you can easily increase your cluster horizontally with new nodes with non-expensive and non-specialized hardware. Spark has a lot of tools for you to manage your cluster.

- **Spark has it all in a single framework**: The capabilities of coarse grained transformations, real-time data-processing functions, SQL-like handling of structured data, graph algorithms, and machine learning.

It is important to mention that Spark was made with **Online Analytical Processing** (**OLAP**) in mind, that is, batch jobs and data mining. Spark was not designed for **Online Transaction Processing** (OLTP), that is, fast and numerous atomic transactions; for this type of processing, we strongly advise the reader to consider the use of Erlang/Elixir.

Apache Spark has these main components:

- Spark Core
- Spark SQL
- Spark Streaming
- Spark MLIB
- Spark Graph

The reader will find that each Spark component normally has several books. In this book, we just mention the essentials of Apache Spark to meet the SMACK stack.

In the SMACK stack, Apache Spark is the data-processing engine; it provides near real-time analysis of data (note the word *near*, because today processing petabytes of data cannot be done in real time).

Akka

Akka is an actor model implementation for JVM, it is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM.

The open source Akka toolkit was first released in 2009. It simplifies the construction of concurrent and distributed Java applications. Language bindings exist for both Java and Scala.

It is message-based and asynchronous; typically no mutable data is shared. It is primarily designed for actor-based concurrency:

- Actors are arranged hierarchically
- Each actor is created and supervised by its parent actor
- Program failures treated as events are handled by an actor's supervisor
- It is fault-tolerant
- It has hierarchical supervision

- Customizable failure strategies and detection
- Asynchronous data passing
- Parallelized
- Adaptive and predictive
- Load-balanced

Apache Cassandra

Apache Cassandra is a database with the scalability, availability, and performance necessary to compete with any database system in its class. We know that there are better database systems; however, Apache Cassandra is chosen because of its performance and its connectors built for Spark and Mesos.

In SMACK, Akka, Spark, and Kafka can store the data in Cassandra as a data layer. Also, Cassandra can handle operational data. Cassandra can also be used to serve data back to the application layer.

Cassandra is an open source distributed database that handles large amounts of data; originally started by Facebook in 2008, it became a top-level Apache Project from 2010.

Here are some Apache Cassandra features:

- Extremely fast
- Extremely scalable
- Multi datacenters
- There is no single point of failure
- Can survive regional faults
- Easy to operate
- Automatic and configurable replication
- Flexible data modeling
- Perfect for real-time ingestion
- Great community

Apache Kafka

Apache Kafka is a distributed commit log, an alternative to publish-subscribe messaging.

Kafka stands in SMACK as the ingestion point for data, possibly on the application layer. This takes data from one or more applications and streams it across to the next points in the stack.

Kafka is a high-throughput distributed messaging system that handles massive data load and avoids back pressure systems to handle floods. It inspects incoming data volumes, which is very important for distribution and partitioning across the nodes in the cluster.

Some Apache Kafka features:

- High-performance distributed messaging
- Decouples data pipelines
- Massive data load handling
- Supports a massive number of consumers
- Distribution and partitioning between cluster nodes
- Broker automatic failover

Apache Mesos

Mesos is a distributed systems kernel. Mesos abstracts all the computer resources (CPU, memory, storage) away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to be built easily and run effectively.

Mesos was build using Linux kernel principles and was first presented in 2009 (with the name Nexus). Later in 2011, it was presented by Matei Zaharia.

Mesos is the foundation of several frameworks; the main three are:

- Apache Aurora
- Chronos
- Marathon

In SMACK, Mesos' task is to orchestrate the components and manage resources used.

Changing the data center operations

And here is the point where data processing changes data center operation.

From scale-up to scale-out

Throughout businesses we are moving from specialized, proprietary, and typically expensive supercomputers to the deployment of clusters of commodity machines connected with a low cost network.

The **Total Cost of Ownership** (**TCO**) determines the fate, quality, and size of a DataCenter. If the business is small, the DataCenter should be small; as the business demands, the DataCenter will grow or shrink.

Currently, one common practice is to create a dedicated cluster for each technology. This means you have a Spark cluster, a Kafka cluster, a Storm cluster, a Cassandra cluster, and so on, because the overall TCO tends to increase.

The open-source predominance

Modern organizations adopt open source to avoid two old and annoying dependencies: vendor lock-in and external entity bug fixing.

In the past, the rules were dictated from the classically large high-tech enterprises or monopolies. Today, the rules come from the people, for the people; transparency is ensured through community-defined APIs and various bodies, such as the Apache Software Foundation or the Eclipse Foundation, which provide guidelines, infrastructure, and tooling for the sustainable and fair advancement of these technologies.

There is no such thing as a free lunch. In the past, larger enterprises used to hire big companies in order to be able to blame and sue someone in the case of failure. Modern industries should take the risk and invest in training their people in open technologies.

Data store diversification

The dominant and omnipotent era of the relational database is challenged by the proliferation of NoSQL .

You have to deal with the consequences: systems of recording determination, synchronizing different stores, and correct data store selection.

Data gravity and data locality

Data gravity is related to considering the overall cost associated with data transfer, in terms of volume and tooling, for example, trying to restore hundreds of terabytes in a disaster recovery case.

Data locality is the idea of bringing the computation to the data rather than the data to the computation. As a rule of thumb, the more different services you have on the same node, the better prepared you are.

DevOps rules

DevOps refers to the best practices for collaboration between the software development and operational sides of a company.

The developer team should have the same environment for local testing as is used in production. For example, Spark allows you to go from testing to cluster submission.

The tendency is to containerize the entire production pipeline.

Data expert profiles

Well, first we will classify people into four groups based on skill: data architect, data analyst, data engineer, and data scientist.

Usually, data skills are separated into two broad categories:

1. **Engineering skills**: All the DevOps (yes, DevOps is the new black): setting up servers and clusters, operating systems, write/optimize/distribute queries, network protocol knowledge, programming, and all the stuff related to computer science
2. **Analytical skills**: All mathematical knowledge: statistics, multivariable analysis, matrix algebra, data mining, machine learning, and so on.

Data analysts and data scientists have different skills but usually have the same mission in the enterprise.

Data engineers and data architects have the same skills but usually different work profiles.

Data architects

Large enterprises collect and generate a lot of data from different sources:

1. **Internal sources**: Owned systems, for example, CRM, HRM, application servers, web server logs, databases, and so on.
2. **External sources**: For example, social network platforms (WhatsApp, Twitter, Facebook, Instagram), stock market feeds, GPS clients, and so on.

Data architects:

- Understand all these data sources and develop a plan for collecting, integrating, centralizing, and maintaining all the data
- Know the relationship between data and current operations, and understand the effects that any process change has on the data used in the organization
- Have an end-to-end vision of the processes, and see how a logical design maps a physical design, and how the data flows through every stage in the organization
- Design data models (for example, relational databases)
- Develop strategies for all data lifecycles: acquisition, storage, recovery, cleaning, and so on

Data engineers

A data engineer is a hardcore engineer who knows the internals of the data engines (for example, database software).

Data engineers:

- Can install all the infrastructure (database systems, file systems)
- Write complex queries (SQL and NoSQL)
- Scale horizontally to multiple machines and clusters
- Ensure backups and design and execute disaster recovery plans
- Usually have low-level expertise in different data engines and database software

Data analysts

Their primary tasks are the compilation and analysis of numerical information.

Data analysts:

- Have computer science and business knowledge
- Have analytical insights into all the organization's data
- Know which information makes sense to the enterprise
- Translate all this into decent reports so the non-technical people can understand and make decisions
- Do not usually work with statistics
- Are present (but specialized) in mid-sized organizations for example, sales analysts, marketing analysts, quality analysts, and so on
- Can figure out new strategies and report to the decision makers

Data scientists

This is a modern phenomenon and is usually associated with modern data. Their mission is the same as that of a data analyst but, when the frequency, velocity, or volume of data crosses a certain level, this position has specific and sophisticated skills to get those insights out.

Data scientists:

- Have overlapping skills, including but not limited to: Database system engineering (DB engines, SQL, NoSQL), big data systems handling (Hadoop, Spark), computer language knowledge (R, Python, Scala), mathematics (statistics, multivariable analysis, matrix algebra), data mining, machine learning, and so on
- Explore and examine data from multiple heterogeneous data sources (unlike data analysts)
- Can sift through all incoming data to discover a previously hidden insight
- Can make inductions, deductions, and abductions of data to solve a business problem or find a business pattern (usually data analysts just make inductions of from data)
- The best don't just address known business problems, they find patterns to solve new problems and add value to the organization

As you can deduce, this book is mainly focused on the data architect and data engineer profiles. But if you're an enthusiastic data scientist looking for more wisdom, we hope to be useful to you, too.

Is SMACK for me?

Some large companies are using a variation of SMACK in production, particularly those looking at how to take their pipeline data projects forward.

Apache Spark is beginning to attract more large software vendors to support it as it fulfils different needs than Hadoop.

SMACK is becoming a new modern requirement for companies as they move from the initial pilot phases into relying on pipeline data for their revenues.

The point of this book is to give you alternatives.

One example involves replacing individual components. Yarn could be used as the cluster scheduler instead of Mesos, while Apache Flink would be a suitable batch and stream processing alternative to Akka. There are many alternatives to SMACK.

The fundamental premise of SMACK is to build an end-to-end data-processing pipeline having these components interacting in a way that makes integration simple and getting tasks up-and-running is quick, rather than requiring huge amounts of effort to get the tools to play nicely with each other.

Summary

This chapter was full of theory. We reviewed the fundamental SMACK architecture. We also reviewed the differences between Spark and traditional big data technologies such as Hadoop and MapReduce.

We also reviewed every technology in SMACK and we briefly exposed each tool's is addressed in the following chapters potential. Each every technology. We will explore connectors and integration practices, as well as describing technology alternatives in every case.

2
The Model - Scala and Akka

This chapter is divided into two parts: Scala (the language) and Akka (the actor model implementation for the JVM).

As this book is about architecture, and Spark is built in Scala following an actor's model, in this book we decided to show examples only using Scala as the language. In this way, we have made room for architectural issues preventing lead content.

In the Apache Spark world, there are four spoken languages: Java, Scala, Python, and R. To continue with our training, we need to know one of these four languages. Most books expose all examples in each of these languages.

If you are reading this section and do not know Scala, welcome to the introduction course for data manipulation. This chapter is a dojo where you will learn some Scala tricks to manipulate data (because it is not a book about Scala, some powerful topics were not mentioned, such as null-less containers, for example, option, either, try, pattern matching, and case classes). It is always good to learn a new language and this is your chance. If you have already mastered Scala, this chapter will allow you to improve your techniques with a series of **Katas** (Kata code is a programming exercise that helps a programmer hone their skills through practice and repetition).

The second part of this chapter focuses on the Akka actor's model implementation, so in the following chapters you will be able to understand the architecture and operation of Spark.

In this chapter, we will cover the following topics:

- The language - Scala
    - Kata 1 - The collections hierarchy
    - Kata 2 - Choosing the right collection
    - Kata 3 - Iterating with `foreach`
    - Kata 4 - Iterating with `for`
    - Kata 5 - Iterators
    - Kata 6 - Transforming with map
    - Kata 7 - Flattening
    - Kata 8 - Filtering
    - Kata 9 - Subsequences
    - Kata 10 - Splitting
    - Kata 11 - Extracting unique elements
    - Kata 12 - Merging
    - Kata 13 - Lazy views
    - Kata 14 - Sorting
    - Kata 15 - Streams
    - Kata 16 - Arrays
    - Kata 17 - ArrayBuffer
    - Kata 18 - Queues
    - Kata 19 - Stacks
    - Kata 20 - Ranges
- The model - Akka
    - Kata 21 - Actors
    - Kata 22 - Actor communication
    - Kata 23 - Actor life cycle
    - Kata 24 - Starting actors
    - Kata 25 - Stopping actors
    - Kata 26 - Killing actors
    - Kata 27 - Shutting down the actor system
    - Kata 28 - Actor monitoring
    - Kata 29 - Looking up actors

The language - Scala

The objective of this section is to think in a functional programming way.

As good data architects, here we will understand collections. We will not cover other issues of the language other than collection management.

We need to be clear regarding the following two statements:

- Scala collections are different from Java collections
- Scala collections are different from Spark collections

So, a list in Java is different from a list in Scala. Lists are a fundamental part of functional languages. The first functional programming language, **LISP**, is an acronym for **List Processing**.

We have to master three key concepts of functional programming to understand Scala collections:

- Predicates
- Literal functions (anonymous functions)
- Implicit loops

A predicate is just a function that receives several parameters and returns a Boolean value.

For example:

```
def isOdd (i: Int) = if (i % 2 != 0) true else false
```

A literal function is an alternate syntax for defining a function. It's useful when we want to pass a function as an argument to a method (especially to higher-order functions such as a fold or a filter operation) but do not want to define a separate function.

For example, the previous function can be rewritten as:

```
(i: Int) => i % 2 != 0
```

When examining this code, it's helpful to think of the => symbol as a *transformer*. It is important to know that it's a keyword used for function types and function literals.

The expression evaluates the parameter list on the left side of the => symbol (in this case, an Int named i) into a new value using the code on the right side of the symbol.

The same function could be written as:

```
_ % 2 != 0
```

Combined with the filter method, we find expressions such as:

```
scala> val nums = List.range(1, 10)
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
scala> val odds = nums.filter(_ % 2 != 0)
events: List[Int] = List(1, 3, 5, 7, 9)
```

As we can see, in the third concept, this code is equivalent to a loop. The filter code could be rewritten using a `for` loop over the collection, but here we are elegantly functional programmers and we avoid loops.

Kata 1 - The collections hierarchy

In the beginning there existed **Traversable**, and its family characteristic is the implementation of the foreach behavior.

Traversable only has one descendant: **Iterable**. A characteristic of the Iterable behavior is the implementation of the iterator method.

Iterable has three children: **Seq**, **Set,** and **Map**.

Seq (an abreviation of **Sequence**) is our first heroine. She has three main children: **IndexedSeq**, **LinearSeq,** and **Buffer.**

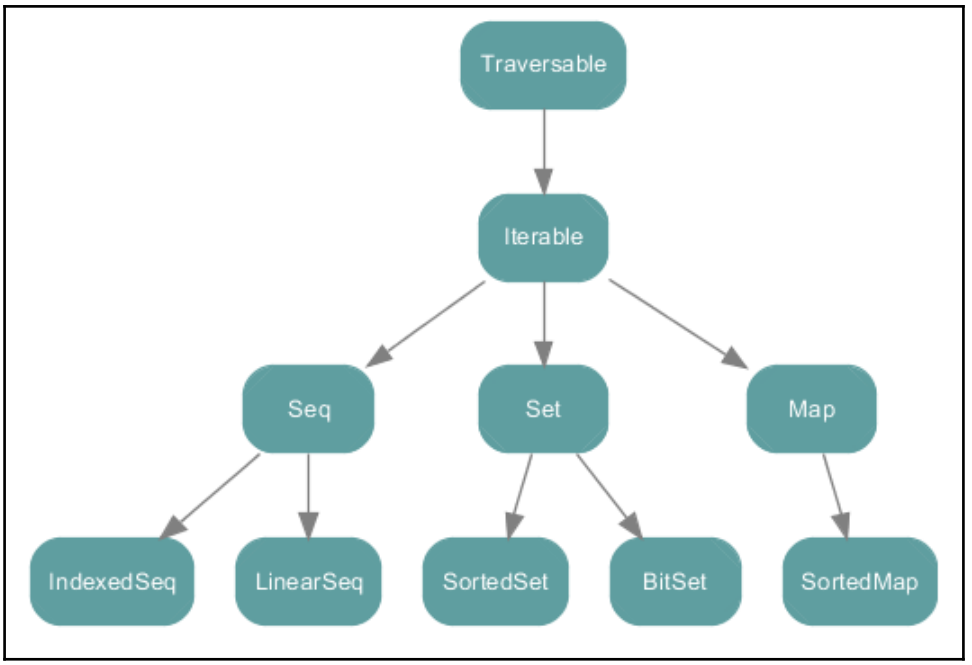

Figure 2-1. The Scala collections top hierarchy.

Sequence

The sequence hierarchy has three main families: IndexedSeq, LinearSeq, and Buffer.

- **IndexedSeq**: This implements the random access of elements in an efficient way. We can say Arry(365), where Arry is an array and we can access the 365th element of Arry in this way.
- **LinearSeq**: This is the implementation of a linked list as we all know it in the functional world. We have three fundamental (functional list) methods: head, tail, and isEmpty

- **Buffer**: This allows updates on existing elements, insertion, removal, and the efficient addition of new elements at the end of the buffer. The two most common implementations of buffers are **ListBuffer** and **ArrayBuffer.**

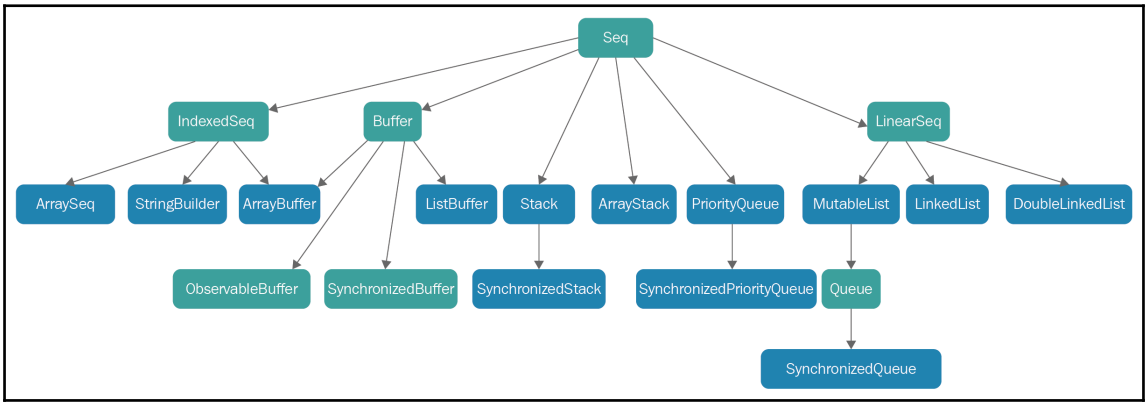

Figure 2-2- The Seq hierarchy

Map

A **Map** is a collection of key/value pairs. This popular data structure in the Java world is called **Map**, in Ruby it is known as Hash, and in Python it's called a dictionary.

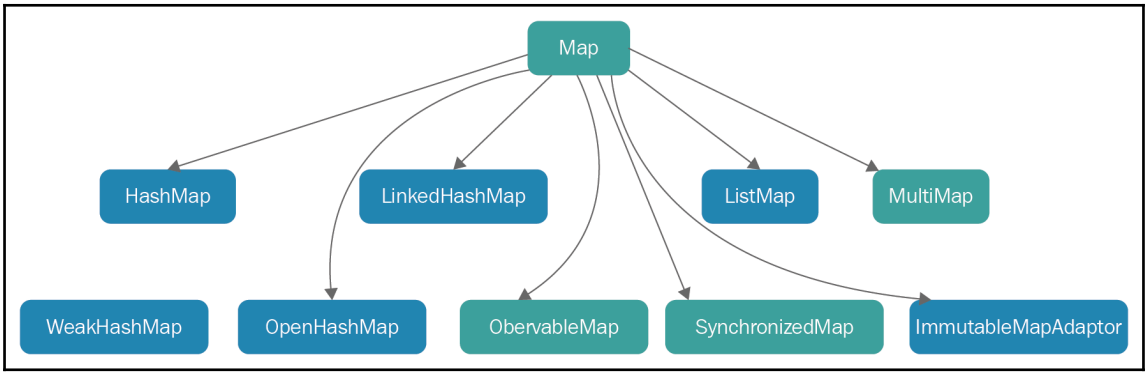

Figure 2-3- The Map hierarchy

When we need an immutable map, we create it without importing it:

```
scala> val m = Map(1 -> "uno", 2 -> "dos")
m: scala.collection.immutable.Map[Int,java.lang.String] = Map(1 -> uno, 2
-> dos)
```

Set

A set is a collection of unique elements. Its behavior is similar to the Java set. Operations on sets fall into four categories:

- **Tests**: `contains`, `apply`, `subsetOf`
- **Additions**: + (one element) and ++ (another set)
- **Removals**: – (one element) and –– (another set)
- **Classic**: `union`, `intersect`, `diff`

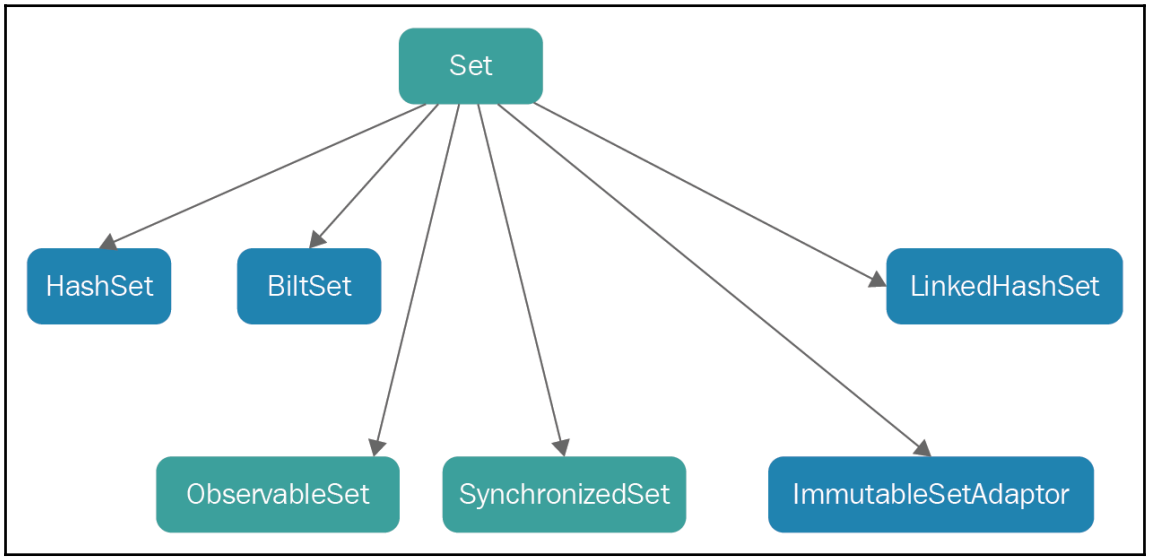

Figure 2-4- The Set hierarchy

When we need a `Set`, we create it without importing it:

```
scala> val mySet = Set(-1, 0, 1)
m: scala.collection.immutable.Set[Int] = Set(-1, 0, 1)
```

Kata 2 - Choosing the right collection

As we have seen in Scala, there are just three types of collection:

- Sequence
- Map
- Set

The crucial decision is whether we want a mutable or an immutable collection for each type.

Sequence

The decision here is whether we need an indexed sequence (such as an array) or a linear sequence (such as a linked list).

	Immutable	**Mutable**
`IndexedSeq`	Vector	ArrayBuffer
`LinearSeq`	List	ListBuffer

Table 2-1. Sequence collections

Let's see each in detail:

- **Mutable collection**: can be updated or extended in place so we can change, add, or remove elements of a collection as a side-effect
- **Immutable collections**: never changes, so we still have operations that simulate additions, removals, or updates, but in each case - return a new collection and leave the old collection unchanged

In the case of `IndexedSeq`:

`IndexedSeq`	**Use in case of**
Range	Limited integer list
String	Sequence of chars
Vector	Immutable and indexed vanilla-flavored list

In the case of `LinearSeq`:

`LinearSeq`	**Use in case of**
List	You need a functional, pure, and recursive list
Queue	First In First Out (FIFO)
Stack	Last In First Out (LIFO)
Stream	Infinite sequence, lazy, and persistent

Table 2-2. Immutable sequences

Immutable collections never change after they are created. So, we can rely on the fact that accessing the same collection value repeatedly at different points in code and time will always yield a collection with the same elements. All modification operations on immutable collections return a new instance with an operation applied.

In the case of `IndexedSeq`:

`IndexedSeq`	**Use in case of**
Array	Elements are mutable but the length is a constant
ArrayBuffer	Mutable and indexed vanilla-flavored list
ArrayStack	LIFO when performance matters
StringBuilder	Efficient string building inside loops

In case of `LinearSeq`:

`LinearSeq`	**Use in case of**
DoubleLinkedList	Linked list with `prev` method
LinkedList	Famous in data structure courses
ListBuffer	Array buffer but as a list
MutableList	List for lovers of non-functional programming
Queue	FIFO for non-functional programmers
Stack	LIFO for non-functional developers

Table 2-3. Mutable sequences

Mutable collections are known to have some operations that change a collection in place. So dealing with them means you need to understand exactly where (which code) changes which collection, and when. This is an aberration for functional programmers.

Map

Consider two decisions. Do you want it mutable? Do you want elements sorted?

Let's view this in tabular format:

Immutable	**Use in case of**
HashMap	Implemented using a tree
ListMap	Elements returned inverse as they were inserted
Map	Map vanilla flavor
SortedMap	Keys stored in sorted order
TreeMap	Sorted Map, the red-black tree of the books

Mutable	**Use in case of**
HashMap	Implemented using a hash table
LinkedHashMap	Elements returned as they were inserted
ListMap	Elements returned inverse as they were inserted
Map	Map vanilla flavor

Table 2-4. Map collections

Fundamental operations on maps are similar to those on sets:

- **Lookup operations**: `apply`, `get`, `getOrElse`, `contains`, `isDefinedAt`.
- **Additions**: + (one element), ++ (another Map)
- **Removals**: – (one element), –– (another Map)
- **Subcollection producers**: `keys`, `keySet`, `keysIterator`, `values`, `valuesIterator` (returns different collections of keys)
- **Transformations**: `filterKeys` and `mapValues` (produces a new map by filtering and transforming an existing map)

Set

The same two questions apply. Do you want it mutable? Do you want elements sorted?

Let's view this in tabular format:

Immutable	Use in case of
BitSet	Saves memory, just integers allowed
HashSet	Implemented using a tree
ListSet	Set in the outside, list in the inside
TreeSet	Immutable, set using a tree
Set	Set for the friends
SortedSet	Tree set ordered

Mutable	Use in case of
BitSet	Saves memory, just integers allowed
HashSet	Implemented using a hash table
LinkedHashSet	Elements returned as they were inserted
TreeSet	The AVL tree in the data structure books

Table 2-5. Set collections

Mutable sets also provide `add` and `remove` as variants of the operators `+=` and `-=`.

We can replace a mutable collection stored in a `val` with an immutable collection stored in a `var`, and vice versa.

Kata 3 - Iterating with foreach

The `foreach` method takes a function as an argument. This function should have one parameter and a unit return type, which could be thought of as void in Java. The parameter type should be the same as the type of the elements in the collection. It proceeds element by element:

```
scala> val x = Vector("un", "dos", "tres")
x: scala.collection.immutable.Vector[String] = Vector(un, dos, tres)

scala> x.foreach(s => print(s))
undostres
```

For example, this function takes one char and prints it:

```
scala> def printChar( c: Char) { print(c) }
printChar: (c: Char)Unit
```

This is the same function is applied to a string (a character sequence):

```
scala> "SMACK".foreach( c => printChar(c) )
SMACK
```

This is the type inference is a powerful tool in functional languages. Here the string is considered a char sequence:

```
scala> "SMACK".foreach( printChar )
SMACK
```

The same as the function literal:

```
scala> "SMACK".foreach( (c: Char) => print(c) )
SMACK
```

The same as using type inference and a function literal:

```
scala> "SMACK".foreach( print )
SMACK
```

An example using an implicit iteration over collections is:

```
scala> "SMACK: Spark Mesos Akka Cassandra Kafka".split(" ")
Array[String] = Array(SMACK:, Spark, Mesos, Akka, Cassandra, Kafka)
```

Kata 4 - Iterating with for

We can iterate a collection's elements with the `for` loop. If we want a new collection, we use a `for`/`yield` combo.

As we said, if it is `Traversable` or is `Iterable`:

```
scala> val smack = Traversable("Spark", "Mesos", "Akka", "Cassandra",
"Kafka")
smack: Traversable[String] = List(Spark, Mesos, Akka, Cassandra, Kafka)

scala> for (f <- smack) println(f)
Spark
Mesos
Akka
Cassandra
Kafka

scala> for (f <- smack) println( f.toUpperCase )
SPARK
MESOS
AKKA
CASSANDRA
KAFKA
```

A new collection is built every time when a transformation is applied to the source collection. To assign the returned result to a named value, we use the `for`/`yield` construct:

```
scala> val smack = Array("Spark", "Mesos", "Akka", "Cassandra", "Kafka")
smack: Array[java.lang.String] = Array(Spark, Mesos, Akka, Cassandra,
Kafka)
scala> val myArray = for (s <- smack) yield s.toUpperCase
newArray: Array[java.lang.String] = Array(SPARK, MESOS, AKKA,, CASSANDRA,
KAFKA)
```

This `for`/`yield` use is called *for comprehension*.

For comprehensions in Scala are syntactic sugar for the composition of multiple operations with `foreach`, `Map`, `flatMap`, `filter`, or `withFilter`. Scala actually translates a `for`-expression into calls to those methods, so any class providing them, or a subset of them, can be used with for comprehensions.

Now, let's iterate a `Map` with a for loop:

```
scala> val smack = Map("S" ->"Spark", "M" -> "Mesos", "A" -> "Akka", "C"
->"Cassandra", "K" -> "Kafka")
smack: scala.collection.immutable.Map[String,String] = Map(A -> Akka, M ->
Mesos, C -> Cassandra, K -> Kafka, S -> Spark)

scala> for ((k,v) <- smack) println(s"letter: $k, means: $v")
letter: A, means: Akka
letter: M, means: Mesos
letter: C, means: Cassandra
letter: K, means: Kafka
letter: S, means: Spark
```

Kata 5 - Iterators

In Java, the way to iterate a collection is with the `hasNext()` and `next()` methods.

In Scala we don't use these methods because we have the `Map` and `foreach` methods.

The only reasonable use of iterators in Scala is when we are reading very large files, because it is impractical to load the entire file in memory.

An `Iterator` is exhausted after it has been used; for example:

```
scala> val ite = Iterator("S","M","A")
ite: Iterator[String] = non-empty iterator
scala> ite.foreach(println)
S
M
A

scala> ite.foreach(println)
```

The last line doesn't produce any output; the iterator is exhausted.

Kata 6 - Transforming with map

Another way to transform collections, in addition to `for/yield` is by calling the Map method by passing it a function. For example:

```
scala> val smack = Vector("spark", "mesos", "akka", "cassandra", "kafka")
smack: scala.collection.immutable.Vector[String] = Vector(spark, mesos,
akka, cassandra, kafka)

// long expression
scala> val cap = smack.map(e => e.capitalize)
cap: scala.collection.immutable.Vector[String] = Vector(Spark, Mesos, Akka,
Cassandra, Kafka)

// short expression
scala> val cap = smack.map(_.capitalize)
cap: scala.collection.immutable.Vector[String] = Vector(Spark, Mesos, Akka,
Cassandra, Kafka)

//producing a Vector of Int
scala> val lens = smack.map(_.size)
lens: scala.collection.immutable.Vector[Int] = Vector(5, 5, 4, 9, 5)

//producing a Vector of XML elements
scala> val elem = smack.map(smack => <li>{smack}</li>)
elem: scala.collection.immutable.Vector[scala.xml.Elem] =
Vector(<li>spark</li>, <li>mesos</li>, <li>akka</li>, <li>cassandra</li>,
<li>kafka</li>)
```

Experience will enable you to identify which comprehension to use, `for/yield` or `map`:

```
scala> val smack = List("spark", "mesos", "akka", "cassandra", "kafka")
smack: List[String] = List(spark, mesos, akka, cassandra, kafka)

// map
scala> val m = smack.map(_.capitalize)
m: List[String] = List(Spark, Mesos, Akka, Cassandra, Kafka)

// for/yield
scala> val y = for (s <- smack) yield s.capitalize
y: List[String] = List(Spark, Mesos, Akka, Cassandra, Kafka)
```

Kata 7 - Flattening

Flattening is when we transform a list of lists (a multilist, a sequence of sequences) to one list (a sequence):

```
scala> val couples = List(List("Rachel","Ross"), List("Monica","Chandler"))
couples: List[List[String]] = List(List(Rachel, Ross), List(Monica,
Chandler))

scala> val friends = couples.flatten
friends: List[String] = List(Rachel, Ross, Monica, Chandler)
```

Here we capitalize, flatten, and sort lexicographically all in one sentence:

```
scala> val friends = couples.flatten.map(_.toUpperCase).sorted
friends: List[String] = List(CHANDLER, MONICA, RACHEL, ROSS)
```

When we work with a network, flattening could benefit some graphs:

```
val myFriends = List("Monica", "Ross")
val monicaFriends = List("Rachel", "Phoebe")
val rossFriends = List("Rachel", "Chandler", "Joy")

val friendsOfFriends = List( monicaFriends, rossFriends)

scala> val uniqueFriends = friendsOfFriends.flatten.distinct
uniqueFriends: List[String] = List(Rachel, Phoebe, Chandler, Joy)
```

Flattening a string is to produce a `List` of its chars:

```
scala> val myList = List("SMACK", "stack")
myList: List[String] = List(SMACK, stack)

scala> myList.flatten
List[Char] = List(S, M, A, C, K, s, t, a, c, k)
```

Flattening removes `None` elements and strips `Some` content:

```
scala> val boxes = Vector(Some("A"), None, Some(3), None)
boxes: scala.collection.immutable.Vector[Option[Any]] = Vector(Some(A),
None, Some(3), None)

scala> boxes.flatten
res1: scala.collection.immutable.Vector[Any] = Vector(A, 3)
```

Kata 8 - Filtering

Filtering a collection involves applying a predicate to each element. For example:

```
scala> val myRange = List.range(1, 10)
myRange: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> val odds = x.filter(_ % 2 != 0)
odds: List[Int] = List(1, 3, 5, 7, 9)

scala> val states = Set("California", "Arizona", "New Mexico", "Texas")
states: scala.collection.immutable.Set[String] = Set(California, Arizona,
New Mexico, Texas)

scala> val c = states.filter(_.startsWith("C"))
c: scala.collection.immutable.Set[String] = Set(California)

scala> val s = states.filter(_.length > 7)
s: scala.collection.immutable.Set[String] = Set(California, New Mexico)
```

As a rule of thumb, filtering has two conditions:

- We only keep elements whose predicate returns `true`
- The filter doesn't modify the `collection`; we must keep the result in a new variable

Kata 9 - Subsequences

There are some methods to extract a sub-sequence from a sequence:

```
// We declare a range of Int from -5 to 5
scala> val myArray = (-5 to 5).toArray
myArray: Array[Int] = Array(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5)

// drop the first N elements
scala> val d = myArray.drop(3)
d: Array[Int] = Array(-2, -1, 0, 1, 2, 3, 4, 5)

// drop all the elements which predicate is true
scala> val dw = myArray.dropWhile(_<2)
dw: Array[Int] = Array(2, 3, 4, 5)

// drop the last N elements
scala> val dr = myArray.dropRight(4)
dr: Array[Int] = Array(-5, -4, -3, -2, -1, 0, 1)
```

```
// take the first N elements
scala> val t = myArray.take(3)
t: Array[Int] = Array(-5, -4, -3)

// take all the elements which predicate is true
scala> val tw = myArray.takeWhile(_<2)
tw: Array[Int] = Array(-5, -4, -3, -2, -1, 0, 1)

// take the last N elements
scala> val tr = myArray.takeRight(3)
tr: Array[Int] = Array(3, 4, 5)


// the subsequence from index A to index B
scala> val sl = myArray.slice(1,3)
sl: Array[Int] = Array(-4, -3)
```

Here is a list methods to be fully functional:

```
// the first (head) element
scala> val h = myArray.head
h: Int = -5

// the first (head) element boxed
scala> val ho = myArray.headOption
ho: Option[Int] = Some(-5)

// all the elements except the last one
scala> val h = myArray.init
h: Array[Int] = Array(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4)

// the last element
scala> val la = myArray.last
la: Int = 5

// the last element boxed
scala> val la = myArray.lastOption
la: Option[Int] = Some(5)

// all the elements except the first one
scala> val t = myArray.tail
t: Array[Int] = Array(-4, -3, -2, -1, 0, 1, 2, 3, 4, 5)
```

Kata 10 - Splitting

Here, as in SQL, we have methods for making groups and splitting the world in two for those who meet the predicate and those who do not:

```
// having this beauty list
scala> val myList = List(-15, -10, -5, 10, 20, 15)
myList: List[Int] = List(-15, -10, -5, 10, 20, 15)

// in one group those greater than 10
scala> val x = myList.groupBy(_ > 10)
x: scala.collection.immutable.Map[Boolean,List[Int]] = Map(false ->
List(-15, -10, -5, 10), true -> List(20, 15))


// the groups generated can be accessed like those
scala> val t = x(true)
t -> List(20, 15))

scala> val f = x(false)
f -> List(-15, -10, -5, 10)

// partition is the same as groupBy but returns a List with two Lists
scala> val x = myList.partition(_ > 10)
x: (List[Int], List[Int]) = (List(20, 15),List(-15, -10, -5, 10))

// One sublist till the longest index who meets the predicate, and the rest
scala> val x = myList.span(_ < 20)
x: (List[Int], List[Int]) = (List(-15, -10, -5, 10),List(20, 15))

// Two lists, one before the index N, and the rest
scala> val x = myList.splitAt(2)
x: (List[Int], List[Int]) = (List(-15, -10),List(-5, 10, 20, 15))

// The result in a Tuple
scala> val (a,b) = myList.partition(_ > 10)
a: List[Int] = List(-15, -10),
b: List[Int] = List(-5, 10, 20, 15))
```

Kata 11 - Extracting unique elements

Let's see how to remove duplicates in a `collection`:

```
scala> val myList = List(-15, -15, 10, 10, 20, 15)
myList: List[Int] = List(-15, -15, 10, 10, 20, 15)
// As seen in previous examples, using distinct
scala> val x = myList.distinct
x: List[Int] = List(-15, 10, 20, 15)

// Another way is converting the Collection to a Set
// (by definition, a set can not contain duplicates)
scala> val s = myList.toSet
s: scala.collection.immutable.Set[Int] = Set(-15, 10, 20, 15)
```

Kata 12 - Merging

For merging and subtracting, we use the methods ++ and --:

```
// The ++= method could be used in any mutable collection

scala> val n = collection.mutable.ListBuffer(-4, -3, -2, -1)
n: scala.collection.mutable.ListBuffer[Int] = ListBuffer(-4, -3, -2, -1)
// The result is a mutable

scala> n ++= Seq(1,2,3,4)
res0: n.type = ListBuffer(-4, -3, -2, -1, 1, 2, 3, 4)


scala> val array1 = Array(-300,-200,-100)
array1: Array[Int] = Array(-300,-200,-100)

scala> val array2 = Array(50,150,250)
array2: Array[Int] = Array(50,150,250)

// The ++ method merge two collections and assign a new variable
scala> val mer = array1 ++ array2
mer: Array[Int] = Array(-300, -200, -100, 50, 150, 250)
```

We have all classic operations in Venn diagrams:

```
scala> val colors1 = Array("blue", "yellow", "green")
colors1: Array[String] = Array("blue", "yellow", "green")

scala> val colors2 = Array("yellow", "red", "orange")
colors2: Array[String] = Array("yellow", "red", "orange")
```

```
// intersect: the elements in both collections
scala> val colors3 = colors1.intersect(colors2)
colors3: Array[String] = Array(yellow)

// union: the elements in both collections
scala> val colors4 = colors1.union(colors2)
colors4: Array[String] = Array(blue, yellow, green, yellow, red, orange)

// distinct: remove duplicates
scala> val colors5 = colors1.union(colors2).distinct
colors5: Array[String] = Array(blue, yellow, green, red, orange)
```

The *set difference* (`diff` method) results depend on which sequence it's called on:

```
scala> val diff1 = colors1 diff colors2
diff1: Array[String] = Array(blue, green)

scala> val diff2 = colors2 diff colors1
diff2: Array[String] = Array(red, orange)
```

Kata 13 - Lazy views

A **lazy view** is a version of a collection computed and returned when it is actually needed.

In the JVM, all the memory is allocated immediately when the collection is created.

The difference between these two lines could save all the memory in your project:

```
scala> -50 to 50
res0: scala.collection.immutable.Range.Inclusive = Range(-50, -49, -48,
-47, -46, -45, -44, -43, -42, -41, -40, -39, -38, -37, -36, -35, -34, -33,
-32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21, -20, -19, -18,
-17, -16, -15, -14, -13, -12, -11, -10, -9, -8, -7, -6, -5, -4, -3, -2, -1,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50)

scala> (-50 to 50).view
res0:
scala.collection.SeqView[Int,scala.collection.immutable.IndexedSeq[Int]] =
SeqView(...)
```

To allocate all the memory in a view, we use the force instruction:

```
scala> val v = (-50 to 50).view
v: scala.collection.SeqView[Int,scala.collection.immutable.IndexedSeq[Int]]
= SeqView(...)

scala> val f = v.force
f: scala.collection.immutable.IndexedSeq[Int] = Vector(-50, -49, -48, -47,
-46, -45, -44, -43, -42, -41, -40, -39, -38, -37, -36, -35, -34, -33, -32,
-31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21, -20, -19, -18, -17,
-16, -15, -14, -13, -12, -11, -10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0,
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48, 49, 50)
```

Using views with the map method improves performance (as functional programming kung-fu):

```
//increase the numbers in both sentences and see your CPU struggle
scala> (-100 to 100).map { _ * 2 }
res0: scala.collection.immutable.IndexedSeq[Int] = Vector(-200, -198, -196,
-194, -192, -190, -188, -186, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56,
58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94,
96, 9...

scala> (-100 to 100).view.map { _ * 2 }
res0: scala.collection.SeqView[Int,Seq[_]] = SeqViewM(...)
```

Functional programmers are very familiar with the two benefits view provide:

- Performance (a matter of life or death when the amount of data is huge)
- Using a data structure similar to the one we used with database views

Database views were created to allow modifications on a large table without compromising performance:

```
// create an array
scala> val a = (0 to 9).toArray
a: Array[Int] = Array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

// a view on the first elements of the array
scala> val view = a.view.slice(0, 5)
view: scala.collection.mutable.IndexedSeqView[Int,Array[Int]] =
SeqViewS(...)

// modify the view elements
scala> view(0) = 10
```

```
scala> view(1) = 20
scala> view(2) = 30

// so, the original array is affected
scala> a
res0: Array[Int] = Array(10, 20, 30, 3, 4, 5, 6, 7, 8, 9)
```

Kata 14 - Sorting

We use the sorted method with the operators <, <=, >, and >=:

```
// sorting a list of Strings
scala> val states = List("California", "Arizona", "New Mexico",
"Texas").sorted
states: List[String] = List(Arizona, California, New Mexico, Texas)

// sorting a list of numbers
scala> val nums = List(10, 1, 8, 3, 5).sorted
nums: List[Int] = List(1, 3, 5, 8, 10)

// sorting ascending
scala> List(10, 1, 8, 3, 5).sortWith(_ < _)
res0: List[Int] = List(1, 3, 5, 8, 10)

// sorting descending
scala> List(10, 1, 8, 3, 5).sortWith(_ > _)
res0: List[Int] = List(10, 8, 5, 3, 1)

// sorting ascending alphabetically
scala> List("California", "Arizona", "New Mexico", "Texas").sortWith(_ < _)
res0: List[String] = List(Arizona, California, New Mexico, Texas)

// sorting descending alphabetically
scala> List("California", "Arizona", "New Mexico", "Texas").sortWith(_ > _)
res0: List[String] = List(Texas, New Mexico, California, Arizona)

// sorting ascending by length
scala> List("California", "Arizona", "New Mexico",
"Texas").sortWith(_.length < _.length)
res0: List[String] = List(Texas, Arizona, California, New Mexico)

// sorting descending by length
scala> List("California", "Arizona", "New Mexico",
"Texas").sortWith(_.length > _.length)
res0: List[String] = List(California, New Mexico, Arizona, Texas)
```

Kata 15 - Streams

In similar way, let's see how views are lazy versions of collection. Stream is the lazy version of `Lists`.

Here we use the power of functional programming to show `Stream` benefits:

```
scala> val s = (-100000000 to 100000000).toStream
s: scala.collection.immutable.Stream[Int] = Stream(-100000000, ?)

scala> s.head
res0: Int = -100000000

scala> s.tail
res0: scala.collection.immutable.Stream[Int] = Stream(-99999999, ?)

scala> s.take(3)
res0: scala.collection.immutable.Stream[Int] = Stream(-100000000, ?)

scala> s.filter(_ < 100)
res1: scala.collection.immutable.Stream[Int] = Stream(-100000000, ?)

scala> s.filter(_ > 100)
res2: scala.collection.immutable.Stream[Int] = Stream(-99999900, ?)

scala> s.map { _ * 2 }
res3: scala.collection.immutable.Stream[Int] = Stream(-200000000, ?)

scala> s(0)
res0: Int = -100000000

scala> s(1)
res1: Int = -99999999

scala> s(2)
res2: Int = -99999998
```

Kata 16 - Arrays

Scala, as a good functional language, determines the type of the `Array` if it is not mentioned (this is called type inference):

```
// the biggest data type determines the Collection type
scala> val a = Array(-1e100,  3.1415,  33D,  666L)
a: Array[Double] = Array(-1.0E100, 3.1415, 33.0, 666.0)

// we can force manually the type
scala> val n = Array[Number] (-1e100,  3.1415,  33D,  666L)
n: Array[Number] = Array(-1.0E100, 3.1415, 33.0, 666)
```

Practical ways to create and fill an `Array` include:

```
// from a Range
scala> val r = Array.range(-5, 5)
r: Array[Int] = Array(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4)

// from a Range with step
scala> val rs = Array.range(-5, 5, 2)
rs: Array[Int] = Array(-5, -3, -1, 1, 3)

// using fill
scala> val f = Array.fill(4)("hoo")
f: Array[String] = Array(hoo, hoo, hoo, hoo)

// using tabulate
scala> val t = Array.tabulate(6)(n => n * n)
t: Array[Int] = Array(0, 1, 4, 9, 16, 25)

// from a List
scala> val a = List("s", "m", "a", "c", "k").toArray
a: Array[String] = Array(s, m, a, c, k)

// from a String
scala> val s = "SMACK".toArray
s: Array[Char] = Array(S, M, A, C, K)
```

Scala arrays access is as follows:

```
scala> val a = Array(-1, 0, 1)
a: Array[Int] = Array(-1, 0, 1)

scala> a(0) = -100
scala> a(1) = 1
scala> a(2) = 100
scala> a
res1: Array[Int] = Array(-100, 1, 100)
```

Kata 17 - ArrayBuffer

An `ArrayBuffer` (as we have seen in the collections hierarchy) is a dynamically sized Array:

```
// initialize with elements
val smack = collection.mutable.ArrayBuffer("Akka", "Scala")

// += add one element
smack += "Spark"

// += add two or more elements
smack += ("Cassandra", "Kafka")

// ++= add multiple elements
smack ++= Seq("Mesos", "Docker")

// append, for add one or more elements
smack.append("Mesos", "Docker")
```

Kata 18 - Queues

A Queue is a **FIFO** (**First-in First-out**) data structure:

```
// first we need to import the Collection
scala> import scala.collection.mutable.Queue
import scala.collection.mutable.Queue

// create a Queue specifying the type
scala> var smack = new Queue[String]
smack: scala.collection.mutable.Queue[String] = Queue()

// += adding an element
scala> smack += "Spark"
```

```
res0: scala.collection.mutable.Queue[String] = Queue(Spark)

// += adding various elements
scala> smack += ("Mesos", "Akka")
res0: scala.collection.mutable.Queue[String] = Queue(Spark, Mesos, Akka)

// ++= adding a Collection
scala> smack ++= List("Cassandra", "Kafka")
res0: scala.collection.mutable.Queue[String] = Queue(Spark, Mesos, Akka,
Cassandra, Kafka)

// can also use enqueue
scala> smack.enqueue("Scala")
scala> smack
res0: scala.collection.mutable.Queue[String] =
Queue(Spark, Mesos, Akka, Cassandra, Kafka, Scala)
// take an element from the head of the queue
scala> smack.dequeue
res0: String = Spark

// the first element was removed from the queue
scala> smack
res0: scala.collection.mutable.Queue[String] = Queue(Mesos, Akka,
Cassandra, Kafka, Scala)

// the next element
scala> val n = q.dequeue
next: String = Mesos

// the first element was removed from the queue
scala> smack
res1: scala.collection.mutable.Queue[String] = Queue(Akka, Cassandra,
Kafka, Scala)

dequeueFirst and dequeueAll methods use a predicate

scala> smack = Queue("Spark", "Mesos", "Akka", "Cassandra", "Kafka",
"Scala")
smack: scala.collection.mutable.Queue[String] = Queue(Spark, Mesos, Akka,
Cassandra, Kafka, Scala)

// remove the first element containing a k
scala> smack.dequeueFirst(_.contains("k"))
res0: Option[String] = Some(banana)

scala> smack
res1: scala.collection.mutable.Queue[String] = Queue(Mesos, Akka,
Cassandra, Kafka, Scala)
```

```
// remove all the elements beginning with S
scala> smack.dequeueAll(_.startsWith("S"))
res2: scala.collection.mutable.Seq[String] = ArrayBuffer(Scala)

scala> smack
res20: scala.collection.mutable.Queue[String] = Queue(Mesos, Akka,
Cassandra, Kafka)
```

Kata 19 - Stacks

A Stack is a **Last-In-First-Out** (**LIFO**) data structure:

```
// first we create a stack
scala> var smack = Stack[String]()
smack: scala.collection.mutable.Stack[String] = Stack()
// we add some elements one by one

scala> smack.push("Spark")
res0: scala.collection.mutable.Stack[String] = Stack(Spark)
scala> smack.push("Mesos")
res1: scala.collection.mutable.Stack[String] = Stack(Spark, Mesos)

// adding multiple elements in one sentence
scala> smack.push("Akka", "Cassandra", "Kafka")
res2: scala.collection.mutable.Stack[String] = Stack(Spark, Mesos, Akka,
Cassandra, Kafka)

// to take the last element we pop
scala> val top = smack.pop
next: String = Kafka
scala> smack
res3: scala.collection.mutable.Stack[String] = Stack(Spark, Mesos, Akka,
Cassandra)

// to access the last element without extract it, use top
scala> smack.top
res4: String = Cassandra

// "Cassandra" is still on the top
scala> smack
res5: scala.collection.mutable.Stack[String] = Stack(Spark, Mesos, Akka,
Cassandra)

// we can use the Seq methods:
scala> smack.size
res6: Int = 4
```

```
scala> smack.isEmpty
res7: Boolean = false

// we can empty all the stack with clear
scala> smack.clear
scala> smack
res8: scala.collection.mutable.Stack[String] = Stack()
```

Kata 20 - Ranges

A range is useful in `for` loops:

```
// A range from 0 to 12 (inclusive)
scala> 0 to 12
res0: scala.collection.immutable.Range.Inclusive = Range(0, 1, 2, 3, 4, 5,
6, 7, 8, 9, 10, 11, 12)

// A range from 0 to 7 (without the upper limit)
scala> 0 until 7
res0: scala.collection.immutable.Range.Inclusive = Range(0, 1, 2, 3, 4, 5,
6)

// A range with a step of 3
scala> 0 to 12 by 3
res2 scala.collection.immutable.Range = Range(0, 3, 6, 9, 12)

// A range of chars
scala> 'a' to 'm'
res3: scala.collection.immutable.NumericRange.Inclusive[Char] =
NumericRange(a, b, c, d, e, f, g, h, i, j, k, l, m)

// A List from a Range
scala> val a = (0 to 12).toList
a: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)

// An Array from a Range
scala> val a = (0 to 12).toArray
a: Array[Int] = Array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)

// A Set from a Range
scala> val a = (0 to 10).toSet
a: scala.collection.immutable.Set[Int] = Set(0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
10)

// The range method in Array (upper limit not included)
scala> val a = Array.range(0, 12)
```

```
a: Array[Int] = Array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

// The range method in Vector (upper limit not included)
scala> val a = Vector.range(0, 11)
a: collection.immutable.Vector[Int] = Vector(0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
10)

// The range method in List (upper limit not included)
scala> val a = List.range(0, 12)
a: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

// A list with numbers in a range with a step of 4
scala> val s = List.range(0, 40, 4)
s: List[Int] = List(0, 4, 8, 12, 16, 20, 24, 28, 32, 36)

// A list with characters in a range
scala> val a = collection.mutable.ArrayBuffer.range('a', 'g')
a: scala.collection.mutable.ArrayBuffer[Char] = ArrayBuffer(a, b, c, d, e,
f)

// The classic: a for loop using a Range
scala> for (i <- 1 to 2) println(i)
1
2
```

The model - Akka

The objective of this section is to think about our systems in the Actor Model.

The Actor Model is a mathematical model. As Obi Wan would say, it's *"An elegant weapon for a more civilized age."* The Actor Model was developed by Carl Hewitt, Peter Bishop, and Richard Steiger in 1973 at the Massachusetts Institute of Technology, in a paper entitled, *A Universal Modular Actor Formalism for Artificial Intelligence*.

It was a more civilized age, because computer science was developed by mathematicians and all the programming was made with their bare hands. Well, if the Actor Model has been around for more than 40 years, at what point did we turn to the dark side? The answer is neither short nor simple to explain.

The quick and dirty answer is: because they developed a very advanced model for the technology of those days. The problem is that we had to develop a lot of technology in software and hardware to reap benefits from the Actor Model. Modern compilers, modern processors, and modern memory were needed.

We can explore the release years for the main functional programming languages: Lisp (1956-1958), Smalltalk (1972), ML (1973), Scheme (1975) , Erlang (1987), and Haskell (1990). With Erlang, for example, the first version was designed to solve telephony switching issues. Since 1980 there has been a predominance in object oriented programming languages, and, as you guessed it, it's because of the hardware costs and the rise of the PC.

Between 1980 and 2003 everything was fine during the dark ages, without the need for functional programming. This was until two concepts became mainstream: parallelism and concurrency. I remember all the efforts to solve problems with the wrong tools; for example, thread programming, C++, and Java programs, which were built using threads, were normally complex and prone to disaster.

In 2003 there was a renaissance of functional programming with the launch of: Scala (2003), F# (2005), Clojure (2007), and Elixir (2012). As you can guess, all that was made with the Actor Model published in 1973. The Actor Model can be declared as the winning approach for concurrency.

The Actor Model is at a much higher abstraction level than threads. Programming threads gets into low-level problems such as locks, semaphores, and shared data.

To be competitive in today's Actor Model, a library must have the following three characteristics:

- **Fault-tolerant**: The *let it crash* paradigm has demonstrated its supremacy
- **Lightweight**: The hardware must be in full use in an efficient way
- **Distributed**: It spans between different servers with transparency

Smalltalk is a perfect example of high-quality designs but a poor implementation, so it has a low adoption rate.

Here, as in life, a high abstraction level is the same as *ease of use*. Language designers always refer to *syntactic sugar*, but the language sweetness has proved to be a catalyst for adoption speed.

The development velocity with the Actor Model is so much higher and safer than the threading approach.

To learn more about fault-tolerant systems, it is recommended you visit the Akka developer's blog at `http://letitcrash.com`.

The Actor Model in a nutshell

Here we have a comparison between the Actor Model and traditional OOP:

Concept	Object Oriented approach	Actor Model
Smallest unit	Object	Actor
Unit encapsulates	State and behavior	State and behavior
Access	It is allowed (but not recommended) to execute object methods from outside and access an object's fields from outside the object	Only through messages
Messages	Could be mutable	Always immutable
State	Represented by the value of its fields at a certain point in time	Has a mailbox for messages. The Actor is always waiting for messages
Exception handling approach	`Try-catch` approach	*Let it crash* approach

Table 2-6. OOP versus Actor Model

Here we highlight Lightbend's (the company behind Scala) recommendations for newcomers:

- Think as actors as people. Think of your Actor Model as a company.
- Think of your actor's siblings as the people in the same hierarchical level as the employee.
- Think of your actor's children as an employee's subordinates.
- An Actor has one and only one supervisor (the actor who created it).
- The success of the Actor Model is delegation. Always delegate, always.
- Especially with blocking tasks, delegation to subordinates is fundamental.

The Akka implementation of the Actor Model has the following peculiarities:

- When we create an Actor, Akka gives the `ActionRef`, so you can know its state.
- Actors run in real Java Threads, and some Actors can share the same Thread.
- There are three mailbox types: unbounded, bounded, and priority. We can create ours.
- Actors can scan their mailboxes looking for specific messages.
- There is a dead letter mailbox, with the messages of all terminated Actors.

In Java, we must throw an exception and handle it with a lot of combinations and scenarios. As a supervisor in the *Let it crash* approach, we have four (and only four) options:

- Resume the actor, so the internal state is retained
- Restart the actor, so the internal state is cleared
- Terminate the actor
- Send a message with the failure, escalating the problem

Kata 21 - Actors

For this example, lets model a **Drone**, an **Unmanned Aerial Vehicle** (**UAV**). Of course, this is our first actor; we just model the basic functionality (don't expect this code to build a full drone, as it is only for illustrative purposes):

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props

class DroneActor extends Actor {
  def receive = {
    case "left"  => println("turning left")
    case "right" => println("turning right")
    case "up"    => println("ascending")
    case "down"  => println("descending")
    case _       => println("command not recognized")
  }
}

object Main extends App {
  // always build the ActorSystem
  val actorSystem = ActorSystem("DroneSystem")
  // create the actor
  val parrot = actorSystem.actorOf(Props[DroneActor], name = "droneactor")
```

```
    // send the actor some messages
    parrot ! "up"
    parrot ! "right"
    parrot ! "drop the bombs"

    // shut down the actor system
    actorSystem.shutdown
  }
```

Running this program, the output is:

```
$ sbt run
[info] Running Main
ascending
turning right
command not recognized
```

The program is explained, as follows:

- For using Akka actors, we need to import some `Akka.actor.*` members.
- We define the `DroneActor` of type `Actor` (an original name, isn't it?).
- All the actor behaviors are defined under the `receive` method.
- The receive method is implemented using pattern matching, an important and powerful part of all functional languages. We don't use `switch-case` statements, as match expressions compare with the first line. If the pattern is not matched it compares with the second line, and so on. Never forget to put the default case at the end of all the known cases. This is always for security and should be the last line in the match expression.
- If the message is not matched, an `UnhandledMessage` exception is thrown in the `ActorSystem`.
- The main object is created to test our new toy (and the model, of course).
- `ActorSystem` receives a name as an argument; alphanumeric characters and hyphens are allowed (but not in the leading char).
- Then we invoke the `actorOf` method in the `ActorSystem` to create our `Actor`. As we will see later, actors can be created inside other actors. Actors start asynchronously when created.
- To send messages to an actor, we use the `!` operator.
- The actor does its stuff; after that the `ActorSystem` shuts down.

The actor system

An actor system is a group of actors. It has the following characteristics:

- It is hierarchical; every actor always has an actor supervisor. An actor could have siblings and children.
- As in an office, the actors under the same actor system share dispatchers, deployments, and addresses.
- The actor system is the central point where the actors are created and looked for.
- As you may guess, the actor system internally is a thread controller. The actor system decides when to allocate threads for an application.
- If the actor system is not shut down (with the `system.shutdown` line), the application doesn't terminate. While the actor system is running, the app is running.

Actor reference

The actor reference has the following characteristics:

- As we can see in the code, the `actorOf` method in the `ActorSystem` has two tasks: it starts the actor asynchronously and returns an actor reference.
- The `ActorRef` is a handle, so we cannot break the actor system.
- The `ActorRef` is an actor's facade, so we cannot manipulate the actor instance directly or alter its variables.
- The `ActorRef` is the way to communicate with the actor. It is the way to place messages in its mailbox.
- The `ActorRef` is immutable, so we cannot change it; it is only a reference.
- One actor has only one `ActorRef`. One `ActorRef` refers to only one actor. It is a one-to-one relationship.
- To complain about the actor model, an `ActorRef` is serializable, and is server-independent, so we can share, pass, and transmit an `ActorRef` through the network.

Kata 22 - Actor communication

For this example, we are building an actor-based communication application. For sending messages to an actor, we use the `!` operator:

```
myActorReference ! message
```

As we explained before, remember that the `!` operator only works in actor references, not in actor instances:

```
import akka.actor._

case object SendPackageMessage
case object AcknowledgeMessage
case object StartTransmissionMessage
case object StopTransmissionMessage

class Transmitter(receiver: ActorRef) extends Actor {
  val messages = 100
  var count = 0

  def increment {
    count += 1;
    println("message sent")
  }

  def receive = {
    case StartTransmissionMessage =>
      increment
      receiver ! SendPackageMessage
    case AcknowledgeMessage =>
      increment
      if (count >= messages) {
        sender ! StopTransmissionMessage
        println("transmitter stopped")
        context.stop(self)
      } else {
        sender ! SendPackageMessage
      }
    case _ => println("Transmitter: message not recognized")
  }
}

class Receiver extends Actor {
  def receive = {
    case SendPackageMessage =>
      println(" received")
```

```
        sender ! AcknowledgeMessage
      case StopTransmissionMessage =>
        println("Receiver stopped")
        context.stop(self)
      case _ => println("Receiver: message not recognized")
    }
  }

  object TransmissionTest extends App {
    val system = ActorSystem("TransmissionSystem")
    val receiver = system.actorOf(Props[Receiver], name = "receiver")
    val transmitter = system.actorOf(Props(new Transmitter(receiver)), name =
  "transmitter")

    // start the transmission
    transmitter ! StartMessage
   //system.shutdown
  }
```

Actor communication is described as follows:

- In Akka, when an actor receives a message from another actor, it also receives the `sender` variable. This is a variable to make reference to the invoking actor. We use the `sender` variable to return messages to the invoker actor. In this context, `sender` is a reserved word. Use it properly.
- In our code, we have modeled four messages; these objects are type cases:
    - case object `SendPackageMessage`
    - case object `AcknowledgeMessage`
    - case object `StartTransmissionMessage`
    - case object `StopTransmissionMessage`
- `TransmissionTest` is the main app. The first line creates the actor system and calls it `TransmissionSystem`.
- Then we create a receiver actor and the reference is loaded in the `val` receiver.
- We create a transmitter actor. In the constructor, it receives a reference to its receiver, already built. We used this to demonstrate how actors could be related to each other. We could define the constructor with no arguments, but we want to explain the concept.
- Next, we send the transmitter a `StartMessage`.
- When the transmitter receives a message we begin the transmission until the count is greater than or equal to the messages. We exchange messages between the transmitter and the receiver until we stop the context.

- The context object is available to all the actors in the actor system. It is used so the actors can stop each other.
- It is important to recall that the receiver, transmitter, and senders are actor references, not actor instances. An actor must be accessed only through its reference, never directly.
- The secret behind the actor reference is that, when you have a high concurrent environment, accessing actor instances directly could be an unsafe environment. Accessing and sending messages through `ActorRef` warrants all the ACID needed.

Kata 23 - Actor life cycle

Here we have a brief description of the Akka Actor life cycle methods:

- `constructor`: This is called when an instance of the class is created (the same as in Java).
- `preStart`: This is called immediately after the actor is started.
- `postStop`: This is called immediately after the actor is stopped, usually, for cleanup work.
- `preRestart`: This is called immediately after the actor is restarted. Usually, a restart is caused by an exception. The `preRestart` receives as parameters a `Throwable` and a message, and the old object receives these parameters.
- `postRestart`: This is called immediately after the actor is restarted. Usually, a restart is caused by an exception. The `postRestart` receives as a parameter a `Throwable`; the new object receives this parameter and calls the `preStart` method.

Within ancient console video games, there exists a character called `Bob-omb`, a walking time bomb.

In this code snippet, we exploit `Bob-omb` to explain the life cycle methods:

```
import akka.actor._

class Bobomb extends Actor {

  println("in the Bob-omb constructor")

  override def preStart {
    println("in the Bob-omb preStart")
  }
```

```
  override def postStop {
    println("in the Bob-omb postStop")
  }

  override def preRestart(reason: Throwable, message: Option[Any]) {
    println("in the Bob-omb preRestart")
    println(s" preRestart message: ${message.getOrElse("")}")
    println(s" preRestart reason: ${reason.getMessage}")
    super.preRestart(reason, message)
  }

  override def postRestart(reason: Throwable) {
    println("in the Bob-omb postRestart")
    println(s" postRestart reason: ${reason.getMessage}")
    super.postRestart(reason)
  }

  def receive = {
    case ForceExploit => throw new Exception("Boom!")
    case _ => println("Bob-omb received a message")
  }
}

case object ForceExploit

object LifecycleDemo extends App {
  val system = ActorSystem("BombSystem")
  val bobomb = system.actorOf(Props[Bobomb], name = "Bobomb")
  println("sending Bob-omb a message")
  bobomb ! "activate"
  Thread.sleep(4000)
  println("making Bob-omb exploit")
  bobomb ! ForceExploit
  Thread.sleep(4000)
  println("stopping Bob-omb")
  system.stop(bobomb)
  println("shutting down bomb system")
  system.shutdown
}
```

When we run this program, the trace is here:

```
[info] Running LifecycleDemo
sending Bob-omb a message
in the Bob-omb constructor
in the Bob-omb preStart
Bob-omb received a message
making Bob-omb exploit
```

```
[ERROR] [03/14/2015 10:21:54.953] [LifecycleDemo-akka.actor.default-
dispatcher-4]
[akka://LifecycleDemo/user/Bobomb] Boom!
java.lang.Exception: Boom!
at Bobomb$$anonfun$receive$1.applyOrElse(Bobomb.scala:19)
(many more lines of exception output ...)
in the Bob-omb preRestart
preRestart message: ForceExploit
preRestart reason: Boom!
in the Bob-omb postStop
in the Bob-omb constructor
in the Bob-omb postRestart
postRestart reason: Boom!
in the Bob-omb preStart
stopping Bob-omb
shutting down system
shutting down bomb system
[success]
```

Complex, isn't it? We leave the reader with an exercise: trace, line by line, this program output.

Use the life cycle wisely

Kata 24 - Starting actors

We have reviewed in previous examples how to create actors through the `ActorSystem`. But, for creating an actor from inside another actor, we use the actor context:

```
class Father extends Actor {
  val son = context.actorOf(Props[Son], name = "Son")
  // add your code here ...
}
```

Let's explore actor life cycle control with an example based on Dilbert, an IT employee whose boss is pointy-haired. Ted is an engineer who is fired and rehired in the comic several times.

```
import akka.actor._

case class Hire(name: String)
case class Name(name: String)
```

```
class Employee extends Actor {
  var name = "Dilbert new character"

  override def postStop {
    println(s"I'm ($name) and my Boss fired me: ${self.path}")
  }

  def receive = {
    case Name(name) => this.name = name
    case _ => println(s"Employee $name can't handle this message.")
  }
}

class Boss extends Actor {
  def receive = {
    case Hire(name) =>
      // here the boss hire personnel
      println(s"($name) is about to be hired")
      val employee = context.actorOf(Props[Employee], name = s"$name")
      employee ! Name(name)

    case _ => println(s"Boss can't handle this message.")
  }
}

object StartingDemo extends App {
  val actorSystem = ActorSystem("StartingDemo")
  val pointyHaired = actorSystem.actorOf(Props[Boss], name =
"PointyHaired")

  // here the boss hires people
  pointyHaired ! Hire("Dilbert")
  pointyHaired ! Hire("Ted")

  // we wait some office cycles
  Thread.sleep(1000)

  // we look for Ted and we fire him
  println("Firing Ted ...")
  val ted = actorSystem.actorSelection("/user/PointyHaired/Ted")

  // PoisonPill is a special message in Akka
  ted ! PoisonPill
  println("now Ted is fired")
  actorSystem.shutdown
}
```

Lets' analyze this Dilbertian code:

- We create and use `Hire` and `Name` classes to send messages among actors.
- When the `Employee` actor receives a `Name` message, it assigns its name variable.
- When the boss receives a Hire message note, it uses the `context.actorOf` method to hire a new employee and then the employee assigns the name.
- The `StartingDemo` object creates a new `ActorSystem`.
- The `StartingDemo` then creates the boss actor using the `ActorSycstem` reference.
- `StartingDemo` sends the `Hire` messages twice to the boss, with Dilbert and Ted.
- After a pause, we look for Ted in the actor system. Then we send him the `PoisonPill` message. It is special message in the Akka actor system that asynchronously sends a stop signal to an actor, so we can use the `postStop` method after `PoisonKill`.

Kata 25 - Stopping actors

There are four ways to stop an actor:

- Calling `system.stop(actorRef)` from the `ActorSystem` level
- Calling `context.stop(actorRef)` from inside the actor
- Sending a `PoisonPill` message to the actor
- Programming a `gracefulStop`

Here we explain every way to stop an actor:

- `stop`: This method when received, the actor will continue processing only the current message (if any). If a new message arrives in the actor's mailbox or there are queued messages, they will be discarded.
- `PoisonPill`: The message `PoisonPill` is a normal message. It is received and queued in the actor's message mailbox. When the `PosionPill` message is processed, the actor stops.
- `gracefulStop`: This method allows you to terminate actors gracefully. It waits for timeout. If we need a specific order of instructions before the stop, this is a good way.

There are some aspects to consider when stopping actors:

- The stop message is asynchronous. Yes, the stop method could return before the actor is actually stopped, so you have been warned.
- The stop process has two subprocesses. The first, as already described, suspends the actor's mailbox. The second sends the stop message to all the actor's children. The parent has to wait for all its children.
- When we can't process messages, these are sent to the `deadLetters` mailbox. We can access them though the `deadLetters` method on the actor system.
- As we have seen, when we stop an actor, the `postStop` life cycle method is invoked. Normally it is used to clean up resources.

Here we have a code example using `system.stop` and `context.stop`:

```
import akka.actor._

class CrashDummy extends Actor {
  def receive = {
    case s:String => println("Message received: " + s)
    case _ => println("Unknown message received")
  }
}

object CrashDummyExample extends App {
  val system = ActorSystem("CrashDummyExample")
  val dummy = system.actorOf(Props[CrashDummy], name = "dummy")
  dummy ! "test123"
  // stop our crash dummy
  system.stop(dummy)
  system.shutdown
}
```

As shown earlier, using `context.stop(actorRef)` is the same as `actorSystem.stop(actorRef)`. We just use the first option within an actor (the context is variable within actors). The second option is used outside the actor.

Here we have the same example using `PosionPill`:

```
import akka.actor._

class CrashDummy extends Actor {
  def receive = {
    case s:String => println("Message received: " + s)
    case _ => println("Unknown message received")
  }
  override def postStop { println("CrashDummy's Will")
}

object CrashDummyExample extends App {
  val system = ActorSystem("CrashDummyExample")
  val dummy = system.actorOf(Props[CrashDummy], name = "dummy")

  // a message
  dummy ! "open your mouth"
  // take the poison
  dummy ! PoisonPill

  // dummy is stopped
  dummy ! "OMG what I've done!"
  // stop our crash dummy
  system.shutdown
}
```

As we mentioned earlier, messages sent after stopping the actor will not be processed and sent to the `deadBox`. This program output is:

```
  Message Received: open your mouth
  CrashDummy's Will
```

Here we have the same example using `gracefullStop`:

```
import akka.actor._
import akka.pattern.gracefulStop
import scala.concurrent.{Await, ExecutionContext, Future}
import scala.concurrent.duration._
import scala.language.postfixOps

class CrashDummy extends Actor {
  def receive = {
    case s:String => println("Message received: " + s)
    case _ => println("Unknown message received")
  }
  override def postStop { println("CrashDummy's Will")
}
```

```
object CrashDummyExample extends App {
  val system = ActorSystem("CrashDummyExample")
  val dummy = system.actorOf(Props[CrashDummy], name = "dummy")
  // try to stop the dummy gracefully
  try {
    val stopped: Future[Boolean] = gracefulStop(dummy, 2
seconds)(system)
    Await.result(stopped, 3 seconds)
    println("dummy was stopped")
  } catch {
    case e:Exception => e.printStackTrace
  } finally {
    system.shutdown
  }
}
```

We use this approach when we specifically want to wait for a period of time. Remember, the future returns when the actor is stopped. If the actor has not finished in that amount of time, an `ActorTimeoutException` is thrown. So, we surround everything with a try-catch. If we put less time in `Await.result` than in the `gracefulStop`, we see an exception thrown.

Kata 26 - Killing actors

There is a black-belt concept called supervisor strategies. When a supervisor strategy is implemented, you can send an actor a `Kill` message, which results in an actor restart.

With the default supervisor strategy, the `Kill` message terminates the target actor. As we can see in this example:

```
import akka.actor._

class Convict extends Actor {
  def receive = {
    case _ => println("Convict received a message")
  }

  override def preStart { println("In Convict preStart") }
  override def postStop { println("In Convict postStop") }
  override def preRestart(reason: Throwable, message:     Option[Any]) {
    println("In Convict preRestart")
  }
  override def postRestart(reason: Throwable) {
    println("In Convict postRestart")
  }
```

```
}

object GreenMile extends App {
  val system = ActorSystem("GreenMile")
  val con = system.actorOf(Props[Convict], name = "Convict")

  con ! "last words"
  // send him towards the light
  con ! Kill
  system.shutdown
}
```

Running this program will produce the following output:

```
In Convict preStart
Convict received a message
[ERROR] [19:00:03.230] [GreenMille-akka.actor.default-dispatcher-2]
[akka://GreenMile/user/Con] Kill (akka.actor.ActorKilledException)
In Convict postStop
```

This code shows the brutality of the system. This approach is used to allow the actor supervisor to restart it. Normally, if we want to stop an actor, we use the methods described earlier.

Kata 27 - Shutting down the actor system

The following code shows how to shut down the actor system, normally because our application has already finished:

```
object Main extends App {
  // create the ActorSystem
  val mySystem = ActorSystem("mySystem")
  // this space is for actors playing ...

  // the play has finished, Ok?
  mySystem.shutdown
}
```

We have already used shutdown systems loosely during this chapter, but we want to dedicate a space due to their importance. If we don't call `mySystem.shutdown` our applications will continue running indefinitely.

Kata 28 - Actor monitoring

The following code shows how an actor should be notified when a monitoring actor has died:

```
import akka.actor._
class Subordinate extends Actor {
  def receive = {
    case _ => println("Subordinate received a message")
  }
}

class Boss extends Actor {
  // start subordinate as a child
  val sub = context.actorOf(Props[Subordinate], name = "sub")
  context.watch(sub)
  def receive = {
    case Terminated(sub) => println("they've killed my sub")
    case _ => println("Boss received a message")
  }
}

object MonitoringTest extends App {
  // create the ActorSystem instance
  val system = ActorSystem("MonitoringTest ")

  // create the Boss (and it will create a Sub)
  val boss = system.actorOf(Props[Boss], name = "Boss")
  // look for sub, then we kill it
  val sub = system.actorSelection("/user/Boss/Subordinate")

  sub ! PoisonPill
  Thread.sleep(3000)
  println("Game over")
  system.shutdown
}
```

Running this code produces the following output:

```
  They've killed my sub
  Game over
```

Using the watch method notifies an actor when a subordinate is stopped (with the methods described earlier) or killed with a `Kill` message. This is useful because it lets the supervisor handle the situation.

Note that, when an exception within an actor is thrown, *the Actor is not killed* (if it doesn't kill you, it makes you stronger). In Akka, an exception makes an actor automatically restart.

Kata 29 - Looking up actors

In previous examples, we have seen how to look up a specific actor:

```
val sub = system.actorSelection("/user/Boss/Subordinate")
```

The `actorSelection` method is available under the `actorSystem` and on every actor context within an actor instance.

We can also look for an actor by its relative paths; for example, from its siblings:

```
//from a sibling actor
val sub = context.actorSelection("../Subordinate")
```

The `actorFor` method in the `actorSystem` could be used to locate actors:

```
val sub = system.actorFor("akka://MonitoringTest/user/Boss/Subordinate ")
val sub = system.actorFor(Seq("user", "Boss", "Subordinate"))
```

And it could also be looked up from a sibling with `actorFor`:

```
val sub = system.actorFor(Seq("...", "Subordinate"))
```

Summary

This chapter was a Scala-Akka dojo where you learnt through several Katas. In the first part we explored the fundamental parts of Scala; in the second part we focused on the Akka actor model.

It is true, there were many important topics not covered in this chapter, such as futures, promises, and parallel collections. But we tried to provide a reference to them, although not an exhaustive guide.

So, as all the book examples are in Scala, we need to master fundamental techniques before delve into the SMACK stack.

The Actor Model is important to understand the architecture and operation of Spark.

In the following chapter, we will explore Spark design and provide some examples using Scala.

3
The Engine - Apache Spark

In this chapter, we'll walk through the process of downloading and running Apache Spark. We'll first see how to run it in local mode on a single computer, and then we'll run it in cluster mode. We'll also see the Spark's core abstraction for data manipulation, the **resilient distributed dataset** (**RDD**). Finally we'll dive into an RDD abstraction called **DStreams** (or **discretized streams**), the core part of this chapter is **Spark Streaming**.

This chapter was written for the Spark newbie, but we don't focus on the data science power of Spark; this chapter is targeted at data engineering and data architecture.

In this chapter, we will learn:

- Spark in single mode
- Spark core concepts
- Resilient distributed datasets
- Spark in cluster mode
- Spark Streaming

Spark in single mode

Although Apache Spark cluster-based installations can become a complex task, when we integrate Mesos, Kafka, and Cassandra, the installation may become an interdisciplinary topic among engineers from: databases, telecommunications, operating systems, and infrastructure.

However, it's so easy to download and install Apache Spark on a laptop in standalone mode for learning and exploration that it has made many developers and data scientists become engaged by, and married to, the platform.

This low barrier to entry makes many small businesses capable of launching pilot projects without production systems interference, without requiring the construction of complex tools, and without hiring expensive expert technicians. As previously mentioned, Spark uses *big data* so nobody is left out.

Apache Spark is open source software and can be downloaded freely from the Apache foundation site. Spark requires at least Java version 6 and at least Maven version 3.0.4. All dependencies on Scala and Akka are automatically downloaded and configured as part of the Spark installation.

Downloading Apache Spark

Go to the Apache Spark download page: `spark.apache.org/downloads.html`

Select a **pre-built for Hadoop** package type (the other options are for a user-provided Hadoop).

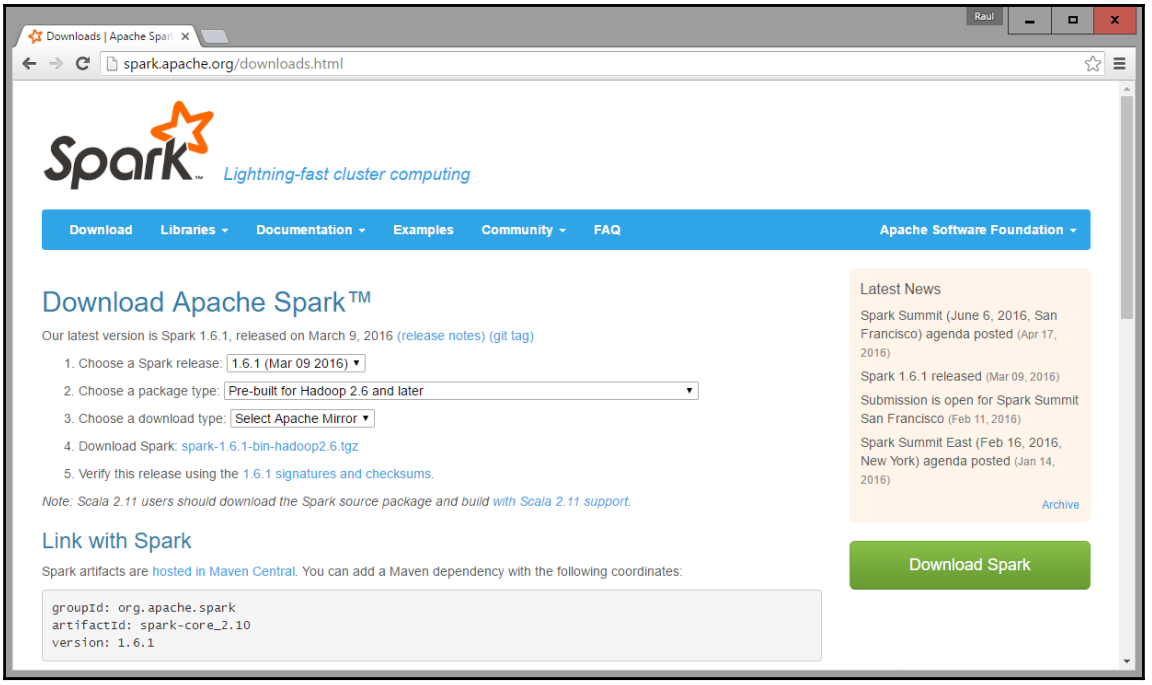

Figure 3-1 Apache Spark download page

Decompress the package in a suitable directory and open a prompt.

Testing Apache Spark

Then, open your favorite command line and move to the directory where you unzipped the package.

To open the Spark interactive shell, go to the bin directory and execute the spark-shell (check the minimum hardware requirements on the page).

```
$> ./bin/spark-shell
```

You will see an output like this (shown on a 64-bit Windows terminal to demonstrate that nobody is excluded):

Figure 3-2 Terminal window with Spark running

Yes, as we all know, Spark runs on Scala, and this is a simple Scala REPL (with some steroids). For example, to create a sequence of numbers from 1 to 300,000:

```
scala>val nums = 1 to 300000
nums: scala.collection.immutable.Range.Inclusive = Range(...
```

As we'll see in a moment, for all its magic, Spark needs special data abstractions called RDD. To transform our humble sequence of numbers in Scala to RDD, we do:

```
scala>val powerfulRdd = sc.parallelize(nums)
powerfullRdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at
parallelize at <console>
```

In this case, we have a numeric RDD. Now, as you may guess, we can do all the math operations we can think of with Scala data types. Let's take only the even numbers:

```
scala>powerfulRdd.filter(_ % 2 == 0).collect()
res1: Array[Int] = Array(2, 4, 6, 8, 10...)
```

Spark returns an `Array[Int]` with the even numbers from 1 to 300,000. With this array we can run all the math operations that we used to do with the Scala Int Arrays.

Now, we are inside Spark and what we learn here can be done in a big corporation cluster, without entry barriers, without excesssive fees. We can do it anywhere.

Spark core concepts

Now that we have Spark running in our shell, we can learn about programming in greater detail. A Spark application consists of a driver program, which is responsible for distribution of the operations among the cluster members. The driver program also distributes the data structure fragments in the cluster, and then applies operations in a distributed way.

The driver programs access the `SparkContext` object representing the connection to the cluster. In the shell, it's always accessed through the `sc` variable. To see what type `sc` is:

```
scala>sc
res1: org.apache.spark.SparkContext = org.apache.spark.SparkContext@e4b54d3
```

To run operations, driver programs have a number of nodes called **executors**. For example, if we run a simple `count()` operation in a cluster, the `count()` operation work is distributed among all the cluster members, each on their portion of file assigned to them by the driver program.

In our examples, as we only have one machine where we run the Spark shell locally, the work is performed in a single node. For distribution within the cluster, we only connect the same shell to the cluster to run in parallel. Figure 3-3 explains how Spark is executed in a standalone cluster.

Finally, the Spark API is about passing functions to their executors to run them on the cluster. A lot of Spark's power is found in the fact that function-based operations such as `union()` also parallelize in the cluster, that is, that the function automatically takes your function and sends it to the executor nodes. The functions within a single stage are chained together and performed at once on one partition. Therefore, you can write code in a single driver program and automatically have portions of it running on multiple nodes.

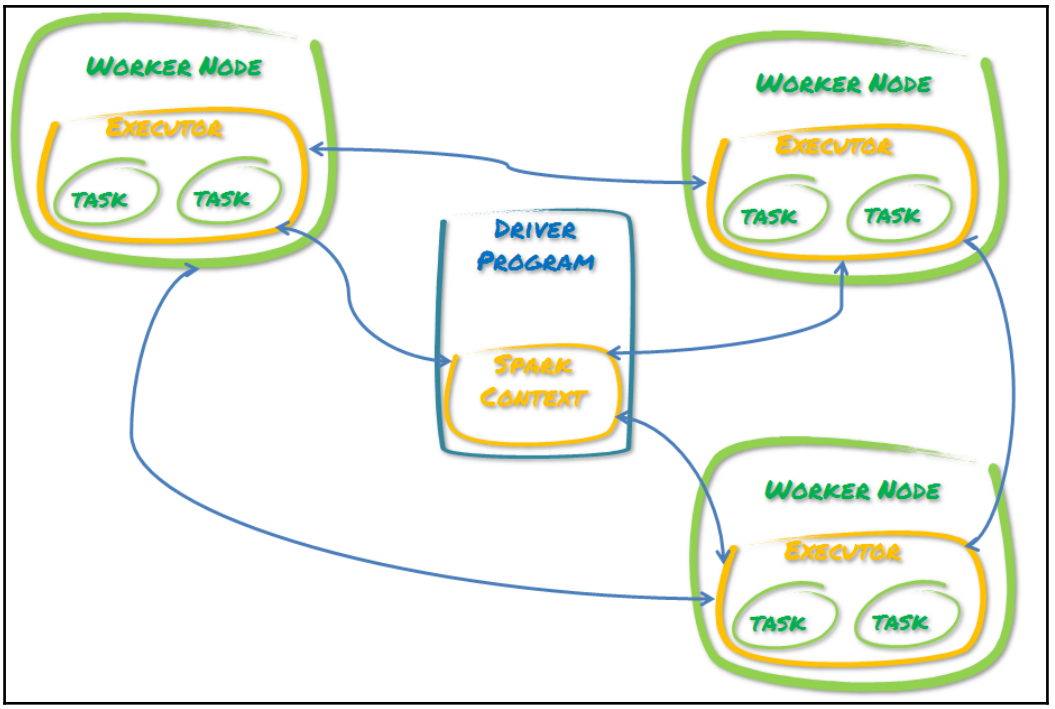

Figure 3-3 One driver program with three worker nodes

Resilient distributed datasets

The Spark soul is the resilient distributed dataset. Spark has four design goals: make in-memory (Hadoop is not in-memory) data storage, distribute in a cluster, be fault tolerant, and be fast and efficient.

Fault tolerance is achieved, in part, by applying linear operations on small data chunks. Efficiency is achieved by parallelization of operations throughout all parts of the cluster. Performance is achieved by minimizing data replication between cluster members.

A fundamental concept in Spark is that there are only two types of operations we can do on an RDD:

- **Transformations:** A new RDD is created from the original; for example, mapping, filtering, union, intersection, sort, join, coalesce
- **Actions**: The original RDD isn't changed; for example, count, collect, first

It's right when people say that computer science is mathematics with a costume. As we've already seen, in functional programming, functions are first-class citizens; the equivalent in mathematics is to talk about function composition.

Thus, in linear algebra there are operations between vectors. The operations between vectors result in another vector (for example, vector addition). In our Spark context these would be **transformations**. On the other hand, there are operations between vectors which result in a scalar value (for example, the inner product). In our Spark context, these operations correspond to **actions**.

In both cases, with actions and transformations the original RDD remains unchanged. As you no doubt thought, this is because the concept of *variable* value in functional programming is an aberration: everything must always be immutable.

In Spark, the chain of transformations from the first RDD until the last RDD is logged. So, if a tragedy happens (for example, a power failure), the process can be repeated.

Since we live in a functional world, transformations, are lazily evaluated. They are not executed until (and only until) the final value is required. As we saw in the section on Scala, this exists to improve performance because it avoids unnecessary data processing and waste of resources (often caused by the programmer).

As we saw, lazy evaluation also prevents deadlocks and bottlenecks, because it prevents a process waiting for the outcome of another process indefinitely. Recall that the lazy evaluation behaves as if all the calculations and processes had already been made and returns an *avatar* to the final result.

When possible, RDDs are kept in memory: This increases performance because it does not need to fetch the intensely used values from disk and/or databases.

In addition to RDDs, we have the dataframes API. Added in 2015, this API offers:

- **Scalability**: We can test on a laptop with kilobytes of data, and run the same on a large cluster with terabytes of data
- **Optimization**: The Spark SQL catalyst optimizer has two advantages: beautification and improvement of awful human-generated SQL, and source code generation from actual SQL
- **Integration**: Smooth integration, with the other Spark tools
- **Multiformat**: Supports a large number of data formats and storage systems

Running Spark applications

Well, now let's look at how to write Spark applications. So far we have seen how to run programs in the interactive shell. The main difference with the shell is that from a standalone application we must initialize the Spark context. An application is submitted to the cluster by the driver program and executed as one whole piece in non-interactive mode compared to Spark shell. Apart from that, everything is the same.

To write a program in Scala or Java, we must import the Maven (or Gradle, or sbt) dependency into our project. At the time of writing, the latest version of Spark is 1.6.1, and the Maven coordinates are:

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.6.1
```

Initializing the Spark context

Once we have the Spark libraries in our project, we need to import the Spark packages in our program and create a `SparkContext`. So, first create an object of type `SparkConf` to configure our application, and then we'll build a `SparkContext` from it:

```
// All the necessary imports
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

// We create the SparkConf object
val conf =
newSparkConf().setMaster("local").setAppName("exampleApp")

// We create the SparkContext from SparkConf
val sc = new SparkContext(conf)
```

When we create the `SparkConf` object we need to pass two parameters:

- **The URL of the cluster**: `local` when we want to run the program on one thread on the local machine, that is, without a cluster.
- **The Application name**: in this case it is called `exampleApp`. We use this name to identify our application in the GUI cluster manager when we run it in cluster mode.

Spark applications

Once we have the necessary imports, the Spark configuration object, and the Spark context object, we can begin to write the program body (`.java`, `.scala`, `.py`). Recall that all we can do from the Spark shell (which is a common Scala REPL on steroids) can be done from a standalone program.

This would not be a classic data book if we did not mention the typical word count program. In this case, we will take a file containing the masterpiece by Miguel De Cervantes Saavedra and see what the most repeated words are (in Spanish obviously).

```
// First, we create an RDD with the don-quijote.txt file contents
val docs = sc.textFile("don-quijote.txt")

// Convert the text of each line in lowercase
val lower = docs.map( line =>line.toLowerCase)

// Separate each text line in words (strings separated by spaces)
// As we already know, the split command flattens the arrays
val words = lower.flatMap(line =>line.split("\\s+"))

// Create the tuples (word, frequency)
// Each word initial frequency is 1
val counts = words.map(word => (word, 1))

// Group by word, the sum of frequencies (peace of cake, isn't?)
val freq = counts.reduceByKey(_ + _)

// Reverse the tuple (frequency, word)
val invFreq = freq.map(_.swap)

// Take the 20 largest and prints
invFreq.top(20).foreach(println)
```

But wait, the most commonly-used words (as in all human languages) are conjunctions and prepositions, so before separating each sentence in words in step 3, we'll filter the unwanted words:

```
valstopWords = Set("que", "de", "y", "la", "a", "el", "en", "", "no",
"los", "se", "con", "por", "las", "lo", "le", "su", "del", "me", "como",
"es", "un", "si", "mÃ¡s", "yo", "al", "mi", "y", "para" )

val words = lower
.flatMap(line =>line.split("\\s+"))
.filter(! stopwords.contains(_))
```

Running programs

Once we have completed our program, we use the `/bin/spark-submit` script to run our small application. Modern Java/Scala IDE's (no names mentioned to avoid injuring susceptibilities) have the ability to easily run Spark programs within themselves.

However, since we know this book will be read by some command-line cowboys, we'll show you here how to run it from the command line with SBT and with Maven:

```
// Run from sbt
sbt clean package
$SPARK_HOME/bin/spark-submit \
--class com.packt.smack.WordFrequency \
./target/WordFrequency-0.0.1.jar \
./README.md ./wordfrequency
// Run from Maven
mvn clean &&mvn compile &&mvn package
$SPARK_HOME/bin/spark-submit \
--class com.packt.smack.WordFrequency \
./target/WordFrequency-0.0.1.jar \
./README.md ./wordfrequency
```

A wise counsel; if everything fails, follow the instructions. We can always refer to the official Spark quick start guide at `http://spark.apache.org/docs/latest/quick-start.html`.

RDD operation

In this chapter, we will repeat the following statement again and again: the RDD's have two types of operations; transformations and actions. Transformations are operations that receive one or more RDDs and return a new RDD. Actions return a result to a driver program and/or write it to storage, and trigger a computation. If you're still confused, as a rule of thumb, transformations return RDD, actions don't.

Transformations

As we all know, transformations on the RDDs are lazy operations. That is, they aren't applied until we perform an action or explicitly ask the program to apply them (collect).

Another important point on transformations is that many of them are element-wise, that is, many of them work as if they were operating on each collection item, one at time.

It is also important to mention that, because of the precepts of functional programming, transformations can't modify the value of the RDDs received as parameters. That is, everything is immutable, as it always should be.

Finally, if we have RDDs derived from other RDDs through transformations, Spark keeps track of each transformation and the dependencies between all RDDs. This record is known as a **lineage graph**. This is kept, as we all know, to recover lost information in case of failure, especially if the operations are clustered and run in a distributed way.

Here we enumerate the main transformations:

```
filter(function)
```

Our purpose is to build a new RDD by selecting those elements on which the function returns `true`.

The following is an example:

```
scala>val rdd = sc.parallelize(List("Spark","Cassandra","Kafka"))
scala>val filtered = rdd.filter(_.contains("k"))
scala>filtered.collect()
```

The result of this is:

```
Array[String] = Array(Spark, Kafka)

map( function )
```

The purpose is to return a new RDD by applying the function on each element:

```
scala>val rdd = sc.parallelize(List(1,2,3,4,5))
scala>val times = rdd.map(_*3)
scala>times.collect()
```

Our result is the following:

```
Array[Int] = Array(3, 4, 9, 12, 15)

flatMap(function)
```

Our purpose is similar to the map function, but this function returns a sequence instead of a value and then flattens the results; for example, by mapping a sequence into a sequence of words:

```
scala>val rdd = sc.parallelize(List("Spark is powerful", "Be like Spark"))
scala>val fm = rdd.flatMap(str => str.split(""))
scala>fm.collect()
```

The result of this is:

```
Array[String] = Array(Spark, is, powerful, Be, like, Spark)

reduceByKey(function,[number])
```

The purpose is to aggregate values of a key using a function, the parameter number is optional to specify the reduce tasks number:

The following is an example:

```
scala>val words = fm.map(w=>(w,1) )
scala>val wordCount = words.reduceByKey(_+_)
scala>wordCount.collect()
```

The result of this is:

```
Array[(String, Int)] =
Array((Spark, 2), (is, 1), (powerful,1), (Be,1), (like,1))

groupByKey([numTasks])
```

The purpose is to convert (K,V) to (K,Iterable<V>):

The following is an example:

```
scala>val countWord = wordCount.map{case (w, c) => (c, w)}
scala>countWord.groupByKey().collect()
```

The result of this is:

```
Array[(Int, Iterable[String])] = Array((1, ArrayBuffer(is, powerful, Be,
like)), (2,ArrayBuffer(Spark)))

distinct([numTasks])
```

The purpose is to eliminate duplicates:

The following is an example:

```
scala>fm.distinct().collect()
```

The result of this is:

```
Array[String] = Array(Spark, is, powerful, be, like)
```

Here we have the main set operations: `union()`.

The purpose is to build a new RDD containing all elements from the source and the argument:

The following is an example:

```
scala>val rdd1 = sc.parallelize(List("Spark","Scala")) scala>val rdd2 =
sc.parallelize(List("Akka","Scala")) scala>rdd1.union(rdd2).collect()
```

The result of this is:

```
Array[String] = Array(Spark, Scala, Akka, Scala)

intersection()
```

The purpose is to build a new RDD containing only common elements between source and argument:

The following is an example:

```
Scala>rdd1.intersection(rdd2).collect()
```

The result of this is:

```
Array[String] = Array(Scala)

cartesian()
```

The purpose is to build an RDD with a cross-product of all elements from the source and the argument:

The following is an example:

```
Scala>rdd1.cartesian(rdd2).collect()
```

The result of this is:

```
Array[(String, String)] = Array(("Spark","Akka"), ("Spark","Scala"),
("Scala","Akka"), ("Scala","Scala"))
```

```
subtract()
```

The purpose is to build a new RDD by removing common data elements between source and argument.

The following is an example:

```
scala>rdd1.subtract(rdd2).collect()
```

The result of this is:

```
Array[String] = Array(Spark)
```

```
join(RDD,[number])
```

The purpose is to invoke on (K,V) and (K,W), to create a new RDD with (K, (V,W)):

The following is an example:

```
scala>valhash1 = sc.parallelize( Seq(("1", "A"), ("2", "B"),
("3", "C"), ("1","D")))
scala>valhash2 = sc.parallelize( Seq(("1", "W"), ("2", "X"),
("3", "Y"), ("2","Z")))
scala>hash1.join( hash2 ).collect()
```

The result of this is:

```
Array[(String, (String, String))] = Array((1,(A,W)), (1,(D,W)), (2,(B,X)),
(2,(B,Z)), (3,(C,Y)))
```

```
cogroup( RDD, [number])
```

The purpose is to convert (K,V) to (K,Iterable<V>):

The following is an example:

```
scala>hash1.cogroup(hash2).collect()
```

The result of this is:

```
Array[(String, (Iterable[String], Iterable[String]))] =
Array((1,(CompactBuffer(A, D),CompactBuffer(W))),
(2,(CompactBuffer(B),CompactBuffer(X, Z))),
(3,(CompactBuffer(C),CompactBuffer(Y))))
```

Actions

Right, we have our RDD but what if we want to do something really important with our RDD? This is where Actions come in to their own. Whenever people ask "what is more difficult, a transformation or an action", there is no correct answer because, although actions return simple values, the process can be very complex.

Actions return a result to a driver program and/or write it to storage for calculation.

At the point in your transformations pipeline that we find an action, everything to that point will be evaluated in that moment. Actions trigger the transformation evaluation, as actions always need to produce an output.

Here we have the main actions:

```
count()
```

The purpose is to get the number of elements in the RDD:

The following is an example:

```
scala>val rdd = sc.parallelize(List('S','M','A','C','K'))
scala>rdd.count()
```

The result of this is:

```
long = 5

collect()
```

The purpose is to get all the elements in the RDD as an array:

The following is an example:

```
scala>val rdd = sc.parallelize( List('S', 'M', 'A', 'C', 'K') )
scala>rdd.collect()
```

The result of this is:

```
Array[Char] = Array(S, M, A, C, K)
```

```
reduce(function)
```

The purpose is to aggregate the data elements in the RDD using the argument function:

The following is an example:

```
scala>val rdd = sc.parallelize(List(5,4,3,1, 2))
scala>rdd.reduce(_+_) // the sum of all
```

The result of this is:

```
Int = 15
```

```
take (n)
```

The purpose is to fetch the RDD's first *n* data elements:

The following is an example:

```
Scala>val rdd = sc.parallelize(List('S', 'M', 'A', 'C', 'K'))
scala>rdd.take(3)
```

The result of this is:

```
Array[Char] = Array(S, M, A)
```

```
foreach(function)
```

The purpose is to execute the function in each element of the RDD:

The following is an example:

```
Scala>val rdd = sc.parallelize(List(3,2,1,4, 5))
scala>rdd.foreach(n=>print( "%s*5=%s".format(n, n*5)))
```

The result of this is:

```
3*10=15 1*10=5 4*10=20 5*10=25 2*10=10
```

```
first()
```

The purpose is to retrieve the first data element in RDD, similarly to take(1):

The following is an example:

```
scala>val rdd = sc.parallelize(list(1,2,3,4))
scala>rdd.first()
```

The result of this is:

```
Int = 1
```

```
saveAsTextFile(path)
```

The purpose is to write the RDD content to a text file on a local filesystem/HDFS:

The following is an example:

```
scala>val myLogs = sc.textFile("/users/smack/reallyHugeLog.log")
scala>myLogs.filter(_.contains("Fatal")).
saveAsTextFile("/users/smack/fatality.txt")
```

The result of this is:

```
smack@localhost~/smack$ ls _SUCCESS part-00000 part-00001
```

Persistence (caching)

DDRs have lazy evaluation, but, what if we want to use the same RDD several times? If we don't approach this with caution, Spark will, by default, recalculate the RDD and all its dependencies each time you take an action over the RDD. That is very costly, especially in iterative algorithms that fetch the same data several times.

We can ask Spark to persist the data to avoid calculating an RDD every time. When we persist an RDD, the nodes working on an RDD store their partitions. If a node fails, Spark recalculates lost partitions as necessary.

We can also replicate our data on multiple nodes if we want to be able to handle a node failure without performance implications. Spark has many persistence levels to choose from based on our needs. The data is always serialized when writing data to disk, and also when using off-heap catching (using Alluxio, formerly Tachyon, `http://alluxio.org/`).

Persistence level	**CPU used**	**Space used**	**On disk**	**In memory**
MEMORY_ONLY	Low	High	No	Yes
MEMORY_AND_DISK(*)	Medium	High	Some	Some
MEMORY_ONLY_SER	High	Low	No	Yes
MEMORY_AND_DISK_SER(*)	High	Low	Some	Some
DISK_ONLY	High	Low	Yes	No
OFF_HEAP (experimental)	Low	Low	Some	Some

* Write to disk if there is a lot of data stored in memory

Table 3.1 Persistence levels, SER stands for Serializable

The following is the example:

```
import org.apache.spark.storage.StorageLevel
val rdd= input.map(foo)
rdd.persist(StorageLevel.DISK_ONLY)
rdd.reduce(bar)
rdd.collect()
```

We must call the `persist()` method before the first action. Note that the `persist()` call itself doesn't force evaluation. The off-heap catching doesn't guarantee recovery after fail.

Spark automatically evicts old partitions using a **Least Recently Used** (**LRU**) cache policy.

The RDDs have the method `unpersist()` to manually remove them from cache.

Spark in cluster mode

So far in this chapter we have focused on running Spark in local mode. As we mentioned, horizontal scaling is what makes Spark so sensual and powerful. You don't need software-hardware integration gurus to run clusters with Apache Spark, and you don't need to stop the organization's entire production to escalate and add more machines to your cluster.

The good news is that the same scripts that you build on your laptop on samples of a few kilobytes, can run on business clusters that handle terabytes of information. There's no need to change the code, and no need to invoke another API. All you have to do is to test again and again to be sure your model runs correctly, and then deploy the cluster.

In this section, we'll describe the runtime architecture of a distributed Spark application, and then we'll see the options we have to run a Spark application running on a cluster.

Apache Spark has its own built-in cluster standalone manager but you can run multiple cluster managers, including: Apache Mesos, Hadoop YARN, and Amazon EC2. This subject is so vast that it has its own chapter in this book.

Runtime architecture

Before discussing Spark running in a cluster, it is important to understand the Spark architecture in distributed mode.

Spark uses a master/slave architecture: As we saw, the master is called **driver** and the slaves are called **executors**. It is important to note that we can have a distributed architecture in a single machine, that is, a driver with several executors. The driver always runs in its own Java process, and each executor runs in a separate Java process. If the reader finds this model similar to the actor's model it is because is based on those principles.

The set of a driver with executors is known as the **Spark application**.

If we have more than one machine, the Spark application must be launched using a service called **cluster manager**. The architecture of the Spark application will always be the same, whether it is clustered or not.

The canonical architecture of Spark running in cluster mode is when each physical machine has its own executor. As we shall see, there are several strategies to follow when an executor dies or goes offline.

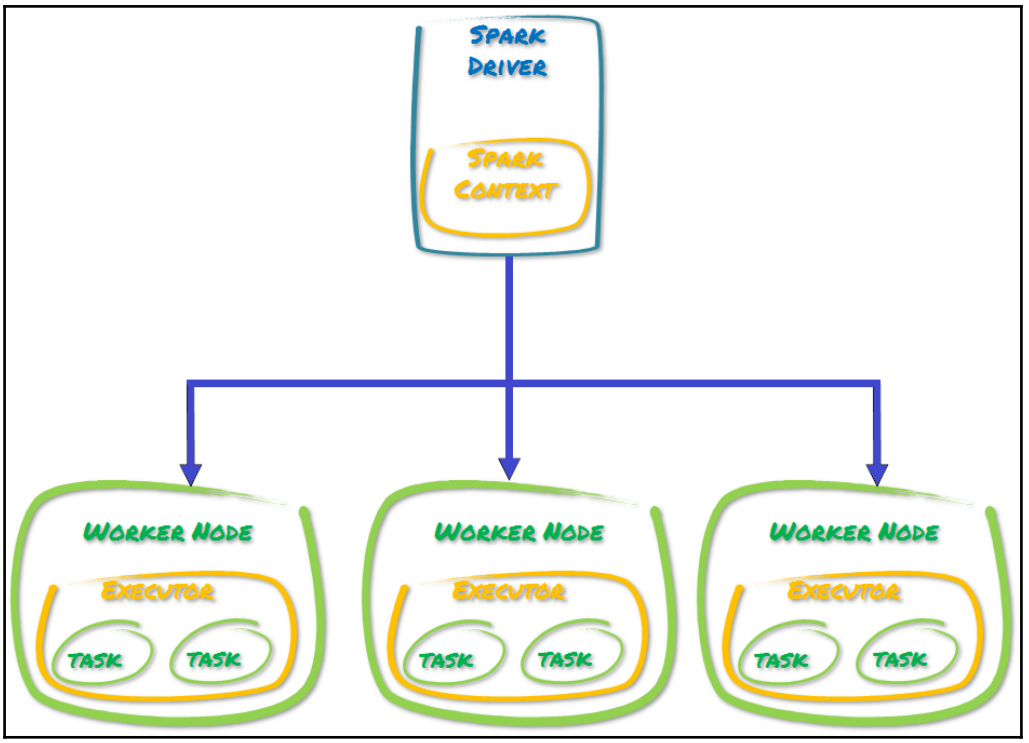

Figure 3-4. Distributed Spark application

Driver

As we saw, the driver is the process where our Spark context runs. It also has the task of creating and executing transformations and actions on RDDs. When we run the famous `spark-shell` command on our machine, we are actually creating a driver program, and its first step is to create the Spark context (`sc` for the friends). When the driver program dies, the application dies.

The two responsibilities in the life of a driver program are:

Dividing a program into tasks

The Spark driver is responsible for splitting the user program (which can be inefficient) into execution units called **tasks**.

A user program, as we know, applies transformations and sections to one or more RDDs, generates new RDDs, and calculates and/or saves data.

The duty of the Spark driver is to generate a **Directed Acyclic Graph** (**DAG**) of the operations. As we mentioned, this graph forms dependencies between DAGs. The driver knows which node will do which task, so if the node dies, the driver knows at what point it stayed and who assigned the task of the deceased node.

Spark also undertakes many pipelining optimizations, splitting the DAG into stages. Each stage has multiple tasks. It is very important to recall that the tasks are the smallest unit of work in Spark, and a normal program can launch thousands of tasks.

Scheduling tasks on executors

Given a physical execution plan, the Spark driver coordinates which task will perform each executor node. When an executor starts operating, it registers itself with the driver so the driver always has a complete view of all the executor's nodes. Each executor is a standalone Java process and can run tasks and store RDDs.

When a program runs, the driver subdivides the program into tasks, checks the available executor nodes, and tries to balance the workload among all executors. The driver also knows what part of the data each node has in order to rebuild everything at the end.

The driver doesn't execute the next stage until the previous stage has fully completed. The reason is that shuffled results are all written to the executors' disks, so in case of further failure the data can be accessed at that point without triggering re-computation from the very beginning.

The driver exposes all the information on the web, so the user knows what is happening at a given point. It is exposed on port `4040`. When we run it locally, it can be accessed at `http://localhost:4040` (go run the Spark shell and check it).

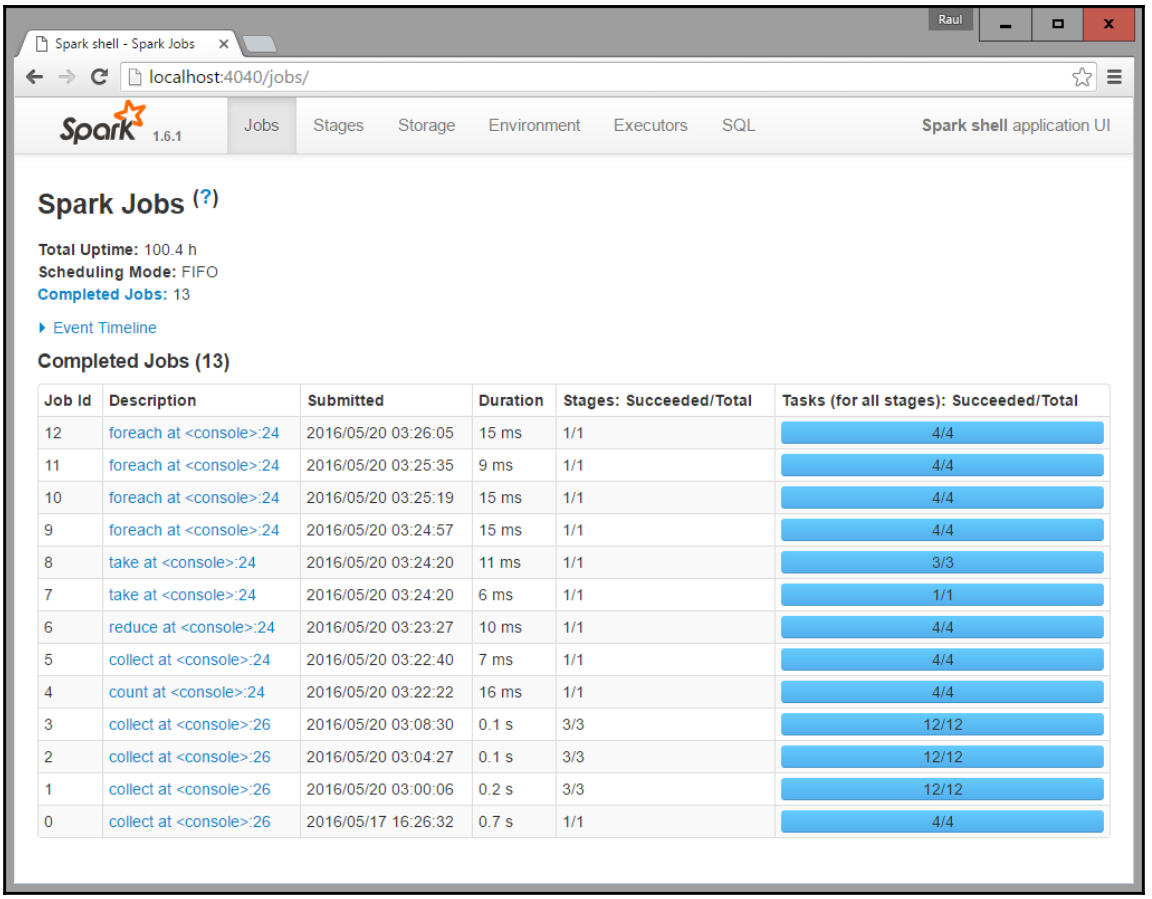

Figure 3-5. Spark shell application web UI

Executor

The executors are responsible for running the individual tasks of a given Spark job. The executors are launched when we start the Spark driver and the goal is that they exist while the Spark application lives.

Executors have two roles:

- Run the assigned tasks and return the results to the driver
- Provide in-memory storage for RDDs: There is a program called block manager within each executor that manages the memory and the RDDs

If we run Spark in local mode, the driver and executors exist in the same Java process. This is just for development purposes, and it's not recommended in a production environment.

Cluster manager

Spark depends on a cluster manager to coordinate and launch the executors. The cluster manager is a pluggable component, that is, we can change and use custom cluster managers (for example, Apache Mesos, Hadoop Yarn, or Amazon EC2).

It is noteworthy that when talking about the Spark application, we use the terms **driver** and **executor**. However, when we talk in terms of the cluster manager we use the terms **master** and **worker**. As mentioned at the beginning of this section, it is important not confuse the terms or exchange them because they are different concepts.

No matter which cluster manager we use, Spark provides a single script to launch our program called `spark-submit`. The `spark-submit` command offers various options to connect to different managers and manage cluster resources that the application needs.

Program execution

When we run a Spark application on a cluster, these are the steps followed by the program:

1. The user runs the `spark-submit` shell.
2. The `spark-submit` launches the driver program, and invokes the `main()` method of the user program.
3. The driver program establishes the connection to the cluster manager, which has a list of slave machines. Thus, the necessary resources are requested to launch the executors.
4. The cluster manager launches executors in each slave node.
5. The driver program runs the user application: analyzes it, divides it, and distributes it, sending each executor their tasks.
6. Tasks run in the executors which calculate and store the results.
7. The program ends when the method `exit()` in `main()` is invoked, or the Spark context `stop()` method is called.
8. The driver program ends the executors and frees the cluster manager resources.

Application deployment

Spark provides the `spark-submit` tool to submit jobs to the cluster manager.

To run our program in standalone mode, we only invoke `spark-submit`, passing our script name or `JAR` file as a parameter in order to run it in local mode.

If we run our program in cluster mode, we have to pass additional parameters, for example, the size of each executor process.

The `--master` flag specifies the cluster URL to which we want to connect. In this case, `spark://` means we are using Spark in standalone mode.

For example:

```
bin/spark-submit --master spark://luckyhost:7077 --executor-memory
10gLotteryCalculator.jar
```

Here we indicate that we will run our `LotteryCalculator` program in standalone cluster mode in the master node `luckyhost` and each executor node will have 10 gigabytes of memory.

In addition to the cluster URL, `spark-submit` has several options to specify how we want to run our application. These options are in two categories:

- Scheduling information: The amount of resources each job will have
- Dependencies: Information such as libraries and files that you want in the slave machines

Here are some `spark-submit` flags:

- The cluster manager to connect to, with the values explained previously:

```
--master
```

- An indication of whether the program launched is **client mode** (local) or **cluster mode**. If local, the driver is launched whenever `spark-submit` is launched. The default value is client mode:

```
--deploy-mode
```

- Your Java/Scala application main class:

```
--class
```

- Our application's human-readable name, as it'll be displayed in Spark's web UI:

```
--name
```

- A list of JAR files to upload and place in the application class path:

```
--jars
```

- A list of files placed in our application's working directory, distributed to each node:

```
--files
```

- Number of executors:

```
--num-executors
```

- Number of cores:

```
--total-executor-cores
```

- Memory for executors: k for kilobytes, m for megabytes, g for gigabytes:

```
--executor-memory
```

- Memory for Driver Process: k for Kilobytes, m for megabytes, g for gigabytes:

```
--driver-memory
```

- Specify a single configuration property value in key-value form:

```
--conf prop=value
```

- Specify a configuration file with properties in key-value form:

```
--properties-file
```

Here are a few example values that `--master` flag could have:

```
spark://host:port
```

Connect to a cluster in standalone mode at a specified host and port. The standalone master default port is `7077`:

```
mesos://host:port
```

Connect to a Mesos cluster at a specified host and port. The Mesos master default port is `5050`:

Master in local mode with a single core:

```
local
```

Master in local mode with *N* cores:

```
local[N]
```

Master in local mode and using as many cores as the machine has:

```
local[*]
```

For example, we can now read this:

```
$ ./bin/spark-submit \
--master spark://masterHelix:7077\
--deploy-mode cluster \
--class com.cyberdyne.DNASequencer\
--name "DNA Sequencer" \
--jars neuralNet.jar,geneticAl.jar\
--total-executor-cores 300 \
--executor-memory 10g\
sequencer.jar
```

Here we indicate that we will run our `sequencer.jar` program:

- In standalone cluster mode in the master node `masterHelix` on port `7077`
- Driver program launched in cluster mode
- The main class is `com.cyberdyne.DNASequencer`
- We use the `neuralNet.jar` and `geneticAl.jar` JAR files
- Each executor node uses 300 cores and has 10 gigabytes of memory

Standalone cluster manager

Spark standalone manager is the simplest option to run the Spark clustered applications. As already stated, there is one master and several workers. By submitting the application, we can specify the memory and the number of cores of each executor node. Also, in this cluster mode we can specify the number of cores in the driver program.

Launching the standalone manager

Here are the instructions to distribute Spark in a cluster environment to run on Linux and Mac OS X. To use the cluster launch scripts we follow these steps:

1. Copy the compiled version of Spark in the same directory on all machines, for example `/home/JohnDoe/spark`.
2. We need to establish a password-less SSH access from the master machine to other machines. This requires that there is the same user account on all machines. Then we have to create an SSH key for the master using the `ssh-keygen` command. We have to add this key to the `.ssh/authorized_keys` file on each worker machine. If you haven't done this before, follow these steps:

- On the master node run `ssh-keygen` entering the default options:

```
$ ssh-keygen -t dsa
Enter file in which to save the key
(/home/JohnDoe/.ssh/id_dsa):  [ENTER]
Enter passphrase (empty for no passphrase): [EMPTY]
Enter same passphrase again: [EMPTY]
```

- On worker nodes:

```
copy the file .ssh/id_dsa.pub from the Master to the Worker,
then:
$ cat ~/.ssh/id_dsa.pub>> ~/.ssh/authorized_keys
$ chmod 644 ~/.ssh/authorized_keys
```

3. Edit the file `conf/slaves` in the master machine and enter the workers hostnames.
4. To start the cluster, run `bin/start-all.sh` on the master (it's very important run it from the master, and never from a worker). If everything starts okay, you should be able to enter without requesting a password, and the web user interface cluster administrator will appear on `http://masterHost: 8080` showing all the workers.
5. To stop the cluster, run. `bin/stop-all sh` in the master node.

If you are not on a Unix-like system or you need to launch the cluster manually, you can start the cluster by hand, using the `spark-class` script:

On the master node, run:

```
bin/spark-class org.apache.spark.deploy.master.Master
```

On every worker, type:

```
    bin/spark-class org.apache.spark.deploy.worker.Worker
spark://urlMasterNode:7077
```

Here `urlMasterNode` is our master hostname. Remember, on Windows, use \ instead of /.

Now the master node will allocate the necessary resources (CPU and memory) on each worker using the default values.

Submitting our application

We are going to submit an application to our standalone cluster manager by typing:

```
spark-submit --master spark://masterNodeHost:masterNodePort appName
```

Previously, we've seen the syntax of the `spark-submit` command, now all we need to do is just change the direction of the master node.

This cluster URL is also displayed in the web UI of the cluster administrator at `http://masterNodeHost: 8080`.

The host name and port used should exactly match the URL present in the web UI. As Spark administrators, we can configure a different port than `7077`.

As we have seen, the standalone cluster manager has a `--deploy-mode` option:

- **Client mode** (local, default): The driver runs the same process as `spark-submit`. If you shut down `spark-submit`, the whole cluster goes down.
- **Cluster mode**: The driver is launched as another process on one of the worker nodes and you can disconnect the machine that `spark-submit` is running on and everything will continue to run.

The `--total-executor-cores` and `-- executor-memory` parameters allow us to specify the number of cores and memory available for each executor node. A common mistake is to ask for more resources than can be assigned. In this case, the cluster manager won't start the executor's nodes. Always check that your cluster has the resources you say in the parameters.

Configuring resources

As we said in the standalone cluster manager, the resource usage is controlled by two variables:

- Memory:

```
--executor-memory
```

This is the argument of the `spark-submit` command.

Each application will have at most one executor in each worker. By default, this value is set to 1GB. In production environments the memory assigned to the executor process is usually bigger than 1GB. When this book was written, most node machines can provide more than 1GB to one process.

- Cores:

```
--total-executor-cores
```

This is the argument of the `spark-submit` command and is the number of cores used for executors. The default is unimited, that is, the application raises one executor on each available node in the cluster.

To check our current parameters, we can check the standalone cluster manager's web UI in `http: // masterNodeHost: 8080`.

By default, the standalone cluster manager disperses the executors to the largest number of machines in the cluster. For example, suppose we have a six-node cluster, each node is a machine with a four-core CPU. We launch the application with `-total-executor-cores` 5. Spark will raise only five executors, and tend to use the largest number of machines possible, that is, an executor on each machine, so we will have one node without an executor.

If we want Spark to use the fewest number of nodes possible, we change the configuration property `spark.deploy.spreadOut` to `false` in the configuration file: `conf / spark-defaults.conf`. In this case, for our example, we are going to use only two machines, one with four executors, one with one executor, and the other four machines will have no executors.

This setting affects all applications on the standalone cluster, so use it wisely.

Working in the cluster

We've already mentioned how to indicate the cluster mode in the deployment mode of the standalone manager so that if the `spark-submit` process dies, the manager won't die. This is because the driver survives in one node of the cluster.

Most of us want our manager to continue to exist while a node is still standing. To ensure the longevity of our manager, there is a powerful tool called **Apache Zookeper**, which we will discuss later, in the Apache Mesos chapter.

Spark Streaming

When studying calculus, one thing that remains clear is that life is not a discreet process, it is continuous; and life does not come in small packages, it is a continuously flowing stream.

As discussed in the first chapter, the fresher the information, the greater the benefit of the data. Many modern applications of machine-learning should be calculated in real-time.

Spark Streaming is the module for managing data flows. Much of Spark is built with the concept of RDD. Spark Streaming provides the concept of DStreams, or Discretized Streams. A DStream is a sequence of information related to time. It is very important to emphasize that an internal DStream is a sequence of RDD, hence the name *discretized*.

Just as RDDs have two transformations, DStreams also offer two types of operations:

- Transformations (whose result is another DStream)
- Output operations aimed at writing information to external systems

DStreams have many of the operations available in the RDDs, plus newer time-related operations, such as sliding windows.

Unlike batch operation, the Spark Streaming applications require additional configurations to provide a 24/7 service. We'll also talk about how to reset applications in case of failure and leave the configuration ready for automatic restart.

Spark Streaming architecture

Spark Streaming uses a *micro-batch* architecture where the streaming is considered to be a continuous series of small batches of data. The magic of Spark Streaming is receiving a continuous flow and splitting it into small pieces.

The batches are generated at regular time intervals; if two pieces of data come in the same time window, they are treated as the same batch of information. The size of the batches is determined by a parameter called **batch interval** which typically has a value between 500 milliseconds and several seconds. The application developer controls this value as required.

The main task of Spark Streaming is to receive a continuous stream of data from multiple sources, and build a DStream, which is a sequence of RDDs. Each RDD corresponds to a slice of time of the original flow.

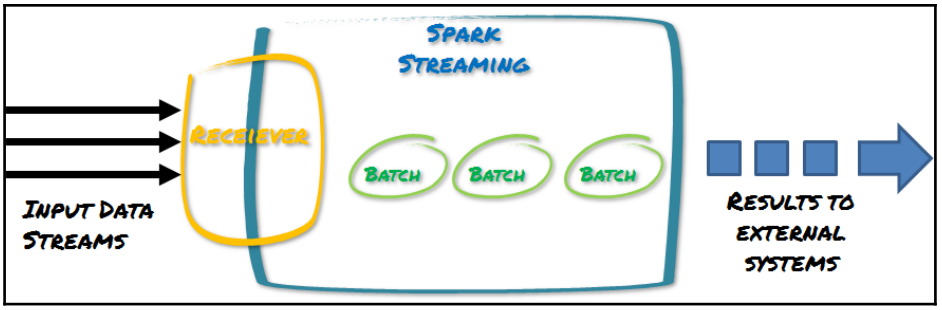

Figure 3-6. Spark Streaming Operation

We can create DStreams, from input sources or by applying transformations to other DStreams. The DStreams support most of the transformations that RDDs support. Additionally, DStreams have stateful transformations that can aggregate data across time.

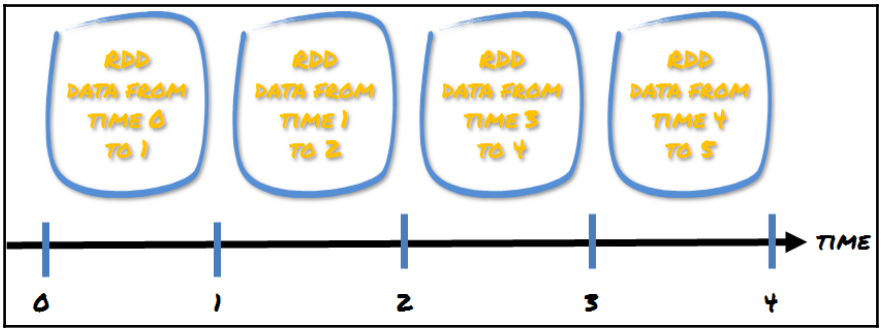

Figure 3-7. A DStream as an RDD series

In addition to transformations, DStreams support output operations which are similar to the actions that RDDs write to external systems, but Spark Streaming batches run periodically producing the output batch.

For each input source, Spark launches streaming receivers, which are tasks that run on executors and gather data from information sources and store it in RDDs. The receivers are also responsible for replicating data among other executors to support fault tolerance. This data is stored in the memory of executors in the same way as the RDDs. The Streaming context in the driver program periodically runs Spark jobs to process this data and combine them with the new RDDs.

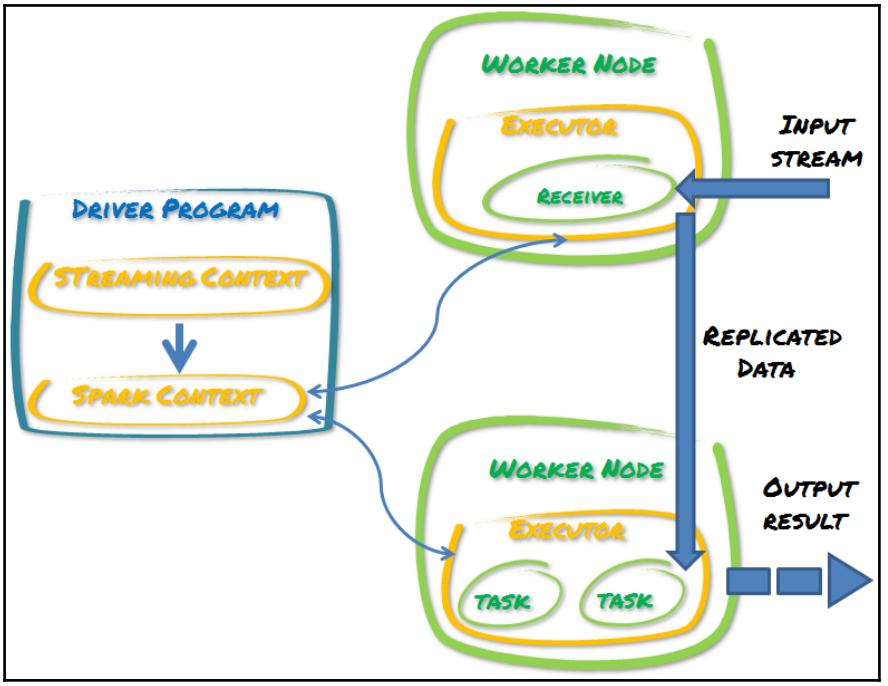

Figure 3-8. Spark Streaming execution with Spark components

Spark Streaming offers the same fault tolerance properties in DStreams that are offered with RDDs. We can always recalculate DStreams in case of failure, at any point in time. However, the recalculation can take time, especially if reconstruction is carried out from the beginning of the program.

Spark Streaming provides a mechanism called **checkpointing** that periodically saves the state to a reliable file system. Typically, a checkpoint is made at between five to ten data batches. When something bad happens, Spark only needs to restore from the last checkpoint.

Transformations

Transformations on streams can be grouped into stateless or stateful (like Session Enterprise JavaBeans) streams:

- **Stateless:** Each batch process is not dependent on data from previous batches. RDD includes transformations already known as `map()`, `reduce()`, `filter()`.
- **Stateful:** In contrast, it uses data, or intermediate results of previous batches, to calculate the result of the current batch. These include transformations based on sliding windows and status tracked over time.

Stateless transformations

The stateless transformations are simple RDD transformations applied to each batch, to each RDD belonging to the DStream. Here we enumerate the main stateless transformations:

The purpose is to apply a function to each RDD in the DStream, returning one DStream as a result:

```
map( function )
```

The following is an example:

```
ds.map(_*3)
flatMap( function )
```

The purpose is to apply a function to each RDD in the DStream, returning one DStream of the contents of the returned iterators:

The following is an example:

```
ds.flatMap(str=>str.split(""))
filter( function )
```

The purpose is to return a DStream consisting of only RDDs evaluated to `true` on the function parameter.

The following is an example:

```
ds.filter(_.contains("k"))
repartition( number )
```

The purpose is to change the number of DStream partitions.

The following is an example:

```
ds.repartition(9)
reduceByKey( function, [number])
```

The purpose is to combine the values with the same key in each batch.

The following is an example:

```
ds.reduceByKey(_+_)
groupByKey()
```

The purpose is to group values with the same key in each batch. The following is an example:

```
ds.groupByKey()
```

It is important that it appears that the transformation applies to the whole DStream. However, this is not the case: actually, it is applied to each batch element (RDD) of the DStream individually. For example, `reduceByKey()` applies the function to each RDD, not all DStreams.

The stateless transformations can also combine data from multiple DStreams, within each time step. For example, DStreams have the same join transformations as RDDs, which are: `cogroup()`, `join()`, and `leftOuterJoin()`. We can use these DStream operations to perform them in each batch.

We also can merge the contents of two different DStreams using the `union()` operation as usual, or using `StreamingContext.union()` to merge several DStreams.

If all this isn't enough, DStreams provide an advanced operator called `transform()` that can operate directly on the RDDs within a DStream. This operation is called on each element of the DStream, producing a new DStream. If we have existing code that we wrote for some RDDs and we now want to use in the Streaming Spark, the `transform()` method is a good way to re-use the same code.

Finally, we can combine several DStreams using `StreamingContext.transform()` or `DStream.transformWith` (another DStream, function).

Stateful transformations

The stateful transformations are DStream operations that track data across time. As previously indicated, data from old batches are used to generate new batches.

There are two types of stateful transformations: Windowed transformations and Update stateby key.

- **Windowed transformations**: Acting on data over a time period window
- **Update stateby key**: Used to track the status among the same key events, for example, a user session

Stateful transformations require checkpointing to enable fault tolerance.

Windowed operations

Windowed operations calculate results in a larger period than the Streaming context batch interval time, allowing the combination of the results of several batches.

All windowed operations require two parameters: the **window duration** and the **slide duration**. Both must be a multiple of the streaming context batch interval.

The duration of the window controls how many previous batches will be considered. The formula is: **Batches considered = window duration/batch interval**

For example, if we have a DStream with an interval of five seconds, and the duration of the window is 30 seconds, then we will only consider the last six batches.

The slide duration indicates how often we want to calculate the results. Its default value is the batch interval duration.

For example, if we have a batch interval of ten seconds and we calculate our window every two seconds, we must change our slide duration to 20 seconds.

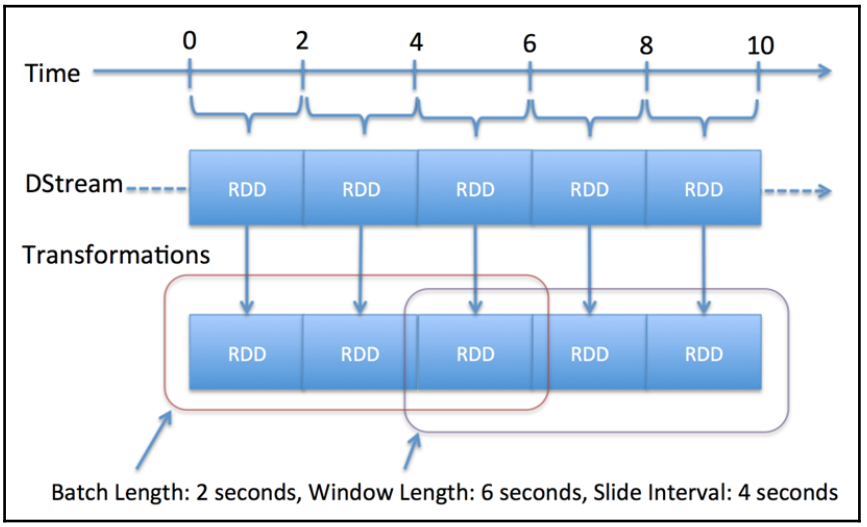

Figure 3-9. Windowed operations example

The simplest operation that can be performed on a DStream is `window()`, which returns the DStream with the information of the current window:

```
window(windowLength, slideInterval)
```

The purpose is to return a new DStream computed from windowed batches of the source DStream.

The following is an example:

```
val wind = lines.window(Seconds(30),Seconds(10));
wind.foreachRDD(rdd => {rdd.foreach(x=>println(x+ " "))})
```

The following is the output:

```
10 10 20 20 10 30 20 30 40 (drops 10)
```

Spark streaming provides other windowed operations to improve efficiency. For example, `reduceByWindow()` and `reduceByKeyAndWindow()` allow us to make reductions in each window in a very efficient way. They have a special form that allows Spark to calculate the reduction incrementally, considering only the information entering and exiting data.

Finally, to count data, DStream offers `countByWindow()` and `countByValueAndWindow()`:

- `countByWindow()` gives us a DStream representing the number of elements in each window
- `countByValueAndWindow()` gives us a DStream representing the counts of each value

```
countByWindow(windowLength, slideInterval)
```

The purpose is to return a new sliding window count of elements in a stream

The following is an example:

```
lines.countByWindow(Seconds(30),Seconds(10)).print()
```

The following is the output:

```
1233
```

Update state by key

Sometimes it's useful to maintain a state between batches in a DStream.

The `updateStateByKey()` provides access to the DStream state variables given a DStream pair (key, event) taking a function that specifies how to update each key status given new events. To use `updateStateByKey()`, provide an update function (event, `oldState`) takes the past events with a key and its previous state, and returns a new state to make the update.

The parameters of this function are:

- `events` : List of events in the current batch (can be empty)
- `oldState` (optional): We may not have a previous state for that key
- `newState` (optional): To specify that you want to delete the previous state

The result of `updateStateByKey()` is a new DStream with RDD in pairs (key, state) at each time step.

Output operations

Output operations specify what to do with the transformed data in a stream (perhaps printed, shown on the screen, or stored in a database).

Like many of the lazy operations in the RDD, if there is not an output operation applied to DStream or any of its dependents, the DStreams won't be evaluated.

Similarly, if there are no output operations in a Streaming context, it will not start.

An output operation used commonly for debugging is the simple `print()`. Printing the results on screen counts as an output operation.

When going into production, it is very important to consider this, as if we remove the `print()` we could be leaving our program without output operations.

Once our program is debugged we can use output operations to store our results. Spark Streaming has the `RDDsave()` operation for DStreams, just like RDDs, and it takes the directory on the file system where we want to store our data.

The results of each batch are stored in sub-directories of the specified directory with the specified suffix as the filename.

Finally, `foreachRDD()` is a generic output operation that allows us to compute each DStream RDD. It is similar to `transform()`. For convenience, it also gives each batch time, making it possible to save each time period in a different place.

Fault-tolerant Spark Streaming

One of the main advantages is that Spark streaming provides a mechanism to guarantee fault tolerant. According to whether the input data is stored reliably, Spark Streaming always calculates the result from it providing the correct semantics, that is, as if the data had been processed without failing nodes.

Spark streaming applications running 24/7 need a special installation. The first step is to enable checkpointing on a reliable storage system and HDFS or Amazon S3. In addition, we must also deal with the driver program fault tolerance by changing some portion of the code.

Checkpointing

Checkpointing is the primary mechanism necessary to enable fault-tolerance Spark streaming. Spark streaming allows us to periodically save the data in the application in a reliable file system, such as HDFS or Amazon S3. As we've seen, checkpointing has two purposes:

- Limits the state to be recomputed when a fault occurs. Remember that Spark streaming recalculates the state using a lineage graph of transformations, but checkpointing tells us how far back we should go.
- Provides driver fault tolerance. If the driver program of a streaming application crashes, you can start it again and tell it to recover from the last checkpoint. In this case, Spark streaming reads how much has processed and will resume from that point.

For these reasons, checkpointing is important when we run production streaming applications.

It should be noted that even running in local mode, Spark streaming won't run a stateful operation if we haven't enabled checkpointing. In this case, we can pass a local file system to do checkpointing. In a production environment, we should use a replicated file system like HDFS, Amazon S3, or NFS.

Spark Streaming performance

Spark streaming has some specific considerations additional to Spark performance considerations:

Parallelism level

A common way to reduce batch processing time is increased parallelism. There are three ways to increase it:

- **Increasing parallelism**: For operations such as `reduceByKey()` we can specify parallelism as the second parameter, as mentioned in the RDDs section.

- **Adding receptors**: Receivers usually cause a bottleneck. If there are many records to read and distribute for a single machine, we can add more recipients creating multiple input DStreams, which create multiple receivers, and then apply a `union()` operation to merge them into a single stream.
- **Repartitioning data**: If you cannot increase the number of receivers, you can redistribute the data received by explicitly repartitioning the received data using the `repartition()` method.

Window size and batch size

A common question is; what is the appropriate batch size? In general, for many applications 500 milliseconds is a good number for the minimum batch size. The best method is to start the minimum size batch with a large number, about 10 seconds, and then keep reducing the number down till you reach the optimal size. If processing times remain consistent, you can continue to reduce the batch size, but if it any point it increases, you've found the optimal size.

In a similar fashion to windowed operations, the interval to calculate a result has a great impact on performance. If it causes a bottleneck, consider increasing this interval for expensive calculations.

Garbage collector

A problem that always causes problems with the JVM is garbage collection. We can minimize the garbage collection pauses, by enabling the concurrent garbage collector `Mark-Sweep`. Generally, this consumes more resources but has fewer pauses.

If you do not know how to enable `Mark-Sweep`, we use -XX: + UseConcMarkSweepGC in `spark.executor.extraJavaOptions` when launching the `spark-submit` command.

In addition, to keep the garbage collection at minimum, we can lower the pressure on the garbage collector; for example, by caching RDDs in serialized form.

Summary

In this chapter, we learned the key points of Apache Spark from scratch. We saw how to download, install, and test Apache Spark. We also saw how to run Spark applications; and we reviewed some Spark core concepts, such as RDD, and the RDD operations (transformations and actions).

Also, we saw how to run Apache Spark in cluster mode, how to run the driver program, and how to achieve high-availability.

Finally, we dived into Spark streaming, the stateless and stateful transformations, the output operations, how to enable it 24/7, and how to improve Spark streaming performance.

In the following chapters, we will see how Apache Spark is the glue to our stack. In each chapter we will see the relationship with this technology.

4
The Storage - Apache Cassandra

We have reached the part where we talk about storage. The C in the SMACK stack refers to **Cassandra**. The reader may wonder, why not use a conventional database? The answer is that Cassandra is the database that propels giants such as Walmart, CERN, Cisco, Facebook, Netflix, and Twitter. Spark uses a lot of Cassandra's power. Application efficiency is greatly increased using the Spark Cassandra Connector.

This chapter has the following sections:

- A bit of history
- NoSQL
- Apache Cassandra installation
- Authentication and authorization (roles)
- Backup
- Recovery
- Spark-Cassandra connector

A bit of history

In Greek mythology, there was a priestess who was chastised for her treason against the God, Apollo. She asked for the power of prophecy in exchange for a carnal meeting; however, she failed to fulfill her part of the deal. So, she received a punishment; she would have the power of prophecy, but no one would ever believe her forecasts. This priestess's name was Cassandra.

Moving to more recent times, let's say 50 years ago, in the world of computing there have been big changes. In 1960, the HDD (Hard Disk Drive) took precedence over the magnetic strips facilitating data handling. In 1966, IBM created the **Information Management System** (**IMS**) for the Apollo space program from whose hierarchical models later developed IBM DB2. In 1970s, a model that is fundamentally changing the existing data storage methods appeared, called the relational data model. Devised by Codd as an alternative to IBM's IMS and its organization mode and data storage in 1985, his work presented 12 rules that a database should meet in order to be considered a relational database.

The Web (especially social networks) appeared and demanded the storage of large amounts of data. The **Relational Database Management System** (**RDBMS**) scales the actual costs of databases, the number of users, amount of data, response time, or the time it takes to make a specific query on a database. In the beginning, it was possible to solve these through vertical scaling: the server machine is upgraded with more RAM, higher processors, and larger and faster HDDs. Now we can mitigate the problem, but it will not disappear.

When the same problem occurs again, and the server cannot be upgraded, the only solution is to add a new server, which itself may hide unplanned costs: OS license, **Database Management System** (**DBMS**), and so on, without mentioning the data replication, transactions, and data consistency under normal use.

One solution to such problems is the use of NoSQL databases. NoSQL was born from the need to process large amounts of data based on large hardware platforms built through clustering servers.

The term *NoSQL* is perhaps not precise. A more appropriate term should be *Not Only SQL*. It is used on several non-relational databases such as Apache Cassandra, MongoDB, Riak, Neo4J, and so on, which have become more widespread in recent years.

NoSQL

In this book, we will read NoSQL as *Not only SQL* (SQL, Structured Query Language). NoSQL is a distributed database with an emphasis on scalability, high availability, and ease of administration, the opposite of established relational databases. Don't think of it as a direct replacement for RDBMS, rather, as an alternative or a complement. The focus is in avoiding unnecessary complexity, the solution for data storage according to today's needs, and fixed schemes. Due its distributed nature, cloud computing is a great NoSQL sponsor.

A NoSQL database model can be:

- Key-value/tuple based

For example, Redis, Oracle NoSQL (ACID-compliant), Riak, Tokyo Cabinet / Tyrant, Voldemort, Amazon Dynamo, and Memcached and is used by Linked-In, Amazon, BestBuy, Github, and AOL.

- Wide Row/column-oriented-based

For example, Google BigTable, Apache Cassandra, Hbase/Hypertable, and Amazon SimpleDB used by Amazon, Google, Facebook, and RealNetworks

- Document-based

For example, CouchDB (ACID-compliant), MongoDB, TerraStore, and Lotus Notes (possibly the oldest) used in various financial and other relevant institutions: the US army, SAP, MTV, and SourceForge

- Object-based

For example, db4o, Versant, Objectivity, and NEO used by Siemens, China Telecom, and the European Space Agency.

- Graph-based

For example, Neo4J, InfiniteGraph, VertexDb, and FlocDb used by Twitter, Nortler, Ericson, Qualcomm, and Siemens.

- XML, multivalue, and others

In Table 4-1, we have a comparison of the afore mentioned data models:

Model	**Performance**	**Scalability**	**Flexibility**	**Complexity**	**Functionality**
key-value	high	high	high	low	depends
column	high	high	high	low	depends
document	high	high	high	low	depends
graph	depends	depends	high	high	graph theory
RDBMS	depends	depends	low	moderate	relational algebra

Table 4-1: Categorization and comparison NoSQL data model of Scofield and Popescu

NoSQL or SQL?

This is the wrong question. It would be better to ask: What do we need?

Basically, it all depends on the application's needs. Nothing is black and white. If consistency is essential, use RDBMS. If you need high-availability, fault tolerance, and scalability then use NoSQL. The recommendation is: in a new project, evaluate the best of each world.

It doesn't make sense to enforce NoSQL where it doesn't fit, because its benefits (scalability, read/write speed in entire orders of magnitude, soft data model) are only *conditioned* advantages achieved in a set of problems that can be solved, per se. It is necessary to carefully weigh, beyond marketing, what exactly is needed, what kind of strategy is needed, and how they will be applied to solve our problem. Consider using a NoSQL database only when you decide that this is a better solution than SQL.

The challenges for NoSQL databases are: elastic scaling, cost-effective, simple, and flexible.

In Table 4-2, we compare the two models:

NoSQL	**RDBMS**
Schema-less	Relational schema
Scalable read/write	Scalable read
Auto high availability	Custom high availability
Limited queries	Flexible queries
Eventual consistency	Consistency
BASE	ACID

Table 4-2: Comparison of NoSQL and RDBMS

CAP Brewer's theorem

In 2000, in Portland Oregon, the United States held the nineteenth international symposium on the principles of distributed computing where the keynote speaker, Eric Brewer, a professor at UC Berkeley gave a talk.

In his presentation, among other things, he said that there are three basic system requirements that have a special relationship when making the design and implementation of applications in a distributed environment, and that a distributed system can have a *maximum of two* of the three properties (which is the basis of his theorem). The three properties are:

- Consistency: This property says that the data on one node must be the same data when read from a second node, the second node must show exactly the same data (there can be a delay, if someone else in between is performing an update, but there can't be a difference).
- Availability: This property says that a failure on one node doesn't mean the loss of its data; the system must be able to display the requested data.
- Partition tolerance: This property says that in the event of a breakdown in communication between two nodes, the system should still work, meaning the data will still be available.

In Figure 4-1, we show CAP Brewer's theorem with some examples.

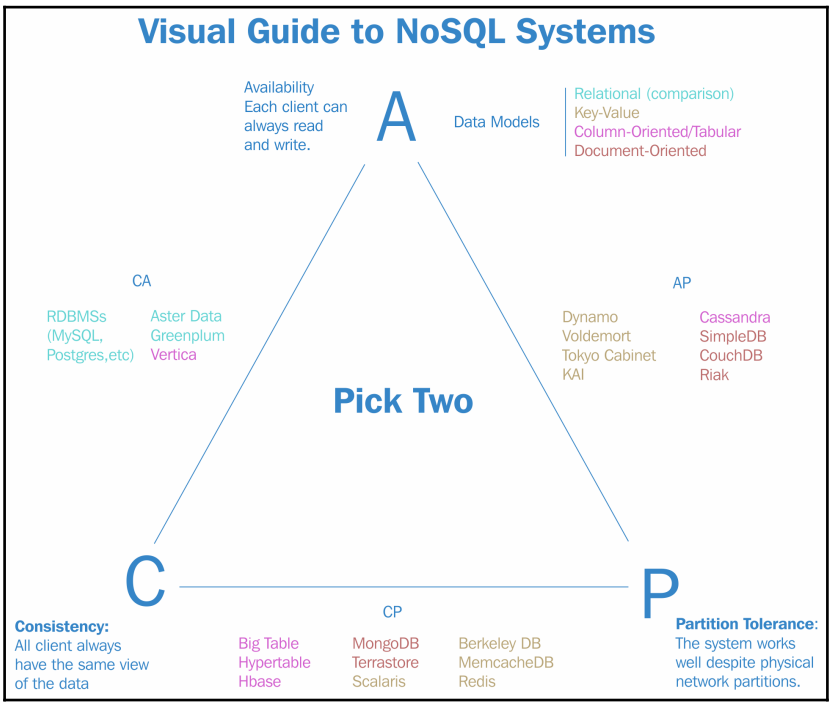

Figure 4-1 CAP Brewer's theorem

Apache Cassandra installation

In Facebook laboratories, although not visible to the public, new software is developed, for example, the junction between two concepts involving the development departments of Google and Amazon. In short, Cassandra is defined as a distributed database. From the start, the authors undertook the task of creating a scalable database massively decentralized, optimized for read operations when possible, painlessly modifying data structures, and , for all this, not difficult to manage. The solution was found by combining two existing technologies: Google's BigTable and Amazon's Dynamo. One of the two authors, A. Lakshman, had earlier worked on BigTable and he borrowed the data model layout, while Dynamo contributed with the overall distributed architecture.

Cassandra is written in Java and for good performance it requires the latest possible JDK version. In Cassandra 1.0, they used another open source project Thrift for client access, which also came from Facebook and is currently an Apache Software project. In Cassandra 2.0, Thrift was removed in favor of CQL. Initially, thrift was not made just for Cassandra, but it is a software library tool and code generator for accessing backend services.

Cassandra administration is done with the command-line tools or via the JMX console; the default installation allows us to use additional client tools. Since this is a server cluster, it has different administration rules and it is always good to review the documentation to take advantage of other people's experiences. Cassandra managed the very demanding tasks successfully. Often used on site, serving a huge number of users (such as Twitter, Digg, Facebook, and Cisco) that, relatively, often change their complex data models to meet the challenges that will come later, they usually do not have to deal with expensive hardware or licenses.

At the time of writing, the Cassandra homepage (`http://cassandra.apache.org`) says that Apple Inc. for example, has a 75,000 node cluster storing 10 Petabytes.

Data model

The storage model of Cassandra could be seen as a sorted HashMap of sorted HashMaps.

Cassandra is a database that stores rows in the form of key-value. In this model, the number of columns is not predefined in advance as in standard relational databases, but a single row can contain several columns. The column (*Figure 4-2*, *Column*) is the smallest atomic unit model. Each element in the column consists of a triplet: a name, a value (stored as a series of bytes without regard to the source type), and a timestamp (the time used to determine the most recent record).

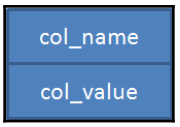

Figure4-2: Column

All data triplets are obtained from the client, and even a timestamp. Thus, the row consists of a key and a set of data triplets (Figure 4-3). Here is how the super column will look:

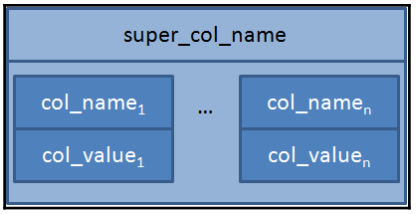

Figure 4-3: Super column

In addition, the columns can be grouped into so-called column families (*Figure 4-4, Column family*), which would be somehow equivalent to the table and can be indexed:

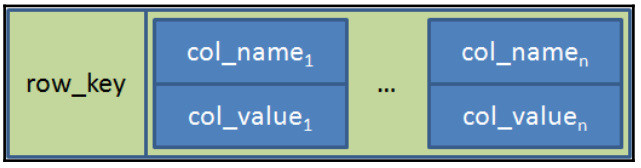

Figure 4-4: Column family

A higher logical unit is the super column family (as shown in the following *Figure 4-5, Super column family*), in which columns contain other columns:

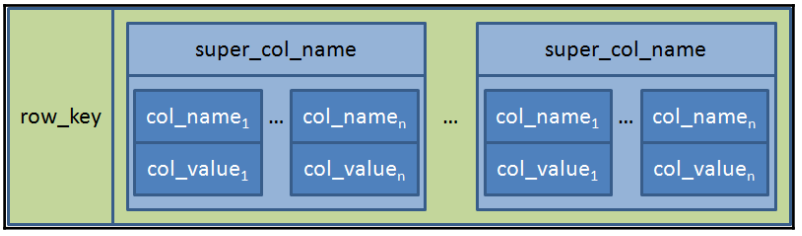

Figure 4-5: Super column family

Above all is the key space (as shown in *Figure 4-6, Cluster with Key Spaces*), which would be equivalent to a relational schema and is typically used by one application. The data model is simple, but at the same time very flexible and it takes some time to become accustomed to the new way of thinking while rejecting all of SQL's syntax luxury.

The replication factor is unique for each key space. Moreover, key spaces could span multiple clusters and have different replication factors for each of them. This is used in geo-distributed deployments.

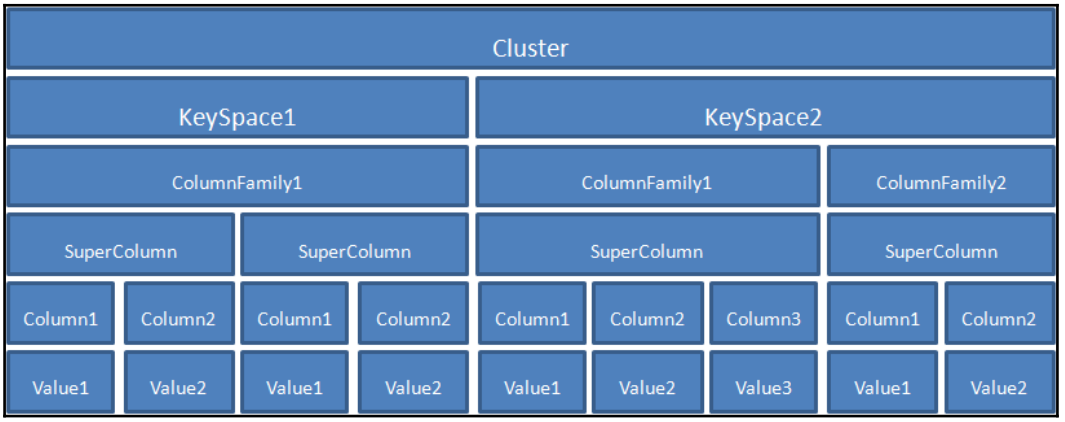

Figure 4-6: Cluster with key spaces

Data storage

Apache Cassandra is designed to process large amounts of data in a short time; this way of storing data is taken from her big brother, Google's Bigtable.

Cassandra has a commit log file in which all new data is recorded in order to ensure their sustainability. When data is successfully written on the commit log file, the recording of the freshest data is stored in a memory structure called memtable (Cassandra considers it a writing failure if the same information is in the commit log and in memtable). Data within memtables is sorted by Row key.

When memtable is full, its contents are copied to the hard drive in a structure called **Sorted String Table** (**SSTable**). The process of copying content from memtable into SSTable is called **flush**. Data flush is performed periodically, although it could be carried out manually (for example, before restarting a node) through node tool flush commands.

The SSTable provides a fixed, sorted map of row and value keys. Data entered in one SSTable cannot be changed, but it is possible to enter new data. The internal structure of SSTable consists of a series of blocks of 64Kb (the block size can be changed); internally a SSTable is a block index used to locate blocks.

One data row is usually stored within several SSTables so reading a single data row is performed in the background combining SSTables and the memtable (which have not yet performed a flush). In order to optimize the process of connecting, Cassandra uses a memory structure called a `Bloom` filter. Every SSTable has a bloom filter that checks if the requested row key is in the SSTable before look up in the disk.

In order to reduce row fragmentation through several SSTables, in the background Cassandra performs another process: the *compaction*, merging several SSTables into a single SSTable. Fragmented data is combined based on the values of a row key. After creating a new SSTable, the old SSTable is labeled as outdated and marked in the garbage collector process for deletion. Compaction has different strategies, size-tiered compaction and leveled compaction, and both have their own benefits for different scenarios.

Installation

To install Cassandra, go to `http://www.planetcassandra.org/cassandra/`.

Installation is simple. After downloading the compressed files, extract them and change a couple of settings in the configuration files (set the new directory path). Run the startup scripts to activate a single node, and the database server. Of course, it is possible to use Cassandra in only one node, but we lose its main power, distribution. The process of adding new servers to the cluster is called a **bootstrap** and is generally not a difficult operation.

Once all the servers are active, they form a ring of nodes, none of which is central, meaning without a main server. Within the ring, the information propagation on all servers is performed through a gossip protocol. In short, one node transmits information about the new instances to only some of their known colleagues, and if one of them already knows from other sources about the new node, the first node propagation is stopped. Thus, the information about the node is propagated in an efficient and rapid way through the network.

It is necessary for a new node activation to seed its information to at least one existing server in the cluster so the gossip protocol works. The server receives its numeric identifier, and each of the ring nodes stores its data. Which nodes store the information depends on the hash MD5 key-value (a combination of key-values) as shown in *Figure 4-7, Nodes within a cluster*.

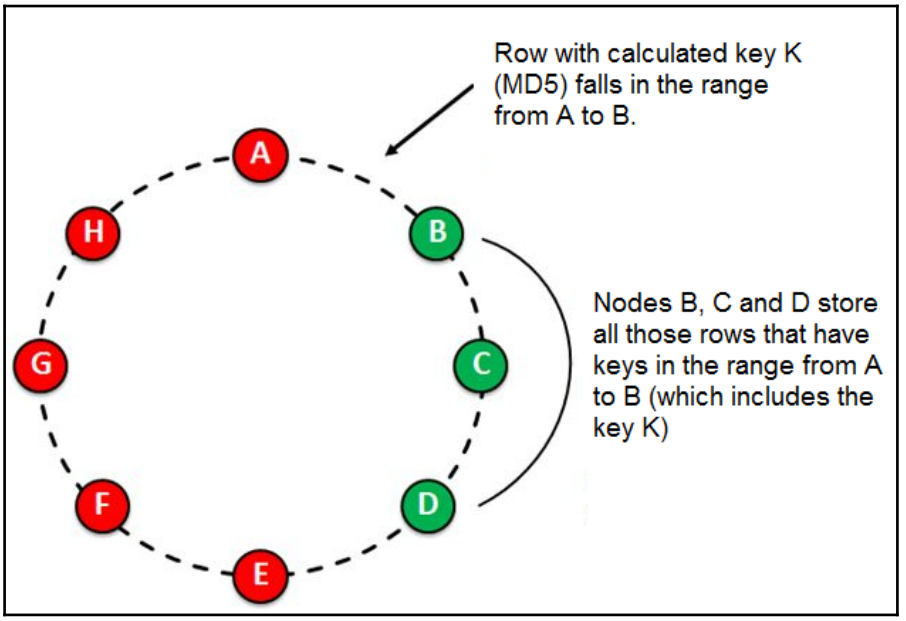

Figure 4-7: Nodes within a cluster

The nodes are in a circular stack, that is, a ring, and each record is stored on multiple nodes. In the case of failure of one of them, the data is still available. Nodes are occupied according to their identifier integer range, that is, if the calculated value falls into a node range, then the data is saved there. Saving is not performed on only one node, more is better; an operation is considered a success if the data is correctly stored on as many nodes as possible. All this is parameterized. In this way, Cassandra achieves sufficient data consistency and provides greater robustness of the entire system; if one node in the ring fails, is always possible to retrieve valid information from the other nodes. In the event that a node comes back online again, it is necessary to synchronize the data on it, which is achieved through a reading operation.

The data is read from all the ring servers, a node saves just the data accepted as valid, that is, the most recent data, the data comparison is made according to the timestamp records. The nodes that don't have the latest information, refresh their data in a low-priority back-end process.

Although this brief description of the architecture makes it sound like it is full of holes, in reality everything works flawlessly. Indeed, more servers in the game implies a better general situation.

DataStax OpsCenter

In this section, we perform the Cassandra installation on a computer with a Windows operating system (to prove that nobody is excluded).

Installing software under the Apache open license can be complicated on a Windows computer, especially if it is new software, such as Cassandra. To make things simpler we will use a distribution package for easy installation, to start up and work with Cassandra on a Windows computer. The distribution used in this example is called DataStax Community Edition. DataStax contains Apache Cassandra, along with the **Cassandra Query Language** (**CQL**) tool and the free version of DataStax OpsCenter for management and monitoring the Cassandra cluster. We can say that OpsCenter is a kind of DBMS for NoSQL databases.

After downloading the installer from the DataStax's official site, the installation process is quite simple, just keep in mind that DataStax supports Windows 7 and Windows Server 2008 and that DataStax used on a Windows computer must have the Chrome or Firefox web browser (Internet Explorer is not supported).

When starting DataStax on a Windows computer, DataStax will open as in *Figure 4-8, DataStax OpsCenter*.

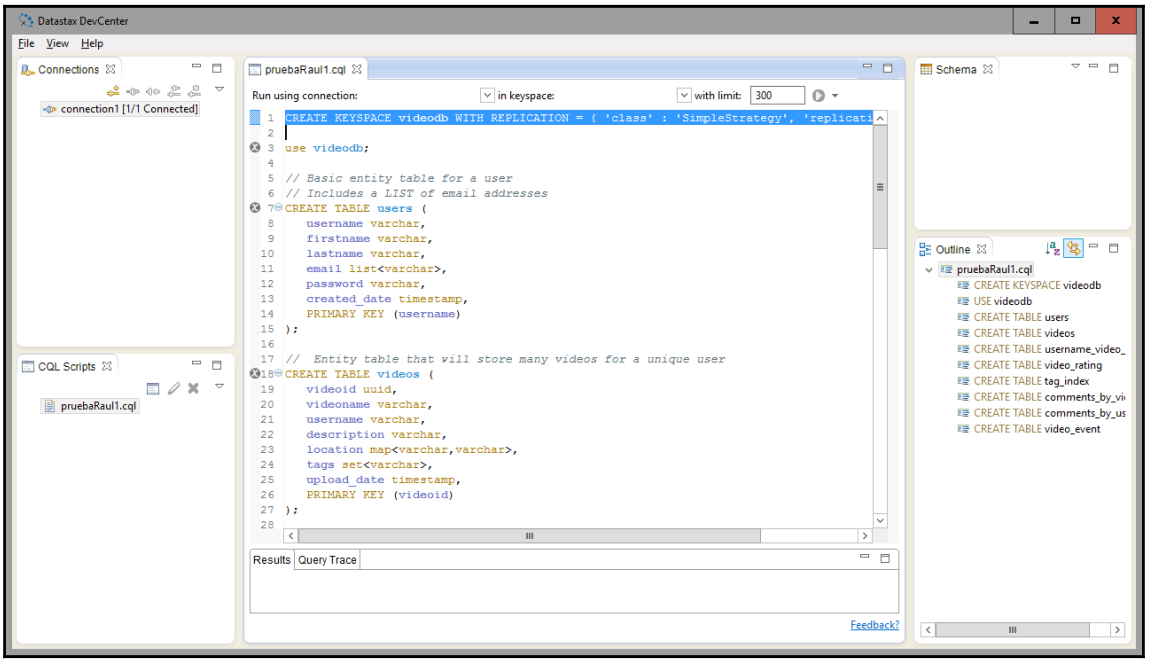

Figure 4-8: DataStax OpsCenter

DataStax consists of a control panel (dashboard), in which we review the events, performance, and capacity of the cluster and also see how many nodes belong to our cluster (in this case a single node). In cluster control, we can see the different types of views (ring, physical, list). Adding a new key space (the equivalent to creating a database in the classic DBMS) is done through the CQLShell using CQL or using the DataStax data modeling. Also, using the data explorer we can view the column family and the database.

Creating a key space

The main tool for managing Cassandra CQL runs in a console interface and this tool is used to add new key spaces from which we will create a column family. A key space is created as follows:

```
    cqlsh> create keyspace hr with strategy_class='SimpleStrategy' and
strategy_options:replication_factor=1;
```

After opening CQL Shell, the command `create keyspace` will make a new key space, the `strategy_class = 'SimpleStrategy'` parameter invokes the class replication strategy used when creating new key spaces. Optionally,
the `strategy_options:replication_factor = 1` command creates a copy of each row in each cluster node, and the value `replication_factor` being set to `1` produces only one copy of each row on each node (if we set it to 2, we will have two copies of each row on each node).

```
cqlsh> use hr;
cqlsh:hr> create columnfamily employee (sid int primary key,
    ... name varchar,
    ... last_name varchar);
```

There are two types of key space, `SimpleStrategy` and `NetworkTopologyStrategy`, whose syntax is as follows:

```
{ 'class' : 'SimpleStrategy', 'replication_factor' : <integer> };

{ 'class' : 'NetworkTopologyStrategy'[, '<data center>' : <integer>, '<data
center>' : <integer>] . . . };
```

When `NetworkTopologyStrategy` is configured as the replication strategy, we set up one or more virtual data centers.

To create a new column family, we use the create command; select the desired Key Space, and with the command `create columnfamily`, for example, we create a new table in which we define the id an integer as a primary key and other attributes such as name and last name.

To make a data entry in a column family, we use the insert command:

```
    insert into <table name> (<attribute_1>, < attribute_2> ... <
attribute_n>);
```

When filling data tables we use the common SQL syntax:

```
cqlsh:hr> insert into employee (sid, name, lastname)  values (1, 'Raul',
'Estrada');
```

So we enter data values. With the `select` command we can review our insert:

```
cqlsh:hr> select * from employee;
sid | name | last_name
----+------+------------
 1  | Raul | Estrada
```

Authentication and authorization (roles)

In Cassandra, the authentication and authorization must be configured on the `cassandra.yaml` file and two additional files. The first file is used to assign rights to users over the key space and column family, while the second assigns passwords to users. These files are called `access.properties` and `passwd.properties`, and are located in the Cassandra installation directory. These files can be opened using our favorite text editor in order to be successfully configured.

Setting up a simple authentication and authorization

Perform the following steps:

1. In the `access.properites` file we add the access rights to users and the permissions to read and write certain key spaces and `columnfamily`. Syntax:

```
keyspace.columnfamily.permits = users

Example 1:
hr <rw> = restrada

Example 2:
hr.cars <ro> = restrada, raparicio
```

- In example 1, we give full rights in the Key Space `hr` to `restrada` while in example 2 we give read-only rights to users to the column family `cars`.

2. In the `passwd.properties` file, user names are matched to passwords; on the left side of the equal sign we write the username and on the right side the password:

Example:

```
restrada = Swordfish01
```

3. After we change the files, before restarting Cassandra it is necessary to type the following command in the terminal in order to reflect the changes in the database:

```
$ cd <installation_directory>
$ sh bin/cassandra –f –Dpasswd.properties =
 conf/passwd.properties
–Daccess.properties = conf/access.properties
```

Note: The third step in setting up authentication and authorization doesn't work on Windows computers and is just needed on Linux distributions. Also, note that user authentication and authorization should not be solved through Cassandra; for safety reasons, in the latest Cassandra versions this function is not included.

Backup

The purpose of making Cassandra a NoSQL database is because when we create a single node, we make a copy of it. Copying the database to other nodes and the exact number of copies depend on the replication factor established when we create a new key space.

But as with any other standard SQL database, Cassandra offers to create a backup on the local computer. Cassandra creates a copy of the base using a snapshot. It is possible to make a snapshot of all the key spaces, or just one column family. It is also possible to make a snapshot of the entire cluster using the **parallel SSH tool** (**pssh**).

If the user decides to snapshot the entire cluster, it can be reinitiated and uses an incremental backup on each node.

Incremental backups provide a way to get each node configured separately, through setting the `incremental_backups` flag to true in `cassandra.yaml`.

When incremental backups are enabled, Cassandra hard-links each flushed SSTable to a backups directory under the key space data directory. This allows storing backups offsite without transferring entire snapshots.

To snapshot a key space we use the `nodetool` command:

Syntax:

```
nodetool snapshot –cf <ColumnFamily> <keypace> –t <snapshot_name>
```

Example:

```
nodetool snapshot –cf cars hr snapshot1
```

The snapshot is stored in the Cassandra installation directory:

```
C:\Program Files\DataStax Community\data\data\en\example\snapshots
```

Compression

Compression increases the cluster nodes capacity, reducing the data size on the disk. With this function, compression also enhances the server's disk performance.

Compression in Cassandra works better when compressing a column family with a lot of columns, when each row has the same columns, or when we have a lot of common columns with the same data. A good example of this is a column family that contains user information such as user name and password because it is possible that they have the same data repeated. The greater the amount of identical data extended through the rows, the higher the compression ratio.

Column family compression is made with the Cassandra-CLI tool. It is possible to update existing columns families or create a new column family with specific compression conditions, for example, the compression shown here:

```
CREATE COLUMN FAMILY users WITH comparator = 'UTF8Type'
AND key_validation_class = 'UTF8Type'
AND column_metadata = [
(column_name: name, validation_class: UTF8Type)
(column_name: email, validation_class: UTF8Type)
(column_name: country, validation_class: UTF8Type)
(column_name: birth_date, validation_class: LongType)
]
AND compression_options=(sstable_compression:SnappyCompressor,
chunk_length_kb:64);
```

We will see this output:

```
Waiting for schema agreement....
... schemas agree across the cluster
```

After opening the Cassandra-CLI, we need to choose the key space where the new column family is. When creating a column family, it is necessary to state that the comparator (UTF8 type) and `key_validation_class` are of the same type. With this we will ensure that when executing the command we won't have an exception (generated by a bug). After printing the column names, we set `compression_options` which has two possible classes: `SnappyCompresor`, which provides faster data compression or `DeflateCompresor` which provides a higher compression ratio. The `chunk_length` adjusts compression size in kilobytes.

Recovery

Recovering a key space snapshot requests all the snapshots made for a certain column family. If you use an incremental backup, it is also necessary to provide the incremental backups created after the snapshot. There are multiple ways to perform a recovery from the snapshot. We can use the SSTable loader tool (used exclusively on the Linux distribution) or can recreate the installation method.

Restart node

If the recovery is running on one node, we must first shutdown the node. If the recovery is for the entire cluster, it is necessary to restart each node in the cluster. Here is the procedure:

1. Shut down the node.
2. Delete all the log files in: `C:\Program Files\DataStax Community\logs`.
3. Delete all `.db` files within a specified key space and column family: `C:\Program Files\DataStax Community\data\data\en\cars`.
4. Locate all Snapshots related to the column family: `C:\Program Files\DataStax Community\data\data\en\cars\snapshots\1,351,279,613,842,`.
5. Copy them to: `C:\Program Files\DataStax Community\data\data\en\cars`.
6. Re-start the node.

Printing schema

Through DataStax OpsCenter or Apache Cassandra CLI we can obtain the schemes (Key Spaces) with the associated column families, but there is no way to make a data export or print it.

Apache Cassandra is not RDBMS and it is not possible to obtain a relational model scheme from the key space database.

Logs

Apache Cassandra and DataStax OpsCenter both use the Apache log4j logging service API. In the directory where DataStax is installed, under Apache-Cassandra and opsCenter is the `conf` directory where the file `log4j-server.properties` is located: `log4j-tools.properties` for apache-cassandra and `log4j.properties` for OpsCenter.

The parameters of the `log4j` file can be modified using a text editor. Log files are stored in plain text in the `...\DataStax Community\logs\` directory; here it is possible to change the directory location to store the log files.

Configuring log4j

log4j configuration files are divided into several parts where all the parameters are set to specify how collected data is processed and written in the log files.

For `RootLoger`:

```
# RootLoger level
log4j.rootLogger = INFO, stdout, R
```

This section defines the data level, respectively, to for all the events recorded in the log file.

As we can see in *Table 4-3*, the possible log levels are:

Level	Record
ALL	The lowest level; all the events are recorded in the log file
DEBUG	Detailed information about events
ERROR	Information about runtime errors or unexpected events
FATAL	Critical error information

INFO	Information about the state of the system
OFF	The highest level; the log file record is off
TRACE	Detailed debug information
WARN	Information about potential adverse events (unwanted/unexpected runtime errors)

Table 4-3 Log4J Log level

For Standard out `stdout`:

```
# stdout
log4j.appender.stdout = org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout = org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=
%5p %d{HH:mm:ss,SSS} %m%n
```

Through the `StandardOutputWriter` class, we define the appearance of the data in the log file. The `ConsoleAppender` class is used for entry data in the log file, and the `ConversionPattern` class defines the data appearance written into a log file. In the diagram, we can see how the data looks stored in a log file, which is defined by the previous configuration.

Log file rotation

In this example, we rotate the log when it reaches 20 Mb and we retain just 50 log files.

```
# rolling log file
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.maxFileSize=20MB
log4j.appender.R.maxBackupIndex=50
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%5p [%t] %d{ISO8601} %F (line %L)
%m%n
```

This part sets the log files. The `RollingFileAppender` class inherits from `FileAppender`, and its role is to make a log file backup when it reaches a given size (in this case 20 MB). The `RollingFileAppender` class has several methods; these two are the most used:

```
public void setMaxFileSize( String value )
```

- Defines the file size and can take a value from 0 to 263 using the abbreviations KB, MB, GB. The integer value is automatically converted (in the example, the file size is limited to 20 MB):

```
public void setMaxBackupIndex( int maxBackups )
```

- Defines how the backup file is stored before the oldest log file is deleted (in this case we retain 50 log files)

To set the parameters of the location where the log files will be stored, use:

```
# Edit the next line to point to your logs directory
log4j.appender.R.File=C:/Program\ Files\ \(x86\)/DataStax\
Community/logs/cassandra.log
```

User activity log

The log4j API has the ability to store user activity logs. In production, it is not recommended to use the DEBUG or TRACE log level.

Transaction log

As mentioned earlier, any new data is stored in the commit log file. Within the `cassandra.yaml` configuration file, we can set the location where the commit log files will be stored:

```
# commit log
commitlog_directory: "C:/Program Files (x86)/DataStax
Community/data/commitlog"
```

SQL dump

It is not possible to make a database SQL dump, only a snapshot of the DB.

CQL

CQL is a language like SQL. CQL means Cassandra Query Language. With this language we make queries on a Key Space. There are several ways to interact with a Key Space; in the previous section we showed how to do it using a shell called CQL shell.

Since CQL is the first way to interact with Cassandra, in *Table 4-4, Shell Command Summary*, we see the main commands that can be used on the CQL Shell:

Command	Description
`Cqlsh`	Captures command output and appends it to a file.
`CAPTURE`	Shows the current consistency level or, given a level, sets it.
`CONSISTENCY`	Imports and exports CSV (comma-separated values) data to and from Cassandra.
`COPY`	Provides information about the connected Cassandra cluster, or about the data objects stored in the cluster.
`DESCRIBE`	Formats the output of a query vertically.
`EXPAND`	Terminates `cqlsh`.
`EXIT`	Enables or disables query paging.
`PAGING`	Shows the Cassandra version, host, or tracing information for the current `cqlsh` client session.
`SHOW`	Executes a file containing CQL statements.
`SOURCE`	Enables or disables request tracing.
`TRACING`	Captures command output and appends it to a file.

Table 4-4. Shell command summary

For more detailed information on shell commands, visit:

`http://docs.datastax.com/en/cql/3.1/cql/cql_reference/cqlshCommandsTOC.html`

CQL commands

CQL is very similar to SQL as we have already seen in this chapter. *Table 4-5, CQL Command Summary* lists the language's commands.

CQL, like SQL, is based on sentences/statements. These sentences are for data manipulation and work with their logical container, the key space.

As with SQL statements, they must end with a semicolon (;)

Command	Description
`ALTER KEYSPACE`	Change property values of a key space.
`ALTER TABLE`	Modify the column metadata of a table.
`ALTER TYPE`	Modify a user-defined type. Cassandra 2.1 and later.
`ALTER USER`	Alter existing user options.
`BATCH`	Write multiple DML statements.
`CREATE INDEX`	Define a new index on a single column of a table.
`CREATE KEYSPACE`	Define a new key space and its replica placement strategy.
`CREATE TABLE`	Define a new table.
`CREATE TRIGGER`	Register a trigger on a table.
`CREATE TYPE`	Create a user-defined type. Cassandra 2.1 and later.
`CREATE USER`	Create a new user.
`DELETE`	Remove entire rows or one or more columns from one or more rows.
`DESCRIBE`	Provide information about the connected Cassandra cluster, or about the data objects stored in the cluster.
`DROP INDEX`	Drop the named index.
`DROP KEYSPACE`	Remove the key space.
`DROP TABLE`	Remove the named table.
`DROP TRIGGER`	Removes registration of a trigger.
`DROP TYPE`	Drop a user-defined type. Cassandra 2.1 and later.
`DROP USER`	Remove a user.
`GRANT`	Provide access to database objects.

`INSERT`	Add or update columns.
`LIST PERMISSIONS`	List permissions granted to a user.
`LIST USERS`	List existing users and their superuser status.
`REVOKE`	Revoke user permissions.
`SELECT`	Retrieve data from a Cassandra table.
`TRUNCATE`	Remove all data from a table.
`UPDATE`	Update columns in a row.
`USE`	Connect the client session to a keyspace.

Table 4-5. CQL command summary

For more detailed information on CQL commands visit:

`http://docs.datastax.com/en/cql/3.1/cql/cql_reference/cqlCommandsTOC.html`

DBMS Cluster

The basic concept of Cassandra is a database working in a cluster, that is databases on multiple nodes. Although primarily intended for Cassandra Linux distributions building clusters on Linux servers, Cassandra offers the possibility to build clusters on Windows computers.

The first task that must be done prior to setting up a cluster on Windows computers is opening the firewall for Cassandra *DBMS* DataStax OpsCenter. Ports that must be open for Cassandra are `7000` and `9160`. For OpsCenter, the ports are `7199`, `8888`, `61620`, and `61621`. These ports are the default when we install Cassandra and OpsCenter; however, we can specify new ports.

Immediately after installing Cassandra and OpsCenter on a Windows computer, it is necessary to stop the DataStax OpsCenter service, the DataStax OpsCenter agent, as in *Figure 4-9, Microsoft Windows display services*.

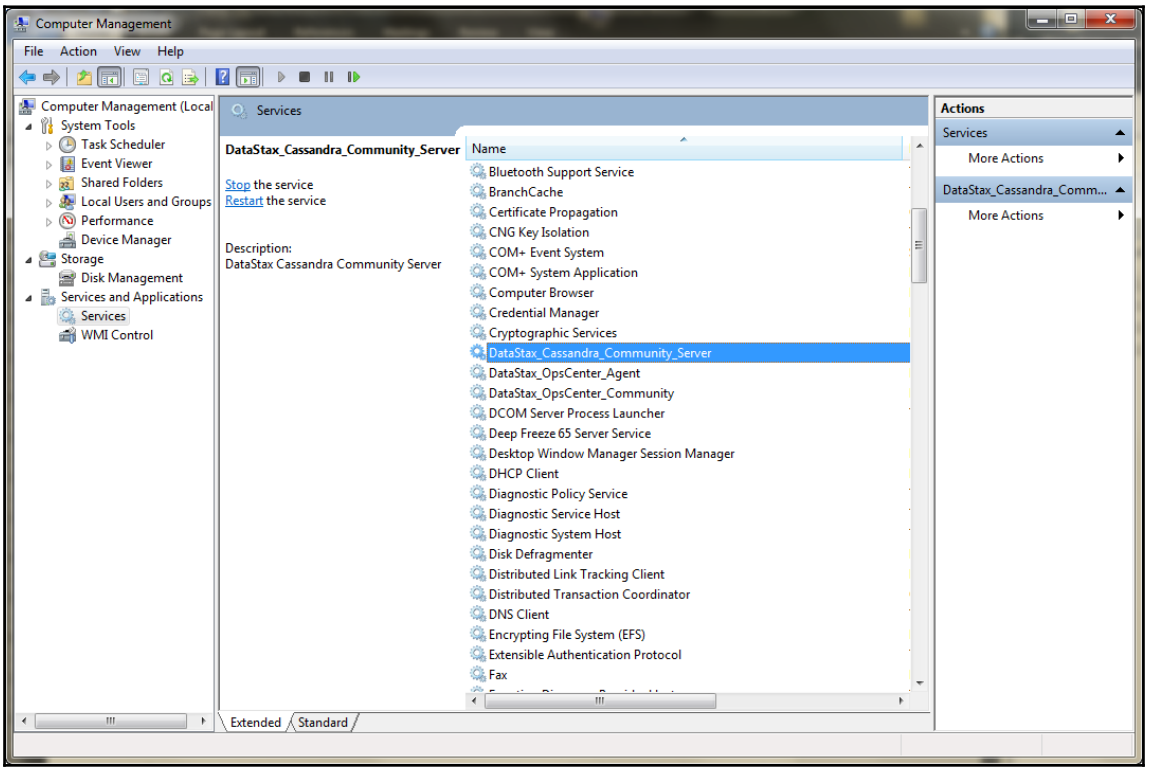

Figure 4-9: Microsoft Windows display services

One of Cassandra's advantages is that it automatically distributes data in the computers of the cluster using the algorithm for the incoming data. To successfully perform this, it is necessary to assign tokens to each computer in the cluster. The token is a numeric identifier that indicates the computer's position in the cluster and the data scope in the cluster responsible for that computer. For successful token generation Python can be used; it that comes within the Cassandra installation located in DataStax's installation directory. In the code for generating tokens, the variable `num = 2` refers to the number of computers in the cluster:

```
$ python -c "num=2; print ""\n"".join([(""token %d: %d""
%(i,(i*(2**127)/num))) for i in range(0,num)])"
```

We will see an output like this:

```
token 0: 0
token 1: 88743298547982745894789547895490438209
```

It is necessary to preserve the value of the tokens because they will be required in the following steps. We now need to configure the `cassandra.yaml` file, which we have already met in the authentication and authorization section. The `cassandra.yaml` file must be configured separately on each computer in the cluster. After opening the file, you need to make the following changes:

```
Initial_token
```

On each computer in the cluster, copy the tokens generated. It should start from the token 0 and assign each computer a unique token.

```
Listen_adress
```

In this section, we will enter the IP of the computer used.

```
Seeds
```

You need to enter the IP address of the primary (main) node in the cluster.

Once the file is modified and saved, you must restart DataStax Community Server as we already saw. This should be done only on the primary node. After that it is possible to check if the cluster nodes have communication using the node tool. In can be used in node tool, enter the following command:

```
nodetool –h localhost ring
```

If the cluster works, we will see the following result:

```
Address  DC  Rack  Status  State  Lead  Owns  Token
-    datacenter1  rack1  Up  Normal  13.41 Kb  50.0%
88743298547982745894789547895490438209
-    datacenter1  rack1  Up  Normal  6.68 Kb  50.0%
88743298547982745894789547895490438209
```

If the cluster is operating normally, select which computer will be the primary OpsCenter (this may not be the primary node). Then on that computer open `opscenter.conf` which can be found in the DataStax's installation directory. In that directory, you need to find the webserver interface section and set the parameter to the value `0.0.0.0`. After that, in the agent section, change the `incoming_interface` parameter to your computer IP address.

In DataStax's installation directory (on each computer in the cluster) we must configure the `address.yaml` file. Within these files, set the `stomp_interface local_interface` parameters and link it to the IP address of the computer where the file is configured.

Now the primary computer should run the DataStax OpsCenter Community and DataStax OpsCenter agent services. After that, run computers the DataStax OpsCenter agent service on all the nodes.

At this point it is possible to open DataStax OpsCenter with an Internet browser and OpsCenter should look like *Figure 4-10, Display cluster in OpsCenter*.

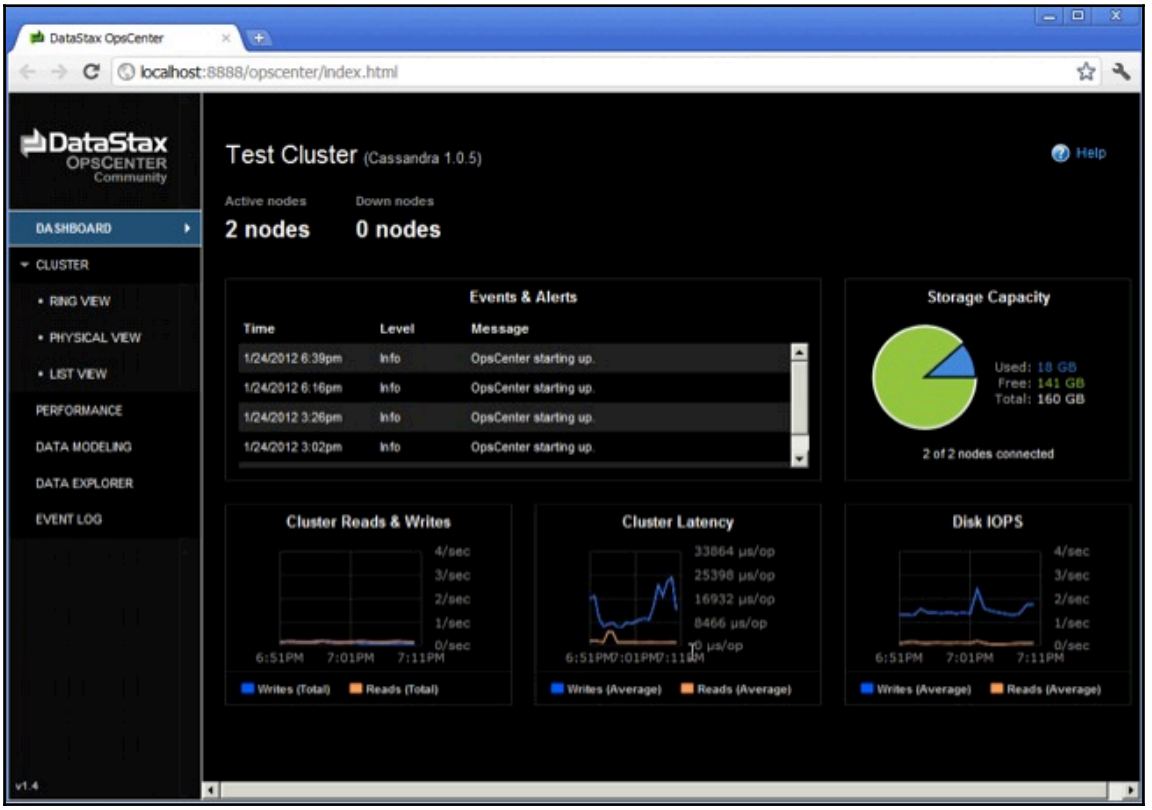

Figure 4-10: Display cluster in OpsCenter

Deleting the database

In Apache Cassandra, there are several ways to delete the database (key space) or parts of the database (column family, individual rows within the family row, and so on).

Although the easiest way to make a deletion is using the DataStax OpsCenter data modeling tool, there are commands that can be executed through the Cassandra-CLI or the CQL shell.

CLI delete commands

In Table 4-6, we have the CLI delete commands:

CLI Command	Function
`part`	Used to delete a great column, a column from the column family, or rows within certain columns
`drop columnfamily`	Delete a column family and all data contained on it
`drop keyspace`	Delete the key space, all the column families, and the data contained on them.
`truncate`	Delete all the data from the selected column family

Table 4-6 CLI delete commands

CQL shell delete commands

In Table 4-7, we have the shell delete commands:

CQL shell command	Function
`alter_drop`	Delete a specified column from the column family
`delete`	Delete one or more columns from one or more rows of the selected column family
`delete_columns`	Delete columns from the column family
`delete_where`	Delete individual rows
`drop_table`	Delete the selected column family and all the data contained on it
`drop_columnfamily`	Delete a column family and all the data contained on it
`drop_keyspace`	Delete the key space, all the column families, and all the data contained on them.

`truncate`	Delete all data from the selected column family.

Table 4-7 CQL Shell delete commands

DB and DBMS optimization

Cassandra optimization is specified in the `cassandra.yaml` file and these properties are used to adjust the performance, and specify the use, of system resources such as disk I/O, memory, and CPU usage.

- `column_index_size_in_kb`:
    - Initial value: 64 Kb
    - Range of values: -
    - Column indices added to each row after the data reaches the default size of 64 Kilobytes
- `commitlog_segment_size_in_mb`
    - Initial value: 32 Mb
    - Range of values: 8-1,024 Mb
    - Determines the size of the commit log segment. The commit log segment is archived to be obliterated or recycled after being transferred to the SRM table.
- `commitlog_sync`
    - Initial value: -
    - Range of values: -
    - In Cassandra, this method is used for entry reception. This method is closely correlated with `commitlog_sync_period_in_ms`, which controls how often the log is synchronized with the disc.
- `commitlog_sync_period_in_ms`
    - Initial value: 1000 ms
    - Range of values: -
    - Decides how often to send the commit log to disk when `commit_sync` is in periodic mode.

- `commitlog_total_space_in_mb`
    - Initial value: 4096 MB
    - Range of values: -
    - When the size of the commit log reaches an initial value, Cassandra removes the oldest parts of the commit log. This reduces the data amount and facilitates the launch of fixtures.
- `compaction_preheat_key_cache`
    - Initial value: true
    - Range of values: true / false
    - When this value is set to true, the stored key rows are monitored during compression, and later resaved to a new location in the compressed SSTable
- `compaction_throughput_mb_per_sec`
    - Initial value: 16
    - Range of values: 0-32
    - Compression damping the overall bandwidth throughout the system. Faster data insertion means faster compression
- `concurrent_compactors`
    - Initial value: 1 per CPU core
    - Range of values: depends on the number of CPU cores
    - Adjusts the number of simultaneous compression processes on the node.
- `concurrent_reads`
    - Initial value: 32
    - Range of values: -
    - When there is more data than the memory can accommodate, a bottleneck occurs in reading data from disk
- `concurrent_writes`
    - Initial value: 32
    - Range of values: -
    - Making inserts in Cassandra does not depend on I/O limitations. Concurrent inserts depend on the number of CPU cores. The recommended number of cores is 8.

- `flush_largest_memtables_at`
    - Initial value: 0.75
    - Range of values: -
    - This parameter clears the biggest memtable to free disk space. This parameter can be used as an emergency measure to prevent memory loss (out of memory errors)
- `in_memory_compaction_limit_in_mb`
    - Initial value: 64
    - Range of values:
    - Limit order size on the memory. Larger orders use a slower compression method.
- `index_interval`
    - Initial value: 128
    - Value range: 128-512
    - Controlled sampling records from the first row of the index in the ratio of space and time, that is, the larger the time interval to be sampled, the less effective. In technical terms, the interval corresponds to the number of index samples skipped between taking each sample.
- `memtable_flush_queue_size`
    - Initial value: 4
    - Range of values: a minimum set of the maximum number of secondary indexes that make more than one Column family
    - Indicates the total number of full-memtable to allow a flush, that is, waiting on the write thread.
- `memtable_flush_writers`
    - Initial value: 1 (according to the data map)
    - Range of values: -
    - Number of memtable flush writer threads. These threads are blocked by the disk I/O, and each thread holds a memtable in memory until it is blocked.
- `memtable_total_space_in_mb`
    - Initial value: 1/3 Java Heap
    - Range of values: -
    - Total amount of memory used for all the Column family memtables on the node
- `multithreaded_compaction`

    - Initial value: false
    - Range of values: true/false
    - Useful only on nodes using solid state disks
- `reduce_cache_capacity_to`
    - Initial value: 0.6
    - Range of values: -
    - Used in combination with `reduce_cache_capacity_at`. When the Java Heap reaches the value of `reduce_cache_size_at`, this value is the total cache size by which to reduce the percentage to the declared value (in this case the size of the cache is reduced to 60%). Used to avoid unexpected out-of-memory errors.
- `reduce_cache_size_at`
    - Initial value: 0.85
    - Range of values: 1.0 (disabled)
    - When Java Heap marked to full sweep by the garbage Collector reaches a percentage stated on this variable (85%), Cassandra reduces the size of the cache to the value of the variable `reduce_cache_capacity_to`
- `stream_throughput_outbound_megabits_per_sec`
    - Initial value: off, that is, 400 Mbps (50 Mb/s)
    - Range of values: -
    - Regulates the stream of output file transfer in a node to a given throughput in Mbps. This is necessary because Cassandra mainly does sequential I/O when it streams data during system startup or repair, which can lead to network saturation and affects Remote Procedure Call performance.

Bloom filter

Every SSTable has a **Bloom filter**. In data requests, the Bloom filter checks whether the requested order exists in the SSTable before any disk I/O. If the value of the Bloom filter is too low, it may cause seizures of large amounts of memory, respectively; a higher Bloom filter value means less memory use. The Bloom filter range of values is from 0.000744 to 1.0. It is recommended you keep the minimum value of the Bloom filter less than 0.1.

The value of the Bloom filter column family is adjusted through the CQL shell as follows:

```
ALTER TABLE <column_family> WITH bloom_filter_fp_chance = 0.01;
```

Data cache

Apache Cassandra has two caches by which it achieves highly efficient data caching. These are:

- Cache key (default: enabled): cache index primary key columns families
- Row cache (default: disabled): holding a row in memory so that reading can be done without using the disc

If the key and row cache are set, querying data is accomplished as shown in *Figure 4-11, Apache Cassandra Cache*.

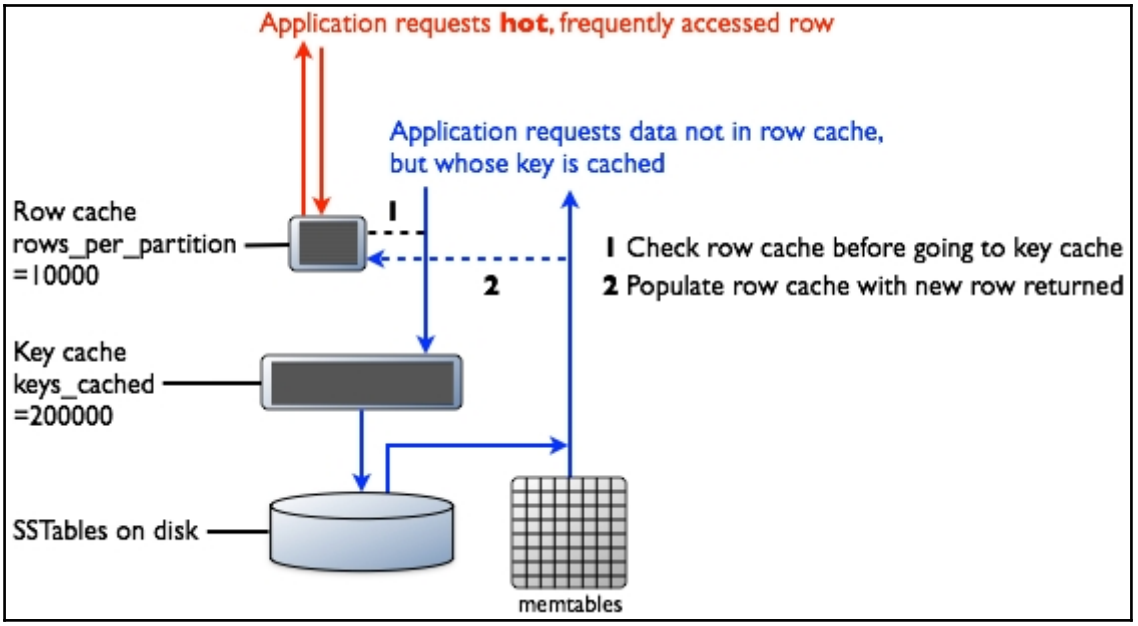

Figure 4-11: Apache Cassandra cache

When information is requested, first it checks in the row cache; if the information is available, then row cache returns the result without reading from the disk.

If it has come from a request and the row cache can return a result, it checks if the data can be retrieved through the key cache, which is more efficient than reading from the disk; the retrieved data is finally written to the row cache.

As the key cache memory stores the key location of an individual column family, any increase in the key cache has a positive impact on reading data for the column family. If the situation permits, a combination of key cache and row cache increases efficiency.

It is recommended that the size of the key cache is set in relation to the size of the Java heap.

Row cache is used in situations where data access patterns follow a normal (Gaussian) distribution of rows, containing often-read data and queries that often return data from most or all of the columns.

Within `cassandra.yaml` files, we have the following options to configure the data cache:

- `key_cache_size_in_mb`
    - Initial value: empty, meaning *Auto* (min (5% Heap (in MB), 100MB))
    - Range of values: blank or 0 (disabled key cache)
    - Variable that defines the key cache size per node
- `row_cache_size_in_mb`
    - Initial value: 0 (disabled)
    - Range of values: -
    - Variable that defines the row cache size per node
- `key_cache_save_period`
    - Initial value: 14400 (i.e. 4 hours)
    - Range of values: -
    - Variable that defines the save frequency of the key cache to disk
- `row_cache_save_period`
    - Initial value: 0 (disabled)
    - Range of values: -
    - Variable that defines the save frequency of the row cache to disk
- `row_cache_provider`
    - Initial value: `SerializingCacheProvider`
    - Range of values: `ConcurrentLinkedHashCacheProvider` or `SerializingCacheProvider`
    - Variable that defines the implementation of the row cache

Java heap tune up

Apache Cassandra interacts with the operating system using the Java virtual machine, so the Java heap size plays an important role. When starting Cassandra, the size of the Java Heap is set automatically based on the total amount of RAM (*Table 4-8, Determination of the Java heap relative to the amount of RAM*). The Java heap size can be manually adjusted by changing the values of the following variables contained on the file `cassandra-env.sh` located in the directory `...\apache-cassandra\conf\`.

```
# MAX_HEAP_SIZE = "4G"
# HEAP_NEWSIZE = "800M"
```

Total system memory	Java heap size
< 2 Gb	Half of the system memory
2 Gb - 4 Gb	1 Gb
> 4 Gb	One quarter of the system memory, no more than 8 Gb

Table 4-8: Determination of the Java heap relative to the amount of RAM

Java garbage collection tune up

Apache Cassandra has a GC Inspector responsible for collecting information on each garbage collection process longer than 200ms. Garbage Collection processes that occur frequently and take a lot of time (as concurrent mark-sweep, which takes several seconds) indicate that there is a great pressure on garbage collection and in the JVM. Recommendations to address these issues include:

- Add new nodes
- Reduce the cache size
- Adjust items related to the JVM garbage collection

Views, triggers, and stored procedures

By definition (In RDBMS) a view represents a virtual table that acts as a real (created) table, which in reality does not contain any data. The obtained data is the result of a `SELECT` query. Views consists of a rows and column combination from one or more different tables.

Respectively in NoSQL, in Cassandra all data for key value rows are placed in one Column family. As in NoSQL, there is no `JOIN` commands and there is no possibility of flexible queries; the `SELECT` command lists the actual data, but there are no display options for a virtual table, that is, a view.

Since Cassandra does not belong to the RDBMS group, there is no possibility of creating triggers and stored procedures. RI Restrictions can be set only in the application code

Also, as Cassandra does not belong to the RDBMS group, we cannot apply Codd's rules.

Client-server architecture

At this point, we have probably already noticed that Apache Cassandra runs on a client-server architecture.

By definition, the client-server architecture allows distributed applications, since the tasks are divided into two main parts:

- On the one hand, service providers: the servers
- On the other hand, service petitioners: the clients

In this architecture, several clients are allowed to access the server; the server is responsible for meeting requests and handles each one according to its own rules.

So far, we have only used one client, managed from the same machine, that is, from the same data network.

CQLs allows us to connect to Cassandra, access a key space, and send CQL statements to the Cassandra server.

This is the most immediate method, but in daily practice, it is common to access the key spaces from different execution contexts (other systems and other programming languages).

Thus, we require other clients different from CQLs. To do it in the Apache Cassandra context, we require connection drivers.

Drivers

A driver is just a software component that allows access to a key space to run CQL statements. Fortunately, there are already a lot of drivers to create clients for Cassandra in almost any modern programming language; you can see an extensive list at this URL: `http://wiki.apache.org/cassandra/ClientOptions`.

Typically, in a client-server architecture there are different clients accessing the server from different clients, which are distributed in different networks. Our implementation needs will dictate the required clients.

Spark-Cassandra connector

Now that we are clear how a connection to a Cassandra server is done, we can talk about a very special client. Everything we have seen previously has been directed at reaching this point. We have seen what Spark can do; now we know Cassandra and we know we can use it as a storage layer to improve Spark performance.

We need a client to achieve this connection but this client is special because it has been designed specifically for Spark and not for a specific language. This special client is called: Spark Cassandra connector.

Installing the connector

The Spark-Cassandra connector has its own GitHub repository, the latest stable version is in master, but we can access a special version through a particular branch.

The Cassandra connector project home page is: `https://github.com/datastax/spark-cassandra-connector`.

At the time of writing, the most stable connector version is 1.6.0.

The connector is basically a `.jar` file loaded when Spark starts. So, if you prefer to access the JAR directly and avoid the build process, you can do it by downloading the official Maven repository.

A widely used repository is: `http://mvnrepository.com`.

The exact URL is: `http://mvnrepository.com/artifact/com.datastax.spark/spark-cassandra-connector_2.10/1.6.0-M2`.

Generating the JAR file directly from the git repository has one main advantage: all the necessary dependencies of the connector are generated. If we choose to download the JAR from the official repository, we must also download all these dependencies.

Fortunately, there is a third way to run the connector, by telling the spark-shell that we require certain packages for the session. This is done by adding the flag:

```
./spark-shell --packages datastax:spark-cassandra-connector:1.6.0-M2-s_2.10
```

The nomenclature of the package is that we know from Gradle or SBT:

```
GroupID: datastax
ArtifactID: spark-cassandra-connector
Version: 1.6.0-M2-s_2.10
```

In the following lines, we tell the shell that we require that artifact, and the shell will handle all units. Let us see how it works.

Establishing the connection

The current connector version is 1.6.0 because it is the latest stable version of Apache Spark. First, we validate if the versions are compatible; we access the Spark shell to see if we have the correct version:

```
{16-04-18 1:10}localhost:~/opt/apache/spark/spark-1.6.0-bin-hadoop2.6/bin
restrada% ./spark-shell
log4j:WARN No appenders could be found for logger
(org.apache.hadoop.metrics2.lib.MutableMetricsFactory).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for
more info.
Using Spark's repl log4j profile: org/apache/spark/log4j-defaults-
repl.properties
To adjust logging level use sc.setLogLevel("INFO")
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/
```

```
Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_60)
Type in expressions to have them evaluated.
Type :help for more information.
....
scala>
```

We stop the shell (exit command) and now, at start time, we indicate the package where we require the connector. The first time, the shell download the dependencies:

```
{16-06-08 23:18}localhost:~/opt/apache/spark/spark-1.6.0-bin-hadoop2.6/bin
restrada% >./spark-shell --packages datastax:spark-cassandra-
connector:1.6.0-M2-s_2.10
Ivy Default Cache set to: /home/restrada/.ivy2/cache
The jars for the packages stored in: /home/restrada/.ivy2/jars
:: loading settings :: url =
jar:file:/home/restrada/opt/apache/spark/spark-1.6.0-bin-
hadoop2.6/lib/spark-assembly-1.6.0-
hadoop2.6.0.jar!/org/apache/ivy/core/settings/ivysettings.xml
datastax#spark-cassandra-connector added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent;1.0
  confs: [default]
  found datastax#spark-cassandra-connector;1.6.0-M2-s_2.10 in spark-
packages
  found joda-time#joda-time;2.3 in local-m2-cache
  found com.twitter#jsr166e;1.1.0 in central
  found org.scala-lang#scala-reflect;2.10.5 in central :: retrieving ::
org.apache.spark#spark-submit-parent
  confs: [default]
  2 artifacts copied, 14 already retrieved (5621kB/32ms)
log4j:WARN No appenders could be found for logger
(org.apache.hadoop.metrics2.lib.MutableMetricsFactory).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for
more info.
Using Spark's repl log4j profile: org/apache/spark/log4j-defaults-
repl.properties
To adjust logging level use sc.setLogLevel("INFO")
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_60)
Type in expressions to have them evaluated.
```

```
Type :help for more information.
16/06/08 23:18:59 WARN Utils: Your hostname, localhost.localdomain resolves
to a loopback address: 127.0.0.1; using 192.168.1.6 instead (on interface
wlp7s0)
16/06/08 23:18:59 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to
another address
Spark context available as sc.
16/06/08 23:19:07 WARN ObjectStore: Version information not found in
metastore. hive.metastore.schema.verification is not enabled so recording
the schema version 1.2.0
16/06/08 23:19:07 WARN ObjectStore: Failed to get database default,
returning NoSuchObjectException
SQL context available as sqlContext.

scala>
```

The connector is loaded and ready for use.

Using the connector

First, we stop the Scala executor from the shell:

```
sc.stop
```

We import the required classes for communication:

```
import com.datastax.spark.connector._, org.apache.spark.SparkContext,
org.apache.spark.SparkContext._, org.apache.spark.SparkConf
```

We set a variable with the required configuration to connect.

```
val conf = new SparkConf(true).set("spark.cassandra.connection.host",
"localhost")
```

And finally, we connect to our well-known key space and our table, both created at the beginning of this chapter:

```
val sc = new SparkContext(conf)
val test_spark_rdd = sc.cassandraTable("mykeyspace", "cars")
```

Given the context and key space, it is possible to consult the values with the following statement:

```
test_spark_rdd.foreach(println)
```

Summary

NoSQL is not just hype, or a young technology; it is an alternative, with known limitations and capabilities. It is not an RDBMS killer. It's more like a younger brother who is slowly growing up and taking some of the burden. Acceptance is increasing and it will be even better as NoSQL solutions mature. Skepticism may be justified, but only for concrete reasons.

Since Cassandra is an easy and free working environment, suitable for application development, we recommended it, especially with additional utilities that ease and accelerate database administration.

Cassandra has some faults (for example, user authentication and authorization are still insufficiently supported in Windows environments) and should preferably be used when there is a need to store large amounts of data.

For start-up companies that need to manipulate large amounts of data with the aim of costs reduction, implementing Cassandra in a Linux environment is a must-have.

In the next chapter, we will explore Kafka, an amazing new technology, to manage message brokers.

5
The Broker - Apache Kafka

The aim of this chapter is to familiarize the reader with Apache Kafka and show you how to solve the consumption of millions of messages in pipeline architecture. Here we show some Scala examples to give you a foundation for the different types of implementation and integration for Kafka producers and consumers.

In addition to the explanation of the Apache Kafka Architecture and principles, we'll explore the Kafka integration with the rest of the SMACK stack, specifically with Spark. At the end of the chapter, we will learn how to administrate Apache Kafka.

This chapter has the following sections:

- Introducing Kafka
- Installation
- Cluster
- Architecture
- Producers
- Consumers
- Integration
- Administration

Introducing Kafka

Jay Kreps, the author of Apache Kafka says about the Kafka name:

> *I thought that since Kafka was a system optimized for writing using a writer's name would make sense. I had taken a lot of lit classes in college and liked Franz Kafka. Plus the name sounded cool for an open source project.*
>
> *So basically there is not much of a relationship.*

Apache Kafka is mainly optimized for writing (in this book when we say optimized we mean two million writes per second on a commodity cluster).

Nowadays, real-time information is continuously generated; this data needs easy ways to be delivered to multiple receivers. Most of the time, generators and consumers of information are inaccessible to each other, and here is when integration tools are required.

In the eighties, nineties and two thousands, the large software vendors (IBM, SAP, BEA, Oracle, Microsoft, Google, and so on) found a very lucrative market in the integration layer. Here we can find enterprise service buses, SOA architectures, integrators and other panaceas that cost several millions of dollars.

As we mentioned in `Chapter 1`, *Introducing SMACK*, in the era of fast data the first challenge is data collection, and the second is to analyze this huge amount of data.

Features of Apache Kafka

Nowadays, all traditional applications tend to have a point of integration, so, there is a need for a mechanism for seamless integration between consumers and producers of data to avoid any application rewriting at either end.

Message publishing is the mechanism for connecting heterogeneous applications by sending messages between them. The message router is known as a **message broker**. Apache Kafka is a software solution to deal with real-time information and route it to consumers in a quick way.

The objective of the message broker is to provide seamless integration but it has two collateral directives: the first is to not block the producers, and the second is to isolate producers and consumers, so as not to let the producers know who the final consumers are.

Apache Kafka is a real-time publish-subscribe solution and a messaging system: It's open source, distributed, partitioned, replicated, commit-log based with a publish-subscribe schema. Moreover, its main characteristics are:

- Distributed: The cluster centric design supports the distribution of the messages over the cluster members maintaining the semantics, i.e. we can grow the cluster horizontally without affect the consumers.
- High throughput: As we mentioned, all the technologies in the SMACK Stack are designed to work on commodity hardware. Kafka can handle hundreds of read and write operations per second from a large number of clients.
- Multi-client: Easy integration with different clients from different platforms: JVM (Clojure, Java, Groovy, Scala, Ceylon), Python, .NET, PHP, Ruby, and so on.
- Persistent: It doesn't allow any data to be lost. Kafka is designed with efficiency O(1), so data structures provide constant time performance access, no matter the data size.
- Real time: The messages produced are immediately seen by consumer threads, these are the basis of the systems called **Complex Event Processing** (**CEP**).

Figure 5-1, *Apache Kafka typical scenario* shows an Apache Kafka messaging system typical scenario.

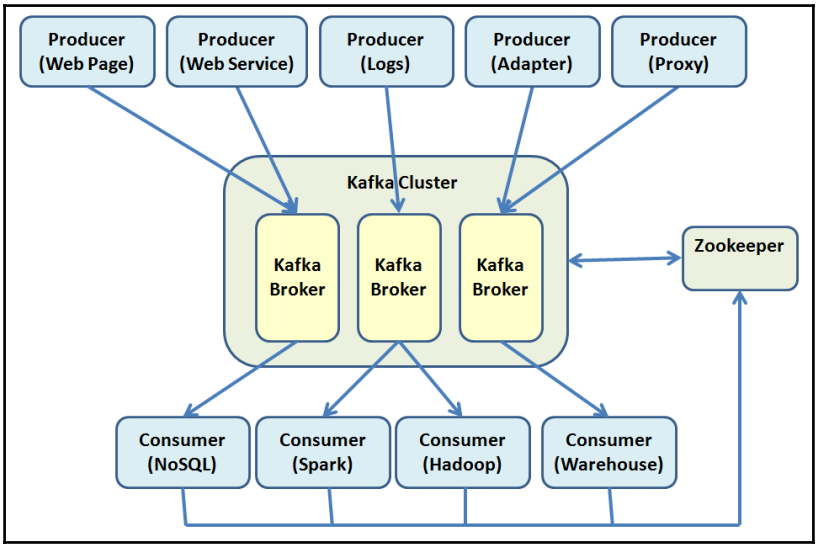

Figure 5-1. Apache Kafka typical scenario

On the **Producer** side we find several types of actors, for example:

- **Adapters**: Generating data transformations, for example, a database listener or a file system listener
- **Logs**: For example, the log files of application servers, and other legacy systems
- **Proxy**: Generating web analytics information for example
- **Web Page**: Front-end applications generating data
- **Web Services**: This is the service layer with invocation traces

We could group the clients into three types:

- **Offline**: The information is stored for a deferred analysis, for example Hadoop and Data Warehouses.
- **Near real-time**: The information is stored but could be requested at the same time, for example NoSQL databases like Apache Cassandra.
- **Real-time**: The information is analyzed as it is generated. We need a modern engine like Apache Spark, or Apache Storm (to make an analysis over HDFS).

Born to be fast data

As we've mentioned, fast data is the new ingredient of Internet based systems. Simply to operate a web page we need to know: user activity, logins, page visits, clicks, scrolls, comments, heat zone analysis, shares, and so on.

Traditionally, the data was handled, and stored with traditional aggregation solutions. Due to the high throughput, the analysis could not be done in the same moment. Today, yesterday's information is generally less valuable. Offline analysis like Hadoop is being left out of the game.

There are several examples of Apache Kafka use cases:

- Application security: Login analysis, brute force attack detection, systemic denial of service attack detection
- Collecting data from device sensor: Including sophisticated sensors like surveillance cameras to GPS cell phone sensors; and traditional sensors like light, temperature, pressure, and humidity
- Collecting logs from business systems, like application server logs, ERP, CRM, and so on

- Focused marketing in real time to huge public spaces
- Recommendations Based on correlation, popularity, and so on
- Sentiment analysis: Market tendencies, consumer segmentation
- Web search: Based on relevance

In all these cases, the analysis is done only in real time: it's now or never!

Apache Kafka is usually compared with traditional messaging systems such as ActiveMQ or RabitMQ. The difference is the data volume that Apache Kafka can handle in real time.

Kafka is a message broker. This means that the message delivery between two points is guaranteed.

An ESB consists of multiple components where a message broker is usually one of them. Besides that, ESBs serve more purposes than just message transport. The ESB also deals with protocol transformations.

An ESB also:

- Monitors, controls and routes message exchanges between services
- Solves connections between communication service components
- Controls service deployments and versioning
- Marshals the use of redundant services

There is a complete chapter about Apache Kafka and enterprise integration patterns later in this book.

Use cases

We have already seen which business scenarios are solved with Apache Kafka. However, this is an architecture book, so in which layer in the Architecture should we put Kafka? Here are some real use cases:

- Commit logs: If is a system without log system, we can use Kafka. Often a system does not have logs, simply because in the past it was not possible to handle huge data volumes effectively. There are a lot of horror stories about application servers falling because they could not write their logs correctly with the verbosity needed by the business. Kafka can also help to start and restart fallen log servers.

- Log aggregation: Contrary to what one might think, much of the work of the on-site support team is log analysis. Kafka not only provides a system for log management, but can also make heterogeneous aggregation of several logs. Kafka is a replacement for traditional log aggregation solutions. Log aggregation typically collects physical log files and puts them in a central place (a file server or HDFS) for processing. Kafka can physically collect the logs removing the implementation details such as the file location or format. In addition its process has low latency and supports multiple data sources when making distributed consumption.
- Messaging: The most popular use for Kafka. Systems are often heterogeneous and instead of rewriting them we have to translate between them. Often the manufacturer's adapters are unaffordable, and for such cases Kafka is the solution, as it's open source and can handle more volume than many traditional commercial brokers.
- Record user activity: Many marketing and advertising companies are interested in recording ALL the customer activity on a web page. This sounds ambitious and until recently it was very difficult to keep track of every click that each user made on a site. For those tasks where the data volume is huge, we can use Kafka for real-time processing and monitoring.
- Stream processing: The tendency we could write a whole book about this new topic. In some business cases, the process of collecting information consists of several stages. A clear example is when a broker is used not only to gather information but to transform it, this is the real meaning and success of the **Enterprise Service Bus** (**ESB**) mega projects. With Kafka, the information can be collected and further enriched, this (very well paid today) enrichment process is known as **stream processing**.

All this seems good, but who is using Kafka today? Here are some examples:

- LinkedIn: Used for activity stream and operational metrics. Can you imagine today's LinkedIn newsfeed without Kafka? Here's more information: `http://engineering.linkedin.com/blog/2016/04/kafka-ecosystem-at-linkedin`
- Uber: Hundreds of rides per minute rely on Kafka data feeds to bulk-load log data into Amazon S3 to stream change-data logs from local datacenters: `http://www.datanami.com/2015/10/05/how-uber-uses-spark-and-hadoop`

- Twitter: The Twitter whale is a new unicorn. Handling five billion sessions a day in real time requires Kafka as their stream processing infrastructure: `http://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time`.
- Netflix: Thousands of hours watched per minute, AND Kafka is the Backbone of Netflix's data pipeline for real-time monitoring and event-processing: `http://techblog.netflix.com/2013/12/announcing-suro-backbone-of-netflixs.html`
- Spotify: Thousands of hours of music per minute, and Kafka is used as part of their log delivery system: `http://www.meetup.com/stockholm-hug/events/121628932/`
- Yahoo: Used by the media analytics team for the real-time analytics pipeline. Their cluster handles 20 GBp/s of compressed data

Installation

Go to the Apache Kafka home page: kafka.apache.org/downloads as in *Figure 5-2*, *Apache Kafka download page*.

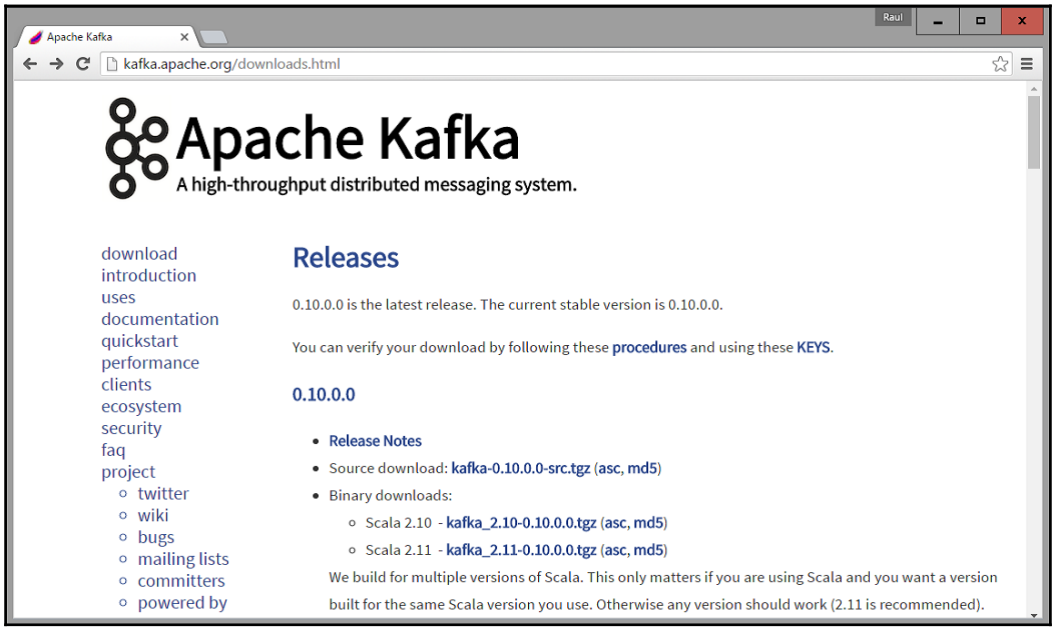

Figure 5-2. Apache Kafka download page

The Apache Kafka current version available is 0.10.0.0 as a stable release. A major limitation with Kafka since 0.8.x is that it is not backward-compatible. So, we cannot replace this version for one prior to 0.8. Once you've downloaded the latest available release, let's proceed with the installation.

Installing Java

We need Java 1.7 or later. Download and install the latest JDK from Oracle's website: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

To install in Linux (as an example):

1. Change the file mode:

```
[master@localhost opt]# chmod +x jdk-8u91-linux-x64.rpm
```

2. Go to the directory in which you want to perform the installation:

```
[master@localhost opt]# cd <directory path name>
```

3. Run the `rpm` installer with the command:

```
[master@localhost java]# rpm -ivh jdk-8u91-linux-x64.rpm
```

4. Finally, add the environment variable `JAVA_HOME`. This command will write the `JAVA_HOME` environment variable to the file `/etc/profile`:

```
[master@localhost opt]# echo "export JAVA_HOME=/usr/java/jdk1.8.0_91"
>> /etc/profile
```

Installing Kafka

To install in Linux:

1. Extract the downloaded file `kafka_2.10-0.10.0.0.tgz`:

```
[master@localhost opt]# tar xzf kafka_2.10-0.10.0.0.tgz
```

2. Create the `KAFKA_HOME` environment variable:

```
[master@localhost opt]# export KAFKA_HOME=/opt/kafka_2.10-0.10.0.0
```

3. Add the Kafka bin directory to the PATH variable:

```
[master@localhost opt]# export PATH=$PATH:$KAFKA_HOME/bin
```

Importing Kafka

To include Kafka 2.10 in our programming projects, we use the following dependencies:

With SBT:

```
// https://mvnrepository.com/artifact/org.apache.kafka/kafka_2.10
libraryDependencies += "org.apache.kafka" % "kafka_2.10" % "0.10.0.0"
```

With Maven:

```
<!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka_2.10 -->
  <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.10</artifactId>
    <version>0.10.0.0</version>
  </dependency>
```

With Gradle:

```
// https://mvnrepository.com/artifact/org.apache.kafka/kafka_2.10
compile group: 'org.apache.kafka', name: 'kafka_2.10', version: '0.10.0.0'
```

Cluster

Now we are ready to program with the Apache Kafka publisher-subscriber messaging system. First, a few terminologies:

In Kafka, there are three types of clusters:

- Single node - single broker
- Single node - multiple broker
- Multiple node - multiple broker

A Kafka cluster has five main actors:

- **Broker**: The server - a Kafka cluster has one or more physical servers where each one may have one or more server processes running.Each server process is called a broker. The topics live on the broker processes.
- **Topic**: The queue is a category or *feed name* in which messages are published by the message producers. Topics are partitioned, and each partition is represented by an ordered immutable messages sequence. The cluster has a partitioned log for each topic. Each message in the partition has a unique sequential ID called **offset**.
- **Producer**: These publish data to topics by choosing the appropriate partition in the topic. To achieve load balancing, the message allocation to the topic partition can be done in a round-robin mode or by defining a custom function.
- **Consumer**: These are applications or processes subscribed to topics and process the feed of published messages.
- **Zookeeper**: the coordinator between brokers and consumers. Zookeeper coordinates the distributed processes through a shared hierarchical namespace of data registers.These registers are called **znodes**.

We can conceptualize Zookeeper as a traditional file system, but there are two main differences between Zookeeper and a traditional file system:

- Every znode has data associated, and is designed to store data about coordination
- The znodes are limited by the amount of data they can hold

Single node - single broker cluster

Figure 5-3, Single Node - Single Broker Kafka Cluster example, shows an example diagram of a single node - single broker cluster.

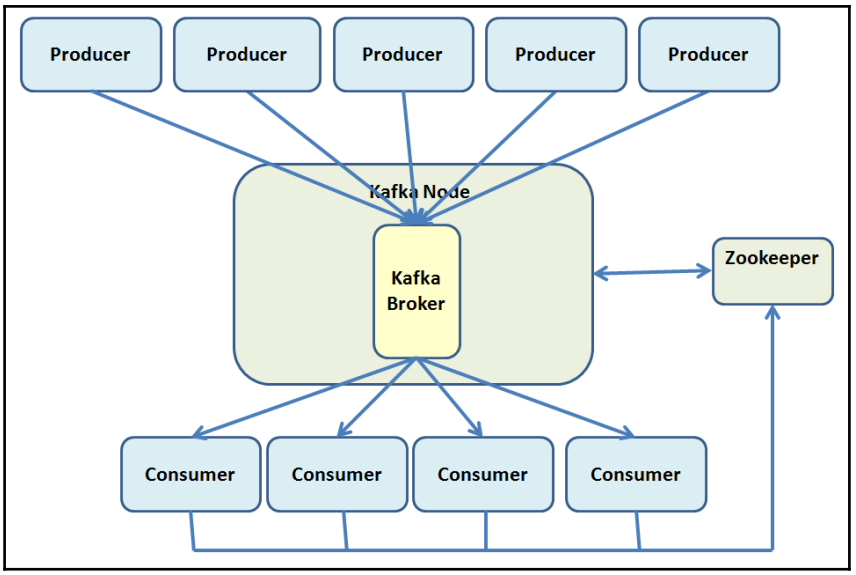

Figure 5-3. Single node - single broker Kafka cluster example

Starting Zookeeper

To start the Zookeeper server, Kafka provides a simple Zookeeper configuration file to launch a single Zookeeper instance. To install the Zookeeper instance we use this command:

```
[master@localhost kafka_2.10-0.0.0.0]# bin/zookeeper-server-start.sh
config/zookeeper.properties
```

The main properties defined in the `zookeeper.properties` are:

- `clientPort`: The listening port for client requests. By default, Zookeeper listens in the `2181` TCP port:

  ```
  clientPort=2181
  ```

- `dataDir`: The directory where the zookeeper is stored:

  ```
  dataDir=/tmp/zookeeper
  ```

- `maxClientCnxns`: The limit for the number of connections per IP (0 = unbounded)

```
maxClientCnxns=0
```

For more information about the Apache Zookeeper project visit its home page: `http://zookeeper.apache.org/`

Starting the broker

After starting Zookeeper, we start the Kafka Broker with this command:

```
[master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-server-start.sh
config/server.properties
```

The main properties defined in the `server.properties` file are:

- `broker.id`: The unique positive integer identifier for each broker:

```
broker.id=0
```

- `log.dir`: Directory to store log files:

```
log.dir=/tmp/kafka10-logs
```

- `num.partitions`: Number of log partitions per topic:

```
num.partitions=2
```

- `port`: Port where the socket server listen on:

```
port=9092
```

- `zookeeper.connect`: The Zookeeper URL connection:

```
zookeeper.connect=localhost:2181
```

Creating a topic

Kafka has a command to create topics. Here we'll create a topic called `smackTopic` with one partition and one replica:

```
    [master@localhost kafka_2.10.0-0.0.0.0]#bin/kafka-topics.sh --create --
zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic
smackTopic
```

We obtain the following output:

```
    Created topic "smackTopic".
```

The parameters:

- `--replication-factor 1` indicates one replica
- `--partition 1` indicates one partition
- `--zookeeper localhost:2181` indicates the zookeeper url

To get the list of topics on a Kafka server, we use the following command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-topics.sh --list --
zookeeper localhost:2181
```

We obtain the following output:

```
    smackTopic
```

Starting a producer

Kafka has a command to start producers: It accepts inputs from the command line and publishes them as messages. By default, each new line is considered a message.

```
    [master@localhost kafka_2.10.0-0.0.0.']# bin/kafka-console-producer.sh
--broker-list localhost:9092 --topic smackTopic
```

Two parameters are required:

- `broker-list`: The broker URL to be connected
- `topic`: The topic name (to send a message to the topic subscribers)

Now we type:

```
Salam Alaikum [Enter]
Shalom Aleijem [Enter]
```

We obtain this output (as expected):

```
Salam Alaikum
Shalom Aleijem
```

The most important properties defined in the `producer.properties` are:

- `metadata.broker.list`: List of brokers used for bootstrapping knowledge on the rest of the cluster information: `host1:port1`, `host2:port2`

```
metadata.broker.list=localhost:9092
```

- `compression codec`: The compression codec for data generated, for example, `none`, `gzip`, `snappy`

```
compression.codec=none
```

Later on in this chapter we will see how to write producers.

Starting a consumer

Kafka has a command to start a message consumer client. It shows the output at the command line as soon as it subscribes to the topic in the broker:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-console-consumer.sh
--zookeeper localhost:2181 --topic smackTopic --from-beginning
```

As we request `from-beginning` parameter, we see the following output:

```
Salam Alaikum
Shalom Aleijem
```

The main property defined in the `consumer.properties` file is:

- group.id: String that identifies a set of consumers in the same group

```
group.id=test-consumer-group
```

Later on in this chapter we will see how to write consumers.

Now play with your new toy technology, open up each program in a different console: Zookeeper, broker, producer, and consumer. Type some messages in the producer and watch them as they are displayed in the consumer.

If you don't recall how to run a program, running the command with no arguments will show the possible parameters.

Single node - Multiple broker cluster

The *Figure 5-4, Single Node - Multiple Broker Kafka Cluster example*, shows an example diagram of a single node - multiple broker cluster.

We begin by starting the Zookeeper server:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/zookeeper-server-start.sh
config/zookeeper.properties
```

We need a different `server.properties` file for every broker. Let's call them: `server-1.properties`, `server-2.properties`, `server-3.properties`, and so on.

In `server-1.properties` we specify these properties:

- `broker.id=1`
- `port=9093`
- `log.dir=/tmp/kafka-logs-1`

Similarly, on `server-2.properties` we specify these properties:

- `broker.id=2`
- `port=9094`
- `log.dir=/tmp/kafka-logs-2`

On `server-3.properties` we specify these properties:

- `broker.id=3`
- `port=9095`
- `log.dir=/tmp/kafka-logs-3`

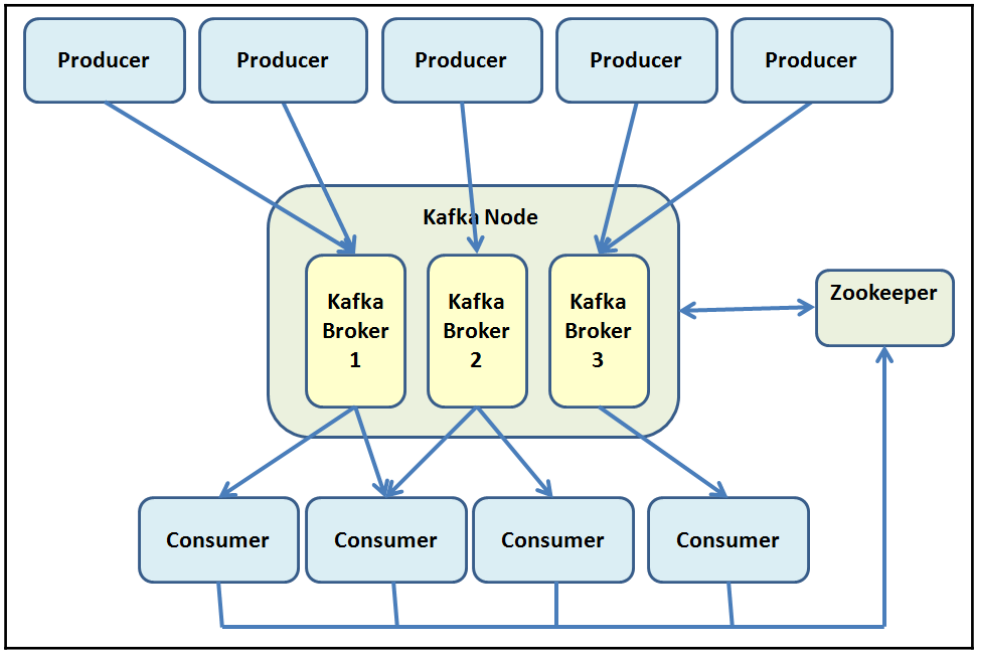

Figure 5-4. Single node - multiple broker Kafka cluster example.

Starting the brokers

With Zookeeper already running, start the Kafka brokers with these commands:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-server-start.sh
config/server-1.properties
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-server-start.sh
config/server-2.properties
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-server-start.sh
config/server-3.properties
```

Creating a topic

Using the command to create topics, we create one called `smackTopic2`, with two partitions and two replicas:

```
    [master@localhost kafka_2.10.0-0.0.0.0]#bin/kafka-topics.sh --create --
zookeeper localhost:2181 --replication-factor 2 --partitions 2 --topic
smackTopic2
```

We obtain this output:

```
    Created topic "smackTopic2".
```

Starting a producer

Now that we dominate the command to start producers, indicate more brokers in the `broker-list` is a trivial task:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-console-producer.sh
-- broker-list localhost:9093, localhost:9094, localhost:9095 --topic
smackTopic2
```

If we need to run multiple producers connecting to different brokers, we need to specify a different broker list for each producer.

Starting a consumer

To start a consumer we use that well-known command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-console-consumer.sh
-- zookeeper localhost:2181 --from-beginning --topic smackTopic2
```

Multiple node - multiple broker cluster

The *Figure 5-5, Multiple node - multiple broker Kafka cluster* shows an example diagram of a Multiple node - multiple broker cluster.

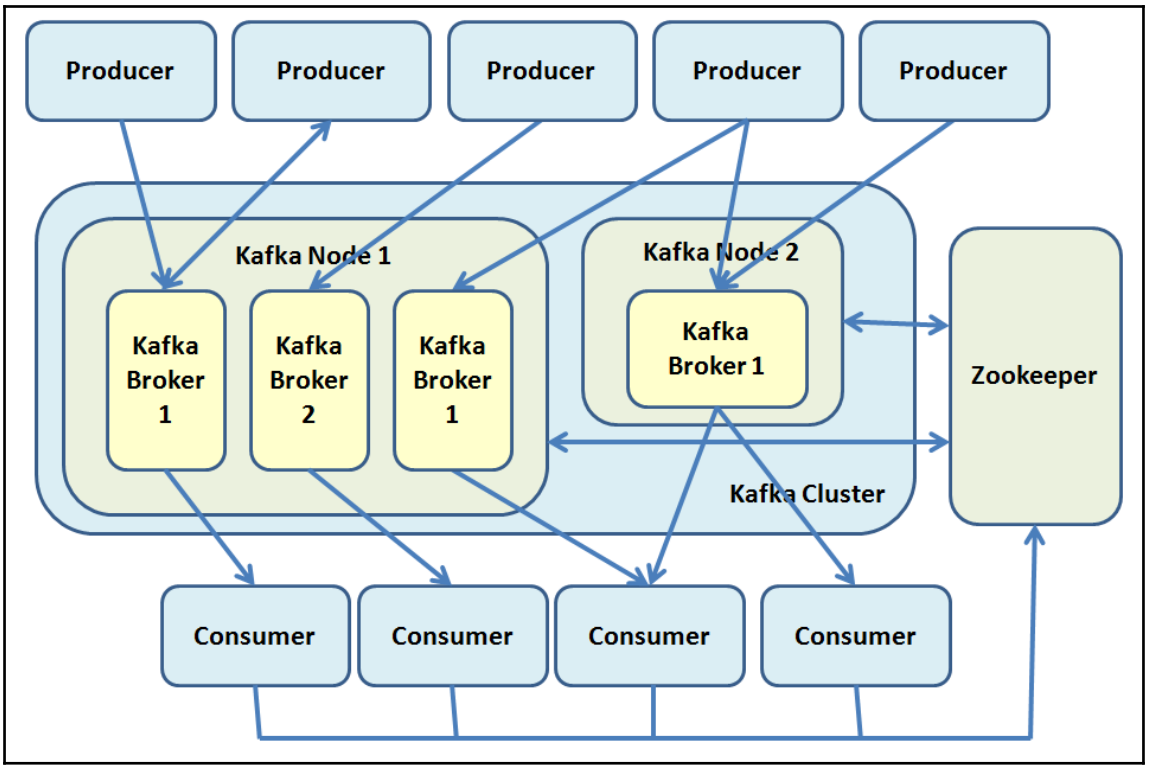

Figure 5-5. Multiple node - multiple broker Kafka cluster

Here we are in front of the real power of the cluster. In this cluster, Kafka should be installed on every machine in the cluster. Here, every physical server could have one or many brokers; all the nodes on the same cluster should connect to the same Zookeeper.

But the good news is that all the previous commands remain equal. The commands for Zookeeper, broker, producer, and consumer don't change.

Broker properties

As a section recapitulation, we show the *Table 5-1, Kafka Broker most important properties*, with the list of the most popular broker properties.

Name	Default value	Description
`broker.id`	`0`	Each broker is identified with a positive integer ID. This ID is the name of the broker and allows the broker to be moved to a different host or port without losing its consumers.
`listeners`	`null`	A comma separated list of listeners' URIs in which we will listen on and their protocols. To bind all interfaces specify the hostname as `0.0.0.0`.
`num.partitions`	`1`	Number of partitions per topic if a partition count is not given at topic creation.
`default.replication.factor`	`1`	Default replication factor for topics automatically created.
`auto.create.topics.enable`	`true`	To enable the auto-creation of the topic on the server. If this is set to `true`, any attempts to produce, consume, or fetch data for a non-existent topic will automatically create a new one with the default replication factor and the default number of partitions.
`log.dirs`	`/tmp/kafka-logs`	Directory where the log data is stored. Each new partition created is placed in the directory with less partitions.

`zookeper.connect`	`localhost:2181`	The Zookeeper's connection string in the form `hostname:port/chroot`, where `chroot` is the base directory for the path operations (namespace to sharing with other applications under the same Zookeeper cluster).

Table 5-1. Kafka Broker's most important properties

The complete list is available at `http://kafka.apache.org/documentation.html#brokerconfigs`.

Architecture

Kafka was created at LinkedIn. To start with LinkedIn used **Java Message Service** (**JMS**). But when they needed more power, that is, a scalable architecture, the LinkedIn development team decided to build the project that today we know as Kafka. In 2011, Kafka was open sourced as theApache Project. Due to size constraints, in this section we'll leave the reader with some reflections on why the architecture is designed in the way it is.

The *Figure 5-6, A topic with 3 partitions*, shows a topic with three partitions. We can see the five Kafka components: Zookeeper, broker, topic, producer and consumer.

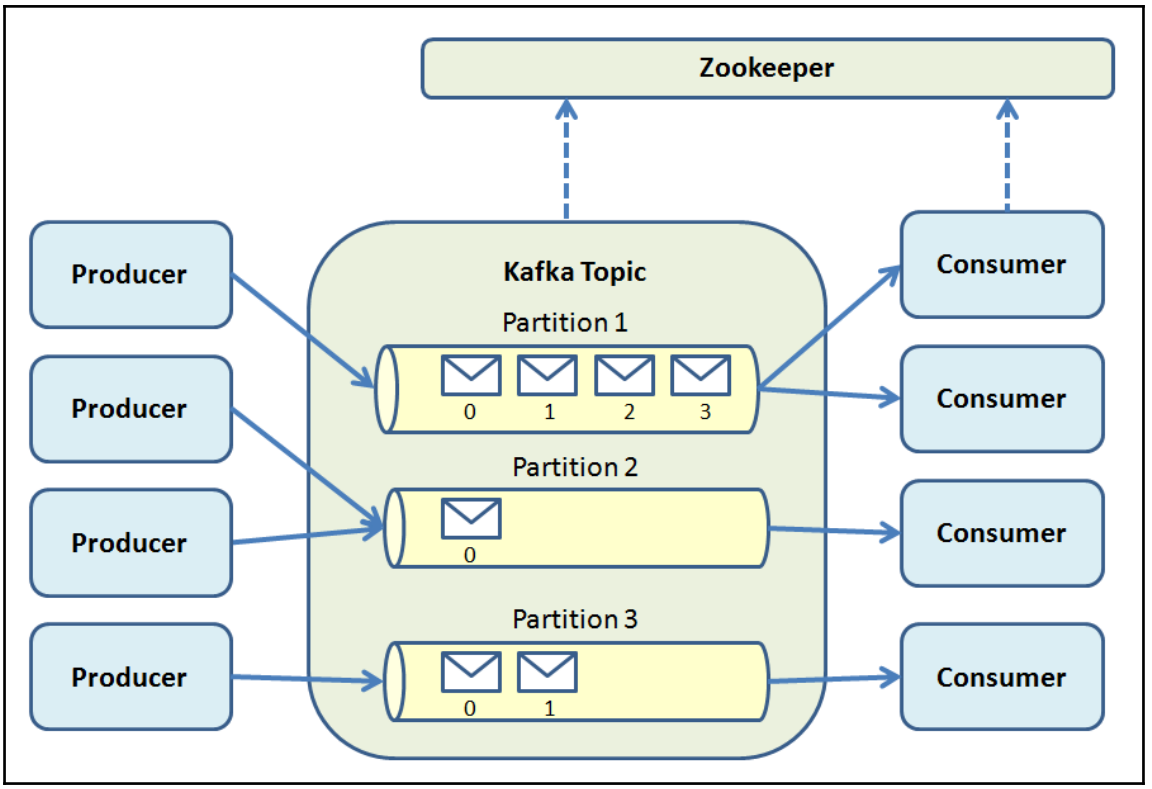

Figure 5-6. A topic with 3 partitions

The Kafka project goals are:

- An API: To support the custom implementation of producers and consumers
- Low overhead: Low network latency and low storage overhead with message persistence on disk
- High throughput: Publishing and subscribing of millions of messages, supporting data feeds in real time
- Distributed: Highly scalable architecture to handle low-latency delivery
- High availability: In case of failure, the consumers will auto-balance
- Fault tolerant: In case of failure, data integrity is guaranteed

Kafka is more than a queuing platform, because the messages are received and queued to be delivered to a consumer pool.

Kafka is more than published-subscriber platform, because the messages are not published to all the consumers.

In a nutshell the Kafka operation is:

- Messages are published on topics: We can consider a topic as a message queue, or a message category
- Topics run in a broker, a broker is a server: Kafka brokers don't just run topics, they also store messages when required
- Consumers use the Zookeeper service to get the data about a message state in order to track the message

Segment files

Internally, every partition is a logical log file, represented as a set of segment files with the same size. A partition is a sequence of ordered messages. When a message is published, the broker appends the message to the last segment of a file. When a certain number of messages is reached, the segment file is flushed to the disk. Once the file is flushed the messages are available for consumption by consumers.

Offset

The partitions are assigned to a unique sequential number called **offset**. The offset is used to identify messages inside the partition. For fault tolerance the partitions are replicated among the servers.

Leaders

Inside Kafka, each partition has one Kafka server as its leader, and the other servers are followers. The leader of a partition coordinates the read and write requests for that partition. The followers replicate asynchronously the data from the leader. If the leader fails, one server becomes the new leader. In a cluster, every server has the two roles: it's leader for some partitions and it's follower for other partitions.

Groups

Consumers are organized into groups: Each consumer is represented as a process, and one process belongs to only one group.

In Kafka there are three ways to deliver messages:

1. Messages are never redelivered but may be lost.
2. Messages may be redelivered but never lost.
3. Messages are delivered once, and only once.

As a reflection exercise: consider why there are just three ways.

Log compaction

There are two types of retention: **coarse-grained** (time based) and **finer-grained** (per message). Log compaction is the process to pass from time-based to per message retention.

In Kafka, the retention policy can be set to: *per-topic* (time-based), *size-based* and *log compaction-based*. Log compaction ensures that:

- Reads begin at the offset 0, if the consumer begins at the start of the log, the messages are in the order they are written
- Messages have sequential offsets and the offset never changes

- Message order is always preserved
- A group of background threads recopy log segment files; the records whose key appears in the loghead are removed

As another reflection exercise, can you deduce why the log compaction ensures these four points?

Kafka design

The Kafka design points are:

- **Master-less**: Like Apache Cassandra, Apache Kafka is master-less, we can remove it at any time any broker and the metadata is maintained by Zookeeper and shared with the customers.
- **Metadata**: Many messaging systems keep the message metadata at server level, but in Kafka the message state is maintained at consumer level. This prevents:
    - Multiple delivery of the same message
    - Losing messages due failures
- **OLTP**: Consumers store their state in Zookeeper, but Kafka also allows to storage in external systems called **Online Transaction Processing systems** (**OLTPs**).
- **Push and pull**: Producers push the message to the broker and consumers pull the message from it.
- **Retention**: Once a message is consumed, the message is not wasted; it is retained allowing the re-consumption of the message.
- **Storage**: The main reason for Kafka is message processing. The main function is caching and storing on the file system. The caching and flushing to disk are configurable.
- **Synchronous**: Producers have the option to be synchronous or asynchronous when sending messages to the broker.

Message compression

There are cases where the network bandwidth is a bottleneck. In Kafka there is a mechanism to compress groups of messages. Note that you don't have to be a compression expert to deduce that it is far better compress a group of messages than compress every message individually.

When we compress a group of messages, the network overhead is reduced. Before Kafka 0.8.0 the group of messages was compressed and were presented to the consumer as a single message, the consumer decompressed it later. But there were issues with decompression that produced overhead.

Since Kafka 0.8.0 some changes have been introduced to the broker to handle offsets, so the problem was moved to the broker, but the overall performance is better. The lead broker is responsible for compressing messages, so we can decrease the network overhead but that could increase the load in the CPU of the broker.

As we saw, Kafka handles the GZIP and Snappy compression protocols. We need to specify this configuration in the producer to use compression:

- `compression.codec`: This parameter indicates the codec for all data generated on this producer. The default value is `none`. Valid values are: `none`, `gzip` and `snappy`.
- `compressed.topics`: This parameter turns on the compression on particular topics. Note that if the list of compressed topics is empty, then we are enabling the compression for all the topics. If the compression codec is `none`, the compression is disabled for all the topics.

If you are DevOps master and you need to mirror data across data centers, consider using Kafka as an option when you have to transfer huge amounts of data between active and passive data centers in compressed format with low network bandwidth overhead.

Replication

When we have message partitioning on Kafka, the decision about which partitioning strategy to use is made at the broker side. The decision about how the message is partitioned is made at the producer end. The broker stores the messages as they arrive. If you recall, the number of partitions is configured for each topic within the Kafka broker.

One of the best features introduced in Kafka 0.8.0 is replication. Replication ensures that messages will be published and consumed in the case of a broker failure. Both producers and consumers are replication-aware.

In replication, each partition has *n* replicas of a message (so we can handle *n-1* failures). One replica acts as the leader. Zookeeper knows which the replica leader is. The lead replica has a list of the follower replicas. The replica stores its chunk of the message in local logs.

Kafka has two replication modes:

- Asynchronous replication process
- Synchronous replication process

Asynchronous replication

In asynchronous replication:

1. The producer identifies the lead replica from Zookeeper.
2. The producer publishes the message.
3. Once the message is published it's written to the lead replica log.
4. The followers start pulling the message.
5. When the message is written on the lead replica, the leader sends the acknowledgement to the consumer.

Synchronous replication

In synchronous replication:

1. The producer identifies the lead replica from Zookeeper.
2. The producer publishes the message.
3. Once the message is published, it is written to the lead replica log.
4. The followers start pulling the message.
5. The leader waits for all the followers to send the written acknowledgement of the replica.
6. When replication is complete, the leader sends the acknowledgement to the producer.

As we can see, the asynchronous mode is faster, but is not fault tolerant.

Replication ensures strong durability and high availability. It guarantees that any successfully published message will not be lost and will be consumed, even in the case of broker failures.

Producers

Producers are applications that create messages and publish them to the broker. Normally, producers are: frontend applications, web pages, web services, backend services, proxies, and adapters to legacy systems. We can write Kafka producers in Java, Scala, C and Python.

The process begins when the producer connects to any live node and requests the metadata about the partitions leaders of a topic, to put the message directly to the partition lead broker.

Producer API

First, we need to understand the classes needed to write a producer:

- Producer: The producer class is `KafkaProducer <K, V>` in `org.apache.kafka.clients.producer.KafkaProducer`

Kafka Producer is a type of Java Generic written in Scala, K specifies the type of the partition key and V specifies the message value.

- `ProducerRecord`: The class is `ProducerRecord <K, V>` in `org.apache.kafka.clients.producer.ProducerRecord`

This class encapsulates the data required for establishing the connection with the brokers (broker list, partition, message serializer, and partition key).

The `ProducerRecord` is a type of Java generic written in Scala, `K` specifies the type of the partition key and `V` specifies the message value.

The producer API encapsulates all the low-level producer implementations. The default mode is asynchronous, but we can specify, in `producer.type`, another configuration.

Scala producers

Now we will write in Scala a simple Kafka producer to send messages to the broker. The `SimpleProducer` class creates a message for a specific topic and sends it using the message partitioning. The examples of this chapter were tested on Scala version 2.10.

Step 1: Import classes

We need these two classes:

```
import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}
```

Step 2: Define properties

We need to define these properties:

```
val props = new Properties()
props.put("metadata.broker.list","192.168.146.132:9093,
192.168.146.132:9094, 192.168.146.132:9095")
props.put("serializer.class", "kafka.serializer.StringEncoder")
props.put("request.required.acks", "1")
producer = new KafkaProducer(props)
```

A brief description of the properties:

- `metadata.broker.list`: To specify the brokers list to the producer connects in the format [`node:port`, `node:port`]. Kafka determines the lead broker of the topic.
- `serializer.class`: This class is used to specify the serializer used when preparing the message for transmission from the producer to the broker.

In this example we use the string encoder provided by Kafka. By default, the `serializer` for the key and message is the same, but we can also implement the custom `serializer` class by extending `kafka.serializer.StringEncoder`.

- `request.required.acks`: It is used to indicate the broker to send an acknowledgment to the producer when a message is received.

1 means the producer receives an acknowledgment once the lead replica has received the message.

The default mode is *fire and forget* meaning not informed in case of message loss.

Step 3: Build and send the message

We make the message with the publishing timestamp:

```
val topic = args(0)
val runtime = new Date().toString
```

```
val msg = "Message Publishing Time - " + runtime
val data = new ProducerRecord[String, String](topic, msg)
producer.send(data)
```

Listing 5-1, SimpleProducer.scala, shows the code of the `SimpleProducer.scala`:

```
package packt.ch05

import java.util.{Date, Properties}
import packt.ch05.SimpleProducer._
import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}
object SimpleProducer {
private var producer: KafkaProducer[String, String] = _
def main(args: Array[String]) {
val argsCount = args.length
if (argsCount == 0 || argsCount == 1)
throw new IllegalArgumentException(
"Provide topic name and Message count as arguments")
// Topic name and the message count to be published is passed from the
  // command line
val topic = args(0)
val count = args(1)
val messageCount = java.lang.Integer.parseInt(count)
println("Topic Name - " + topic)
println("Message Count - " + messageCount)
val simpleProducer = new SimpleProducer()
    simpleProducer.publishMessage(topic, messageCount)
   }
}
class SimpleProducer {
val props = new Properties()
// Set the broker list for requesting metadata to find the lead broker
props.put("bootstrap.servers",
 "192.168.146.132:9093, 192.168.146.132:9094, 192.168 146.132:9095 ")
//This specifies the serializer class for keys props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer")
props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer")
// 1 means the producer receives an acknowledgment once the lead replica
// has received the data. This option provides better durability as the
// client waits until the server acknowledges the request as successful.
props.put("request.required.acks", "1")
producer = new KafkaProducer(props)
private def publishMessage(topic: String, messageCount: Int) {
for (mCount <- 0 until messageCount) {
val runtime = new Date().toString
val msg = "Message Publishing Time - " + runtime
println(msg)
```

```
// Create a message
val data = new ProducerRecord[String, String](topic, msg)
// Publish the message
producer.send(data)     }
// Close producer connection with broker.
producer.close()
   }
}
```

Step 4: Create the topic

Before running the program, we must create the topic: We can create it using the API (yes, we can create topics from a program) or from the command line as follows:

```
    [master@localhost kafka_2.10.0-0.0.0.0]#bin/kafka-topics.sh --create --
zookeeper localhost:2181 --replication-factor 1 --partitions 3 --topic
smackTopic
```

Step 5: Compile the producer

We compile the program using this command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# scalac .
packt/ch05/SimpleProducer.scala
```

Step 6: Run the producer

We run the `SimpleProducer` class with this command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# scala
packt.ch05.SimpleProducer smackTopic 10
```

Our program receives two arguments: the topic name and the number of messages to publish.

Step 7: Run a consumer

As usual, we start the consumer program with this command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-console-consumer.sh
-- zookeeper localhost:2181 --from-beginning --topic smackTopic
```

Producers with custom partitioning

Now we can make more sophisticated producers, let's write another program that implements customized messages partitioning. The example consists of recollecting the IPs visiting a website, which we record and publish. The message has three parts: time stamp, website name and IP address.

Step 1: Import classes

For our example, we need to import these classes:

```
import java.util.Date
import java.util.Properties
import java.util.Random
import org.apache.kafka.clients.producer.KafkaProducer
import org.apache.kafka.clients.producer.ProducerRecord
```

Step 2: Define properties

We define the following properties:

```
val props = new Properties() props.put("metadata.broker.list",
"192.168.146.132:9092, 192.168.146.132:9093, 192.168.146.132:9094")
props.put("serializer.class", "kafka.serializer.StringEncoder")
props.put("partitioner.class", "packt.ch05.SimplePartitioner")
props.put("request.required.acks", "1")
producer = new KafkaProducer(props)
```

Step 3: Implement the partitioner class

We write the `SimplePartitioner` class that extends the abstract class `Partitioner`. The class takes a key, in this case the IP address, and makes a modulo operation with the number of partitions.

The *Listing 5-2, SimplePartitioner.scala,* shows the complete code of the `SimplePartitioner.scala`

Listing 5-2. SimplePartitioner.scala:

```
package packt.ch05
import java.util
import kafka.utils.VerifiableProperties
import org.apache.kafka.clients.producer.KafkaProducer
import org.apache.kafka.clients.producer.Partitioner
import org.apache.kafka.common.Cluster
object SimplePartitioner {
private var producer: KafkaProducer[String, String] = _
}
class SimplePartitionerextends Partitioner {
def partition(key: AnyRef, a_numPartitions: Int): Int = {
var partition = 0
val partitionKey = key.asInstanceOf[String]
val offset = partitionKey.lastIndexOf('.')
if (offset >0) {
       partition = java.lang.Integer.parseInt(partitionKey.substring(offset
+ 1)) %
          a_numPartitions
     }
     partition
   }
override def partition(topic: String,
key: AnyRef,
keyBytes: Array[Byte],
value: AnyRef,
valueBytes: Array[Byte],
cluster: Cluster): Int = partition(key, 10)
override def close() {
   }
override def configure(configs: util.Map[String, _]) {
   }
}
```

Step 4: Build and send the message

The *Listing 5-3, CustomPartitionProducer.scala* shows the complete code of the `CustomPartitionProducer.scala`.

Listing 5-3. `CustomPartitionProducer.scala`:

```
package packt.ch05 import java.util.Date import java.util.Properties import
java.util.Random import org.apache.kafka.clients.producer.KafkaProducer
import org.apache.kafka.clients.producer.ProducerRecord import
CustomPartitionProducer._ object CustomPartitionProducer { var producer:
KafkaProducer[String, String] = _ def main(args: Array[String]) { val
argsCount = args.length if (argsCount == 0 || argsCount == 1) throw new
IllegalArgumentException( "Please provide topic name and Message count as
arguments") // Topic name and message count to be published received from
the     // command line val topic = args(0) val count = args(1) val
messageCount = java.lang.Integer.parseInt(count) println("Topic Name - " +
topic) println("Message Count - " + messageCount) val simpleProducer = new
CustomPartitionProducer()     simpleProducer.publishMessage(topic,
messageCount)   } } class CustomPartitionProducer { val props = new
Properties() // Set the broker list for requesting metadata to find the
lead broker props.put("metadata.broker.list", "192.168.146.132:9092,
192.168.146.132:9093, 192.168.146.132:9094") // This specifies the
Serializer class for keys props.put("serializer.class",
"kafka.serializer.StringEncoder") // Defines the class to be used for
determining the partition   // in the topic where the message needs to be
sent. props.put("partitioner.class", "packt.ch05.SimplePartitioner") // 1
means the producer receives an acknowledgment once the lead replica   //
has received the data.
props.put("request.required.acks", "1") producer = new KafkaProducer(props)
private def publishMessage(topic: String, messageCount: Int) { val random =
new Random() for (mCount <- 0 until messageCount) { val clientIP =
"192.168.14." + random.nextInt(255) val accessTime = new Date().toString
val msg = accessTime + ",kafka.apache.org," + clientIP println(msg) //
Create a ProducerRecord instance val data = new ProducerRecord[String,
String](topic, clientIP, msg) // Publish the message producer.send(data)
} producer.close()   }
}
```

Step 5: Create the topic

As usual, before run the program, we must create the topic `pageHits`:

```
    [master@localhost kafka_2.10.0-0.0.0.0]#bin/kafka-topics.sh --create --
zookeeper localhost:2181 --replication-factor 3 --partitions 5 --topic
pageHits
```

Step 6: Compile the programs

We compile the programs with the following commands:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# scalac .
packt/ch05/SimplePartitioner.scala
    [master@localhost kafka_2.10.0-0.0.0.0]# scalac .
packt/ch05/CustomPartitionProducer.scala
```

Step 7: Run the producer

We run the `CustomPartitionProducer` with the following command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# scala
packt.ch05.CustomPartitionProducer pageHits 100
```

Our program receives two arguments: the topic name and the number of messages to publish.

Step 8: Run a consumer

We run the consumer program with the following command line:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-console-consumer.sh
--zookeeper localhost:2181 --from-beginning --topic pageHits
```

Producer properties

As a section review, we have the *Table 5-2, Kafka Production most important properties,* with the list of the most popular producer properties.

Name	Type	Default	Description
`bootstrap.servers`	list		Producer uses this property to get metadata about topics, partitions, and replicas. The format is `host1:port1,host2:port2`.
`key.serializer`	class		Specifies the serializer class for the messages. The default encoder accepts and returns the same byte.
`value.serializer`	class		Specifies the serializer value for the messages.

`acks`	string	1	Controls when the producer request is considered complete and when the producer receives an acknowledgment from the broker:
			0 = producer will never wait for an acknowledgment from the broker, lowest latency, but with weakest durability.
			1 = producer receives an acknowledgment once the lead replica has received the data, affording better durability as the client waits until the server acknowledges a successful request.
			-1 = producer will receive an acknowledgment once all the in-sync replicas have received the data, providing the best durability.
`buffer.memory`	long	33554432	The total bytes of memory the producer can use to buffer records waiting to be sent to the server.
`compression.type`	string	none	Specifies the compression codec for all data generated by this producer. Values accepted: `none`, `gzip`, and `snappy`.
`retries`	int	0	A value greater than zero causes the client to resend any record with a potentially transient error or whose sending fails.

Table 5-2. Kafka producers' most important properties

You can find complete list on: `http://kafka.apache.org/documentation.html#producerconfigs`

Consumers

Consumers are applications that consume the messages published by the broker. Normally, producers are: real-time analysis applications, near real-time analysis applications, noSQL solutions, data warehouses, backend services or subscriber-based solutions. We can write Kafka producers in JVM (Java, Groovy, Scala, Clojure, Ceylon) Python and C/C++.

The consumer subscribes for message consumption to a specific topic on the Kafka broker. The consumer then makes a fetch request to the lead broker to consume from the message partition by specifying the message offset. The consumer works in a pull model and always pulls all available messages from its current position.

Consumer API

In the Kafka 0.8.0 version there was two API types for consumers: High level APIs and low level APIs. In version 0.10.0 they were unified.

To use the consumer API with Maven we should use the coordinates:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>0.10.0.0</version>
</dependency>
```

The use the consumer API classes with SBT:

```
// https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients
libraryDependencies += "org.apache.kafka" % "kafka-clients" % "0.10.0.0"
```

To use the consumer API classes with Gradle we use:

```
// https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients
compile group: 'org.apache.kafka', name: 'kafka-clients', version:
'0.10.0.0'
```

Simple Scala consumers

We will write a single thread Scala program using the consumer API for ingesting the messages from a topic. This `SimpleConsumer` class is used to fetch messages from a topic and consume them. Here, we assume that we have a single partition on the topic.

Step 1: Import classes

We need to import the following classes:

```
import java.util
import java.util.Properties
import kafka.consumer.ConsumerConfig
```

Step 2: Define properties

We need to define these properties:

```
val props = new Properties()
props.put("zookeeper.connect", zookeeper)
props.put("group.id", groupId)
props.put("zookeeper.session.timeout.ms", "500")
props.put("zookeeper.sync.time.ms", "250")
props.put("auto.commit.interval.ms", "1000")
new ConsumerConfig(props)
```

Now let's see the most important properties used in the code:

- `auto.commit.interval.ms`: Defines the frequency in milliseconds at which consumer offsets get committed.
- `group.id`: Specifies the consumer group name (shared by all the consumers on the group). Also, this is the process name used by Zookeeper to store offsets.
- `zookeeper.connect`: Specifies the Zookeeper <node:port> connection to find the running Zookeeper instance in the cluster. Zookeeper is used to store offsets of messages consumed by this consumer group for a specific topic and partition.
- `zookeeper.session.timeout.ms`: Specifies the Zookeeper session timeout in milliseconds, it represents the amount of time Kafka will wait for a Zookeeper response to a request before giving up and continuing to consume messages.
- `zookeeper.sync.time.ms`: Specifies the Zookeeper sync time in milliseconds between the leader and its followers.

Step 3: Code the SimpleConsumer

The *listing 5-4, SimpleConsumer.scala,* shows the whole code of the `SimpleConsumer.scala`:

Listing 5-4. SimpleConsumer.scala:

```
package packt.ch05
import java.util
import java.util.Properties
import kafka.consumer.ConsumerConfig import SimpleConsumer._
import scala.collection.JavaConversions._
object SimpleConsumer {
private def createConsumerConfig(zookeeper: String, groupId: String):
ConsumerConfig = {
val props = new Properties()
    props.put("zookeeper.connect", zookeeper)
    props.put("group.id", groupId)
```

```
    props.put("zookeeper.session.timeout.ms", "500")
    props.put("zookeeper.sync.time.ms", "250")
    props.put("auto.commit.interval.ms", "1000")
new ConsumerConfig(props)
 }
def main(args: Array[String]) {
val zooKeeper = args(0)
val groupId = args(1)
val topic = args(2)
val simpleHLConsumer = new SimpleConsumer(zooKeeper, groupId, topic)
simpleHLConsumer.testConsumer()
 }
}
class SimpleConsumer(zookeeper: String, groupId: String, private val topic:
String) {
private val consumer =
kafka.consumer.Consumer.createJavaConsumerConnector(createConsumerConfig(zo
okeeper, groupId))
def testConsumer() {
val topicMap = new util.HashMap[String, Integer]()
    topicMap.put(topic, 1)
val consumerStreamsMap = consumer.createMessageStreams(topicMap)
val streamList = consumerStreamsMap.get(topic)
for (stream <- streamList; aStream <- stream)
println("Message from Single Topic :: " + new String(aStream.message()))
if (consumer != null) {
       consumer.shutdown()
   }
  }
}
```

Step 4: Create the topic

Before running the program, we create the topic `smackTopic` with this command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]#bin/kafka-topics.sh --create --
zookeeper localhost:2181 --replication-factor 1 --partitions 3 --topic
smackTopic
```

Step 5: Compile the program

We compile the program with the following command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# scalac .
packt/ch05/SimpleConsumer.scala
```

Step 6: Run the producer

We run the `SimpleProducer` with this command line:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# scala
packt.ch05.SimpleProducer smackTopic 100
```

Step 7: Run the consumer

We run the `SimpleConsumer` with this command line:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# scala
packt.ch05.SimpleConsumer localhost:2181 testGroup smackTopic
```

The `SimpleConsumer` class takes three arguments: the Zookeeper connection string in form <host:port>, the group ID, and the Kafka topic name.

Multithread Scala consumers

A multithreaded consumer API design is based on the number of partitions in the topic and has a one-to-one mapping approach between the thread and the partitions on the topic.

If we don't have the one-to-one relationship we will have some conflicts such as threads that never receive a message or a threads that receive messages from multiple partitions. Now, let's code the `MultiThreadConsumer`.

Step 1: Import classes

We need to import these classes:

```
import java.util
import java.util.Properties
import java.util.concurrent.ExecutorService
import java.util.concurrent.Executors
import kafka.consumer.ConsumerConfig
```

Step 2: Define properties

We define these properties:

```
val props = new Properties()
props.put("zookeeper.connect", zookeeper)
```

```
props.put("group.id", groupId)
props.put("zookeeper.session.timeout.ms", "500")
props.put("zookeeper.sync.time.ms", "250")
props.put("auto.commit.interval.ms", "1000")
new ConsumerConfig(props)
```

Step 3: Code the MultiThreadConsumer

The *listing 5-5, MultiThreadConsumer.scala,* shows the full code of `MultiThreadConsumer.scala`

Listing 5-5. MultiThreadConsumer.scala:

```
package packt.ch05
import java.util
import java.util.Properties
import java.util.concurrent.ExecutorService
import java.util.concurrent.Executors
import kafka.consumer.ConsumerConfig
import MultiThreadConsumer._
import scala.collection.JavaConversions._
object MultiThreadConsumer {
private def createConsumerConfig(zookeeper: String, groupId: String):
ConsumerConfig = {
  val props = new Properties()
      props.put("zookeeper.connect", zookeeper)
      props.put("group.id", groupId)
      props.put("zookeeper.session.timeout.ms", "500")
      props.put("zookeeper.sync.time.ms", "250")
      props.put("auto.commit.interval.ms", "1000")
new ConsumerConfig(props)
 }
 def main(args: Array[String]) {
 val zooKeeper = args(0)
 val groupId = args(1)
 val topic = args(2)
 val threadCount = java.lang.Integer.parseInt(args(3))
 val multiThreadHLConsumer = new MultiThreadConsumer(zooKeeper,
     groupId, topic)
     multiThreadHLConsumer.testMultiThreadConsumer(threadCount)
 try {
        Thread.sleep(10000)
     } catch {
     case ie: InterruptedException =>
  }
  multiThreadHLConsumer.shutdown()
 }
```

```
 }
class MultiThreadConsumer(zookeeper: String, groupId: String, topic:
String) {
private var executor: ExecutorService = _
private val consumer =
kafka.consumer.Consumer.createJavaConsumerConnector(createConsumerConfig(zo
okeeper, groupId))
def shutdown() {
if (consumer != null) consumer.shutdown()
if (executor != null) executor.shutdown()
 }
def testMultiThreadConsumer(threadCount: Int) {
val topicMap = new util.HashMap[String, Integer]()
// Define thread count for each topic
topicMap.put(topic, threadCount)
// Here we have used a single topic but we can also add
   // multiple topics to topicCount MAP
val consumerStreamsMap = consumer.createMessageStreams(topicMap)
val streamList = consumerStreamsMap.get(topic)
// Launching the thread pool
executor = Executors.newFixedThreadPool(threadCount)
// Creating an object messages consumption
var count = 0
for (stream <- streamList) {
  val threadNumber = count
  executor.submit(new Runnable() {
def run() {
val consumerIte = stream.iterator()
while (consumerIte.hasNext)
println("Thread Number " + threadNumber + ": " + new
String(consumerIte.next().message()))
println("Shutting down Thread Number: " + threadNumber)
 }
})
count += 1 }
if (consumer != null) consumer.shutdown()
if (executor != null) executor.shutdown()
  }
}
```

Step 4: Create the topic

Before running the program, we must create the topic `smackTopic` with this command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]#bin/kafka-topics.sh --create --
zookeeper localhost:2181 --replication-factor 2 --partitions 4 --topic
smackTopic
```

Step 5: Compile the program

We compile the program with the following command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# scalac .
packt/ch05/MultiThreadConsumer.scala
```

Step 6: Run the producer

We run the `SimpleProducer` with this command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# scala
packt.ch05.SimpleProducer smackTopic 100
```

Step 7: Run the consumer

We run the `MultiThreadConsumer` with this command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# scala
packt.ch05.MultiThreadConsumer localhost:2181 testGroup smackTopic 4
```

The `MultiThreadConsumer` program takes four arguments:

- The Zookeeper connection string in the form <host:port>
- A unique group ID
- The topic name
- The thread count

This program will print all the partitions of messages associated with each thread.

Consumer properties

As a section review, the *table 5-3, Kafka Consumer most important properties*, shows the list of the most important consumer properties.

Name	Default	Type	Description
`group.id`	""	string	This string identifies the consumer group that the consumer belongs to.
`zookeeper.connect`		string	The Zookeper connection string in the format hostame:port. If we want to connect to other servers when the first server is down we specify the connection string in the form `host1:port1`, `host2:port2`...
`consumer.id`	null		If not set, this value is generated automatically.
`fetch.min.bytes`	1	int	The Minimum amount of data the server should return for a fetch request. Setting this to something greater than 1 will cause the server to wait to accumulate larger data amounts so it can improve server throughput a bit at the cost of additional latency.
`heartbeat.interval.ms`	3000	int	Expected time between heartbeats to the consumer coordinator when using Kafka's group management facilities.
`key.deserializer`		class	The deserializer class for the key implementing the deserializer interface.
`key.serializer`		class	The serializer class for the key implementing the serializer interface.
`max.partition.fetch.bytes`	1048576	int	The maximum amount of data per-partition returned by the server. The maximum total memory used for a request is given by `#partitions * max.partition.fetch.bytes`

`session.timeout.ms`	30000	int	The timeout to detect failures when using Kafka's group management. If it hasn't received a consumer's heartbeat within the session timeout, the broker marks the consumer as failed and rebalances the group.
`value.deserializer`		class	The deserializer class for value implementing the deserializer interface.
`value.serializer`		class	The serializer class for value implementing the serializer interface.

Table 5-3. Kafka Consumers' most important properties

You can find the complete list at: `http://kafka.apache.org/documentation.html#consumerconfigs`

Integration

Processing small data amounts in real time is not a challenge when we use **Java Messaging Service** (**JMS**), but, if we learn from the LinkedIn experience, we will see that these processing systems have serious performance limitations when dealing with large data volumes. Moreover, these systems are a nightmare when we try to scale horizontally, because they don't.

Integration with Apache Spark

For this demo, we need a Kafka cluster up and running. Also, we need Spark installed on our machine and ready to be deployed.

Apache Spark has one utility class to create a data stream to be read from Kafka. As with any Spark project, we first need to create `SparkConf` and the Spark `StreamingContext`:

```
val sparkConf = new SparkConf().setAppName("SparkKafkaTest")
val jssc = new JavaStreamingContext(sparkConf, Durations.seconds(10))
```

The `JavaStreamingContext` is a Java friendly version of `StreamingContext` which is the main entry point for Spark streaming functionality.

We create the `Hashset` for the topic and Kafka consumer parameters:

```
val topicsSet = new HashSet[String]()
topicsSet.add("smackTopic")
val kafkaParams = new HashMap[String, String]()
kafkaParams.put("metadata.broker.list", "localhost:9092")
```

We create a direct Kafka stream with brokers and topics:

```
val messages = KafkaUtils.createDirectStream(
jssc,
classOf[String],
classOf[String],
classOf[StringDecoder],
classOf[StringDecoder],
kafkaParams,
topicsSet)
```

With this stream we can run our usual data processing algorithms:

1. We create a Spark streaming context that sets up the entry point for all stream functionality. Then we set up the stream processing batch interval to 10 seconds.
2. We create the Hash Set for the topics to read from.
3. We set the parameters for the Kafka producer using a hash map. This map must have a value for `metadata.broker.list`, which is the comma separated list of host and port numbers.
4. We create the input DStream using the `KafkaUtils` class.

Once we have the DStream ready, we can apply our algorithms to it. How to do that is beyond the scope of this book but Spark streaming is explained in detail in the Spark chapter.

Administration

There are numerous tools provided by Kafka to manage features such as: cluster management, topic tools, and cluster mirroring. Let's see some of these tools.

Cluster tools

When replicating multiple partitions, we obtain several replicas of which one acts as leader, and the rest as followers. When there is no leader, a follower takes the leadership.

When we have to shut down the broker for maintenance activities, the new leader is elected sequentially. This brings significant I/O operations on Zookeeper. With a big cluster, this means delays in service.

To reach high availability, Kafka provides a tool for shutting down the brokers. This tool transfers the leadership among the replicas or to another broker. If we don't have in-sync replicas available, the tool fails to shut down the broker to ensure the data integrity.

This tool is used with this command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-run-class.sh
kafka.admin. ShutdownBroker --zookeeper <zookeeper_host:port/namespace> --
broker <brokerID> --num.retries 3 --retry.interval.ms 100
```

The Zookeeper URL and the Broker ID are mandatory parameters. There are other optional parameters, for example: `num.retries` (the default value is 0), and `retry.interval.ms` (the default value is 1000).

When the server is stopped gracefully, it will sync all its logs to disk to avoid any log recovery when it it's restarted again. Due to log recovery this is a time-consuming task. Before shutdown, it moves the leader partitions to other replicas, so it ensures low downtime for each partition.

Controlled shutdown is enabled in this way:

```
controlled.shutdown.enable=true
```

When we have a big cluster, Kafka ensures that the lead replicas are equally distributed among the brokers. If a broker fails in shutdown, this distribution could not be balanced.

To maintain a balanced distribution, Kafka has a tool to distribute lead replicas across the brokers in the cluster. This tool is used with this syntax:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-preferred-replica-
election.sh --zookeeper <zookeeper_host:port/namespace>
```

This tool updates the Zookeeper path with the list of topic partition whose lead replica needs to be moved. If a controller finds that the preferred replica is not the leader, it sends a request to the broker to make the preferred replica the partition leader. If the preferred replica is not in the ISR list, the controller fails the operation to avoid data loss.

We can specify a json list for this tool in this format:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-preferred-
replicaelection.
    sh --zookeeper <zookeeper_host:port/namespace> --path-to-jsonfile
    topicPartitionList.json
```

The `topicPartitionList.json` file format should be:

```
{"partitions": [
    {"topic": "SmackTopic", "partition": "0"},
    {"topic": "SmackTopic", "partition": "1"},
    {"topic": "SmackTopic", "partition": "2"},
    {"topic": "smackTopic2", "partition": "0"},
    {"topic": "smackTopic2", "partition": "1"},
    {"topic": "smackTopic2", "partition": "2"},
  ]
}
```

Adding servers

When we add servers to the cluster, a unique broker ID needs to be assigned to each new server. This way to add servers doesn't assign data partitions to the server. So,a new server won't perform any work till new partitions are migrated to it or new topics are created.

To move partitions between brokers there is a tool to reassign partitions, located in `bin/kafka-reassign-partitions.sh`. This tool takes care of everything. When migrating, Kafka makes the new server a follower of the migrating partition. This enables the new server to fully replicate the existing data in the partition.

The reassign-partition tool runs in three different modes:

- `--generate`: To move the partitions using the topics and brokers list shared with the tool
- `--execute`: To move the partitions using the user plan specified in the `--reassignment-json-file`
- `--verify`: To move the partitions using the resulting status (successful/failed/in progress) of the last `--execute`.

The partition reassignment tool could be used to move particular topics form current brokers to new brokers. The administrator provides a list of topics and a target list of new Broker IDs. This tool distributes the partitions of a given topic across the new brokers. For example:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# cat topics-for-new-server.json
    {"partitions":
      [{"topic": "smackTopic",
      {"topic": "smackTopic2"}],
      "version":1
    }
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-reassign-
partitions.sh --zookeeper localhost:2181 --topics-to-move-json-file topics-
for-new-server.json --broker-list "4,5" --generate new-topic-
reassignment.json
```

This command generates the assignment (`new-topic-reassignment.json`) plan to move all partitions for topics `smackTopic` and `smackTopic2` to the new set of brokers having IDs 4 and 5. At the end of this move, all partitions will only exist on brokers 5 and 6. To initiate the assignment with the `kafka-reassign-partitions.sh` tool we type:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-reassign-partitions.
sh --zookeeper localhost:2181 --reassignment-json-file new-topic-
reassignment.json --execute
```

We could use this tool to selectively move the partitions from the existing broker to a new broker:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# cat partitions-
reassignment.json
    {"partitions": [{"topic": "smackTopic", "partition": 1,"replicas":
[1,2,4] }], }], "version":1 }
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-reassign-
partitions.sh --zookeeper localhost:2181 --reassignment-json-file
partitions-reassignment.json --execute
```

This command moves some replicas for selected partitions to the new server. Once the reassignment is done, we can verify the operation:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-reassign-
partitions.sh --zookeeper localhost:2181 --reassignment-json-file new-
topic-reassignment.json --verify
    Status of partition reassignment:
    Reassignment of partition [smackTopic, 0] completed successfully
    Reassignment of partition [smackTopic, 1] is in progress
    Reassignment of partition [smackTopic, 2] completed successfully
    Reassignment of partition [smackTopic2, 0] completed successfully
```

```
Reassignment of partition [smackTopic2, 1] completed successfully
Reassignment of partition [smackTopic2, 2] is in progress
```

To confiscate a server from the Kafka cluster, we have to move the replica for all partitions hosted on the server to be confiscated to the remaining brokers. We can also use the `kafka-reassign-partitions.sh` tool to increase the partition's replication factor with the following commands:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# cat increase-replication-
factor.json
{"partitions":[{"topic":"smackTopic","partition":0,"replicas":[2,3]}],
    "version":1 }
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-reassign-
partitions.sh --zookeeper localhost:2181 --reassignment-json-file increase-
replication-factor.json --execute
```

This command assumes that partition 0 of the `smackTopic` has replication factor 1 (replica is on broker 2); and now it increases the replication factor to 2 and creates the new replica on the next: broker 3.

Kafka topic tools

When Kafka creates topics, it uses the default number of partitions (1) and the default replication factor (1). In real life, to redefine these parameters we use this command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-topics.sh --create -
-zookeeper localhost:2181/chroot --replication-factor 3 --partitions 10 --
topic smackTopic
```

We can read this command as: replication factor 3 means that up to two servers can fail before data access is lost. ten partitions are defined for a topic, which means the full dataset will be handled by no more than ten brokers excluding replicas.

To modify existent Kafka topics we use this command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-topics.sh --alter --
zookeeper localhost:2181/chroot --partitions 20 --topic smackTopic
```

With this command, we are adding ten more partitions to the topic created in the previous example:

We use the following command to delete a topic:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-topics.sh --delete --
zookeeper localhost:2181/chroot --topic smackTopic
```

Using the `kafka-topics.sh` utility, the configuration can also be added to the Kafka topic:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-topics.sh --alter --
zookeeper localhost:2181/chroot --topic smackTopic --config <key>=<value>
```

To remove a configuration from a topic, we use the following command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-topics.sh --alter --
zookeeper localhost:2181/chroot --topic smackTopic --deleteconfig
<key>=<value>
```

Kafka has one utility to look for the list of topics in the server. By querying Zookeeper, the list tool provides a listing of topics and information about partitions, replicas, and leaders.

To obtain a list of topics, we use the following command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-topics.sh --list --
zookeeper localhost:2181
```

The result table of this command has the following headers:

- **leader**: The randomly selected node for a specific portion of the partitions, responsible for the reads and writes on this partition.
- **replicas**: Node list that holds the log for a specified partition.
- **isr**: The subset of the in-sync list of replicas that is currently alive and in-sync with the leader

Cluster mirroring

Mirroring is used to create a replication of an existing cluster, for example replicating an active data center into a passive data center. Kafka has the mirror maker tool for mirroring the source cluster into a target cluster.

To mirror the source cluster, bring up the target cluster and start the mirror maker processes with the following command:

```
    [master@localhost kafka_2.10.0-0.0.0.0]# bin/kafka-run-class.sh
kafka.tools. MirrorMaker --consumer.config sourceClusterConsumer.config --
num.streams 2 --producer.config targetClusterProducer.config --
whitelist=".*"
```

Kafka also has tools to check the position of the consumers while mirroring or in general. The tool shows the position of all the consumers in a consumer group and how far they are to the log's end. It also indicates how well the cluster mirroring is performing. This tool is used as follows:

```
    [master@localhost kafka_2.10.0-0.0.0.0]#bin/kafka-run-class.sh
kafka.tools. ConsumerOffsetChecker --group MirrorGroup --zkconnect
localhost:2181 --topic kafkatopic
```

Summary

This was a complete review of Apache Kafka and we have touched upon many important facts about it. We have also learned the reason why Kafka was developed, Kafka installation, and its support for different types of clusters. We also explored Kafka's design approach, and wrote a few basic producers and consumers.

In this chapter, we learned how to set up a Kafka cluster with single and multiple brokers on a single node, run producers and consumers from the command line, and exchange some messages. We also discussed important settings about the broker. Finally, we discussed Kafka's integration with technologies such as Spark.

In the next chapter we will review some integration patterns with examples. Take a look at `Chapter 7`, *Study Case 1 - Spark and Cassandra*, where we take a look at an in-depth example of Kafka integration with the other technologies.

6
The Manager - Apache Mesos

If we stop to analyze our current point in the book and we look back, we now know that we have technologies that require more memory and CPU than traditional computers can offer. How is this achieved at low cost? The answer is Apache Mesos.

In addition to the explanation of the Apache Mesos architecture and principles, we explore how to run the principal frameworks and how to run Spark, Cassandra, and Kafka on Apache Mesos.

This chapter has the following sections:

- The Apache Mesos architecture
- Resource allocation
- Running a Mesos cluster on AWS
- Running a Mesos cluster on a private data center
- Scheduling and management frameworks
- Apache Spark on Apache Mesos
- Apache Cassandra on Apache Mesos
- Apache Kafka on Apache Mesos

The Apache Mesos architecture

Mesos is an open source platform for sharing the resources of commodity machines between different distributed applications (later we see they are called frameworks in the Mesos ecosystem), such as Spark, Cassandra, and Kafka among others. Mesos objective is to run as a centralized cluster manager that pools all the physical resources of each cluster member and makes them available as a single source of highly available resources for all the different applications.

Let's take a simple example, a startup has bought eight machines for its humble data center, each one has 8 CPUs and 64 GB of RAM, and previously had a four node cluster where each machine had 4 CPUs and 16 GB of RAM. With Apache Mesos, we can make a virtual cluster that emulates a single machine with (8*8 + 4*4) 80 CPUs and (8*64 + 4*16) 576 GB of RAM. So easily we can have at our fingertips the power of ancestral mainframes. On this cluster we can run multiple distributed applications. The sharing of resources improves the hardware utilization and eliminates replication (of processes and data) across the cluster.

Since the beginning, Mesos objectives have been:

- **Efficiency**: Mesos schedules (like an operating system) and manages the CPU and the memory across the applications (frameworks)
- **High availability**: Here (again) Apache Zookeeper comes to the rescue
- **Monitoring Interface**: Mesos has a fancy, non-blocking web UI for cluster monitoring
- **Resource isolation**: Special mention to Linux and Docker container architectures
- **Scalability**: To date, it indicates that the current version supports up to 50,000 nodes

Mesos is based on the Linux kernel principle and provides an efficient, highly available, scalable, and fault-tolerant base for enabling various frameworks to share cluster resources efficiently in isolation. As we know, the distributed applications are diverse and in continuous evolution.

One Mesos design goal is to expose a thin interface that allows an efficient resource allocation among different frameworks. Mesos delegates the job scheduling and execution to the applications (frameworks) themselves.

So, we achieve two advantages:

- **The choice of application**: Developers can focus, for example, on data locality or fault tolerance depending on the framework needs
- **Design**: It allows Mesos to be scalable, flexible, robust, and agile

The architecture of Mesos delegates the task scheduling responsibility to the frameworks. The Mesos master node decides the quantity of resources to offer each framework and each application decides which resources to accept and which tasks to execute on these assigned resources. This resource allocation method has proved to achieve a good degree of data locality for each framework on the same cluster.

Mesos has an alternative design because it implements a global scheduler whose process is: take the frameworks requirements, organize priorities, and check the resource availability then it produces a detailed schedule of frameworks and resources, for example, functions as weighting between the resources availability, requesting frameworks, and priorities between applications.

One Mesos challenge was the development of an API where the frameworks could express their resources requirements, and concurrently attend millions of jobs and schedule them. Another challenge for the Mesos creator is to anticipate and enable the creation of new frameworks.

Frameworks

A Mesos framework is the interface between Apache Mesos and the application.

The framework is a layer where the task is scheduled and executed. As the framework implementation is application specific, the terms framework and application are used indistinctly. In the beginning, a Mesos framework could only interact with the Mesos API using the C++ library libmesos, so the bindings for other languages such as Java, Scala, Python, and Go were developed to use the libmesos. Since Mesos v.0.19.0, there was introduced an HTTP-based protocol to enable developers to use their preferred language without the need of the libmesos library invocation.

A framework has two parts: scheduler and executor.

- **Scheduler**: Responsible for making decisions on the resource offers made and tracking the current state of the cluster.

  Communication with the Mesos master is made through the `SchedulerDriver` module, whose responsibilities are: registering the framework with the master, task launching, and message passing to the other components

- **Executor**: Responsible for the slave node's task execution

  Communication with the Mesos slaves is made through the `ExecutorDriver` module, whose responsibility is to send status updates to the scheduler

As we can infer, the Mesos API allows programmers to build their own frameworks to run its applications on Mesos. Given the vastness of the subject, in this book we only see the API methods, but not how to build our own framework. For example, we will see how to create with the API the administration, authentication, and authorization of users.

Existing Mesos frameworks

Every day new frameworks are added to communicate new applications with Mesos. In this section, we mention some important frameworks for the purposes of this book, the full list can be found at:

`http://mesos.apache.org/documentation/latest/frameworks/`

Frameworks for long running applications

The following are the frameworks for long running applications:

- **Marathon**: A **Platform as a Service** (**PaaS**) built on Mesos that handles software and hardware failures and ensures that an application is always running.
- **Aurora**: A service scheduler that runs on Mesos. It enables the running of services for long-running jobs taking advantage of the Mesos fault-tolerance, scalability, and resource isolation.
- **Singularity**: A scheduler with an HTTP API and a web interface for running Mesos tasks, long-running processes, scheduled jobs, and single tasks.
- **SSSP**: A simple web application that enables the massive upload to store and share files on Amazon S3.

Frameworks for scheduling

The following are the frameworks for scheduling:

- **Chronos**: A distributed job scheduler that supports complex topologies, the most radical developers refer to it as the alternative to the cron command for the new millennium.
- **Jenkins:** A server for continuous integration. The Mesos-Jenkins plug-in allows dynamically according to the workload, launch workers on a Mesos cluster.
- **JobServer:** A distributed job scheduler and processor. It allows you to build custom batch processing *Tasklets* using a fancy *point and click* web UI.

Frameworks for storage

The following are the frameworks for storage:

- **Cassandra**: The C in the SMACK stack.
- **Elastic**: (Formerly ElasticSearch) A distributed search engine, Mesos allows to scale it, producing a top-notch seek n' destroy engine.
- **Solr**: An enterprise search platform built on Apache Lucene. In recent years. Elastic and Solr fought a battle for supremacy in the world of **Search Engine Optimization** (**SEO**).
- **Storm**: A distributed real-time computation system. Storm makes it easy to process *unbounded* data streams. Storm does for real-time processing what Hadoop does for batch processing.

Attributes and resources

The way to describe Mesos slave nodes in the cluster is done through:

Attributes

They are used to describe the slave node information: OS version, hardware type, and so on.

They are expressed as key-value pairs and support three different types: `scalar`, `range`, and `text`.

They are sent with the offers to frameworks. The attribute's syntax is:

```
attributes : attribute ( ";" attribute )*
attribute : text ":" ( scalar | range | text )
```

Resources

The resources also manage three types: `scalar`, `range`, and `set`. A resource represents the offer of a Mesos slave, for example, the amount of memory on a slave. The resource's syntax is:

```
resources : resource ( ";" resource )*
resource : key ":" ( scalar | range | set )
resourceRole : text | "*"
key : text ( "(" resourceRole ")" )?
```

The Mesos master has predefined how to handle: Disks, CPU cores, memory, and ports.

When a slave doesn't offer processors and memory, it won't have its resources advertised to the frameworks. The **Mesos Master User Interface** interprets all the scalars in megabytes, that is, is not decimal, all is 1024 multiple, for example, the value 16000 is displayed as 15.625 GB.

Now we can read the following example expressions:

- resources='cpus:36;mem:65530;disk:512000;ports:[20000-24000];wallet:{x,y,z}'
- attributes='rack:uvw;zone:east;os:centos5;level:12;keychain:[1000-1200]'

For this example, our resources are called `cpus`, `mem`, `disk`, `ports`, and `wallet`:

- The resource `cpus` is a scalar with value `36`
- The resource `mem` is a scalar with value `65530`
- The resource `disk` is a scalar with value `512000`
- The resource `ports` is a range of values 20000 through 24000 (inclusive)
- The resource `wallet` is of type set containing the values `x`, `y`, and `z`

Likewise, our example attributes were called `rack`, `zone`, `os`, `level`, and `keychain`:

- The attribute `rack` with the text value `uvw`
- The attribute `zone` with the text value `east`
- The attribute `os` with the text value `centos5`
- The attribute `level` with the scalar value `12`
- The attribute `keychain` is of type range with the values 1000 through 1200 (inclusive)

The Apache Mesos API

As we already said, Mesos has an API that allows programmers to develop custom frameworks to run over the distributed infrastructure. In this section, we show several API methods, how to access the API will be explained further in detail.

Messages

As mentioned at the beginning of the book, the Actor System is present everywhere and Mesos could not be the exception. Mesos implements the message passing programming model that we already know, allowing the no-blocking concurrent communication between the Mesos components. A simple example close at hand is when the scheduler tells an executor to use certain resources, the executor communicates its status update to the scheduler through messages with information related to the executed tasks.

The **protocol buffers** provide the messages and the mechanism to allow developers to define their custom formats and protocols to enable the communication. The Mesos code is open so we can check the algorithm direct from the source in this URL:

`https://github.com/apache/mesos/blob/master/include/mesos/mesos.proto`

The Executor API

Here we enumerate the Executor API methods, for a full description visit the following URL: `http://mesos.apache.org/api/latest/java/org/apache/mesos/Executor.html`

The Executor API methods are:

- `registered`: When the executor is connected with Mesos we register it with this code:

```
void registered(ExecutorDriver driver,
    ExecutorInfo executorInfo,
    FrameworkInfo frameworkInfo,
    SlaveInfo slaveInfo)
```

- `reregistered`: The code invoked when the executor re-registers with a restarted slave:

```
void reregistered(ExecutorDriver driver, SlaveInfo slaveInfo)
```

- `disconnected`: The code invoked when the executor becomes disconnected from the slave:

```
void disconnected(ExecutorDriver driver)
```

- `launchTask`: The code invoked when a task has been launched on this executor:

```
void launchTask(ExecutorDriver driver, TaskInfo task)
```

- `killTask`: The code invoked when a task running within this executor has been killed:

```
void killTask(ExecutorDriver driver, TaskID taskId)
```

- `frameworkMessage`: The code invoked when a framework message has arrived for this executor:

```
void frameworkMessage(ExecutorDriver driver, byte[] data)
```

- `shutdown`: The code invoked when the executor should terminate all of its currently running tasks:

```
void shutdown(ExecutorDriver driver)
```

- `error`: The code invoked when a fatal error has occurred with the executor and/or executor driver:

```
void error(ExecutorDriver driver, java.lang.String message)
```

Executor Driver API

Here we enumerate the Executor Driver API methods, for a full description visit the following URL:

`http://mesos.apache.org/api/latest/java/org/apache/mesos/ExecutorDriver.html`

The Executor Driver API methods are:

- `start`: This code starts the executor driver:

```
Protos.Status start()
```

- `stop`: This code stops the executor driver:

```
Protos.Status stop()
```

- `abort`: This code aborts the driver so that no more callbacks can be made to the executor:

```
Protos.Status abort()
```

- `join`: This code waits for the driver to be stopped or aborted:

```
Protos.Status join()
```

- `run`: This code starts the driver and immediately calls the `join()` method:

```
Protos.Status run()
```

- `sendStatusUpdate`: This code sends a status update to the framework scheduler:

```
Protos.Status sendStatusUpdate(Protos.TaskStatus status)
```

- `sendFrameworkMessage`: This code sends a message to the framework scheduler:

```
Protos.Status sendFrameworkMessage(byte[] data)
```

The Scheduler API

Here we enumerate the Scheduler API methods, for a full description visit the following URL:

`http://mesos.apache.org/api/latest/java/org/apache/mesos/Scheduler.html`

The Scheduler API methods are:

- `disconnected`: This code is invoked when the scheduler becomes "disconnected" from the master:

```
void disconnected(SchedulerDriver driver)
```

- `error`: This code is invoked when there is an unrecoverable error in the scheduler or driver:

```
void error(SchedulerDriver driver, java.lang.String message)
```

- `executorLost`: This code is invoked when an executor has exited/terminated:

```
void executorLost(SchedulerDriver driver, Protos.ExecutorID
                  executorId, Protos.SlaveID slaveId, int
                  status)
```

- `frameworkMessage`: This code is invoked when an executor sends a message:

```
void frameworkMessage(SchedulerDriver driver, Protos.ExecutorID
                      executorId, Protos.SlaveID slaveId,
                      byte[] data)
```

- `offerRescinded`: This code is invoked when an offer is no longer valid:

```
void offerRescinded(SchedulerDriver driver, Protos.OfferID
                    offerId)
```

- `registered`: This code is invoked when the scheduler successfully registers with a Mesos master:

```
void registered(SchedulerDriver driver, Protos.FrameworkID
                frameworkId, Protos.MasterInfo masterInfo)
```

- `reregistered`: This code is invoked when the scheduler re-registers with a newly elected Mesos master:

```
void reregistered(SchedulerDriver driver, Protos.MasterInfo
                  masterInfo)
```

- `resourceOffers`: This code is invoked when resources have been offered to this framework:

```
void resourceOffers(SchedulerDriver driver,
                    java.util.List<Protos.Offer> offers)
```

- `slaveLost`: This code is invoked when a slave has been determined unreachable:

```
void slaveLost(SchedulerDriver driver, Protos.SlaveID slaveId)
```

- `statusUpdate`: This code is invoked when the status of a task has changed:

```
void statusUpdate(SchedulerDriver driver, Protos.TaskStatus
                  status)
```

The Scheduler Driver API

Here we enumerate the Scheduler Driver API methods, for a full description visit the following URL:

```
http://mesos.apache.org/api/latest/java/org/apache/mesos/SchedulerDriver.html
```

The Scheduler Driver API methods are:

- `abort`: This code aborts the driver so that no more callbacks can be made to the scheduler:

```
Protos.Status abort()
```

- `acceptOffers`: This code accepts the given offers and performs a sequence of operations on those accepted offers:

```
Protos.Status acceptOffers(java.util.Collection<Protos.OfferID>
offerIds, java.util.Collection<Protos.Offer.Operation>
operations, Protos.Filters filters)
```

- `acknowledgeStatusUpdate`: This code acknowledges the status update:

```
Protos.Status acknowledgeStatusUpdate(Protos.TaskStatus status)
```

- `declineOffer`: This code declines an offer in its entirety:

```
Protos.Status declineOffer(Protos.OfferID offerId)
```

- `declineOffer`: This code declines an offer in its entirety and applies the specified filters on the resources:

```
Protos.Status declineOffer(Protos.OfferID offerId,
                           Protos.Filters filters)
```

- `join`: This code waits for the driver to be stopped or aborted, possibly blocking the current thread indefinitely:

```
Protos.Status join()
```

- `killTask`: This code kills the specified task:

```
Protos.Status killTask(Protos.TaskID taskId)
```

- `launchTasks`: This code launches the given set of tasks:

```
Protos.Status launchTasks(java.util.Collection<Protos.OfferID>
                          offerIds,
                          java.util.Collection<Protos.TaskInfo>
                          tasks)
```

- `reconcileTasks`: This code allows the framework to query the status for non-terminal tasks:

```
Protos.Status reconcileTasks(java.util.Collection
                            <Protos.TaskStatus> statuses)
```

- `requestResources`: This code requests resources from Mesos:

```
Protos.Status requestResources(java.util.Collection
                              <Protos.Request> requests)
```

- `reviveOffers`: This code removes all filters previously set by the framework:

```
Protos.Status reviveOffers()
```

- `run`: This code starts and immediately joins (for example, blocks on) the driver:

```
Protos.Status run()
```

- `sendFrameworkMessage`: This code sends a message from the framework to one of its executors:

```
Protos.Status sendFrameworkMessage(Protos.ExecutorID
                                   executorId, Protos.SlaveID
                                   slaveId, byte[] data)
```

- `start`: This code starts the scheduler driver:

```
Protos.Status start()
```

- `stop`: This code stops the scheduler driver assuming no failover:

```
Protos.Status stop()
```

- `stop`: This code stops the scheduler driver:

```
Protos.Status stop(boolean failover)
```

- `suppressOffers`: This code informs Mesos master to stop sending offers to the framework:

```
Protos.Status suppressOffers()
```

Resource allocation

Mesos has a resource allocation module that contains the policy Mesos master uses to determine the quantity of resource offers made to each framework. As developers, we can customize the module to implement our own allocation policy, for example, we can manipulate the priority and weight of resources, to meet the business requirements. We can also develop custom allocation modules.

One objective of the resource allocation module is to ensure fair resource distribution among the frameworks. The efficiency of a cluster manager lies in the choice of the correct sharing policy algorithm.

For example, Hadoop is governed by the max-min fairness allocation algorithm, in which resource requirements are distributed equitably among competitors. The effectiveness of this algorithm is proven in homogeneous environments. Unfortunately, fast data requires heterogeneous environments.

The distribution of resources between frameworks with heterogeneous demands for resources brings an interesting challenge.

At first glance it may seem trivial, but we will explain this point with a simple but illustrative example. Suppose we have two users A and B. Each user A's task requires 2 CPUs and 8 GB of RAM and each user B's task requires 4 CPUs and 2 GB of RAM. As we can see, user A's tasks require more RAM than user B's tasks, and user B's tasks require more CPU than user A's tasks.

The problem is how the resources should be distributed between the two users to achieve the maximum number of tasks running and ensure the most efficient use of the resources. If we use a traditional algorithm it would assign the same amount of resources to both users and that is not what we want. This situation is what we call a heterogeneous environment.

Apache Mesos implements an algorithm called **Dominant Resource Fairness** (**DRF**) and this is the default policy for resource allocation. This algorithm is more effective for heterogeneous environments.

The DRF algorithm

The DRF algorithm is usually taught in the operating systems courses at college level. We know that job scheduling is not only limited to the CPU, there are multiple resources such as memory, network, and disk. Still, simplifying the problem by reducing the types of resources, we can see that the algorithm min-max fairness fails (not strongly, but it is not efficient) this is because some tasks are processor intensive, others are disk intensive, and others are memory intensive.

Here there is a need for a resource scheduling mechanism that provides each user in a heterogeneous environment a fair share of the resources. In a nutshell, the DRF algorithm is an adaptation of the max-min fairness algorithm to systems with heterogeneous resources.

In our example, we have a restricted universe with only two types of resource: CPU and RAM.

For the purpose of simplifying our example, we will assume that resources become immediately available after use them.

Let's continue with our example, for practical purposes we consider that resources are divisible. Now consider that we have in total 12 CPUs and 16 GB of RAM. Each user A's task requires 2 CPUs and 2 GB of RAM. Each user B's task requires 1 CPU and 3 GB of RAM. There are two concepts that we must master before continuing:

- **Dominant resource**: This refers to the resource that the user requires more. In our example, every user A's task would consume two-twelfths of the total CPU and two-sixteenths of the total RAM, as two-twelfths are bigger than two-sixteenths then the CPU is the user A's dominant resource. Every user B's task would consume one-twelfth of the total CPU and three-sixteenths of the total RAM, as three-sixteenths are bigger than one-twelfth then the RAM is the user B's dominant resource.

- **Dominant share**: This is the fraction of the user's dominant resource allocated. In our example the user A's dominant share is two-twelfths because it uses 2 CPUs of 12 available while the user B's dominant share is three-sixteenths because it uses 3 GB of 16 GB available.

One premise of the DRF algorithm is that it always knows the dominant share per user at any time.

On each DRF algorithm's round it offers resources to the competing user with the lowest dominant share.

The algorithm ends when we cannot allocate the resources for one task.

So, we have table 6-1 with the DRF rounds:

Round	User attended	User A resource share CPU, RAM	User A dominant share	User B resource share CPU, RAM	User B dominant share	CPU Total allocation	RAM Total allocation
0		0/12, 0/16	0/12	0/12, 0/16	0/16	0/12	0/16
1	A	**2/12, 2/16**	2/12	0/12, 0/16	0/16	2/12	2/16
2	B	2/12, 2/16	2/12	**1/12, 3/16**	**3/16**	3/12	5/16
3	A	**4/12, 4/16**	**4/12**	1/12, 3/16	3/16	5/16	7/16
4	B	4/12, 4/16	4/12	**2/12, 6/16**	**6/16**	6/16	10/16
5	A	**6/12, 6/16**	**6/12**	2/12, 6/16	6/16	8/16	12/16
6	B	6/12, 6/16	6/12	**3/12, 9/16**	**9/16**	9/16	15/16
7	A	**8/12,8/16**	**8/12**	3/12, 9/16	9/16	11/12	**17/16**

Table 6-1. DRF algorithm run, the shares of the user attended on each round are in bold.

Each row provides the following information:

- **Round**: The iteration number of the DRF algorithm
- **User attended**: The user selected (the competing user with the lowest dominant share in the previous round)
- **Resource share**: The fraction used of the total amount of CPU and RAM

- **Dominant share**: The fraction used by the dominant resource
- **Total allocation**: The sum of the specific resource allocated to all users in the current round

The following shows the explanation:

At beginning of round 0, both users have a dominant share of 0 (no resources allocated). The algorithm could select both users, in this case we selected user A, the final outcome is the same if we select user B (as an exercise, try to fill table 6-1 choosing user B first). So, the rounds are as follows:

1. User A receives 2 CPUs and 2 GB of RAM. At the end of round 1, user A's dominant share is now two-twelfths.
2. As in the end of round 1, the dominant share for user A is two-twelfths and for user B is 0, user B is attended. In round 2, user B receives 1 CPU and 3 GB of RAM. User B's dominant share is now three-sixteenths.
3. As at the end of round 2, the dominant share for user A is two-twelfths and for user B is three-sixteenths, user A is attended. In round 3, user A receives 2 CPU and 2 GB of RAM. User A's dominant share is now four-twelfths.
4. As at the end of round 3, the dominant share for user A is four-twelfths and for user B is five-sixteenths, user B is attended. In round 4 user B receives 1 CPU and 3 GB of RAM. User B's dominant share is now six-twelfths.
5. The process continues until there are no more resources to allocate to run the user tasks. In this case, after step 6, the CPU resources will get saturated the step 7 doesn't allocate any.
6. The process continues when any resources are freed or if the resource requirement changes.

One objective of the DRF is to maximize the minimum dominant share across all users. At the end of this run, DRF worked with the tasks allocating the following:

- Three tasks to user A with a total allocation of 6 CPUs, 6 GB memory, and a dominant share of six-twelfths, 50% of the CPUs.
- Three tasks to user B with a total allocation of 3 CPUs, 9 GB memory, and a dominant share of nine-sixteenths, 56% of the RAM.

Weighted DRF algorithm

In the previous example, we assumed that both users have the same chances of being attended for resources. We can put weights to each user task in order to favor one user over the others. We say that the DRF algorithm is weighted if the resources are not distributed equally among users. The sharing could be weighted per-user and per-resource-level, the first is more popular.

Let's consider the same parameters as in the previous example, just with one modification, user A has a weight of 3, and user B has a weight of 1. That means that user A has three times more resources than user B. If both users have the same weight, the algorithm runs as in the unweighted mode.

So, following we have *Table 6-2*, with the weighted DRF rounds:

Round	User attended	User A resource share CPU, RAM	User A dominant share	User A dominant share weighted	User B resource share CPU, RAM	User B dominant share	CPU Total allocation	RAM Total allocation
0		0/12, 0/16	0/12	0/36	0/12, 0/16	0/16	0/12	0/16
1	A	**2/12, 2/16**	**2/12**	**2/36**	0/12, 0/16	0/16	2/12	2/16
2	B	2/12, 2/16	2/12	2/36	1/12, 3/16	**3/16**	3/12	5/16
3	A	**4/12, 4/16**	**4/12**	**4/36**	**1/12, 3/16**	3/16	5/16	7/16
4	A	**6/12, 6/16**	**6/12**	6/36	1/12, 3/16	3/16	7/16	9/16
5	A	**8/12, 8/16**	**8/12**	**8/36**	1/12, 3/16	3/16	9/16	11/16
6	B	8/12, 8/16	8/12	8/36	**2/12, 6/16**	**6/16**	10/16	12/16
7	A	**10/12,10/16**	**10/12**	**10/36**	2/12, 6/16	6/16	**12/12**	**16/16**

Table 6-2. DRF weighted algorithm run, the shares of the user attended on each round are in bold.

At beginning of round 0, both users have a dominant share of 0 (no resources allocated). The algorithm could select both users, in this case we selected user A, the final outcome is the same if we select user B (as an exercise, try to fill table 6-1 choosing user B first). So, the rounds are as follows:

1. User A receives 2 CPUs and 2 GB of RAM. At the end of round 1, User A's dominant share is now two-twelfths, divided by the weight 3, gives 2/36.

2. As at the end of round 1, the dominant share for user A is 2/36 and for user B is 0, User B is attended. In round 2, user B receives 1 CPU and 3 GB of RAM. User B's dominant share is now three-sixteenths divided by user B's weight 1, gives three-sixteenths.
3. As at the end of round 2, the dominant share for user A is 2/36 and for user B is three-sixteenths, user A is attended. In round 3, user A receives 2 CPUs and 2 GB of RAM. User A's dominant share is now four-twelfths, divided by user A's weight 3, gives 4/36.
4. As at the end of round 3, the dominant share for user A is 4/36 and for user B is three-sixteenths, user A is attended. In round 4, user A receives 2 CPUs and 2 GB of RAM. User A's dominant share is now six-twelfths, divided by user A's weight 3, gives 6/36.
5. As at the end of round 4, the dominant share for user A is 6/36 and for user B is three-sixteenths, user A is attended. In round 5, user A receives 2 CPUs and 2 GB of RAM. The user A's dominant share is now 8/12, divided by the user A's weight 3, gives 8/36.
6. As in the end of round 5, the dominant share for user A is 8/36 and for user B is 3/16, the user B is attended. In the round 6 user B receives 1 CPU and 3 GB of RAM. User B's dominant share is now six-sixteenths divided by the users B's weight 1, gives six-sixteenths.
7. The process continues until there are no more resources to allocate to run the user tasks. In this case, after step 7, the CPU and RAM resources will be depleted.
8. The process continues when any resources are freed or if the resource requirement changes.

At the end of this run, DRF worked with the tasks allocating the following:

- Five tasks to user A with a total allocation of 10 CPUs, 10 GB memory, and a dominant share of ten-twelfths, 83% of the CPUs.
- Two tasks to user B with a total allocation of 2 CPUs, 6 GB memory, and a dominant share of six-sixteenths, 38% of the RAM.

Besides this, we can create custom modules to adapt to organization-specific resource allocation needs. Some important features of the DRF algorithm are:

- **Envy freeness**: The DRF algorithm is free from envy because there is no need for any user to envy the other's resource allocation. As it offers resources to the competing user with the lowest dominant share, all the users have the same chances.

- **Pareto efficiency**: Increasing the controlling interest of a given user, the dominant participation of the others for this resource is reduced proportionately. The allocation for more resources to one user harms the others.
- **Progressive filling**: If we look, the algorithm increases dominant shares for all the users at the same rate. Other algorithms allocate resources based on demand. The DRF ends when a resource is depleted, when the user frees the resource, the other users proceed in a recursive way, the process continues till no user left with dominant share can be increased.
- **Share guarantee**: All the users dominant share allocation is increased at the same rate, so the users are treated equally and are guaranteed a part of at least one resource.
- **Strategy proof**: The users cannot falsify their resource demands. If one user demands more resources, the algorithm works in a way deterrent to this user.

Resource configuration

Sometimes some frameworks don't accept resource offers and reject resource offers due to incorrect configuration on the slaves. A classic example is the Elastic framework that requires ports `9200` and `9300` but the default port in the Mesos slaves is in the range `31000` to `32000`.

We can configure the slaves to make right resource offers to frameworks as following:

1. Add the parameters of the `mesos-slave` command, for example:

```
--resources='ports:[9200-9200,9300-9300]' ...
```

2. Make a file under `/etc/mesos-slave` called `resources` whose content is the resource string with the following command:

```
$ cat /etc/mesos-slave/resources
ports:[9200-9200,9300-9300]
```

Resource reservation

We can also reserve resources on specific slaves. With this, important services guarantee their resource offers from a specific slave. If an important service has to wait a lot for a resource it will have a negative impact on the performance.

We must use the service reservation wisely, because we can be doing what Mesos tries to avoid. The Mesos access control ensures that any framework that requests a resource has the authorization to do it.

Mesos has two methods to make resource reservations:

1. Static reservation.
2. Dynamic reservation.

Static reservation

In this type, we specify the resources to be reserved for specific slave nodes on a specific framework or a framework group. To reserve resources for a particular framework it must be assigned to a role.

Several frameworks can be assigned to a role. Just the frameworks assigned to a specific role can get the offers for the role. First we must define the roles, then assign frameworks to them, and finally set resource policies for each role.

Defining roles

To define a role, we must start the Mesos master with the following parameters:

```
--roles = "role1, role2, role3"
```

We can also define the role with this command:

```
echo role1 > /etc/mesos-master/role
```

Then we have to restart the Mesos master:

```
sudo service mesos-master restart
```

Assigning frameworks to roles

The next step is to assign the frameworks to roles. The way to do it depends on the framework, for example a framework called Marathon is configured using the flag: `–mesos_role`. In the case of HDFS, we should change the `mesos.hdfs.role` in the file `mesos-site.xml` to the value of `role1` defined previously:

```
<property>
  <name>mesos.hdfs.role</name>
  <value>role1</value>
</property>
```

We can specify custom roles setting the `role` option in `FrameworkInfo` to the desired value (default value is *).

Setting policies

To reserve resources on each slave for a specific role, we use the parameter `–resources`. We must use this carefully because the slave level resource policy can cause overhead as the cluster size increases.

For example, if we have 8 CPUs and 32 GB RAM available on a specific slave and we want to reserve 4 cores and 8 GB of RAM for the `role1` role, (recall that the MB is the unit on Mesos) we have to make these changes on the slave parameters:

```
--resources="cpus:8;mem:32768;cpus(role1):4;mem(role1):8192"
```

Then we have to stop the `mesos-slave` with the command:

```
sudo service mesos-slave stop
```

To remove the older state on the slave we run the following command:

```
rm -f /tmp/mesos/meta/slaves/latest
```

Then we restart the `mesos-slave` with the command:

```
sudo service mesos-slave start
```

Dynamic reservation

On static reservation, the reserved resources can't be used by other roles during downtime, neither to be unreserved. Since Mesos version 0.23.0, there is support for dynamic reservation to overcome this drawback. Dynamic reservation allows users to reserve and unreserve resources as needed.

The frameworks can send back the following message to a resource offer (using the `acceptOffers` API) as response:

```
Offer::Operation::Reserve
Offer::Operation::Unreserve
```

The reserve operation

As part of the offer cycle, a framework can reserve resources. With the following code we are reserving 8 CPUs and 16 GB RAM.

The first step is to receive the offer:

```
{
  "id": <offerId>,
  "framework_id": <frameworkId>,
  "slave_id": <slaveId>,
  "hostname": <hostName>,
  "resources": [
  {
    "name": "cpus",
    "type": "SCALAR",
    "scalar": { "value": 8 },
    "role": "*",
  },
  {
    "name": "mem",
    "type": "SCALAR",
    "scalar": { "value": 16384 },
    "role": "*",
  }]
}
```

To specify that we want to reserve 8 CPUs and 16 GB RAM for a specific framework, we use:

```
{
  "type": Offer::Operation::RESERVE,
  "reserve": {
  "resources": [
    {
      "name": "cpus",
      "type": "SCALAR",
      "scalar": { "value":8  },
      "role": <roleName>,
      "reservation": {
        "principal": <frameworkName>
      }
    },
    {
      "name": "mem",
      "type": "SCALAR",
      "scalar": { "value": 16384 },
      "role": <roleName>,
      "reservation": {
        "principal": <frameworkName>
      }
    }
  ] }
}
```

So, on the next resource offer will be included our reservation data:

```
{
  "id": <offerId>,
  "framework_id": <frameworkId>,
  "slave_id": <slaveId>,
  "hostname": <hostName>,
  "resources": [
    {
      "name": "cpus",
      "type": "SCALAR",
      "scalar": { "value": 8 },
      "role": <roleName>,
      "reservation": {
        "principal": <frameworkName>
      }
    },
    {
      "name": "mem",
      "type": "SCALAR",
```

```
        "scalar": { "value": 16384 },
        "role": <roleName>,
        "reservation": {
          "principal": <frameworkName>
        }
      }
    ]
  }
```

The unreserve operation

As the frameworks have the reserve power, they also have the courtesy to unreserve. With the following code we are unreserving the 8 CPUs and 16 GB RAM:

First, we will receive the reserved resource offer, as follows:

```
  {
    "id": <offerId>,
    "framework_id": <frameworkId>,
    "slave_id": <slaveId>,
    "hostname": <hostName>,
    "resources": [
      {
        "name": "cpus",
        "type": "SCALAR",
        "scalar": { "value": 8 },
        "role": <roleName>,
        "reservation": {
          "principal": <frameworkName>
        }
      },
      {
        "name": "mem",
        "type": "SCALAR",
        "scalar": { "value": 16384 },
        "role": <roleName>,
        "reservation": {
          "principal": <frameworkName>
        }
      }
    ]
  }
```

Next, we specify that we want to unreserve our 8 CPUs and 16 GB RAM as follows:

```
  {
    "type": Offer::Operation::UNRESERVE,
```

```
  "unreserve": {
    "resources": [
    {
      "name": "cpus",
      "type": "SCALAR",
      "scalar": { "value": 8 },
      "role": <roleName>,
      "reservation": {
        "principal": <framework_principal>
      }
    },
    {
      "name": "mem",
      "type": "SCALAR",
      "scalar": { "value": 16384 },
      "role": <roleName>,
      "reservation": {
        "principal": <frameworkName>
      }
    }
    ]
  }
}
```

Similarly to reserve, in the future resource offers will be included our unreserved data as part of the message, so the resource can be offered to other frameworks.

Since Mesos version 0.25.0, the reserve and unreserve methods were introduced as HTTP endpoints as the `/reserve` and `/unreserve` methods. All this to ease the use from HTTP clients, nice isn't it?

In the following paragraphs we explain how to use them.

HTTP reserve

We want to reserve our 8 CPUs and our 16 GB RAM, but we need to send the request through an HTTP POST. We need to use the `/reserve` HTTP endpoint. As true masters of the command line, we use the curl command to send this file using the HTTP protocol:

```
$ curl -i \
-u <operator_principal>:<password> \
-d slaveId=<slaveId> \
-d resources='[ \
{ \
  "name": "cpus", \
  "type": "SCALAR", \
```

```
    "scalar": { "value": 8 }, \
    "role": <roleName>, \
    "reservation": { \
    "principal": <operator_principal> \
  } \
  }, \
  { \
    "name": "mem", \
    "type": "SCALAR", \
    "scalar": { "value": 16384 }, \
    "role": <roleName>,\
    "reservation": { \
    "principal": <operator_principal> \
  } \
  } \
  ]' \
  -X POST http://<master_ip>:<master_port>/master/reserve
```

The Mesos master responds with one these HTTP responses:

- `200 OK`: Successful reservation
- `400 BadRequest`: We have the wrong arguments or missing parameters
- `401 Unauthorized`: The user or password doesn't match or user is not authorized
- `409 Conflict`: Request ok, but the master has not the resources requested

HTTP unreserve

Similarly, we can unreserve our 8 CPUs and our 16 GB RAM sending an HTTP POST request to the `/unreserve` HTTP endpoint:

```
$ curl -i \
-u <operator_principal>:<password> \
-d slaveId=<slaveId> \
-d resources='[ \
{ \
  "name": "cpus", \
  "type": "SCALAR", \
  "scalar": { "value": 8 }, \
  "role": <roleName>, \
  "reservation": { \
  "principal": <operator_principal> \
} \
}, \
{ \
```

```
    "name": "mem", \
    "type": "SCALAR", \
    "scalar": { "value": 16384 }, \
    "role": <roleName>\
    "reservation": { \
    "principal": <operator_principal> \
  } \
  } \
  ]' \
  -X POST http://<master_ip>:<master_port>/master/unreserve
```

The response can be one of the following:

- `200 OK`: Successful unreservation
- `400 BadRequest`: We have the wrong arguments or missing parameters
- `401 Unauthorized`: The user or password doesn't match or user is not authorized
- `409 Conflict`: Request ok, but the master cannot unreserve the resources requested

Running a Mesos cluster on AWS

For Amazon Web Services, Amazon has divided the world in to 11 physical regions and each can be accessed remotely. The services offered have usage-based pricing. And include several services such as EC2 (computing or processing), S3 (Storage), Dynamo DB (the Amazon Database), RDS, EBS, and so on.

AWS includes an EC2 trial to start developing on the platform. The free trial includes a machine with 700 MB of RAM for a year without cost. We need to pay if we need more power (more CPUs or more RAM), or if we want to use the S3 storage service. Prices are at `https://aws.amazon.com/ec2/pricing/`.

- **Amazon account**: To create an account we should go to `http://aws.amazon.com` and follow the instructions. The steps include phone and e-mail verification. The confirmation e-mail contains the account number needed in the following steps.
- **Key pairs**: Amazon uses public-key authentication. We can choose the key pair from a drop-down list or we can create a new one when launching the instance. With the EC2 console we can create the key pair. For each world region a different key pair is needed.

- **Security groups**: The groups act as a firewall for the instances. They control the inbound and outbound traffic to the instances. If we want to manage our instances from a specific IP we need to enable the rule for an SSH connection from the IP specified. We can build specific rules, for example the connection from SSH just from a specific IP. We can find an updated guide at the following URL: `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html`.

AWS instance types

On AWS, the virtual servers are known as instances. We can customize our instances by selecting storage, RAM, CPU, and network capacity. Each instance type includes one or more resource sizes that allow customization and fine-tuning to meet the requirements of our target workload.

The AWS instance types are:

- **General purpose**: For applications that require an even mix of all resources
- **Computer optimized**: An intensive calculation is required and the CPU number matters
- **Memory optimized**: When our applications require a lot of RAM (such as Spark)
- **Storage optimized**: When our applications deal with large volumes of data (such as Cassandra)
- **Micro instances**: Quick trials, fast demos, prototypes, and lightweight applications
- **GPU instances**: We are intrepid and use GPU as a processing unit, not just for graphics

The instances are created from preconfigured templates for example, as suggested by the operating system. These templates are called **Amazon Machine Images** (**AMI**). The AMIs can be either provided by Amazon or can be obtained on the AWS Marketplace. There is a wider community of developers sharing their own AMIs.

AWS instances launching

We can launch instances from the command line and from the EC2 console. To launch an instance from the Amazon EC2 console, we follow these steps:

1. Open the Amazon EC2 console.

2. Click the **Launch Instance** button from the console.
3. From the AMI options, choose **Ubuntu Server 14.04 LTS HVM**, the 64-bit Ubuntu.
4. From the **Choose an Instance Type** page, select the instance that meets your requirements. For this example, we use the m4.xlarge instance with 4 CPUs and 16 GB RAM on each node.
5. On the **Configure Instance Type** page, change the number of instances to 4, because we are running a 4 node cluster.
6. On the **Configure Security Group** page, add Mesos Web UI port `5050` as custom TCP rule, and set `My IP` as the source address, so we restrict other IPs from connecting to the cluster.
7. Skip storage instances and tagging instances.
8. Click on **Launch** button.
9. On this page we will be prompted to choose the private key to log into the machines. As we mentioned, we can create a new key pair or use an existing one.
10. Click on **Download Key Pair** and the file `mesos-cluster.pem` will be downloaded to your machine. We use this file to make an SSH login into the machines.

Installing Mesos on AWS

We need to log in to each one of our four nodes, for example:

```
ssh -i mesos-cluster.pem ec2-54-220-190-120.compute
-1.amazonaws.com (master)
ssh -i mesos-cluster.pem ec2-54-220-191-121.compute
-1.amazonaws.com (slave1)
ssh -i mesos-cluster.pem ec2-54-220-192-122.compute
-1.amazonaws.com (slave2)
ssh -i mesos-cluster.pem ec2-54-220-193-133.compute
-1.amazonaws.com (slave3)
```

We will call the first machine `master` machine and the others `slave1`, `slave2`, and `slave3`.

We need to install the dependencies and software on each machine individually, with these sentences:

Following is the command to update the packages:

```
$ sudo apt-get update
```

We require the JDK to deploy Java projects on Mesos. The following command installs Java 7:

```
$ sudo apt-get install -y openjdk-7-jdk
```

If we are building from the git repository, it will install the tools:

```
$ sudo apt-get install -y autoconf libtool
```

To install the Mesos project dependencies:

```
    $ sudo apt-get -y install build-essential python-dev python-boto
libcurl4-nss-dev libsasl2-dev maven libapr1-dev libsvn-dev
```

Building the Mesos binary is a time consuming task. Instead of building the Mesos binary on each machine, just build it on the master machine and copy it to the slaves.

Downloading Mesos

There are two ways to get Mesos:

- Download the latest stable release from Apache at the URL: `http://mesos.apache.org/downloads/`. At the time of writing this book, the latest version of Mesos is `1.0.0`:

```
restrada@master:~$ wget
http://www.apache.org/dist/mesos/1.0.0/mesos-1.0.0.tar.gz
restrada@master:~$ tar -zxf mesos-1.0.0.tar.gz
restrada@master:~$ mv mesos-1.0.0 mesos
```

- For the hardcore developers, clone the Mesos git repository:

```
restrada@master:~$ git clone https://git-wip-
us.apache.org/repos/asf/mesos.git
```

Building Mesos

To build Mesos, we type the following sentences:

- Go to the Mesos directory:

```
restrada@master:~$ cd mesos
```

- This step is only required if you are building from the git repository (or else you can skip this step):

```
restrada@master:~$ ./bootstrap
```

- Make a build directory that will contain the compiled Mesos binaries:

```
restrada@master:~$ mkdir build
```

- Go to the build directory:

```
restrada@master:~$ cd build
```

- Run configure and make commands:

```
restrada@master:~$ ../configure
restrada@master:~$ make
```

To speed up the building and reduce the log's verbosity, we can append the parameters `-j <number_of_cores> V=0` to our make command. This step could take several hours.

Once the make command is executed (finally!) we test the make with the command:

```
restrada@master:~$ make check
```

This step is optional--we use it if we're installing system-wide.

```
restrada@master:~$ make install
```

Copy the build directory from the master machine to `slave1`, `slave2`, and `slave3`:

```
restrada@master:~$ rsync -za mesos ip-10-154-19-121:
restrada@master:~$ rsync -za mesos ip-10-154-19-122:
restrada@master:~$ rsync -za mesos ip-10-154-19-123:
```

Now, start the Mesos master with this command:

```
restrada@master:~/mesos/build$ ./bin/mesos-master.sh --
work_dir=/var/lib/mesos
```

Start the Mesos slaves:

```
restrada@slave1:~/mesos/build$ ./bin/mesos-slave.sh --master=mesos-
master:5050
restrada@slave2:~/mesos/build$ ./bin/mesos-slave.sh --master=mesos-
master:5050
restrada@slave3:~/mesos/build$ ./bin/mesos-slave.sh --master=mesos-
master:5050
```

By default, the Mesos web UI runs on port `5050` on the master machine, it is how we check the installation. To open the web UI, got to the URL on your web browser: `http://ec2-54-220-190-120.compute-1.amazonaws.com:5050`.

Launching several instances

Apache Mesos has scripts to create EC2 clusters of several configurations. The script is located on the EC2 directory and allows launch, runs jobs, and stops Mesos clusters. We can use this script without building Mesos, we require Python version 2.6 or later. We can manage several clusters with different names.

We need our AWS key pair to use the script, and access to the secret key created before:

```
restrada@local:~ $ export AWS_ACCESS_KEY_ID=<access-key>
restrada@local:~ $ export AWS_SECRET_ACCESS_KEY=<secret-key>
```

To launch a new cluster we use the following sentence:

```
restrada@local:~/mesos/ec2 $ ./mesos-ec2 -k <key-pair> -i
<identity-file> -s 10 launch mesos-cluster
```

This command launches a cluster named `mesos-cluster` with ten slave servers. We can confirm that the cluster is running by checking the Mesos web UI in `<master-hostname>:8080`.

We can list the script options typing `mesos-ec2 --help`. The options provided by the script are the following:

- `--slave or -s`: The number of slaves in the cluster
- `--key-pair or -k`: The SSH key pair for authentication
- `--identity-file or -i`: The SSH identity file used to log in to the instances
- `--instance-type or -t`: A slave instance type, 64-bit
- `--master-instancetype or -m`: The master instance type, 64-bit
- `--ebs-vol-size`: The size of an EBS volume, used to store persistent HDFS data

- `--zone or -z`: The Amazon availability zone to launch instances
- `--resume`: Resumes the installation from the previous run

We can log into the launched cluster specifying the cluster name:

```
restrada@local:~/mesos/ec2 $ ./mesos-ec2 -k <key-pair> -i
<identity-file> login mesos-cluster
```

The script sets up an HDFS instance, user by commands in the `/root/ephemeral-hdfs/` directory.

To destroy a cluster we can use the following command, ensure a backup before using it:

```
restrada@local:~/mesos/ec2 $ ./mesos-ec2 destroy ec2-test
```

The script can pause and restart clusters. On the Mesosphere site there is a lot of information about creating clusters on Amazon EC2, Google Cloud, Azure, and other platforms: `http://mesosphere.com`.

Because these topics are very extensive (and useful), the way to generate clusters in Google Compute Engine and Microsoft Azure is not explained in this book.

Running a Mesos cluster on a private data center

If we don't want to use cloud services from Amazon, Google, or Microsoft; we can set up our cluster on out private data center. Here we assume that our data center has four machines, and we are going to install a Mesos cluster on all. We also assume that the each machine has the same CentOS 6.6 Linux.

For this example, our machines' IP addresses are:

```
machine-1: 192.168.2.190
machine-2: 192.168.2.191
machine-3: 192.168.2.192
machine-4: 192.168.2.193
```

For convenience, we chose `machine-1` as the Mesos cluster master and `machine-2`, `machine-3`, and `machine-4` will be slaves.

Mesos installation

To install a multi node Mesos cluster on our data center we follow these steps:

Setting up the environment

As we already seen, we must download and install the libraries and dependencies to run Mesos on CentOS. We need to log onto the machine and run the following commands:

- We need the `wget` command, to install it we type:

```
$ sudo yum install -y tar wget
```

- Mesos > 0.21.0 requires a C++ compiler with full C++11 support (such as GCC > 4.8) which is available via `devtoolset-2`. Fetch the Scientific Linux CERN `devtoolset repo` file.

```
$ sudo wget -O /etc/yum.repos.d/slc6-devtoolset.repo
http://linuxsoft.cern.ch/cern/devtoolset/slc6-devtoolset.repo
```

- To use the Scientific Linux CERN we need to import the CERN GPG key:

```
$ sudo rpm --import
http://linuxsoft.cern.ch/cern/centos/7/os/x86_64/RPM-GPG-KEY-
cern
```

- Download the Apache Maven repo file:

```
$ sudo wget http://repos.fedorapeople.org/repos/dchen/apache-
maven/epel-apache-maven.repo -O /etc/yum.repos.d/epel-apache-
maven.repo
```

- Since Mesos version 0.21.0 is required Subversion v. 1.8 (or later) devel package, it is NOT available in the default repositories. To install the correct version, Add the WANdisco SVN repo file: '`/etc/yum.repos.d/wandisco-svn.repo`' with this content:

```
[WANdiscoSVN]
name=WANdisco SVN Repo 1.8
enabled=1
baseurl=http://opensource.wandisco.com/centos/6/ svn-
1.8/RPMS/$basearch/
gpgcheck=1
gpgkey=http://opensource.wandisco.com/RPM-GPG-KEY-WANdisco
```

- Now install the development tools with utilities such as the `make` command:

```
$ sudo yum groupinstall -y "Development Tools"
```

- Download and install `devtoolset-2-toolchain` which includes GCC 4.8.2 and its dependencies:

```
$ sudo yum install -y devtoolset-2-toolchain
```

- Install the Mesos related libraries for CentOS:

```
$ sudo yum install -y apache-maven python-devel java-1.7.0-
openjdk-devel zlib-devel libcurl-devel openssl-devel cyrus-
sasl-devel cyrus-sasl-md5 apr-devel subversion-devel apr-util-
devel
```

- Don't forget to enable the `devtoolset` for the shell:

```
$ scl enable devtoolset-2 bash $ g++ --version
```

Finally, check if we have installed the `gcc+ version 4.8`.

For the Mesos download, follow the same steps from the AWS installation in previous section, *Building Mesos*

Follow the same steps for *Building Mesos* from the AWS installation in previous section.

Once we have the Mesos build directory, copy it from the master to the slaves:

```
machine-1:~$ rsync -za mesos machine-2:
machine-1:~$ rsync -za mesos machine-3:
machine-1:~$ rsync -za mesos machine-4:
```

Start the master

Run the following command from the master to start the Mesos master:

```
machine-1:~/mesos/build$ ./bin/mesos-master.sh --
work_dir=/var/lib/mesos --ip=192.168.2.190
```

Start the slaves

Run the following command from the Slaves to start the slave services:

```
machine-2:~/mesos/build$ ./bin/mesos-slave.sh --
master=192.168.2.190:5050
machine-3:~/mesos/build$ ./bin/mesos-slave.sh --
master=192.168.2.190:5050
machine-4:~/mesos/build$ ./bin/mesos-slave.sh --
master=192.168.2.190:5050
```

Check if the installation is complete opening the Mesos web UI on the port `5050` on the master. Open the browser and go to the following URL: `http://192.168.2.190:5050`.

Process automation

To set up a Mesos cluster, we could run the procedure manually starting the `mesos-slave` on each slave.

But, for those guys running a huge cluster, Mesos includes a set of scripts in the "deploy directory" to deploy on a cluster. These scripts use SSH to make the deployment.

In this example, we will set up a 12 slave cluster (`slave1`, `slave2`, `slave12`) and a `master` node.

First, we have to install the prerequisites on every node. The following command generates an SSH key and copies it to the slaves:

```
master:~ $ ssh-keygen -f ~/.ssh/id_rsa -P ""
master:~ $ ssh-copy-id -i ~/.ssh/id_rsa.pub slave1
master:~ $ ssh-copy-id -i ~/.ssh/id_rsa.pub slave2
...
master:~ $ ssh-copy-id -i ~/.ssh/id_rsa.pub slave12
```

Then we have to copy the Mesos build directory to all the nodes on the same location as the master:

```
master:~ $ scp -R build slave1:[install-directory]
master:~ $ scp -R build slave2:[install-directory]
...
master:~ $ scp -R build slave12:[install-directory]
```

Edit the master's file in the route: `[install-directory]/var/mesos/deploy/masters`.

This file lists all the Mesos masters, one per line. In our example, we just have one master:

```
master:~ $ cat [install-directory]/var/mesos/deploy/masters
master
```

Analogously, edit the slave's file listing the machines that we want to be Mesos slaves:

```
master:~ $ cat [install-directory]/var/mesos/deploy/slaves slave1
slave2 slave3 slave4 slave5 slave6 slave7 slave8 slave9 slave10
slave11 slave12
```

Finally, start the cluster with the `mesos-start-cluster` script and the `mesos-stop-cluster` to stop it:

```
master:~ $ mesos-start-cluster.sh
```

This script will call the `mesos-start-masters` and `mesos-start-slaves` that start the appropriate processes on the master and slave nodes. The script looks for the environment configuration in `[install-directory]/var/mesos/deploy/mesosdeploy-env.sh`.

For a better configuration management, specify the configuration options on separate files:

- `[install-directory]/var/mesos/deploy/mesos-master-env.sh`
- `[install-directory]/var/mesos/deploy/mesos-slave-env.sh`

Common Mesos issues

Often, when we install Mesos, some errors are thrown at the configure step. Here we enumerate common issues on the Mesos set up.

Missing library dependencies

Often, the `libz-dev` package is missing, showing this message:

```
configure: error: cannot find libz
libz is required for Mesos to build.
```

When we have a missing package error, we need to install it manually and execute the `configure` command again.

For example, to resolve the missing `libz` library, we have to type in the following on Ubuntu:

```
$ sudo apt-get install libz-dev
```

And on CentOS the command is as follows:

```
$ yum install zlib-devel
```

Directory permissions

Mesos will try to write in `/var/lib/mesos` but if you missed assigning the permissions on the directory, it leads to the following error:

```
mkdir: cannot create directory '/var/lib/mesos': Permission denied
```

To solve it, assign proper permission to this directory:

```
$ sudo chown 'whoami' /var/lib/mesos
```

Missing library

The following error is:

```
configure: error: libmesos*.so not found
```

To solve it, the best way is to copy manually the `libmesos*.so` from the Mesos installation to the `/lib` directory.

This message is another expression of this error:

```
/home/restrada/mesos/build/src/.libs/test-executor:
error while loading shared libraries: libmesos-1.0.0.so:
cannot open shared object file: No such file or directory
```

Debugging

If our cluster is not properly configured we could have a framework with bugs that will not succeed in its execution. To see this, open the Mesos Web Interface and on the Frameworks tab click the **failed framework**. You will see the tasks marked as **KILLED**.

Directory structure

Every slave machine in the Mesos work directory is by default on `/tmp/mesos`.

Under `/tmp/mesos/slaves/` we will find the `slaveId` which keeps track of the frameworks running on it on the `frameworks` directory.

Each attempt to run a task is logged on the `runs` subdirectory, on the classic `stderr` and `stdout` files.

To solve our issue, go to the failed frameworkID (obtained from the Mesos Web Interface) and check the `stderr` file, which will show the log with our issue.

Here we have an example:

```
$ cat /tmp/mesos/slaves/
63033100-1d1d-4490-a3dd-a4f55fe66a44-S3/
frameworks/ed4b4921-6c5b-1b15-6574-4a937539276503d-
0002/executors/default/runs/latest/stderr
mkdir: cannot create directory '/var/lib/mesos': Permission denied
```

VoilÃ , from this log we can deduce that the problem is with permissions.

Slaves not connecting with masters

When we have a connection problem we can proceed in two ways:

- Test the network interface between master and slaves (black box approach). Test if the master and slaves bind to the correct network interface. Always it is safer to use the `--ip` option.
- Look into the logs (white box approach). Open the logs (as shown previous) and try to find messages such as: connection refused, operation timed out, network unreachable, and so on.

Multiple slaves on the same machine

When we try to launch several slave processes on the same port, we have the exception:

```
Failed to initialize, bind: Address already in use [98]
```

To launch several slaves on the same machine we have to specify a different `port` and `workdir`.

```
./mesos-slave.sh --master=<master_ip>:<master_port> --ip=<slave_ip>
--work_dir=<different_work_dir> --port=<different_port>
```

Recall that running several processes on the same node is recommended only on test clusters.

Scheduling and management frameworks

Here we mention some popular Mesos frameworks to deploy, discover, balance load, and handle failure of services. Unlike other application frameworks, these frameworks are used to service management.

In this context, we define two concepts as follows:

- **Load balancing**: To ensure an equitable workload distribution among the instances.
- **Service discovery**: To keep track of the instances on which a particular service is running

Some important Mesos frameworks are:

- **Marathon**: Framework to launch and manage long-running applications
- **Chronos**: A cluster scheduler
- **Apache Aurora**: Framework to manage long-running services and cron jobs
- **Singularity**: Platform-as-a-service (PaaS) for running services
- **Marathoner**: Service discovery for Marathon
- **Consul**: Framework for orchestration and service discovery
- **HAProxy**: Framework used for load balancing
- **Bamboo**: Framework to automatically configure HAProxies
- **Netflix Fenzo**: A task scheduler
- **Yelp's PaaSTA**: A PaaS for running services.

In this section we review Marathon, Chronos, Aurora, and Singularity.

Marathon

The Marathon framework features are as follows:

- A framework for long running applications is a replacement for `init`, `upstart`, and `init.d` of traditional systems
- Can control a high-availability environment and check the application's health
- Comes with a **REST** endpoint (**Representational State Transfer**) to start, stop, and scale applications
- Can automatically scale up and downsize the cluster based on the workload
- Able to start a new instance in case the one available goes down
- Designed to run other technologies on it: Hadoop, Kafka, Storm, and so on
- Ensures that an application started through it keeps running if the slave goes down
- Can have multiple schedulers but there is only one leader, if an application requests a non-leader, the request is dispatched to the active leader
- Can be used in conjunction with HAProxy for service discovery and load balancing
- Supports basic authentication mechanisms, uses SSL to encrypt connections

Marathon installation

For installing Marathon, go to `https://mesosphere.github.io/marathon/` and download the latest Marathon release, at the time of writing this book, the latest version is 1.1.1. Download Marathon as follows:

```
$ wget http://downloads.mesosphere.com/marathon/v1.1.1/marathon-
1.1.1.tgz
```

Once download completes, extract the files:

```
$ tar xf marathon-1.1.1.tgz
```

You can see the following files once you extract Marathon:

```
restrada@master:~ /marathon-1.1.1$ ls
bin Dockerfile docs examples LICENCE README.md target
restrada@master:~ /marathon-1.1.1$ ls bin/
build-distribution
run-tests.config-template.sh
servicerouter.py
haproxy-marathon-bridge
```

```
run-tests-in-loop.sh
start
marathon-framework
run-tests.sh
```

We can run Marathon in a development mode, so we don't need a Mesos installation, this is called Marathon local mode and it should be used just for test purposes, not recommended on production. As usual we need to install Zookeeper (as in Cassandra and Kafka) to store the Marathon state.

Installing Apache Zookeeper

To save the Marathon state we require Zookeeper:

1. Go to `https://zookeeper.apache.org` to check the latest version of Zookeeper, at the time of writing this book, the current version is 3.4.8.
2. Download Zookeeper with this command:

```
$ wget https://archive.apache.org/dist/zookeeper/zookeeper-
3.4.8/zookeeper-3.4.8.tar.gz
```

3. Extract the archive as here:

```
$ tar xf zookeeper-3.4.8.tar.gz
```

4. To configure Zookeeper, edit the file `conf/zoo.cfg` with the following contents:

```
tickTime=2000
dataDir=/var/zookeeper
clientPort=2181
```

5. To start Zookeper, run the following command:

```
$ bin/zkServer.sh start
```

You can see the following messages once you start it successfully:

```
restrada@master:~/zookeeper-3.4.8$ bin/zkServer.sh start
ZooKeeper JMX enabled by default
Using config: /home/restrada/zookeeper-3.4.8/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
```

Running Marathon in local mode

This command launches Marathon in local mode:

```
$ ./bin/start --master local --zk zk://localhost:2181/marathon
```

You will see the following message:

```
INFO All services up and running. (mesosphere.Marathon.Main$.main)
```

When Marathon is running, we can visit the Marathon UI by pointing the browser to the `8080` port.

Multi-node Marathon installation

For this part we need a high-availability Mesos cluster running. We are going to install Marathon on the cluster master machines. To install Marathon, log in to the master machines and type these commands:

On `Debain/Ubuntu`, run these commands:

- To update the repositories:

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --
recv E57252BF
$ DISTRO=$(lsb_release -is | tr '[:upper:]' '[:lower:]')
$ CODENAME=$(lsb_release -cs)
```

- To add the Mesosphere repository:

```
$ echo "deb http://repos.mesosphere.com/${DISTRO} ${CODENAME}
main" | \
sudo tee /etc/apt/sources.list.d/mesosphere.list
$ sudo apt-get update
```

- To install Marathon:

```
$ sudo apt-get -y install marathon
```

On `RedHat/CentOS`, run these commands:

```
$ sudo yum -y install marathon
```

Now open the browser and go to the master machine on the `8080` port to look at the Marathon web UI.

Running a test application from the web UI

To launch a test application from the Marathon web user interface:

1. Click the + Create button.
2. The ID is used to identify the task, name it `marathonTest`.
3. Indicate the number of CPUs required for the job, here we use one.
4. RAM is expressed in MB, we use the default: 16 MB.
5. The number of instances for our test application is 1.
6. Type this bash script in the command text box:

```
while [ true ] ; do echo 'Running my first Marathon test' ;
sleep 5 ; done
```

If everything is Ok, you'll see the `marathonTest` application status as *Running*.

Application scaling

As you can see, we gave our application just one instance. Clicking on the button **Scale Application** we can change the number of instances.

Terminating the application

To terminate the `marathonTest` application, click on the application name and then click the **Destroy** button. Note that this is an irreversible process and cannot be undone, so the confirmation box.

Chronos

The Chronos framework is a time-based job scheduler, like the typical cron Unix command.

The Chronos key points are:

- Runs on Apache Mesos and is distributed and fault tolerant
- By default, executes shell scripts and commands, but also supports Mesos executors
- Can interact with Apache Hadoop and Apache Kafka
- Can run tasks on Mesos slave (where the execution happens) even if it doesn't have Chronos installed

- Can be used to start a service or run a script on a remote machine and in the background
- The wrapper script can have an asynchronous callback to alert Chronos about the job status
- Some skilled DevOps use Chronos to run Dockerized applications
- Comes with a web user interface to consult the configuration, statistics, history, job status, and retries.

Chronos installation

To install Chronos, log in to any Mesos master machine and execute these commands on:

- On Debain/Ubuntu machines:

  To get Chronos:

  ```
  $ sudo apt-get -y install chronos
  ```

  To start the server:

  ```
  $ sudo service chronos start
  ```

- On `RedHat/CentOS` machines:

  Install Chronos:

  ```
  $ sudo yum -y install chronos
  ```

  Start the server:

  ```
  $ sudo service chronos start
  ```

When the installation is complete, open the web browser on `4400` port to see the Chronos web UI.

Job scheduling

To schedule a new job, follow these steps:

1. Click the **+New Job** button.
2. Fill the **NAME** and **DESCRIPTION** fields.
3. The **COMMAND** is the job to be scheduled (for now, run a sleep command).

4. In the **OWNER** field put the name and e-mail of the DevOps hero whom will be alerted in the morning in the case of job failure (and the fire starts).
5. The field **SCHEDULE** is the frequency at which the job runs (Default: empty, that is, infinity). If the value is zero the task repeats once.
6. Check the summary on the UI to see if everything is Ok.
7. Check the status of the job and see if it is **Running**.
8. Finally, check the Mesos UI (`5050` port) and see the Chronos task.

Chronos and Marathon

Chronos and Marathon is the winning combination to create a top-notch distributed application. As we already know, Chronos is the scheduler and Marathon runs jobs continuously. Both schedulers have a REST endpoint for job management. In this section, we review how to use this endpoint to start, manage, and terminate jobs.

Chronos REST API

We can send HTTP messages to Chronos using the REST API (JSON based). By default, the nodes have a Chronos running at the `8080` port listening for API requests. Here we will learn how to:

- List running jobs
- Start a job manually
- Add a job
- Delete a job

For detailed information go to: `http://mesos.github.io/chronos/docs/api.html`

Listing running jobs

Invoking the `HTTP GET` method on `/scheduler/jobs` we will get a list of the running jobs in JSON format. For example:

```
$ curl -L -X GET localhost:8080/scheduler/jobs
```

The response has this data

- Success count
- Error count

- Last success
- Last error
- Executor
- Parents

Starting a job manually

To start a job manually, send an `HTTP PUT` request to the endpoint `/scheduler/job` with the parameters at the end of the command. For example:

```
$ curl -L -X PUT localhost:8080/scheduler/job/job_1?arguments=-debug
```

Adding a job

To schedule a job, send an `HTTP POST` request to the endpoint `/scheduler/iso8601` with the JSON data of the job. The JSON message must contain these fields:

- `name`
- `command`
- `schedule`
- Number of times to repeat the job
- Start time of the job in ISO 8601 format
- Standard ISO 8601 date time format
- Schedule time zone
- `epsilon`
- `owner`
- `async`

For example:

```
$ curl -L -H 'Content-Type: application/json' -X POST -d '{
  "name": "job_1",
  "command": "echo 'A MAN HAS NO NAME'>> /tmp/job_1_OUT",
  "schedule": "R10/2015-03-21T05:52:00Z/PT2S",
  "epsilon": "PT15M",
  "owner": "jaqen@manyfacedgodtemple.org",
  "async": false
}' localhost:8080/scheduler/iso8601
```

Deleting a job

To delete a job we use `HTTP DELETE` on the endpoint `/scheduler/job/<jobName>`, we get the `jobName` from the running jobs list.

For example:

```
$ curl -L -X DELETE localhost:8080/scheduler/job/job_1
```

Deleting all the job tasks

To delete all the job tasks, we use the `HTTP DELETE` request on `/scheduler/task/kill/<jobName>`.

For example:

```
$ curl -L -X DELETE localhost:8080/scheduler/task/kill/job_1
```

Marathon REST API

We can send HTTP messages to Marathon using the REST API. Here we will learn how to:

- List the running applications
- Add an application
- Change the configuration
- Delete an application

Listing the running applications

Invoking the endpoint `/v2/apps` with the `HTTP GET` request, we get the list of the running applications deployed on Marathon. This method's parameters help as filters for displaying the results:

The endpoint parameters are:

- `cmd`: To filter the applications containing the given command
- `embed`: To specify multiple values several times. It embeds the resources that match the path

For example:

```
$ curl -L -X GET "localhost:8080/v2/apps?cmd=sleep 60"
```

The JSON response is similar to this:

```
{ "apps": [
  {
    "id": "/store/delivery/appTest",
    "cmd": "env && sleep 60",
    "constraints": [[
      "hostname",
      "UNIQUE",
      ""
      ]],
    "container": null,
    "cpus": 0.1,
    "env": {
      "LD_LIBRARY_PATH": "/usr/local/lib/myLib"
    },
    "executor": "",
    "instances": 4,
    "mem": 3.0,
    "ports": [
      15193,
      14678
    ],
    "tasksRunning": 1,
    "tasksStaged": 0,
    "uris": [
      "https://raw.github.com/Mesosphere/Marathon/master/README.md"
    ],
    "version": "2015-03-21T11:11:11.348Z"
  }
]}
```

Adding an application

We use the REST endpoint `/v2/apps` to create and start an application, through an `HTTP POST` request.

The parameters required are:

- `id`: The name of the application
- `cmd`: The command to be executed
- `args`: Optional arguments
- `cpus`: The number of cores for this application

- `mem`: The amount of memory for this application
- `ports`: Reserved for the application
- `instances`: The number of instances to deploy the application

If the given application ID is already running on Marathon, a duplication error will be thrown and no application will be launched.

Changing the application configuration

We use the REST endpoint `/v2/apps/<appId>` to change the configuration of an application through a `HTTP POST` request. To obtain the `appId` value we use the previous methods. When the request is processed, the application's running tasks will be restarted with the new configuration.

The `force` parameter is false by default. Setting to true will override the current deployment.

For example:

```
$ curl -L -X PUT localhost:8080/v2/apps/testApp -d '{
"cmd": "sleep 60",
"constraints": [[
  "hostname",
  "UNIQUE",
  ""
]],
"cpus": "0.5",
"instances": "4",
"mem": "16",
"ports": [ 9000 ]
}'
```

If the update is successful, it will give us a JSON response similar to this:

```
{
  "deploymentId": "6b3492a6-4562-4e39-8412-552eba6caca8",
  "version": "2015-03-21T11:11:34.423Z"
}
```

Deleting the application

We use the REST endpoint `/v2/apps/<appId>` with the `HTTP DELETE` request to destroy an application.

For example:

```
$ curl –X DELETE localhost:8080/v2/apps/testApp
```

Apache Aurora

The Apache Aurora key features are as follows:

- It is a Mesos framework for cron jobs, long-running services, and job management.
- Conceived at Twitter Inc. and later open sourced under Apache license.
- Keeps long-running jobs across a shared resources pool over a long duration. If one machine falls, Aurora reschedules jobs on other healthy machines.
- Not recommended for systems with specific scheduling requirements since it is a scheduler itself.
- Provides coarse grained resources for a specific job at any point of time.
- Supports multiple users.
- Its configuration is specified with a **Domain Specific Language** (**DSL**) to avoid configuration redundancy.

Aurora and Marathon offer similar feature sets, both are classified as *service schedulers*. There are three main differences:

- **Ease of use**: Aurora is not easy to install. It exposes a thrift API, which means you'll need a thrift client to interact with it programmatically. On the other hand, Marathon helps you run `Hello World` as quickly as possible. It has great docs to do this in many environments and there's little overhead to get going. It has a REST API and Marathon uses JSON for configuration.
- **Targeted use cases**: Aurora is designed to handle large organizations. For example, Twitter clusters have tens of thousands of machines and hundreds of engineers using them. Marathon has built out features quickly, but is felt a lack of productive experience, a good example is the Docker support. Marathon does not provide preemption.

- **Ownership**: Aurora is owned by the Apache Software Foundation, it means it is subject to the governance model of the ASF, driven by the community. Aurora does not have paying customers, and there is not currently a software company receiving payments for its development. Marathon is owned by Mesosphere, the Mesos Company. This may seem beneficial in some aspects, because a company can pay to receive support and features. It also implies that there is a commercial interest and the project direction is decided by Mesosphere's interests.

The recommendation is: if you are learning to schedule services on Mesos, Marathon is a good beginning because it is easy to learn, to install, and the relationship with the rest of the Mesos ecosystem is natural. If you need to go to production formally with a company, the suggestion is to consider Aurora.

Installing Aurora

Aurora has a web user interface and a command-line utility. We require `vagrant` to run Aurora:

- Install vagrant with the following command:

```
$ sudo apt-get install vagrant
```

- Log in to any of the machines on the cluster and clone the Aurora repository with this command:

```
$ git clone git://git.apache.org/aurora.git
```

- Change the working directory:

```
$ cd aurora
```

- To install Aurora on the machine, type the following command:

```
$ vagrant up
```

The `vagrant` uses the configuration shipped with the Aurora distribution to install and start the Aurora services on the virtual machines.

Some `vagrant` features are:

- It downloads, configures, and starts the corresponding Linux virtual machine image
- It installs Mesos and ZooKeeper on the virtual machine with the build tools
- It compiles the Aurora source and builds it on the virtual machine
- It starts the Aurora services on the virtual machine

The process takes some minutes to complete.

If the installation throws this error **Virtual Box not being present on the machine**, try to install it:

```
$ sudo apt-get install virtualbox
```

Singularity

Singularity key points are as follows:

- Acts as an API and a web application
- Conceived at HubSpot and later open sourced under Apache license
- Used to launch and schedule long-running Mesos processes, scheduled jobs, and tasks
- All its components can be considered as a PaaS to end users
- Non-experimented users can use it to deploy tasks on Mesos without so much knowledge
- Shares Mesos features such as fault tolerance, scalability, and resource allocation
- Can run a task scheduler for other Mesos frameworks

Singularity installation

To install Singularity we need to have Docker installed, to do it follow the steps at:

`https://docs.docker.com`

Once installed, clone the singularity repository with this command:

```
$ git clone https://github.com/HubSpot/Singularity
```

Change the `singularity` directory:

```
$ cd Singularity
```

Now, use Docker Compose `pull` and `up` commands to test Singularity.

These commands set up the following in the container:

- Mesos master and slave
- Zookeeper
- Singularity
- Baragon service and Baragon agent

It is also possible to install singularity without Docker, compile the source code:

```
$ mvn clean package
```

Once the command is done, this is the content of the `SingularityService/target` directory:

```
classes
generated-resources
generated-sources
generated-test-sources
jacoco.exec
maven-archiver
maven-status
SingularityService- 0.10.0-SNAPSHOT.jar
SingularityService- 0.10.0-SNAPSHOT-shaded.jar
SingularityService- 0.10.0-SNAPSHOT-sources.jar
SingularityService- 0.10.0-SNAPSHOT-tests.jar
SingularityService- 0.10.0-SNAPSHOT-test-sources.jar
```

To run singularity we use `SingularityService-0.10.0-SNAPSHOT-shaded.jar`.

The Singularity configuration file

The Singularity configuration is in a YAML file. For this example to run `SingulartiyService`, we use the port `7099`:

```
server:
  type: simple
  applicationContextPath: /singularity
  connector:
    type: http
```

```
    port: 7099
  requestLog:
    appenders:
      type: file
      currentLogFilename: /var/log/singularity-access.log
      archivedLogFilenamePattern: /var/log/singularity-access-%d.log.gz
```

For the Mesos configuration here we put the content from `/etc/Mesos/zk` as Mesos master:

```
mesos:
master:
  zk://100.76.90.36:2181,100.76.126.34:2181,100.72.150.2:2181 /Mesos
```

The number of CPUs that will be used by the job:

```
defaultCpus: 1
```

The memory used by the job on MB:

```
defaultMemory: 128
frameworkName: Singularity
frameworkId: Singularity
frameworkFailoverTimeout: 1000000
```

Quorum is the `host:port` comma separated list:

```
Zookeeper:
  quorum: 10.110.31.101:2181,10.110.31.102:2181,10.110.31.103:2181
zkNamespace: singularity
sessionTimeoutMillis: 60000
connectTimeoutMillis: 5000
retryBaseSleepTimeMilliseconds: 1000
retryMaxTries: 3

logging:
  loggers:
    "com.hubspot.singularity" : TRACE

enableCorsFilter: true
sandboxDefaultsToTaskId: false
```

Enable this using `SingularityExecutor`:

```
ui:
  title: Singularity (local)
  baseUrl: http://localhost:7099/singularity
```

Save this configuration as `singularity_config.yaml` and use this command to start Singularity:

```
java -jar SingularityService/target/SingularityService -*-
shaded.jar server singularity_config.yaml
```

Point the browser to this URL to access the Singularity UI:
`http://<server_IP_Address>:7099/singularity/`.

Apache Spark on Apache Mesos

Here we explain in detail how to run Apache Spark on Mesos.

We have two options:

- We need the Spark binary uploaded to a place accessible by Mesos and Spark driver configured to connect to Mesos
- Install Spark in the same location in all the Mesos slaves and set `Spark.Mesos.executor.home` to point to this specific location

Follow these steps for the first option:

The first time that we run a Mesos task on the Mesos slave, this slave must have the Spark binary package to run the Executor in backend. The location accessible by Mesos could be HDFS, HTTP, or S3.

At the time of writing this book, Spark's version is 2.0.0. To download and upload it on HDFS we use the following command:

```
$ wget http://apache.mirrors.ionfish.org/spark/spark-2.0.0/spark-
2.0.0-bin-hadoop2.6.tgz
$ hadoop fs -put spark-2.0.0-bin-hadoop2.7.tgz /
```

In the Spark driver program, give the master URL as the Apache Mesos master URL in the form:

- For a single Mesos master cluster: `Mesos://master-host:5050`
- For a multimaster Mesos cluster: `Mesos://zk://host1:2181,host2:2181`

There are two ways to submit Spark jobs to Mesos.

Submitting jobs in client mode

The Spark Mesos framework is launched on the client machine, and it waits for the driver output. We need to specify some Mesos configurations in the `Spark-env.sh` file to interact with Apache Mesos:

```
export MESOS_NATIVE_JAVA_LIBRARY=<libMesos.so full path>
export SPARK_EXECUTOR_URI=<SparkURL-2.0.0.tar.gz uploaded before>
```

When starting a Spark application on the cluster, we pass the Mesos URL as the master when we create the Spark Context as follows:

```
val conf = new SparkConf()
  .setMaster("Mesos://<hostname>:5050")
  .setAppName("ApplicationName")
  .set("Spark.executor.uri",
  "<path to Spark-2.0.0.tar.gz uploaded above>")

val sc = new SparkContext(conf)
```

Submitting jobs in cluster mode

In cluster mode, the Spark driver is lunched in the cluster, and the client finds the driver results on the Mesos web UI. We need to start the `MesosClusterDispatcher` to use the cluster mode. The script to start the `MesosClusterDispatcher` is located in `sbin/start-Mesos-dispatcher.sh`.

We can submit jobs to the Mesos cluster specifying the master URL as `Mesos://dispatcher:7077`.

The driver will be available on the Spark Cluster web UI.

For example:

```
./bin/Spark-submit \
--class org.apache.Spark.examples.SparkPi \
--master Mesos://<hostname>:7077 \
--deploy-mode cluster
--supervise
--executor-memory 16G \
--total-executor-cores 64 \
http://path_to_examples/example.jar \
1000
```

The Jars passed to Spark submit should be URIs reachable by Mesos slaves, the Spark driver doesn't automatically upload jars.

Advanced configuration

To run on Mesos, Spark supports two modes:

- **Coarse-grained mode**: The default mode. It will launch one long-running Spark task on each Mesos machine and schedule dynamically mini-tasks on it. This mode is used when we require lower start-up overhead, but it reserves the Mesos resources for all the application duration. We control the resources acquired for Spark in coarse-grained mode setting the `Spark.cores.max` property in `SparkConf`, as by default it acquires all the resources available in the cluster.
- **Fine-grained mode**: Here each Spark task runs a separate Mesos task, allowing better cluster resource shares among frameworks at fine granularity. Each application gets additional or fewer machines depending on the workload. The drawback is that each task launch requires additional overhead. This mode is not recommended for low latency requirements such as web interactive queries. To run in fine-grained mode just turn the coarse-grained mode off through the following property:

```
conf.set("Spark.Mesos.coarse", "false")
```

Specific Apache Spark configuration properties for Apache Mesos can be found at this URL:

`http://spark.apache.org/docs/latest/running-on-mesos.html#configuration`

Apache Cassandra on Apache Mesos

The easiest way to deploy Apache Cassandra on Apache Mesos is through Marathon. Mesosphere has already packaged the Cassandra executor and the necessary JAR files on tarball that can be submitted to Mesos with Marathon with the JSON code located at: `https://downloads.mesosphere.io/cassandra-mesos/artifacts/0.2.0-1/marathon.json`

```
{
  "id": "/cassandra/dev-test",
  "instances": 1,
  "cpus": 0.5,
  "mem": 512,
  "ports": [
    0
```

```
  ],
  "uris": [
"https://downloads.mesosphere.io/cassandra-mesos/artifacts/0.2.0-1/cassandr
a-mesos-0.2.0-1.tar.gz",
    "https://downloads.mesosphere.io/java/jre-7u76-linux-x64.tar.gz"
  ],
  "env": {
    "MESOS_ZK": "zk://localhost:2181/mesos",
    "JAVA_OPTS": "-Xms256m -Xmx256m",
    "CASSANDRA_CLUSTER_NAME": "dev-test",
    "CASSANDRA_ZK": "zk://localhost:2181/cassandra-mesos",
    "CASSANDRA_NODE_COUNT": "3",
    "CASSANDRA_RESOURCE_CPU_CORES": "2.0",
    "CASSANDRA_RESOURCE_MEM_MB": "2048",
    "CASSANDRA_RESOURCE_DISK_MB": "2048",
    "CASSANDRA_HEALTH_CHECK_INTERVAL_SECONDS": "60",
    "CASSANDRA_ZK_TIMEOUT_MS": "10000"
  },
  "cmd": "$(pwd)/jre*/bin/java $JAVA_OPTS -classpath cassandra-mesos-
framework.jar io.mesosphere.mesos.frameworks.cassandra.framework.Main",
  "healthChecks": [
    {
      "gracePeriodSeconds": 120,
      "intervalSeconds": 30,
      "maxConsecutiveFailures": 0,
      "path": "/health/cluster",
      "portIndex": 0,
      "protocol": "HTTP",
      "timeoutSeconds": 5
    },
    {
      "gracePeriodSeconds": 120,
      "intervalSeconds": 30,
      "maxConsecutiveFailures": 3,
      "path": "/health/process",
      "portIndex": 0,
      "protocol": "HTTP",
      "timeoutSeconds": 5
    }
  ]
}
```

We have to adjust this JSON code to point to `MESOS_ZK` and the other parameters. Then save your JSON code in `cassandra-mesos.json`, and then submit it to Marathon with this command:

```
$ curl -X POST -H "Content-Type: application/json" -d cassandra-
mesos.json http://marathon-machine:8080/v2/apps
```

Once the framework is submitted, it will bootstrap itself. We need to expand the port ranges managed by each Mesos node to include the Cassandra standard ports. When starting the process, pass the port ranges:

```
--resources='ports: [31000-32000,7000-7001,7199-7199,9042-
9042,9160-9160]'
```

Cassandra on Mesos provides a REST endpoint to allow fine-tuning, the default port is `18080`.

Advanced configuration

Cassandra running on Mesos takes its configuration through environment variables. To bootstrap the framework configuration, use the following environment variables (read from the framework state stored in Zookeeper):

- Cassandra cluster name, this is part of the framework name in Mesos:

  ```
  CASSANDRA_CLUSTER_NAME=dev-cluster
  ```

- Mesos ZooKeeper URL to locate the leading master:

  ```
  MESOS_ZK=zk://localhost:2181/mesos
  ```

- ZooKeeper URL used to store framework state:

  ```
  CASSANDRA_ZK=zk://localhost:2181/cassandra-mesos
  ```

- Number of nodes in the cluster (default: `3`):

  ```
  CASSANDRA_NODE_COUNT=3
  ```

- Number of seed nodes in the cluster (default: `2`) set this to 1 if you only want to spawn one node:

  ```
  CASSANDRA_SEED_COUNT=2
  ```

- Number of CPU cores for each Cassandra node (default: `2.0`):

  ```
  CASSANDRA_RESOURCE_CPU_CORES=2.0
  ```

- Each Cassandra Node RAM in MB (default: `2048`):

  ```
  CASSANDRA_RESOURCE_MEM_MB=2048
  ```

- Each Cassandra Node Disk in MB (default: `2048`) :

```
CASSANDRA_RESOURCE_DISK_MB=2048
```

- Seconds between each Cassandra node health check (default: `60`):

```
CASSANDRA_HEALTH_CHECK_INTERVAL_SECONDS=60
```

- Default bootstrap grace time (minimum interval between two node starts). Set this to a lower value in local development environments:

```
CASSANDRA_BOOTSTRAP_GRACE_TIME_SECONDS=120
```

- Seconds for the Mesos framework timeout (default: `604800` seconds, that is, 7 days):

```
CASSANDRA_FAILOVER_TIMEOUT_SECONDS=604800
```

- Mesos role used to reserve resources (default `*`). If set, the framework accepts offers for that role or the default role `*`:

```
CASSANDRA_FRAMEWORK_MESOS_ROLE=*
```

- Predefined data directory specifying where Cassandra should write its data. Ensure that this directory can be created by the user on which the framework is running (default: [mesos sandbox]):

```
CASSANDRA_DATA_DIRECTORY=.
```

For more references go to:

- `https://github.com/mesosphere/cassandra-mesos`
- `http://mesosphere.github.io/cassandra-mesos/`

Apache Kafka on Apache Mesos

Ensure that the following applications are available on the machine:

- Java version 7 or later (`http://openjdk.java.net/install/`)
- Gradle (`http://gradle.org/installation`)

To download the Kafka on Mesos project from the repository:

```
$ git clone https://github.com/mesos/kafka
$ cd kafka
$ ./gradlew jar
```

Use this command to download the Kafka executor:

```
$ wget https://archive.apache.org/dist/kafka/0.10.0.0/kafka_2.10-
0.10.0.0.tgz
```

Set the following environment variable to point to the `libmesos.so` file:

```
$ export MESOS_NATIVE_JAVA_LIBRARY=/usr/local/lib/libmesos.so
```

Use the `kafka-mesos.sh` script to launch and configure Kafka on Mesos. But before, create the `kafka-mesos.properties` file with this contents:

```
storage=file:kafka-mesos.json
master=zk://master:2181/mesos
zk=master:2181
api=http://master:7000
```

These properties are used to configure `kafka-mesos.sh` so we don't need to pass arguments to the scheduler all the time. The scheduler supports the following command-line arguments:

`--api`

The API URL, for example `http://master:7000`

`--bind-address`

The scheduler bind address (for example: master, 0.0.0.0, 192.168.50.*, if:eth1). The default is all.

`--debug <Boolean>`

The debug mode, by default is `false`.

`--framework-name`

The framework name, by default is Kafka.

`--framework-role`

The framework role, by default is *.

`--framework-timeout`

The framework timeout (30s, 1m, or 1h), by default is 30d.

`--jre`

JRE zip file (`jre-7-openjdk.zip`), the default is none.

`--log`

The log file to use, the default is `stdout`.

`--master`

The master connection settings, some examples are:

- master:`5050`
- master:`5050`,master2:`5050`
- `zk://master:2181/mesos`
- `zk://master:2181,master2:2181/mesos`

`--principal`

Username used to register the framework, the default is none.

`--secret`

Password used to register the framework, the default is none.

`--storage`

The storage for the cluster state, the default is `file:kafka-mesos.json`. For example:

- `file:kafka-mesos.json`
- `zk:/kafka-mesos`

`--user`

The Mesos user to run tasks, the default is none.

`--zk`

The Kafka zookeeper.connect, for example:

- `master:2181`
- `master:2181,master2:2181`

Now we can start the Kafka scheduler with this command:

```
$ ./kafka-mesos.sh scheduler
```

Now, start the Kafka broker with the default settings with this command:

```
$ ./kafka-mesos.sh broker add 0
```

The cluster has one broker not started, check this with this command:

```
$ ./kafka-mesos.sh broker list
```

Start the broker with the following command:

```
$ ./kafka-mesos.sh broker start 0
```

Now our broker is ready to produce and consume messages. Test the setup with `kafkacat`.

Install the `kafkacat` with this command:

```
$ sudo apt-get install kafkacat
$ echo "test" | kafkacat -P -b "10.213.128.5:31000" -t testTopic -p
0
```

To read back the message pushed to the broker, use this command:

```
$ kafkacat -C -b "10.213.128.5:31000" -t testTopic -p 0 -e
test
```

To add more brokers to the cluster on one command:

```
$ ./kafka-mesos.sh broker add 0..2 --heap 1024 --mem 2048
```

Now we have added three brokers to the cluster, to start these three brokers with one command:

```
$ ./kafka-mesos.sh broker start 0..2
```

To stop one broker use the command:

```
$ ./kafka-mesos.sh broker stop 0
```

To change the Kafka logs location, the broker must be stopped, use this command:

```
$ ./kafka-mesos.sh broker update 0 --options
log.dirs=/mnt/kafka/broker0
```

Kafka log management

To get the last 100 lines of the logs (`stdout`, `default` and `stderr`) use this command:

```
$ ./kafka-mesos.sh broker log 0
```

To read from the `stderr` file use this command:

```
$ ./kafka-mesos.sh broker log 0 --name stderr
```

To read a file in the `*/log/ directory` for example the file `server.log`, use this command:

```
$ ./kafka-mesos.sh broker log 0 --name server.log
```

To read more lines use the `--lines` option:

```
$ ./kafka-mesos.sh broker log 0 --name server.log --lines 200
```

Summary

In this chapter, we have seen the Mesos architecture. We also reviewed how Mesos makes the resource allocation, the DRF algorithm, and reviewed how to run Mesos on AWS and on a private data center.

We visited the most important Mesos frameworks: Marathon, Chronos, Aurora, and Singularity. We also reviewed the Frameworks API.

In the last section, we reviewed how to run Spark, Cassandra, and Kafka on Apache Mesos, also covered topics such as the setup, configuration, and management of these frameworks on a distributed infrastructure using Mesos.

I hope that this chapter has armed you with all the resources that you require to effectively manage the complexities of today's modern DevOps.

The following chapters show study cases, considering Mesos as an infrastructure technology. The examples not focused on Mesos assume you are already running on it.

7
Study Case 1 - Spark and Cassandra

The three last chapters are study cases. In the first study case we discuss the relationship between Spark and Cassandra; in the second study case we explore the relationship among the other technologies; and in the last chapter we analyze the Mesos frameworks and containers.

Remember that in all the examples we use Scala as language and Akka as the actor model. Also Mesos is considered an infrastructure technology, so we assume that all the use cases can be deployed on Mesos and use Scala and Akka.

This chapter has the following parts:

- Spark Cassandra connector
- Study case: The Calliope project

Spark Cassandra connector

To use Apache Spark and Apache Cassandra together, we could develop the calls with our bare hands, but thanks to the open source community coordinated by the DataStax people we have the Spark Cassandra connector. If you remember the history, Cassandra was a project conceived on Facebook that became an Apache project and reached such a size that a whole company was created to support it: DataStax.

DataStax is the company responsible for Apache Cassandra's fate. DataStax has developed, among other useful tools, the Spark-Cassandra connector, which is a powerful open source library that hast three main directives:

1. Expose Cassandra tables as Spark RDDs.
2. Write Spark RDDs to Cassandra.
3. Execute CQL queries within Spark applications.

The Spark-Cassandra connector main features are:

- Supports Apache Spark version 1.0 through 1.6
- Supports Apache Cassandra version 2.0 or later
- Supports Scala versions 2.10 and 2.11
- Supports all the Cassandra data types including collections
- Can convert data types between Scala and Cassandra
- Can expose Cassandra tables as Spark RDDs
- Can map table rows to CassandraRow objects or tuples
- Can map Rows to instances of **User Defined Classes** (**UDC**)
- Can store RDDs into Cassandra calling the elegant method: `saveToCassandra`
- Can make a join with a Cassandra data subset calling the method: `joinWithCassandraTable`
- Can make RDD partitions according to Cassandra replication using the method: `repartitionByCassandraReplica`
- Can filter rows on the server side via the CQL `WHERE` clause
- Can execute *for* cycles over arbitrary CQL statements
- Cassandra Virtual Nodes ready
- Can work with PySpark DataFrames

For obvious reasons, the development of the connector is made after the versions of Apache Spark, Apache Cassandra and Scala are released. Usually, the connector does not support the latest versions.

The Spark Cassandra connector version compatibility is described in table 7-1:

Connector	Apache Spark	Apache Cassandra	Cassandra Java Driver
1.6	1.6	2.1.5, 2.2, 3.0	3.0
1.5	1.5, 1.6	2.1.5, 2.2, 3.0	3.0
1.4	1.4	2.1.5	2.1
1.3	1.3	2.1.5	2.1
1.2	1.2	2.1, 2.0	2.1
1.1	1.1, 1.0	2.1, 2.0	2.1
1.0	1.0, 0.9	2.0	2.0

Table 7-1 Spark-Cassandra connector version compatibility

Requisites

First we have to configure a new Scala project with the Apache Spark dependency. The dependencies are retrieved via the `spark-packages.org` website. For example, if we are using sbt, our `build.sbt` should include something like this:

```
resolvers += "Spark Packages Repo" at
"https://dl.bintray.com/spark-packages/maven"
libraryDependencies += "datastax" % "spark-cassandra-connector" % "1.6.0-
s_2.11"
```

The driver version doesn't depend on the Cassandra server code. Make sure the Connector version you are using coincides with the Spark version (that is, Spark 1.2.x with Connector 1.2.x)

The `spark-cassandra-connector` jar and its dependency jars:

```
"com.datastax.spark" %% "spark-cassandra-connector" % Version
```

Add these jars to the following classpaths:

- Your project classpath
- The classpath of each Spark node on the cluster

Preparing Cassandra

To create a simple `KEYSPACE` called `testKs` and a table in Cassandra run the following `cqlsh` statements:

```
CREATE KEYSPACE testKs WITH replication = {'class': 'SimpleStrategy',
'replication_factor': 1 };
CREATE TABLE testKs.kv(key text PRIMARY KEY, value int);
```

To insert some example data:

```
INSERT INTO testKs.kv(key, value) VALUES ('key1', 1);
INSERT INTO testKs.kv(key, value) VALUES ('key2', 2);
```

Now we are ready to write our first Spark application using Cassandra.

SparkContext setup

To make the SparkContext setup, follow these steps:

1. The first step is to import Spark using:

   ```
   import org.apache.spark._
   ```

2. Set the property `spark.cassandra.connection.host` to point to the address of one of the Cassandra nodes:

   ```
   val conf = new
   SparkConf(true).set("spark.cassandra.connection.host",
   "CassandraNodeIP")
   ```

3. To create the SparkContext, replace `SparkMasterNodeIP` with the Spark master address, use local to run in local mode:

   ```
   val sc = new SparkContext("spark://SparkMasterNodeIP:7077",
   "test", conf)
   ```

4. To enable the Cassandra specific functions on SparkContext, RDD, and DataFrame import:

   ```
   import com.datastax.spark.connector._
   ```

5. To load and analyze data from Cassandra use the `sc.cassandraTable` method to view a Cassandra table as a Spark RDD:

```
val testRDD = sc.cassandraTable("testKs", "kv")
println( testRDD.count )
println( testRDD.first )
println( testRDD.map(_.getInt("value")).sum )
```

Here we save an RDD data to Cassandra, we will add two rows to the table:

```
val col = sc.parallelize(Seq(("key3", 3), ("key4", 4)))
col.saveToCassandra("testKs", "kv", SomeColumns("key", "value"))
```

Cassandra and Spark Streaming

As we saw in the chapter about Spark, Spark Streaming is the Spark module that allows the handling and processing of high throughput and fault tolerant live data streams. We also saw how the data is entered into Spark from multiple data sources such as Akka Streaming, Apache Kafka, Apache Flume, ZeroMQ, RabbitMQ, raw TCP sockets, and so on. We also saw how the results produced by Spark are stored in Cassandra.

Spark Streaming setup

Here we will use the classic Spark Streaming word count example. If we recall the Spark chapter, we output the results with `wordCounts.print()`. The following steps are:

1. Create a `StreamingContext` with the `SparkConf`

```
val scontext = new StreamingContext(sparkConf, Seconds(1))
```

2. Create a `DStream` to connect to server at IP and Port:

```
val lines = scontext.socketTextStream(serverIP, serverPort)
```

3. Count each word in each batch:

```
val words = lines.flatMap(_.split(" "))
val pairs = words.map( word => (word, 1) )
val counts = pairs.reduceByKey(_ + _)
```

4. Print the count to the console, the start() method is to begin the execution:

```
counts.print()
scontext.start()
```

Cassandra setup

To invite Cassandra to this streaming, we have Cassandra specific functions in the StreamingContext and in the RDD. Now simply connect the pipe so that, instead of ending at the console, the output will end to Cassandra. All the abstraction is made by the Spark Cassandra connector. To import the connector:

```
import com.datastax.spark.connector.streaming._
counts.saveToCassandra("streamingTest", "words")
```

Streaming context creation

The second parameter in the Streaming context is the `batchDuration` which sets the intervals that streaming data will be divided into batches. Note that the Spark API supports milliseconds, seconds and minutes, all accepted as duration. Don't confuse this duration with the `scala.concurrent.duration.Duration`

```
val scontext = new StreamingContext(conf, Seconds(n))
```

Stream creation

We can create any stream from the available types or create a custom Spark stream. The Spark Cassandra connector also supports Akka actor streams and it will support more streams types in the future. We can also extend the existing types:

```
import com.datastax.spark.connector.streaming.TypedStreamingActor
```

Kafka Streams

The `kafkastream` creates an input stream that pulls messages from a Kafka Broker:

```
val kafkaStream = KafkaUtils.createStream[String, String, StringDecoder,
StringDecoder]( scontext, kafka.kafkaParams, Map(topic -> 1),
StorageLevel.MEMORY_ONLY)
```

Akka Streams

The `akkastream` creates an input stream that pulls messages from an Akka:

```
val akkaStream =
scontext.actorStream[String](Props[TypedStreamingActor[String]], "stream",
StorageLevel.MEMORY_AND_DISK)
```

Enabling Cassandra

To enable Cassandra-specific functions on the StreamingContext, DStream and RDD we import:

```
import com.datastax.spark.connector.streaming._
```

Write the Stream to Cassandra

In our example `testKs` is the Key space name and our table name is words:

To save the data from the Stream to Cassandra:

```
val wc = stream.flatMap(_.split("\\s+"))
  .map(x => (x, 1))
  .reduceByKey(_ + _)
  .saveToCassandra("testKs", "words", SomeColumns("word", "count"))
```

To start the computation:

```
scontext.start()
```

To save data from an RDD to Cassandra, for example to add two more rows to the table we use:

```
val col = sc.parallelize(Seq(("key3", 3), ("key4", 4)))
col.saveToCassandra("testKs", "kv", SomeColumns("key", "value"))
```

Read the Stream from Cassandra

To read the Streaming Context from Cassandra:

```
val rdd = ssc.cassandraTable("testKs", "key_value")
  .select("key", "value").where("foo = ?", 3)
```

To load and analyze data from Cassandra we use the `sc.cassandraTable` method to view a table as a Spark RDD:

```
val rdd = sc.cassandraTable("testKs", "key_value")
println( rdd.count )
println( rdd.first )
println( rdd.map(_.getInt("value")).sum )
```

Saving datasets to Cassandra

It is possible to save RDDs to Cassandra, not just a Cassandra RDD. The prerequisites are that the object class of RDD is a tuple and has property names corresponding to the Cassandra column names.

It is also possible to save an RDD to an existing Cassandra table. Also, we can let the connector create the appropriate table automatically based on the definition of the RDD items.

To save an RDD on an existing table, we need to import `com.datastax.spark.connector` and call the `saveToCassandra` method with the key space name, table name and the columns list. It is important to include at least all the primary key columns.

To save an RDD into a new table, instead of calling `saveToCassandra` we call `saveAsCassandraTable` or `saveAsCassandraTableEx` with the name of the table we want to create.

Saving a collection of tuples to Cassandra

Assume the following table definition:

```
CREATE TABLE testKs.words (word text PRIMARY KEY, count int);
save("foo", 10);
save("bar", 20);
```

And we have the following Spark code:

```
val collection = sc.parallelize(Seq(("cat", 30), ("dog", 40)))
collection.saveToCassandra("testKs", "words", someColumns("word", "count"))

cqlsh:testKs> select * from words;

 word | count
```

```
------+-------
  foo |    10
  bar |    20
  cat |    30
  dog |    40

(4 rows)
```

Also, a custom mapper is supported for tuples:

```
val collection = sc.parallelize(Seq((30, "cat"), (40, "dog")))
collection.saveToCassandra("testKs", "words", SomeColumns("word" as "_2",
"count" as "_1"))

cqlsh:test> select * from words;

word | count
-----+-------
  foo|    10
  bar|    20
  cat|    30
  dog|    40

(4 rows)
```

Saving collections to Cassandra

When saving a collection of objects of a user defined class, the items to be saved must provide appropriately named public property accessors for getting every column to be saved. This example provides more information on property column naming conventions.

```
case class WordCount(word: String, count: Long)
val collection = sc.parallelize(Seq(WordCount("fox", 50), WordCount("cow",
60)))
collection.saveToCassandra("testKs", "words", SomeColumns("word", "count"))

cqlsh:test> select * from words;

 word | count
------+-------
  foo |    10
  bar |    20
  cat |    30
  dog |    40
  fox |    50
  cow |    60
```

```
(6 rows)
```

We can specify custom column to property mapping with `someColumns`. If the property names in objects to be saved don't correspond to the column names in the destination table, use the `as` method on the column names you want to override. The parameters order is: first table column name, then object property name.

For this example, recall that the table definition in Cassandra is:

```
CREATE TABLE testKs.words (word text PRIMARY KEY, count int);
```

If we want to save the object `wordCount` to the table which has columns word **TEXT** and `num` INT the code in Spark is:

```
case class WordCount(word: String, count: Long)
val collection = sc.parallelize(Seq(WordCount("fox", 50), WordCount("cow",
60)))
collection.saveToCassandra("testKs", "words2", SomeColumns("word", "num" as
"count"))
```

Modifying collections

The default behavior of the Spark Cassandra connector is to overwrite collections when inserted into a Cassandra table. To override this behavior we can specify the instructions on how to treat the collection to the custom mapper.

The operations supported are:

- append/add (lists, sets, maps)
- prepend (lists)
- remove (lists, sets) not supported for maps
- overwrite (lists, sets, maps)

For example let's take the elements from `rddSetField` and remove them from the corresponding C* column `"a_set"`; and take elements from `rddMapField` and add them to C* column `"a_map"` where C* column key == key in the RDD elements:

```
("key", "a_set" as "rddSetField" remove , "a_map" as "rddMapField"
 append)
```

For this example, the schema is:

```
CREATE TABLE testKs .collections_mod (
      key int PRIMARY KEY,
      list_col list<text>,
      map_col map<text, text>,
      set_col set<text>
  )
```

To append and prepend lists we use:

```
val listElements = sc.parallelize(Seq(
  (1,Vector("One")),
  (1,Vector("Two")),
  (1,Vector("Three"))))

val preElements = sc.parallelize(Seq(
  (1,Vector("PreOne")),
  (1,Vector("PreTwo")),
  (1,Vector("PreThree"))))

listElements.saveToCassandra("testKs", "collections_mod",
SomeColumns("key", "list_col" append))
preElements.saveToCassandra("testKs", "collections_mod", SomeColumns("key",
"list_col" prepend))
```

The result is:

```
cqlsh> select * from testKs.collections_mod where key = 1;

key   | list_col                                    |map_col | set_col
------+---------------------------------------------+--------+----------
   1  | ['PreThree', 'PreTwo', 'PreOne', 'One', | null   | null
         'Two', 'Three']
(1 rows)
```

Saving objects of Cassandra (user defined types)

To store structures with fields having user defined types use a case class or a `com.datastax.spark.connector.UDTValue` class. An instance of this class can be easily obtained from a Scala Map by calling the `fromMap` method.

For example, with the following table definition:

```
CREATE TYPE testKs.address (city text, street text, number int);
CREATE TABLE testKs.companies (name text PRIMARY KEY, address
FROZEN<address>);
CREATE TABLE testKs.udts (key text PRIMARY KEY, name text, addr
FROZEN<address>);
```

We use a case class to make inserts into the UDT:

```
case class Address(street: String, city: String, zip: Int)
val address = Address(city = "San Jose", zip = 95126, street = "Santa
Clara")
val col = Seq((1, "Raul", address))
sc.parallelize(col).saveToCassandra(testKs, "udts", Addresses("key",
"name", "addr"))
```

Or use the `UDTValue.fromMap` to create the UDT:

```
import com.datastax.spark.connector.UDTValue
case class Company(name: String, address: UDTValue)
val address = UDTValue.fromMap(Map("city" -> "Palo Alto", "street" ->
"Infinite Loop", "number" -> 1))
val company = Company("Apple", address)
sc.parallelize(Seq(company)).saveToCassandra("testKs", "companies")
```

Scala options to Cassandra options conversion

Cassandra options can be dealt with as if they were normal Scala options. For the reverse transformation, from a Scala option into a Cassandra option, we need to define the `None` behavior. This is done via `CassandraOption.deleteIfNone` and `CassandraOption.unsetIfNone`.

For example:

1. Import:

```
import com.datastax.spark.connector.types.CassandraOption
```

2. Setup some sample data (1, 1, 1) ... (6, 6, 6):

```
sc.parallelize(1 to 6).map(x =>
(x,x,x)).saveToCassandra(testKs, "table1")
```

3. To setup Scala options RDD (1, None, None) (2, None, None) ... (6, None, None):

```
val optionRdd = sc.parallelize(1 to 6).map(x => (x, None,
None))
```

4. To delete just the middle column:

```
optionRdd.map{ case (x: Int, y: Option[Int], z: Option[Int]) =>
(x, CassandraOption.deleteIfNone(y),
CassandraOption.unsetIfNone(z))}.saveToCassandra(testKs,
"tab1")
```

5. To collect the result:

```
val results = sc.cassandraTable[(Int, Option[Int],
Option[Int])](ks, "tab1").collect
```

```
    (1, None, Some(1)),
    (2, None, Some(2)),
    (3, None, Some(3)),
    (4, None, Some(4)),
    (5, None, Some(5)),
    (6, None, Some(6))
```

Saving RDDs as new tables

As we know, we use the `saveAsCassandraTable` method to automatically create a new table with a given name and save RDDs into it. The key space we are saving to must exist. The following code will create a new table, with `words_new` in the key space `testKs` with columns `word` and `count`, where `word` is the primary key:

```
case class WordCount(word: String, count: Long)
val collection = sc.parallelize(Seq(WordCount("dog", 50), WordCount("cow",
60)))
collection.saveAsCassandraTable("testKs", "words_new", SomeColumns("word",
"count"))
```

To customize the table definition, we call `saveAsCassandraTableEx`. The following code demonstrates how to add another column of type `Int` to the table definition, creating new table `words_new_2`:

```
import com.datastax.spark.connector.cql.{ColumnDef, RegularColumn,
TableDef}
import com.datastax.spark.connector.types.IntType
```

```
case class WordCount(word: String, count: Long)
val table1 = TableDef.fromType[WordCount]("testKs", "words_new")
val table2 = TableDef("testKs", "words_new_2", table1.partitionKey,
table1.clusteringColumns,
table1.regularColumns :+ ColumnDef("additional_column", RegularColumn,
IntType))
val collection = sc.parallelize(Seq(WordCount("dog", 50), WordCount("cow",
60)))
collection.saveAsCassandraTableEx(table2, SomeColumns("word", "count"))
```

The following code creates a table with a custom definition and defines which columns are partitions and which clustering column keys:

1. Import the necessary classes:

```
import com.datastax.spark.connector.cql.{ColumnDef,
RegularColumn, TableDef, ClusteringColumn, PartitionKeyColumn}
import com.datastax.spark.connector.types._
```

2. Define the RDD structure:

```
case class outData(col1:UUID, col2:UUID, col3: Double,
col4:Int)
```

3. Define the columns:

```
val p1Col = new ColumnDef("col1",PartitionKeyColumn,UUIDType)
val c1Col = new ColumnDef("col2",ClusteringColumn(0),UUIDType)
val c2Col = new ColumnDef("col3",ClusteringColumn(1),DoubleType)
val rCol = new ColumnDef("col4",RegularColumn,IntType)
```

4. Create the table definition:

```
val table = TableDef("test","words ",Seq(p1Col),Seq(c1Col,
                    c2Col),Seq(rCol))
```

5. Map RDD into the custom data structure and create the table:

```
val rddOut = rdd.map(s => outData(s._1, s._2(0), s._2(1),
                    s._3))
             rddOut.saveAsCassandraTableEx(table,
                    SomeColumns("col1", "col2", "col3",
                    "col4"))
```

Cluster deployment

The objective of this section is to explain how Spark and Cassandra work together in a cluster. In figure 7-1 we can find a canonical Spark Cassandra cluster:

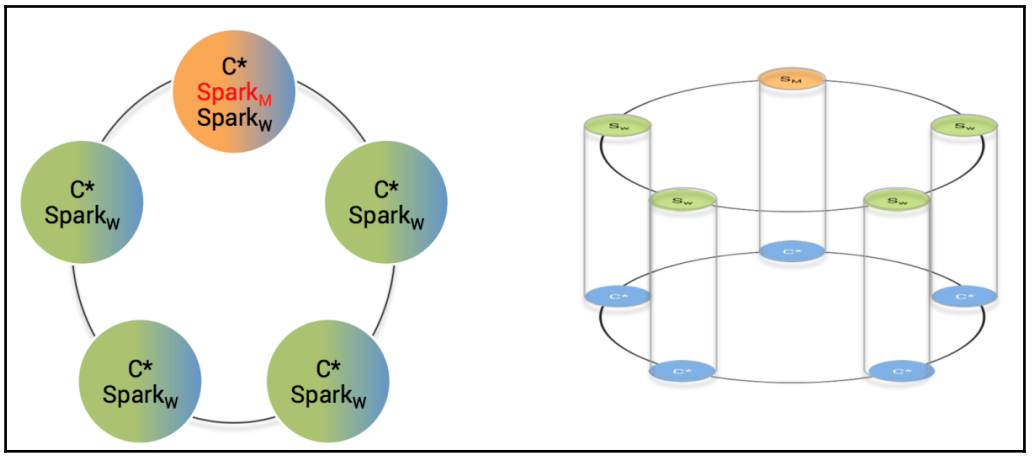

Figure 7-1. Canonical Spark Cassandra cluster

In the canonical Spark Cassandra cluster every Spark Worker has its correspondent Cassandra process.

To explain the steps of the data flow between Spark and Cassandra we use the following figures.

In figure 7-2 we can see that every Spark worker has one and only one Cassandra process associated to it. As we know, one Spark Worker is the cluster Spark Master:

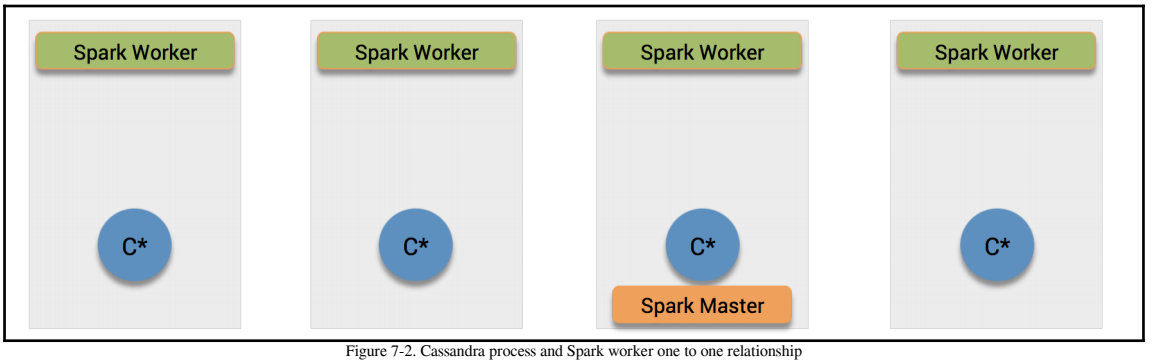

Figure 7-2. Cassandra process and Spark worker one to one relationship

Figure 7-3 shows the first step: To program the business logic in the Driver Program:

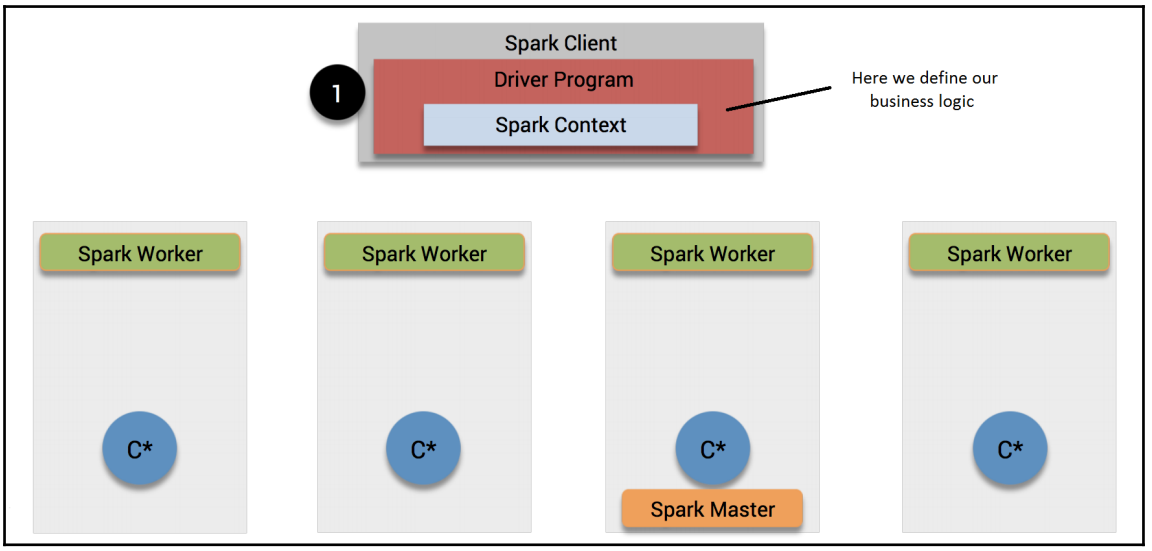

Figure 7-3. Step 1 - Define the business logic

In figure 7-4 we can see the second step: The driver program sends the task to be executed to the cluster Spark Master:

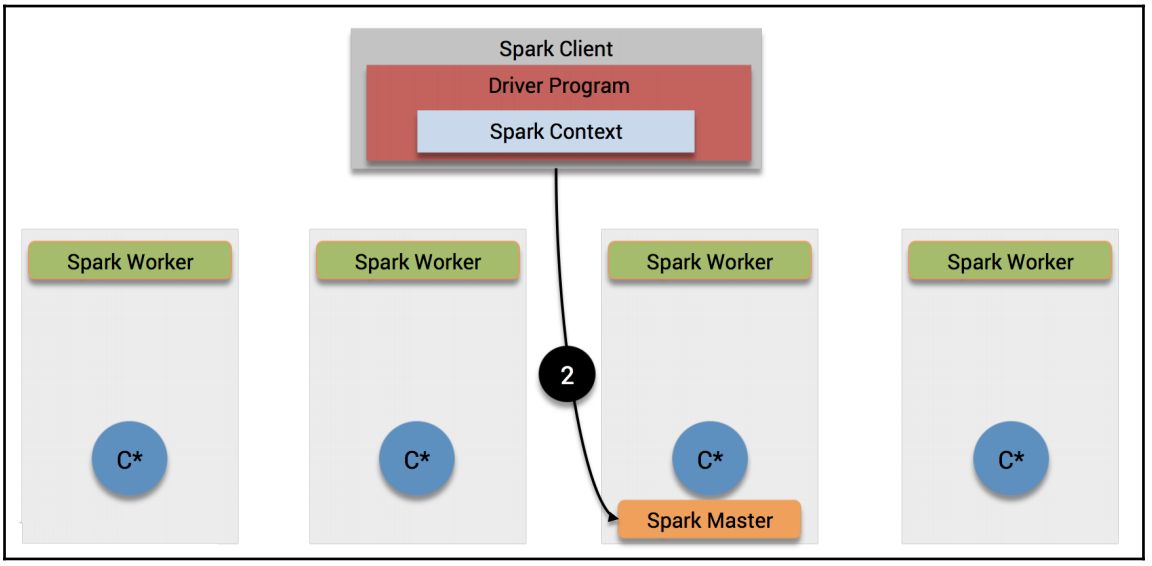

Figure 7-4. Step 2 - Driver send the tasks to the Spark Master

Figure 7-5 shows the third step: This is the distribution of the tasks among the Spark workers.

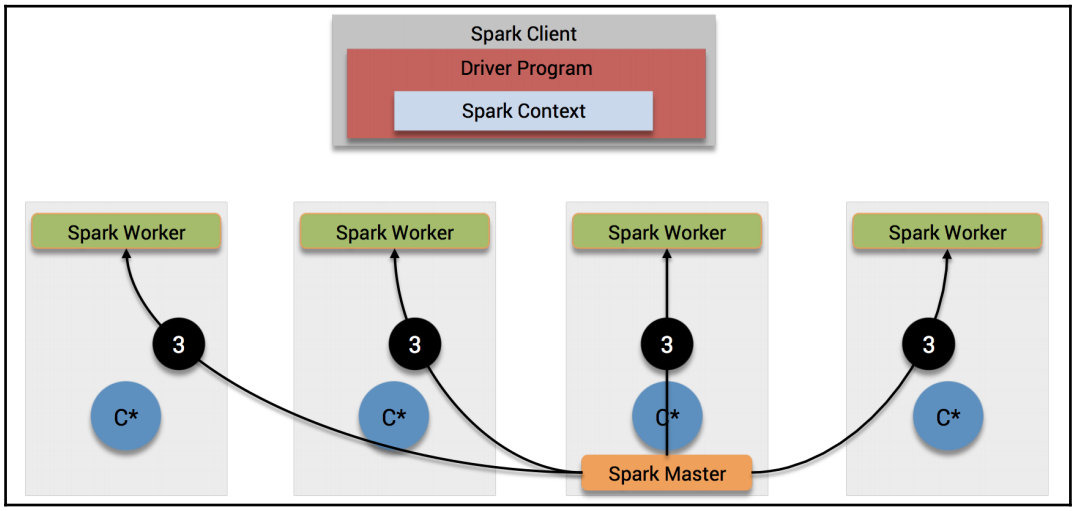

Figure 7-5. Step 3 - The Spark Master distributes the task among the workers

Figure 7-6 shows the fourth step: Every Spark worker sends the task to its corresponding Spark executor.

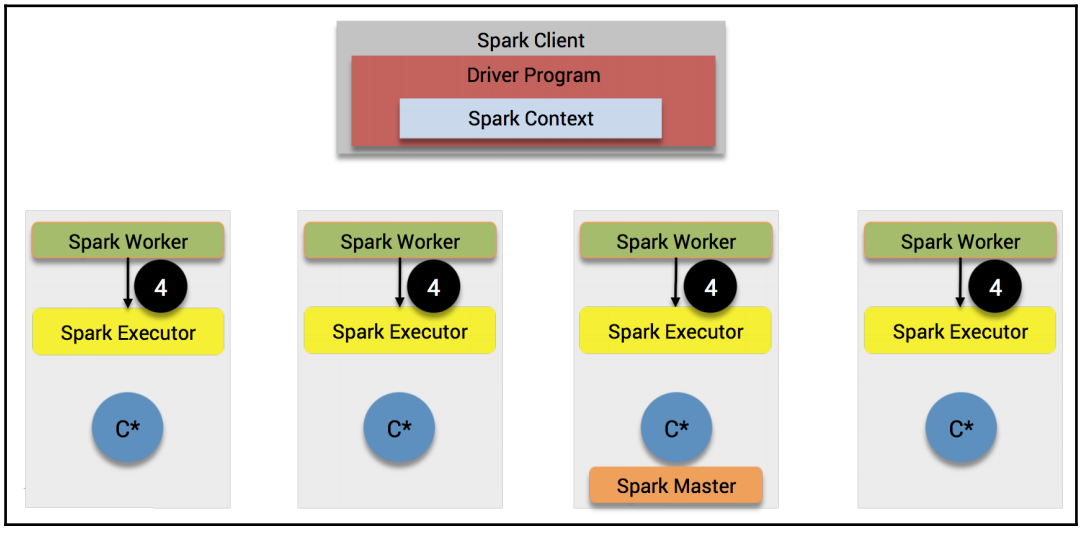

Figure 7-6. Step 4 - The Spark worker executes the task with the Spark executor

Figure 7-7 shows the final step; Every Spark executor *executes* its tasks interacting with its corresponding Cassandra process.

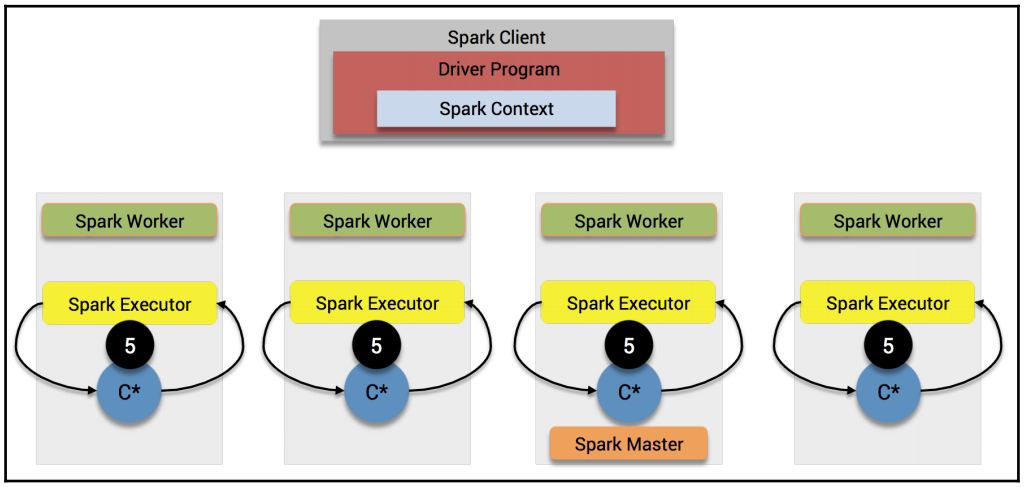

Figure 7-7. Step 5 - The Spark Executor executes the task with the Cassandra Process

Figure 7-8 shows all the actors involved: Spark Master, Spark Workers, Cassandra Processes, Spark Partitions, RDDs and Cassandra tokens.

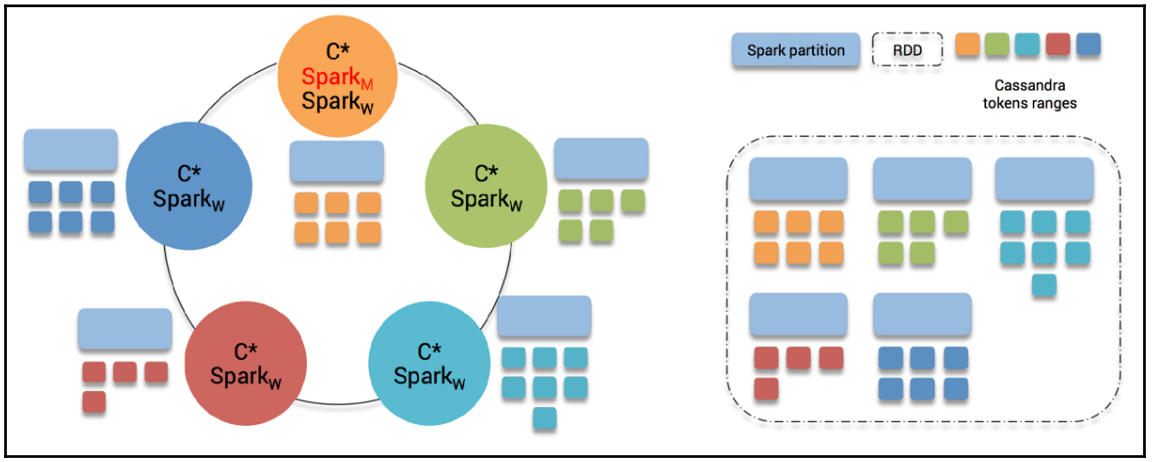

Figure 7-8. Data Locality

Figure 7-9 shows the read process: Every Cassandra process sends the results to its corresponding Spark Worker:

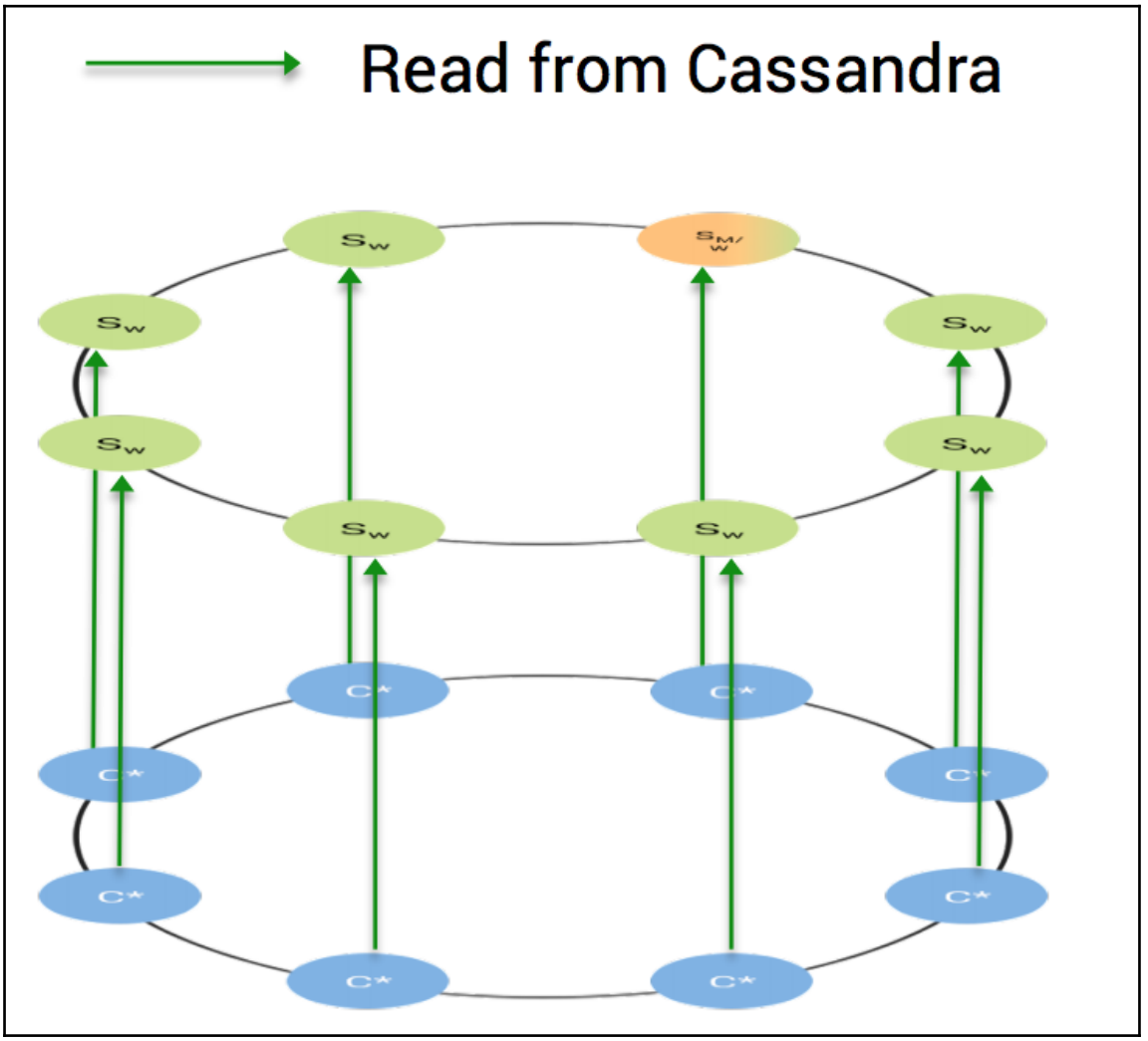

Figure 7-9. Read data from Cassandra

Figure 7-10 shows the Spark shuffle operations. As we know, the function of these operations is to redistribute data so that it is grouped differently across partitions. This process involves copying data across exectors and machines:

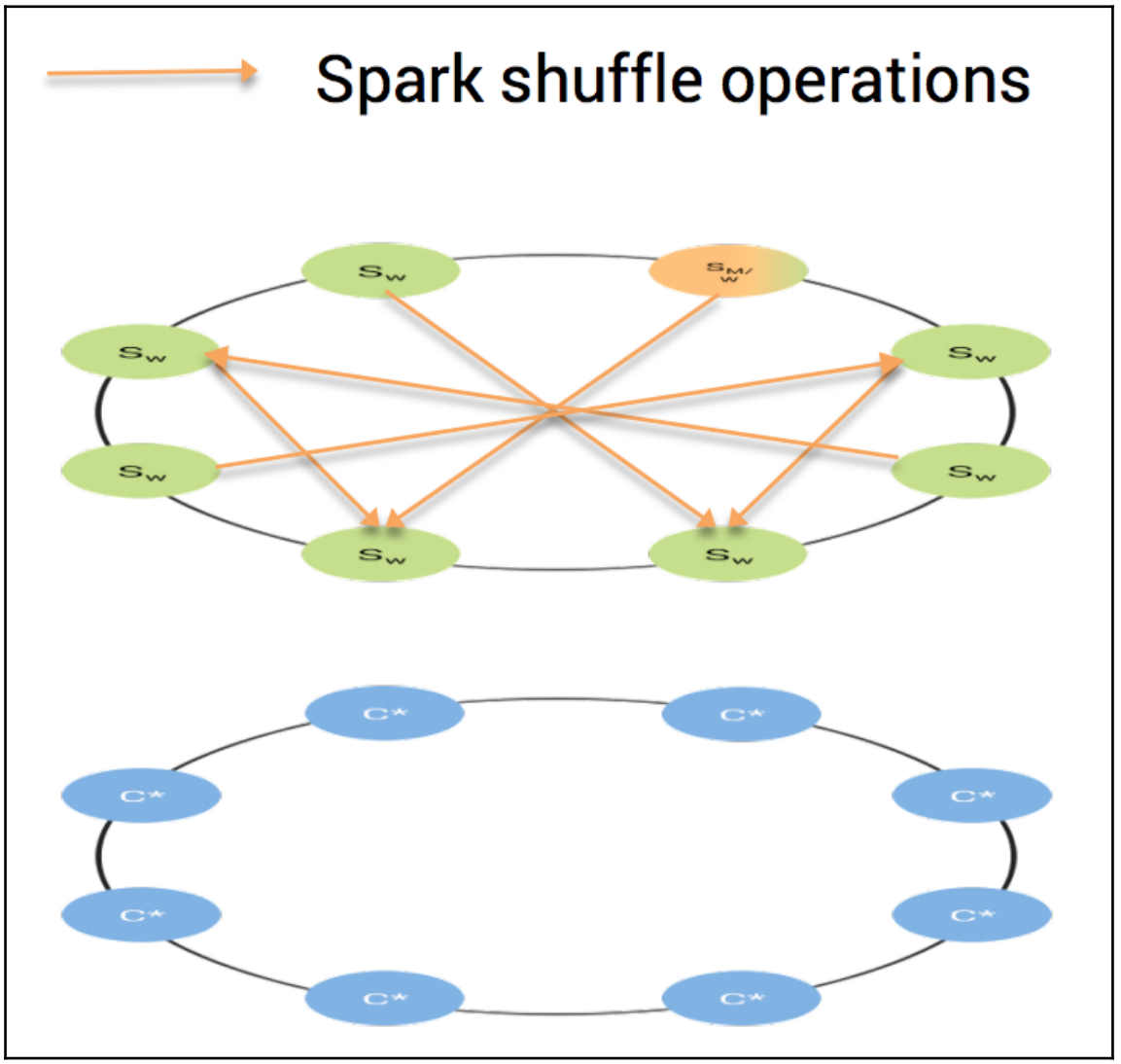

Figure 7-10. Spark shuffle operations

Figure 7-11 shows the process to write to Cassandra when we have asynchronous writes. The problem is that Spark shuffle operations distribute the data across the partitions, so the original reference to the Cassandra process is lost. Every Spark worker has to write to several Cassandra processes with all the cost implications associated with this:

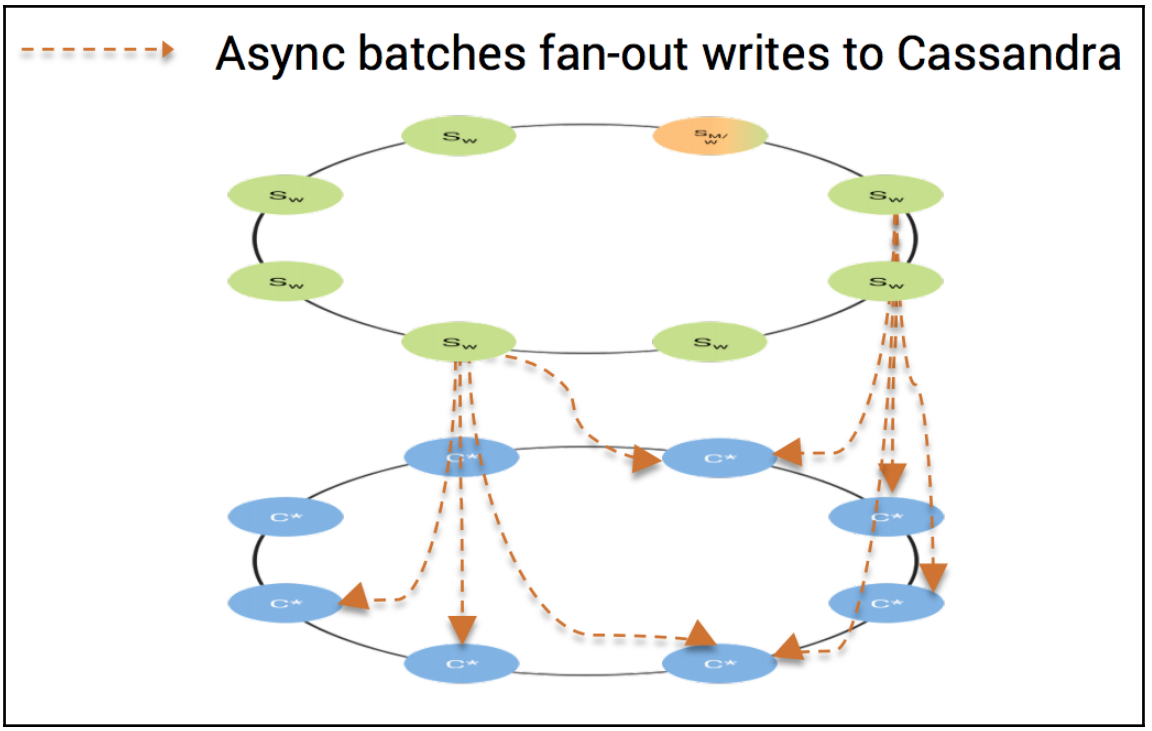

Figure 7-11. Async writes to Cassandra (without data locality)

Figure 7-12 shows how the method `repartitionByCassandraReplica` solves this problem, so that every Spark worker works to the original Cassandra Replica. This powerful and efficient process is known as "data locality". Remember that there will still be data movement across the network, but it's smaller than async writes:

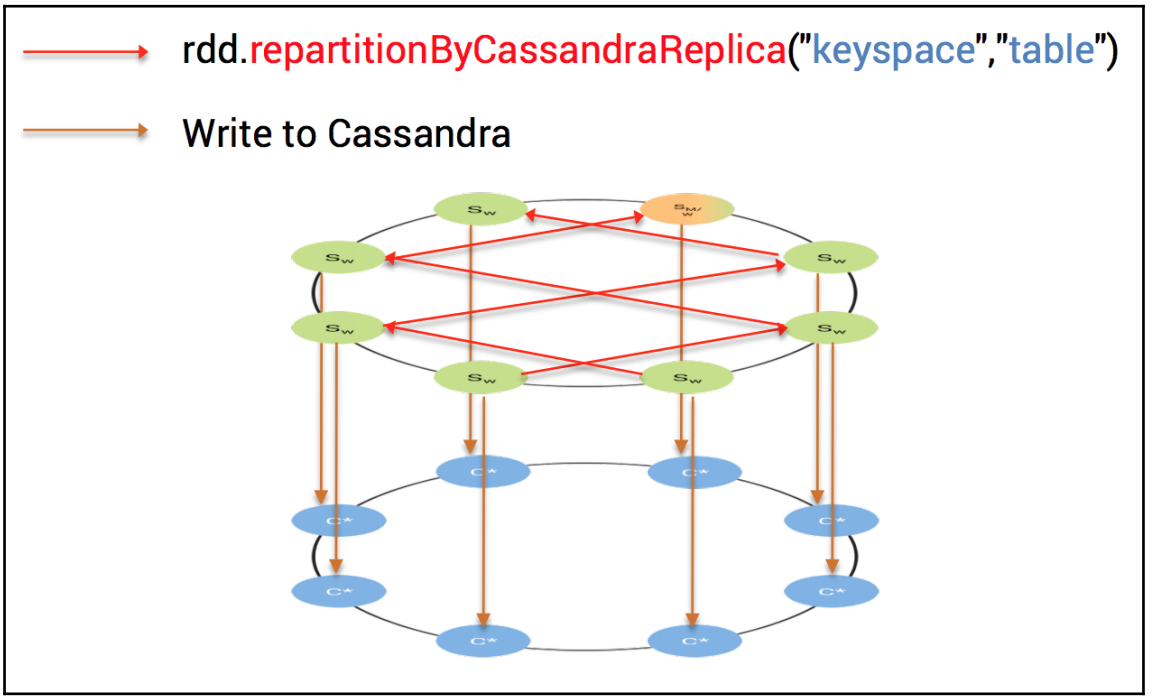

Figure 7-12. Write to Cassandra with Data Locality

The perfect data locality scenario is when:

- The data is read locally from Cassandra
- Operations are used that do not require shuffling in Spark (you already know them: map, filter, and so on.)
- Using `repartitionByCassandraReplica` we ensure that each table has the same partition as the original table
- The data is saved back into the same Cassandra table

However, the reader may wonder *What happens when a Node is down, disconnected, overloaded or another catastrophe occurs?* Well, the magic behind data locality is shown in figure 7-13. The Spark master chooses the next preferred location which is a replica, so, no data is lost:

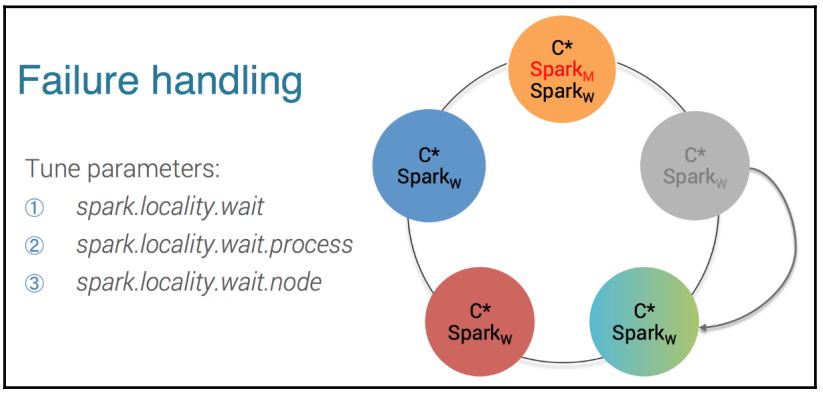

Figure 7-13. Failure handling

Spark Cassandra use cases

Finally, figure 7-14 shows the four main Spark Cassandra use cases:

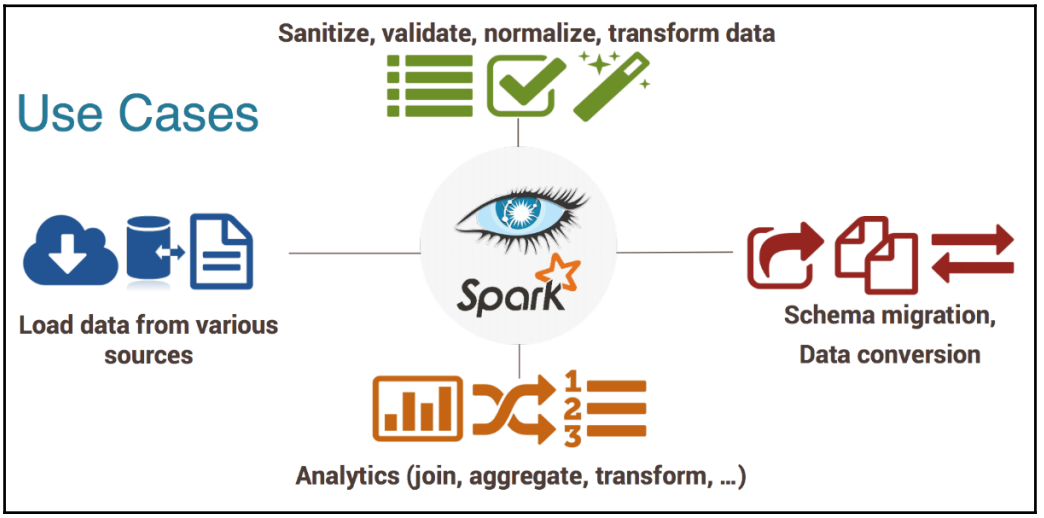

Figure 7-14. Spark Cassandra use cases

Data cleaning use cases:

- If our data process has dirty input data, the ideal scenario is to put in a Spark job to clean it up.
- Nobody is perfect. What happens when our application has a bug? A fresh idea is to program a Spark job to debug our application.

Schema migration use-cases:

- As usual, business requirements change with time. Programming a Spark job that validates the Cassandra schema is not a bad practice.
- How do we know which parts of the Cassandra schema are "less relevant"? Our data model is not perfect, it could have bottlenecks, unused segments, and so on. To detect these issues in the program a Spark job is a good option.

Analytics use cases:

- This is the most used use case; To compute analytics in real time is the reason for all the Connector and Spark-Cassandra architecture. Programming a Spark job to count the number of events in real time is a common and valuable task.

Study case: The Calliope project

In Greek mythology, Calliope (/kla.pi / k-ly-pee; Ancient Greek: Καλλιόπη Kalliopē "beautiful-voiced") was the muse of epic poetry. Calliope was the daughter of Zeus and Mnemosyne, and is believed she was the muse of the poet Homer who inspired the Odyssey and the Iliad.

Calliope is the bridge between Cassandra and Spark that allows us to create fast real-time data apps with ease. Calliope is a library that provides an interface to consume Cassandra data into Spark and vice versa; and to store Spark Resilient Distributed Datasets into Cassandra. As we saw, we can use Spark on Cassandra without Calliope, but Calliope make it all easier.

Calliope was started by Tuplejump Inc in 2013, when there was no other solution available to work with Cassandra Data in Spark. In 2014 Tuplejump worked on the core stabilization while Calliope was adopted and deployed at many organizations.

Installing Calliope

To use the Calliope jar from the Spark shell, add this jar to the executor classpath: `http://downloads.tuplejump.com/calliope-core-assembly-1.1.0-CTP-U2.jar`

To use Calliope with Maven, the coordinates are:

```
<dependency>
    <groupId>com.tuplejump</groupId>
    <artifactId>calliope-core_2.10</artifactId>
    <version>1.1.0-CTP-U2</version>
</dependency>
```

To use Calliope with SBT, use

```
libraryDependencies += "com.tuplejump" % "calliope-core_2.10" % "1.1.0-CTP-
U2"
```

Calliope can be used with two methods: CQL3 and Thrift.

CQL3

The CQL3 method uses `CqlPagingInputFormat` and `CqlPagingOutputFormat`. It can read any column families, whether created with CQL3 or not, and use composite keys.

Read from Cassandra with CQL3

To read a column family name `words` from a Key Space called `calliopeDemo` and create an RDD from it. Here is the code required when using Calliope; consider all the columns as String types:

```
import com.tuplejump.calliope.utils.RichByteBuffer._
import com.tuplejump.calliope.Implicits._
import com.tuplejump.calliope.CasBuilder

val cas = CasBuilder.cql3.withColumnFamily("calliopeDemo", "words")
val rdd = sc.cql3Cassandra[Map[String, String], Map[String, String]](cas)
```

So here we don't need to customize any of the advanced options for Cassandra connections, we can use the simplified API as follows:

```
val rdd = sc.cql3Cassandra[Map[String, String], Map[String,
String]]("words", "calliopeDemo")
```

If Cassandra is not running on the same host as the SparkContext, use the following:

```
val rdd = sc.cql3Cassandra[Map[String, String], Map[String, String]]
("casserver.local", "9160", Words", "calliopeDemo")
```

Now that we have an RDD, we can also use a `where` predicate with the CQL:

```
val cas = CasBuilder.cql3.withColumnFamily("calliopeDemo", "Words")
                                  .where("book = 'Learning SMACK'")

val rdd = sc.cql3Cassandra[Map[String, String], Map[String, String]](cas)
```

This uses the Cassandra index to filter the data and give faster results.

Write to Cassandra with CQL3

To write an RDD to a column family name `words` from the `calliopeDemo` key space:

```
import com.tuplejump.calliope.Implicits._
import com.tuplejump.calliope.CasBuilder
import com.tuplejump.calliope.utils.RichByteBuffer._

val cas = CasBuilder.cql3.withColumnFamily("calliopeDemo", "Words")
            .saveWithQuery(
    "UPDATE calliopeDemo.words set book_name = ?, book_content = ?")

rdd.cql3SaveToCassandra(cas)
```

This will use the keys from the first map in the RDD to write the values of the second. It only allows update queries.

If the key didn't previously exist, then a new row is created.

If the key already exists, the row will be updated.

If the row exists and a column value is not provided, the column is deleted.

Thrift

This method internally uses `ColumnFamilyInputFormat` and `ColumnFamilyOutputFormat`. This method is older and more tested than CQL3, but is not able to read CQL3 column families with composite keys.

Read from Cassandra with Thrift

To read a column family named `words` from a Key Space called `calliopeDemo` and create an RDD from it, here is the code required using Calliope; consider all the columns as String types:

```
import com.tuplejump.calliope.utils.RichByteBuffer._
import com.tuplejump.calliope.Implicits._
import com.tuplejump.calliope.CasBuilder

val cas = CasBuilder.thrift.withColumnFamily("calliopeDemo", "words")
val rdd = sc.thriftCassandra[String, Map[String, String]](cas)
```

So, here we don't need to customize any of the advanced options for Cassandra connections, we can use the simplified API as follows:

```
val rdd = sc.thriftCassandra[String, Map[String, String]]("words",
"calliopeDemo")
```

If Cassandra is not running on the same host as the SparkContext, use the following:

```
val rdd = sc.thriftCassandra[String, Map[String, String]]
("casserver.local", "9160", words", "calliopeDemo")
```

The output of the thrift command is a Spark RDD,

Write to Cassandra with Thrift

To write an RDD to a column family name `words` from the `calliopeDemo` key space:

```
import com.tuplejump.calliope.RichByteBuffer._
import com.tuplejump.calliope.Implicits._
import com.tuplejump.calliope.CasBuilder

val cas = CasBuilder.thrift.withColumnFamily("calliopeDemo", "words")
rdd.thriftSaveToCassandra(cas)
```

Calliope SQL context creation

Calliope provides a custom `SQLContext` (`CassandraAwareSQLContext`) that extends the `SQLContext` of Spark SQL. Creating a `CassandraAwareSQLContext` is the same as creating a normal `SQLContext` with an existing SparkContext:

```
val sc: SparkContext
val sqlContext = new org.apache.spark.sql.CassandraAwareSQLContext(sc)

import sqlContext.createSchemaRDD
```

With this `SQLContext` we can use it as documented in the Spark SQL programming guide: Calliope Cassandra Queries

With Calliope's `sqlContext` we can access Cassandra tables directly with SQL queries.

Let's assume that we have a table called *leaks* in a Cassandra Key Space *wiki*:

```
sqlContext.sql("SELECT * FROM wiki.leaks")
```

We can run any SQL query that Spark SQL can handle. If we combine Spark SQL with C* queries, Calliope understands our query and chooses the best option to get the fastest results. Calliope handles the use of indexes and clustering keys in the queries to reduce the data amount to transfer from C* to Spark.

Calliope SQL Configuration

By default, Calliope assumes that the driver application is running on the same node as C*, so it connects to 127.0.0.1 as the root node. If the driver application is not running on the same host we can specify this configuration through the Spark properties.

Calliope also has these additional configuration properties:

- `spark.cassandra.connection.host`: The contact point in the Cassandra cluster. This IP address must be reachable by the driver application (default: `127.0.0.1`).
- `spark.cassandra.connection.native.port`: The port in which the contact node is listening (default: `9042`).

- `spark.cassandra.connection.rpc.port`: The port for the Thrift protocol where the contact node is listening (default: `9160`).
- `spark.cassandra.auth.username`: The username for authenticating to the listening node.
- `spark.cassandra.auth.password`: The password for authenticating to the listening node.

Loading Cassandra tables programmatically

Sometimes we need to connect to different Cassandra clusters or we need access to the RDD schema. The SQL context of Calliope provides the method `cassandraTable` with alternative signatures. We can specify `host`, `port`, `keyspace`, `table`, `username`, and `password`:

```
sqlContext.cassandraTable("c.host", "c.native.port", "keyspace", "table",
"username", "password")
```

Summary

In the case study in this chapter we have covered the connection between Spark and Cassandra.

We looked at the Spark Cassandra connector and how to make the Cassandra and Spark Context setup, Cassandra and Spark streaming, streaming context creation, reading and writing a stream from Cassandra, saving datasets, collections and tuples to Cassandra, modifying collections, saving UDTs and RDDs as tables.

We also reviewed the Calliope project: installing Calliope, reading and writing from Cassandra with CQL3 and writing and reading from Cassandra with Thrift.

This chapter was about the relation between Spark and Cassandra. In the next chapter we will examine the relationship between the remaining SMACK technologies.

8
Study Case 2 - Connectors

In this chapter, we analyze the Connectors, that is, the software pieces that enable SMACK stack technologies to communicate with each other. The relationship between Spark and Kafka, was covered in the Kafka chapter, and we also dealt with the relationship between Spark and Cassandra in the previous study case in `Chapter 7`, *Study Case 1 - Spark and Cassandra*.

This chapter has the following sections, along with the remaining relationships:

- Akka and Cassandra
- Akka and Spark
- Kafka and Akka
- Kafka and Cassandra

Akka and Cassandra

For this example, we will use the DataStax Cassandra driver and Akka to build an application that downloads tweets and then stores their ID, text, name, and date in a Cassandra table. Here we will see:

- How to build a simple Akka application with just a few actors
- How to use Akka IO to make HTTP requests
- How to store the data in Cassandra

The first step is to build our core example. It contains three actors: two actors interact with the database and one actor downloads the tweets. The `TwitterReadActor` reads from Cassandra, the `TweetWriteActor` writes to Cassandra, and the `TweetScanActor` downloads the tweets and passes them to the `TweetWriteActor` to be written to Cassandra:

```
class TweetReadActor(cluster: Cluster) extends Actor { ... }

class TweetWriterActor(cluster: Cluster) extends Actor { ... }

class TweetScanActor(tweetWrite: ActorRef, queryUrl: String => String)
extends Actor {  ... }
```

Figure 8-1 shows the relationship between the Twitter downloader actors:

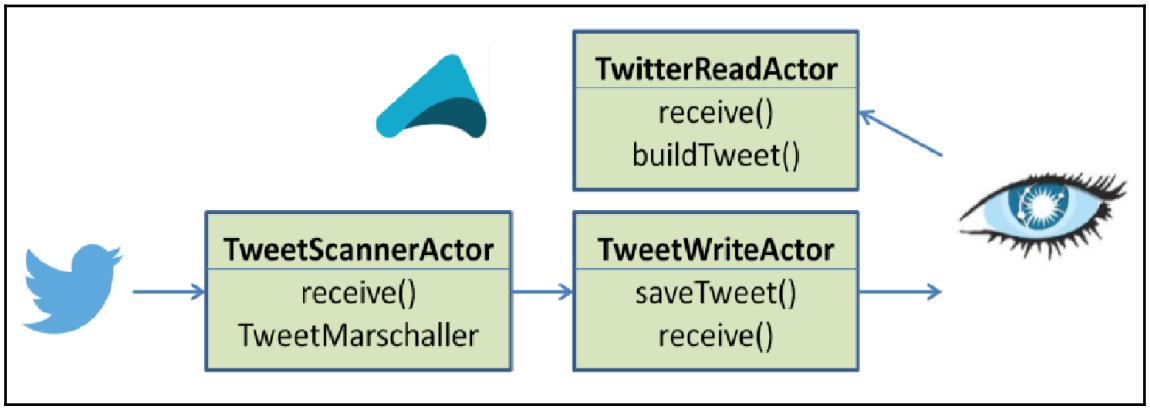

Figure 8-1. Twitter downloader actors

The constructors of the read and write actors receives the Cassandra Cluster instance as a parameter. The scan actor takes an actor reference of the write actor and a function that, given a String query, can construct the query URL to download the tweets. In this scenario, our application instantiates the actors in the right sequence:

```
val system = ActorSystem()
def queryUrl(query: String): String = ???
val cluster: Cluster = ???
val reader  = system.actorOf(Props(new TweetReaderActor(cluster)))
val writer  = system.actorOf(Props(new TweetWriterActor(cluster)))
val scanner = system.actorOf(Props(new TweetScannerActor(writer,
queryUrl)))
```

Writing to Cassandra

For this example, suppose that we have a key spaces set called `akkaCassandra`. Now that we have our program structure, we can give body to the `TwitterWriterActor`. It receives tweet instances and it writes to a key space in Cassandra called `tweets`:

```
class TweetWriterActor(cluster: Cluster) extends Actor {
  val session = cluster.connect(Keyspaces.akkaCassandra)
  val preparedStatement = session.prepare(
   "INSERT INTO tweets(key, user, text, creation_date) VALUES (?, ?, ?,
?);")

  def receive: Receive = {
    case tweets: List[Tweet] =>  // TO_DO
    case tweet:  Tweet       =>  // TO_DO
  }
}
```

To store the tweets, we need to connect to the tweets' key spaces which gives us the Cassandra session. To achieve an elegant and efficient solution, we take advantage of the Cassandra's Prepared Statements and Bound Statements. A Prepared Statement is a CQL statement to be defined; and a Bound Statement is a Prepared Statement with defined parameters, as we can see here:

```
class TweetWriterActor(cluster: Cluster) extends Actor {
  val session = cluster.connect(Keyspaces.akkaCassandra)
  val preparedStatement = session.prepare(
   "INSERT INTO tweets(key, user, text, creation_date) VALUES (?, ?, ?,
?);")

  def saveTweet(tweet: Tweet): Unit =
    session.executeAsync( preparedStatement.bind(
         tweet.id, tweet.user, tweet.text, tweet.creation_date))

  def receive: Receive = {
    case tweets: List[Tweet] =>  // TO_DO
    case tweet:  Tweet       =>  // TO_DO
  }
}
```

Now we have to give a body to the `receive` function:

```
class TweetWriterActor(cluster: Cluster) extends Actor {
  val session = cluster.connect(Keyspaces.akkaCassandra)
  val preparedStatement = session.prepare(
    "INSERT INTO tweets(key, user, text, creation_date) VALUES (?, ?, ?,
?);")
```

```
  def saveTweet(tweet: Tweet): Unit =
    session.executeAsync(preparedStatement.bind(
         tweet.id, tweet.user, tweet.text, tweet.creation_date))

  def receive: Receive = {
    case tweets: List[Tweet] => tweets foreach saveTweet
    case tweet:  Tweet       => saveTweet(tweet)
  }
}
```

We have the code that stores tweet instances into the key space in our Cassandra cluster.

Reading from Cassandra

Reading the data is a slightly more complex task, because we need to build Cassandra queries; then, transform from a Cassandra row to turn it into our tweet object. In this code, we use the asynchronous nature of the Cassandra driver:

```
object TweetReaderActor {
  case class FindAll( maximum: Int = 100 )
  case object CountAll
}

class TweetReaderActor(cluster: Cluster) extends Actor {
  val session = cluster.connect(Keyspaces.akkaCassandra)
  val countAll =
    new BoundStatement(session.prepare("select count(*) from tweets;"))

  def receive: Receive = {
    case FindAll(maximum) => // return a list of Tweets
    case CountAll         => // return a long with the count
  }
}
```

In this code, we defined two messages, `FindAll` and `CountAll` to which our actor reacts. In the first line of our class, we create a session and with this we build a Bound Statement that counts the number of rows. The next step is to code a class method that, given a row, builds a tweet instance:

```
  def buildTweet(r: Row): Tweet = {
    val id   = r.getString("key")
    val user = r.getString("user")
    val text = r.getString("text")
    val creation_date = r.getDate("creation_date")
```

```
      Tweet(id, user, text, createdAt)
    }
```

Here we simply pick the values of the row columns and assign them to the tweet attributes. The next step is to complete the receive method: asynchronously execute the query, map the rows returned by that query to tweet objects, and then pipe these results to the sender:

```
import scala.collection.JavaConversions._
import cassandra.resultset._
import context.dispatcher
import akka.pattern.pipe

class TweetReaderActor(cluster: Cluster) extends Actor {
  val session = cluster.connect(Keyspaces.akkaCassandra)
  val countAll =
    new BoundStatement(session.prepare("select count(*) from tweets;"))


  def buildTweet(r: Row): Tweet = { //folded code...}

  def receive: Receive = {

    case FindAll(maximum)  =>
      val query = QueryBuilder.select().all().from(
    Keyspaces.akkaCassandra, "tweets").limit( maximum )
            session.executeAsync(query)
        map(_.all().map(buildTweet).toList) pipeTo sender

    case CountAll =>
      session.executeAsync(countAll)
        map(_.one.getLong(0)) pipeTo sender
  }
}
```

We make the query using Cassandra's `QueryBuilder`. We call the `executeAsync` method on the `session`, which returns `ResultSetFuture`. We use the implicit conversion in the package, `cassandra.resultset._`, We turn the `ResultSetFuture` into Scala's `Future[ResultSet]`. This allows the use of the `Future.map` method to turn the `ResultSet` into a `List[Tweet]`.

Calling the `session.executeAsync(query)` map expects as its parameter a function from `ResultSet` to some type X. For this case X is `List[Tweet]`. The `ResultSet` contains the method `all()`, which returns `java.util.List[Row]`. To allow the map over the `java.util.List[Row]`, we need to turn it into the Scala `List[Row]`. To do so, we bring in the implicit conversions in the package `scala.collection.JavaConversions._`. Now, we can complete the parameter of the `Future.map` function.

The `session.executeAsync` method gives us `Future[List[Tweet]]`, which is close to what we need. As we don't want to block the result or use the `onSuccess` function, we just want to send the result to the sender so we pipe the success of the future to the sender.

If you are confused with the Scala Futures and the *Ask* pattern, visit the following URL:

`http://doc.akka.io/docs/akka/snapshot/scala/actors.html`

Connecting to Cassandra

We need to explain where the cluster value comes from. Thinking about the example we are writing, we may need to have different values of clusters for the test and production systems. The test cluster needs some special make-up.

We just simply define that there is a Cassandra cluster trait that returns the cluster, and two implementations: one loads the configuration from the `ActorSystem` configuration, and the other is hardcoded and used in tests:

```
trait CassandraCluster
{
  def cluster: Cluster
}
```

The configuration-based implementation and the test configuration differ only in the values that were used to build the Cluster instance.

For the fail located in `src/scala/main` we use:

```
import scala.collection.JavaConversions._

trait ConfigCassandraCluster extends CassandraCluster {
  def system: ActorSystem

  private def config = system.settings.config

  private val cassandraConfig =
```

```
      config.getConfig("akka-cassandra.main.db.cassandra")
    private val port = cassandraConfig.getInt("port")
    private val hosts = cassandraConfig.getStringList("hosts").toList

    lazy val cluster: Cluster =
      Cluster.builder().
        addContactPoints(hosts: _*).
        withCompression(ProtocolOptions.Compression.SNAPPY).
        withPort(port).
        build()
}
```

For the fail located in `src/scala/test` we use:

```
import scala.collection.JavaConversions._

trait TestCassandraCluster extends CassandraCluster
{
  def system: ActorSystem

  private def config = system.settings.config

  private val cassandraConfig =
    config.getConfig("akka-cassandra.test.db.cassandra")
  private val port = cassandraConfig.getInt("port")
  private val hosts = cassandraConfig.getStringList("hosts").toList

  lazy val cluster: Cluster =
    Cluster.builder().
      addContactPoints(hosts: _*).
      withPort(port).
      withCompression(ProtocolOptions.Compression.SNAPPY).
      build()
}
```

This allows us to use the appropriate trait and create a properly configured cluster. We want to have the cluster in a good state, so we create the `CleanCassandra` trait that resets the cluster given by `CassandraCluster.cluster`:

```
trait CleanCassandra extends SpecificationStructure
{
    this: CassandraCluster =>

  private def runClq(session: Session, file: File): Unit =
  {
    val query = Source.fromFile(file).mkString
```

```
      query.split(";").foreach(session.execute)
    }

    private def runAllClqs(): Unit =
      {
        val session = cluster.connect(Keyspaces.akkaCassandra)
        val uri = getClass.getResource("/").toURI
        new File(uri).listFiles().foreach
        { file =>
           if (file.getName.endsWith(".cql")) runClq(session, file)
        }
        session.shutdown()
      }

    override def map(fs: => Fragments) = super.map(fs) insert
  Step(runAllClqs())
  }
```

When we mix this trait into our test, it registers the `runAllClqs()` steps to be executed before any other step in the test.

Scanning tweets

Now we have the components that store and retrieve tweets from Cassandra safely. Next, we need to write the component that downloads them. In our system, this is the `TweetScannerActor` that receives a message of the type String, and performs the HTTP request to download the tweets:

```
class TweetScannerActor(tweetWrite: ActorRef, queryUrl: String => String)
  extends Actor with TweetMarshaller {

  import context.dispatcher
  import akka.pattern.pipe

  private val pipeline = sendReceive ~> unmarshal[List[Tweet]]

  def receive: Receive = {
    case query: String => pipeline(Get(queryUrl(query))) pipeTo tweetWrite
  }
}

trait TweetMarshaller {
  type Tweets = List[Tweet]

  implicit object TweetUnmarshaller extends Unmarshaller[Tweets] {
```

```
      val dateFormat = new SimpleDateFormat("EEE MMM d HH:mm:ss Z yyyy")

      def mkTweet(status: JsValue): Deserialized[Tweet] = {
        val json = status.asJsObject
        ...
      }

      def apply(entity: HttpEntity): Deserialized[Tweets] = {
        val json = JsonParser(entity.asString).asJsObject
        ...
      }
    }
  }
```

The type class instance is the `TweetUnmarshaller` singleton, which extends `Unmarshaller[Tweets]`. Notice that we have also defined an alias type `Tweets = List[Tweet]` by extending `Unmarshaller[Tweets]`. We implement the `apply` method to the `HttpEntity` and this should return the deserialized tweets or indicate an error.

Testing the scanner

We implement a mock service and use it in our tests:

```
class TweetScanActorSpec extends TestKit(ActorSystem())
  with SpecificationLike with ImplicitSender {

  sequential

  val port = <our_port>
  def testQueryUrl(query: String) = s"http://localhost:$port/q=$query"

  val tweetScan = TestActorRef(new TweetScannerActor(testActor,
testQueryUrl))

  "Getting all 'smack' tweets" >> {

    "should return more than 10 last entries" in {
      val twitterApi = TwitterApi(port)
      tweetScan ! "smack"
      Thread.sleep(1000)
      val tweets = expectMsgType[List[Tweet]]
      tweets.size mustEqual 10
      twitterApi.stop()
      success
    }
  }
```

```
}
```

When constructing the `TweetScannerActor`, we give it the `testActor` and a function that returns URLs on the local host on a port. In the body of the example, we start the mock `TwitterApi` on the given port; and use our `TweetScannerActor` to make the HTTP request. Because we gave the `testActor` as the writer `ActorRef`, we should now be able to see the `List[Tweet]` that would have been sent to the `TweetWriterActor`.

So, the components in the system work as expected, we can assemble the app object, which brings everything together within a command-line interface:

```
object Main extends App with ConfigCassandraCluster {
  import Commands._
  import akka.actor.ActorDSL._

  def twitterSearchProxy(query: String) =
    s"http://twitter-search-proxy.herokuapp.com/search/tweets?q=$query"

  implicit lazy val system = ActorSystem()
  val write = system.actorOf(Props(new TweetWriterActor(cluster)))
  val read = system.actorOf(Props(new TweetReaderActor(cluster)))
  val scan = system.actorOf(Props(new TweetScannerActor(write,
twitterSearchProxy)))

  implicit val _ = actor(new Act {
    become {
      case x => println(">> " + x)
    }
  })

  @tailrec
  private def commandLoop(): Unit = {
    Console.readLine() match {
      case QuitCommand            => return
      case ScanCommand(query)     => scan ! query.toString
      case ListCommand(count)     => read ! FindAll(count.toInt)
      case CountCommand           => read ! CountAll
      case _                      => return
    }

    commandLoop()
  }

  commandLoop()
  system.shutdown()

}
```

We have the `commandLoop()` function, which reads the line from a standard input, matches it against the commands and sends the appropriate message to the right actors. It also mixes in the real source of Cassandra cluster values and specifies the live function that constructs the URL to retrieve the tweets.

Akka and Spark

We start developing the Spark Streaming application by creating a `SparkConf` followed by a `StreamingContext`:

```
val conf = new SparkConf(false)
  .setMaster("local[*]")
  .setAppName("Spark Streaming with Akka")
  .set("spark.logConf", "true")
  .set("spark.driver.port", s"$driverPort")
  .set("spark.driver.host", s"$driverHost")
  .set("spark.akka.logLifecycleEvents", "true")
val ssc = new StreamingContext(conf, Seconds(1))
```

This gives us a context to access the actor system that is of the type `ReceiverInputDStream`:

```
val actorName = "salutator"
val actorStream: ReceiverInputDStream[String] =
ssc.actorStream[String](Props[Salutator], actorName)
```

Now that we have a DStream, let's define a high-level processing pipeline in Spark Streaming:

```
actorStream.print()
```

In the preceding case, the `print()` method is going to print the first 10 elements of each RDD generated in this DStream. Nothing happens until `start()` is executed:

```
ssc.start()
```

With the context up and running, the code connects to an Akka remote actor system in Spark Streaming that hosts the `salutator` actor and sends messages that, as the above code shows, displays them all to a standard output:

```
import scala.concurrent.duration._
val actorSystem = SparkEnv.get.actorSystem
val url =
s"akka.tcp://spark@$driverHost:$driverPort/user/Supervisor0/$actorName"
```

```
val timeout = 100 seconds
val salutator =
  Await.result(actorSystem.actorSelection(url).resolveOne(timeout),
timeout)
salutator ! "Hello"
salutator ! "from"
salutator ! "Apache Spark"
salutator ! "and Akka"
```

Kafka and Akka

The connector is available at Maven Central for Scala 2.11 in the coordinates:

```
    libraryDependencies += "com.typesafe.akka" %% "akka-stream-kafka" %
"0.11-M4"
```

If you remember, a producer publishes messages to Kafka topics. The message itself contains information about what topic and partition to publish, so one can publish to different topics with the same producer. The underlying implementation uses the `KafkaProducer`.

When creating a consumer stream we need to pass `ProducerSettings` defining:

- Kafka cluster bootstrap servers
- Serializers for the keys and values
- Tuning parameters

Here we have a `ProducerSettings` example:

```
import akka.kafka._
import akka.kafka.scaladsl._
import org.apache.kafka.common.serialization.StringSerializer
import org.apache.kafka.common.serialization.ByteArraySerializer

val producerSettings = ProducerSettings(system, new ByteArraySerializer,
new StringSerializer).withBootstrapServers("localhost:9092")
```

The easiest way to publish messages is by using `Producer.plainSink`. The sink consumes `ProducerRecord` elements which contains a topic name to send the record, an optional partition number, and an optional key and value.

Produce messages example:

```
Source(1 to 10000)
```

```
  .map(_.toString)
  .map(elem => new ProducerRecord[Array[Byte], String]("topic1", elem))
  .to(Producer.plainSink(producerSettings))
```

Produce messages in a flow example:

```
Source(1 to 10000).map(elem => ProducerMessage.Message(new
ProducerRecord[Array[Byte], String]("topic1", elem.toString), elem))
    .via(Producer.flow(producerSettings))
    .map { result =>
      val record = result.message.record
      println(s"${record.topic}/${record.partition} ${result.offset}:
${record.value} (${result.message.passThrough}")
      result
    }
```

Consumer settings example:

```
import akka.kafka._
import akka.kafka.scaladsl._
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.kafka.common.serialization.ByteArrayDeserializer
import org.apache.kafka.clients.consumer.ConsumerConfig

val consumerSettings = ConsumerSettings(system, new ByteArrayDeserializer,
new StringDeserializer)
  .withBootstrapServers("localhost:9092")
  .withGroupId("group1")
  .withProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest")
```

Consume messages and store a representation, including offset, in DB example:

```
db.loadOffset().foreach { fromOffset =>
   val subscription = Subscriptions.assignmentWithOffset(new
TopicPartition("topic1", 1) -> fromOffset)
    Consumer.plainSource(consumerSettings, subscription)
      .mapAsync(1)(db.save)}
```

Consume messages at-most-once example:

```
Consumer.atMostOnceSource(consumerSettings.withClientId("client1"),
Subscriptions.topics("topic1"))
    .mapAsync(1) { record =>
      rocket.launch(record.value
}
```

Consume messages at-least-once example:

```
Consumer.committableSource(consumerSettings.withClientId("client1"),
Subscriptions.topics("topic1"))
    .mapAsync(1) {
      msg => db.update(msg.value).flatMap(_ =>
        msg.committableOffset.commitScaladsl())
}
```

```
Connect a consumer to a producer example:
Consumer.committableSource(consumerSettings.withClientId("client1"))
    .map(msg => ProducerMessage.Message(
        new ProducerRecord[Array[Byte], String]("topic2", msg.value),
          msg.committableOffset))
    .to(Producer.commitableSink(producerSettings))
```

Consume messages at-least-once, and commit in batches example:

```
Consumer.committableSource(consumerSettings.withClientId("client1"),
Subscriptions.topics("topic1"))
    .mapAsync(1) { msg =>
      db.update(msg.value).map(_ => msg.committableOffset)
    }
    .batch(max = 10, first =>
        CommittableOffsetBatch.empty.updated(first)) { (batch, elem) =>
          batch.updated(elem)
    }.mapAsync(1)(_.commitScaladsl())
```

A reusable Kafka consumer example:

```
val consumer: ActorRef =
system.actorOf(KafkaConsumerActor.props(consumerSettings))
```

Manually assign topic partition to it:

```
val stream1 = Consumer
    .plainExternalSource[Array[Byte], String](consumer,
Subscriptions.assignment(new TopicPartition("topic1", 1)))
    .via(business)
    .to(Sink.ignore)
```

Manually assign another topic partition:

```
val stream2 = Consumer
    .plainExternalSource[Array[Byte], String](consumer,
Subscriptions.assignment(new TopicPartition("topic1", 2)))
    .via(business)
    .to(Sink.ignore)
```

Consumer group example:

```
val consumerGroup =
Consumer.committablePartitionedSource(consumerSettings.withClientId("client
1"), Subscriptions.topics("topic1"))
  //Process each assigned partition separately
  consumerGroup.map {
    case (topicPartition, source) =>
      source
        .via(business)
        .toMat(Sink.ignore)(Keep.both)
        .run()
  }.mapAsyncUnordered(maxPartitions)(_._2)
```

Use case example:

```
import akka.actor.ActorSystem
import akka.stream.ActorMaterializer
import akka.stream.scaladsl.{Sink, Source}
import com.softwaremill.react.kafka.KafkaMessages._
import org.apache.kafka.common.serialization.{StringSerializer,
StringDeserializer}
import com.softwaremill.react.kafka.{ProducerMessage, ConsumerProperties,
ProducerProperties, ReactiveKafka}
import org.reactivestreams.{ Publisher, Subscriber }

implicit val actorSystem = ActorSystem("ReactiveKafka")
implicit val materializer = ActorMaterializer()

val kafka = new ReactiveKafka()
val publisher: Publisher[StringConsumerRecord] =
kafka.consume(ConsumerProperties(
 bootstrapServers = "localhost:9092",
 topic = "lowercaseStrings",
 groupId = "groupName",
 valueDeserializer = new StringDeserializer()
))

val subscriber: Subscriber[StringProducerMessage] =
kafka.publish(ProducerProperties(
```

```
  bootstrapServers = "localhost:9092",
  topic = "uppercaseStrings",
  valueSerializer = new StringSerializer()
))

Source.fromPublisher(publisher).map(m =>
ProducerMessage(m.value().toUpperCase))
  .to(Sink.fromSubscriber(subscriber)).run()
```

Kafka and Cassandra

We need to use the `kafka-connect-cassandra` which is published on Maven Central by Tuplejump.

It can be defined as a dependency in the build file. For example, with SBT:

```
    libraryDependencies += "com.tuplejump" %% "kafka-connect-cassandra" %
"0.0.7"
```

This code polls Cassandra with a specific query. Using this, data can be fetched from Cassandra in two modes:

- Bulk
- Timestamp based

The modes change automatically based on the query. For example:

Bulk:

```
SELECT * FROM userlog;
```

Timestamp based:

```
SELECT * FROM userlog WHERE ts > previousTime();
SELECT * FROM userlog WHERE ts = currentTime();
SELECT * FROM userlog WHERE ts >= previousTime() AND  ts <= currentTime() ;
```

Here, `previousTime()` and `currentTime()` are replaced before fetching the data.

CQL Type	Schema Type
ASCII	STRING
VARCHAR	STRING
TEXT	STRING
BIGINT	INT64
COUNTER	INT64
BOOLEAN	BOOLEAN
DECIMAL	FLOAT64
DOUBLE	FLOAT64
FLOAT	FLOAT32
TIMESTAMP	TIMESTAMP

Table 8.1 CQL types supported

The types `BLOB, INET, UUID, TIMEUUID, LIST, SET, MAP, CUSTOM, UDT, TUPLE, SMALLINT, TINYINT, DATE,TIME` are not currently supported.

Cassandra Sink stores Kafka Sink Records in Cassandra tables. It currently only supports the STRUCT type in the Sink Record. The STRUCT can have multiple fields with primitive field types. Assume one-to-one mapping between the column names in the Cassandra sink table and the field names.

The `SinkRecord` has this STRUCT value:

```
{
    'id': 1,
    'username': 'user1',
    'text': 'This is my first tweet'
}
```

The library doesn't create the Cassandra tables, and it is expected that users create these before starting the Sink.

Summary

We have reviewed the connectors between all the SMACK stack technologies. The Spark and Kafka connection was explained in the `Chapter 5`, *The Broker - Apache Kafka* and the Spark and Cassandra connector was explained in the `Chapter 7`, *Study Case 1 - Spark and Cassandra*.

In this chapter, we reviewed the connectors between:

- Akka and Cassandra
- Akka and Spark
- Kafka and Akka
- Kafka and Cassandra

In the following chapter, we will review container technologies.

9
Study Case 3 - Mesos and Docker

In this chapter, we'll analyze how to develop Mesos frameworks and we'll study how to use Mesos containerizers and Docker containerizers.

This chapter has the following parts:

- Mesos frameworks API
- Mesos containerizers
- Docker containerizers

Mesos frameworks API

As we saw in the `Chapter 6`, *The Manager - Apache Mesos*, a Mesos framework is a layer between Mesos and the application, and is used for managing task scheduling and execution. As the framework implementation is specific to the application, the term is usually used to refer to the application.

Initially, the Mesos framework only could communicate with the Mesos API using the `libmesos` written on C++. So, in order to interact with Java, Scala, Python, and Go, one had to develop language bindings using `libmesos`. Since Mesos version 0.19.0, an HTTP-based protocol has been included to enable developers to build frameworks in the language of their choice without having to work with a C++ wrapper.

A Mesos framework has two components:

- **Scheduler**: Responsible for taking decisions on resource offerings and tracking the current cluster state. The `SchedulerDriver` module has these responsibilities: to handle the communication with the Mesos master, to register frameworks with the master, launch tasks, and message passing to other components.
- **Executor**: Responsible for task execution on the slave nodes. The `ExecutorDriver` module has the responsibility of handling the communication with the slaves and of sending status updates to the scheduler.

Authentication, authorization, and access control

Often these three concepts seem to be the same because they are usually implemented together, but there are differences between them:

- **Authentication**: The process of verifying that someone is who he claims to be, or that something is what it claims to be. It involves several methods to demonstrate identity. The result is a binary answer: Yes or no.
- **Authorization**: The process used to establish if the authenticated user/application is allowed to perform a specific task. The authorization defines and determines what the user/application can do. The module typically has a way to define rules, roles, privileges, and so on.
- **Access control**: The process to ensure that a non-authenticated or non-authorized user/application cannot bypass the system and perform restricted actions. It usually involves the implementation of mechanisms to ensure that the system has no security holes.

Framework authentication

The support for framework authentication was included in Mesos version 0.15 and slave authentication support was included in the Mesos version 0.19. It prevents unauthorized applications running on Mesos. To register a framework with the Mesos master it has to be authenticated beforehand.

Slave authentication is necessary for the slaves to register with the master. An unauthorized process could launch a Denial of Service attack. The access to the HTTP endpoint is not direct, so the frameworks can't interfere to terminate each other.

The CRAM-MD5 is an authentication mechanism that implements a shared secret key in which the framework (principal) and Mesos (authenticator) share a secret key to encrypt/decrypt information. In this way, Mesos knows if an authorized framework wishes to communicate with it.

Note that the Mesos framework authentication is not the same as the one executor uses to run tasks. Full documentation is available at: `http://mesos.apache.org/documentation/latest/authorization/`.

Authentication configuration

The authentication module has these configuration options:

- Master:
  - `--[no-]authenticate`: If it is `true`, the registration of authenticated frameworks is allowed. If it is `false`, other frameworks can register too.
  - `--[no-]authenticate_slaves`: If it is `true`, only the registration of authenticated slaves is allowed. If it is `false`, other slaves can register too.
  - `--authenticators`: Specifies the authentication mechanism to authenticate the master, the default is CRAM-MD5.
  - `--modules`: Usually used to add other authentication mechanisms.
  - `--credentials`: The location of the file containing a valid credentials list and it specifies which authentication mechanism is employed.
- Slave:
  - `--authenticate`: To specify which authentication mechanism is to be used to authenticate the slave, the default is CRAM-MD5.
  - `--credential`: The location of the file containing valid credentials to be used to authenticate the slave.

Framework authorization

Since Mesos version 0.24, the authorization API lets sys admin implement their favorite backend protocols, for example, an LDAP. When starting the Mesos master, we specify the authorization with the `--acls` flag.

Supported authorizations include:

1. Framework registration with specified roles.
2. Frameworks as authorized users for task launching.
3. Framework shutdown by authorized principals.
4. Resources reservation and release by authorized principals.
5. Persistent volume creation and destruction by authorized principals.
6. Quota setting by authorized principals.

Access control lists

The ACLs are used to implement access control on Mesos. The ACLs are defined in JSON format for each of the six possible authorizations. Every ACL has a list of subjects to perform actions on an object group. The Mesos master is responsible for checking whether the requests are authorized or not.

The ACLs are checked in setup order; the first relevant ACL is used to determine the request authorization. The ACLs permissive fields determines how a request without a match should be treated. The default value is `true` so, if no matching ACL exists, then the request is authorized.

The ACL of supported `Actions` are:

- `register_frameworks`: For framework registration
- `run_tasks`: To run tasks
- `shutdown_frameworks`: For framework termination
- `set_quotas`: For quota setting
- `reserve_resources`: For resource reservation
- `unreserve_resources`: To free up resources
- `create_volumes`: For persistent volume creation
- `destroy_volumes`: For persistent volume destruction

The ACL of supported objects include:

- Roles
- Users
- Resources
- volume_types
- framework_principals
- creator_principals
- reserver_principals

Here we have two examples:

- `run_tasks` ACL: When a framework tries to launch a set of tasks, the master checks the ACL to verify whether the framework is authorized to run these tasks with the specified user. If it has no authorization a `TASK_LOST` message is sent to the framework and the task is not launched.
- `register_frameworks` ACL: When a framework tries to register with the master, the master checks the ACL to verify whether the framework is authorized to receive resource offers to that role. If it has no authorization an error message is sent to the framework and the scheduler is terminated.

Spark Mesos run modes

Spark can run over Mesos in two modes: **coarse-grained** (default) and **fine-grained** (deprecated).

Coarse-grained

In coarse-grained mode, each Spark executor runs as a single Mesos task. Spark executors are sized according to the following configuration variables:

- Executor memory: `spark.executor.memory`
- Executor cores: `spark.executor.cores`
- Number of executors: `spark.cores.max/spark.executor.cores`

Executors are brought up when the application starts, until `spark.cores.max` is reached. If `spark.cores.max` is not set, the Spark application will reserve all resources offered to it by Mesos, so it is highly recommended to set this variable in any sort of multi-tenant cluster, including those running multiple concurrent Spark applications.

The scheduler will start the executors round-robin on the offers Mesos gives it, but there are no spread guarantees, as Mesos does not provide such guarantees on the offer stream.

The benefit of the coarse-grained mode is a much lower startup overhead, but this is at the cost of reserving Mesos resources for the complete duration of the application.

Fine-grained

Fine-grained mode is deprecated for Spark 2.0.0. Consider using dynamic allocation to obtain the benefits.

In fine-grained mode, each Spark task inside the Spark executor runs as a separate Mesos task. This allows multiple instances of Spark (and other frameworks) to share cores at a very fine granularity, where each application gets more or fewer cores as it ramps up and down, but it comes with an additional overhead in launching each task. This mode may be inappropriate for low-latency requirements such as interactive queries or serving web requests.

Note that, while Spark tasks in fine-grained will relinquish cores as they terminate, they will not relinquish memory, as the JVM does not give memory back to the operating system. Neither will executors terminate when they're idle.

To run in fine-grained mode, set the `spark.mesos.coarse` property in your `SparkConf` to `false`:

```
conf.set("spark.mesos.coarse", "false")
```

You may also make use of `spark.mesos.constraints` to set attribute-based constraints on Mesos resource offers. By default, all resource offers will be accepted:

```
conf.set("spark.mesos.constraints", "os:centos7;us-east-1:false")
```

For example, let's say `spark.mesos.constraints` is set to `os:centos7;us-east-1:false`, then the resource offers will be checked to see if they meet both these constraints and only then it will be accepted to start new executors.

Apache Mesos API

Mesos has an API so that developers can build custom frameworks to run on top of the infrastructure. As you can imagine, Mesos implements the actor model for message passing, because the complexity increases without a non-blocking communication. This also leverages protocol buffers.

Scheduler HTTP API

Support for the new HTTP API was introduced in the Mesos version 0.24. The Mesos Master has the `/api/v1/scheduler` endpoint which the scheduler communicates.

For detailed information about the scheduler HTTP API, go to:

`http://mesos.apache.org/documentation/latest/scheduler-http-api/`

Requests

The master accepts the following request calls.

SUBSCRIBE

To enable communication with the master, the scheduler sends a `SUBSCRIBE` message via `HTTP POST`. The response contains the subscription confirmation and the framework ID to continue the conversation.

The `SUBSCRIBE` JSON request structure is:

```
POST /api/v1/scheduler HTTP/1.1 {
  "type" : "SUBSCRIBE",
  "force" : true,
  "subscribe" : {
    "framework_info" : {
      "user" : "<user>",
      "name" : "<name>"
    }
  }
}
```

The `SUBSCRIBE` response is:

```
HTTP/1.1 200 OK
```

TEARDOWN

To shutdown itself, the scheduler sends a `TEARDOWN` message. Mesos terminates the executors and kills all the running tasks of the process, then removes the framework and ends the communication with the scheduler.

The `TEARDOWN` JSON request structure is:

```
POST /api/v1/scheduler HTTP/1.1 {
  "type" : "TEARDOWN",
  "framework_id" : {
    "value" : "<framework_id>"
  }
}
```

The `TEARDOWN` response is:

```
HTTP/1.1 202 Accepted
```

ACCEPT

To accept resource offers from the master, the scheduler sends an `ACCEPT` message. The actions are specified as parameters in the request, for example, task launching, volume creation, and resource reservation.

The `ACCEPT` JSON request structure is:

```
POST /api/v1/scheduler HTTP/1.1 {
  "type" : "ACCEPT",
  "framework_id" : {
    "value" : "<framework_id>"
  },
  "accept" : {
    "offer_ids" : [
      {"value" : "<value_1>"},
      {"value" : "<value_2>"}
    ],
    "operations" : [
      {"type" : "LAUNCH",
      "launch" : {...}} ],
    "filters" : {...}
  }
}
```

The `ACCEPT` response is:

```
HTTP/1.1 202 Accepted
```

DECLINE

To decline the resources offered by the master, the scheduler sends a `DECLINE` message.

The `DECLINE` JSON request structure is:

```
POST /api/v1/scheduler HTTP/1.1 {
  "type" : "DECLINE",
  "framework_id" : {
    "value" : "<framework_id>"
  },
  "decline" : {
    "offer_ids" : [
      {"value" : "<value_1>"},
      {"value" : "<value_2>"}
    ],
  "filters" : {...}
  }
}
```

The `DECLINE` response is:

```
HTTP/1.1 202 Accepted
```

REVIVE

To remove the filters set by an `ACCEPT` request or a `DECLINE` request the scheduler sends a `REVIVE` message.

The `REVIVE` JSON request structure is:

```
POST /api/v1/scheduler HTTP/1.1 {
  "type" : "REVIVE",
  "framework_id" : {
    "value" : "<framework_id>"
  },
}
```

The `REVIVE` response is:

```
HTTP/1.1 202 Accepted
```

KILL

To kill a specific task, the scheduler sends a `KILL` message. The request is passed to the executor. If the master is not aware of the task then it generates a `TASK_LOST` message.

The `KILL` JSON request structure is:

```
POST /api/v1/scheduler HTTP/1.1 {
  "type" : "KILL",
  "framework_id" : {
    "value" : "<framework_id>"},
  "kill" : {
    "task_id" : { "value" : "<task_id>" },
    "agent_id" : {"value" : "<agent_id>"}
  }
}
```

The `KILL` response is:

```
HTTP/1.1 202 Accepted
```

SHUTDOWN

To end specific custom executors the scheduler sends a `SHUTDOWN` message to the master.

The `SHUTDOWN` JSON request structure is:

```
POST /api/v1/scheduler HTTP/1.1 {
  "type" : "SHUTDOWN",
  "framework_id" : {"value" : "<framework_id>"},
  "shutdown" : {
    "executor_id" : {"value" : "<executor_id>"},
    "agent_id" : {"value" : "<agent_id>"}
  }
}
```

The `SHUTDOWN` response is:

```
HTTP/1.1 202 Accepted
```

ACKNOWLEDGE

The scheduler sends this message to acknowledge a status update.

The `ACKNOWLEDGE` JSON request structure is:

```
POST /api/v1/scheduler HTTP/1.1 {
  "type" : "ACKNOWLEDGE",
  "framework_id" : {"value" : "<framework_id>"},
  "acknowledge" : {
    "agent_id" : {"value" : "<agent_id>"},
    "task_id" : {"value" : "<task_id>"},
    "uuid" : "<uuid>"
  }
}
```

The `ACKNOWLEDGE` response is:

```
HTTP/1.1 202 Accepted
```

RECONCILE

The scheduler sends this request when the status of any terminal tasks (teardown, kill, shutdown, and so on) needs to be queried. An update event for every task in the list is sent back by the master.

The `RECONCILE` JSON request structure is:

```
POST /api/v1/scheduler HTTP/1.1 {
  "type" : "RECONCILE",
  "framework_id" : {"value" : "<framework_id>"},
  "reconcile" : {
    "tasks" : [   {
      "task_id" : { "<task_id>" },
      "agent_id" : { "<agent_id>" }
    }]
  }
}
```

The `RECONCILE` response is:

```
HTTP/1.1 202 Accepted
```

MESSAGE

The scheduler sends this request when any arbitrary binary data needs to be sent to the executor.

The `MESSAGE` JSON request structure is:

```
POST /api/v1/scheduler HTTP/1.1
{
  "type" : "MESSAGE",
  "framework_id" : {"value" : "<framework_id>"},
  "message" : {
    "agent_id" : {"value" : "<agent_id>"},
    "executor_id" : {"value" : "<executor_id>"},
    "data" : "<data>"
  }
}
```

The `MESSAGE` response is:

```
HTTP/1.1 202 Accepted
```

REQUEST

To request resources from the master, the scheduler uses this request.

The `REQUEST` JSON request structure is:

```
POST /api/v1/scheduler HTTP/1.1 {
  "type" : "REQUEST",
  "framework_id" : {"value" : "<framework_id>"},
  "requests" : [{
    "agent_id" : {"value" : "<agent_id>"},
    "resources" : {...}
  }]
}
```

The `REQUEST` response is:

```
HTTP/1.1 202 Accepted
```

Responses

The master sends the following responses.

SUBSCRIBED

When the scheduler sends a subscribe request, the master first sends the `SUBSCRIBED` event:

The `SUBSCRIBED` JSON event structure is:

```
<event-length> {
  "type" : "SUBSCRIBED",
  "subscribed" : {
    "framework_id" : {"value":"<framework_id>"},
    "heartbeat_interval_seconds" : 10
  }
}
```

OFFERS

When the master can offer a new resource set to the frameworks, it sends this event. Every offer matches a resources group in a slave. Until the scheduler sends an `ACCEPT` or `DECLINE` call to the master, the resources are assumed to be allocated. If the slave is lost or timed-out the offer is rescinded.

The `OFFERS` JSON event structure is:

```
<event-length> {
  "type" : "OFFERS",
  "offers" : [{
    "offer_id":{"value": "<offer_id>"},
    "framework_id":{"value": "<framework_id>"},
    "agent_id":{"value": "<agent_id>"},
    "hostname":"host_name",
    "resources":[...],
    "attributes":[...],
    "executor_ids":[...]
  }]
}
```

RESCIND

If the validity of the offer expires, the master sends this event to rescind the offer. All the future scheduler calls are considered invalid.

The `RESCIND` JSON event structure is:

```
<event-length> {
  "type":"RESCIND",
  "rescind":{
    "offer_id":{"value":"<offer_id>"}
  }
}
```

UPDATE

If the executor creates a status update, the master sends this event.

The `UPDATE` JSON event structure is:

```
<event-length> {
  "type" : "UPDATE",
  "update" : {
    "status" : {
      "task_id" : { "value" : "<task_id>"},
      "state" : "TASK_FINISHED",
      "source" : "SOURCE_EXECUTOR",
      "uuid" : "<uuid>",
      "bytes" : "<some_data>"
    }
  }
}
```

MESSAGE

The Mesos master forwards messages generated by the executor to the scheduler. Note that interpretation or delivery are not guaranteed; if the delivery fails, the executor needs to resend the message.

The `MESSAGE` JSON event structure is:

```
<event-length> {
  "type":"MESSAGE",
  "message":{
    "agent_id":{"value":"<agent_id>"},
    "executor_id":{"value":"<executor_id>"},
    "data":"<some_data>"
  }
}
```

FAILURE

The master sends this event about a slave removal or an executor termination.

The `FAILURE` JSON event structure is:

```
<event-length> {
  "type":"FAILURE",
  "failure":{
    "agent_id":{"value":"<agent_id>"},
    "executor_id":{"value":"<executor_id>"},
    "status": 1
  }
}
```

ERROR

When an error happens, the master sends this event, for example, an unauthorized framework makes a resource request. If a framework receives this event, it should be subscribed again.

The `ERROR` JSON event structure is:

```
<event-length> {
  "type":"ERROR",
  "message":" Not authorized framework"
}
```

HEARTBEAT

To test a scheduler communication, the master sends this message periodically. Remember that a connection could be terminated by insufficient data interchange.

The `HEARTBEAT` JSON event structure is:

```
<event-length> {
  "type":"HEARTBEAT",
}
```

Mesos containerizers

In this section we'll learn about containers and Docker. We will expose an overview of the options for Mesos containerizers. We'll also discuss advanced topics such as networking and cache.

Containers

A Linux container, for simplicity called simply **container**, is an environment to run applications with a common share of resources. As all the containers share the host machine operating system, their creation is very fast.

Container technology based on operating system virtualization has been around since 2004. In OS level virtualization the OS kernel creates many user containers.

The containers are individual encapsulated components running isolated instances on the same kernel. It's easy for the developer to just package the application and its dependencies into a container and deploy it in another environment.

In today's DevOps era, our portable applications are easy to update and move between environments. If we have a desktop development environment, the move to the production environment is really easy.

In the modern virtualization war, there are two types of technologies:

- **Hypervisor-based**: This model includes redundant OS kernels and other libraries used to create an efficient setup. For example VMWare and Xen bare metal.
- **Container-based**: This approach involves the creation of encapsulated deployable components running in isolation on the same kernel. For example, Docker.

The following are the benefits of using containers:

- Agile development
- Application deployment simplification
- Centric management
- Consistency of infrastructure environment
- Continuous deployment and integration
- Deployment and build separation

- High portability
- Micro-services breakdown
- Resource isolation
- Resource utilization

Docker containerizers

We can define Docker as an open source platform that automates application deployment. We define a container as a portable, lightweight, auto sufficient module to be deployed on a container platform.

Docker is based on the **Linux Container** (**LXC**) model. Docker is not just an underlying technology; it acts as an abstraction layer to package and containerize applications and dependencies. Imagine Docker containers as shipping containers for applications and dependencies.

Docker has three main benefits:

- **Agility**: The application development is done quickly and efficiently, because the IT Ops department also has flexible interaction
- **Control**: We have a version control, to ensure code authorship
- **Portability**: We can move from a single developer environment to a cloud environment immediately

Docker is a DevOps tool, reconciling the Development team and the IT Operations team:

- **Development team**: Concerned with what is shipped on the container: code, data, apps, libraries
- **IT Operations**: Focused on the container environment: logging, monitoring, access control, and network configuration

The Docker process has three phases:

- **Build**: With Docker we can create micro services applications, so there is no restriction on the platform and language.
- **Ship**: We can model the entire software model: development, testing and distribution. The modern user interface simplifies the shipment management.
- **Rkun**: Docker is available on a wide variety of platforms to deploy scalable services.

In the Docker eco-system, we have the following main concepts:

- **Docker image**. An ordered collection of root filesystem changes and the corresponding execution parameters to use within the container runtime.
- **Base image**. A Docker image without parents.
- **Dockerfile**: A text document that contains the commands to build a Docker image. Docker builds images reading the instructions from a Dockerfile.
- **Docker engine**: The container runtime instance can be installed on physical, virtual or cloud environments.
- **Docker container**. A runtime instance of a Docker image. A Docker container has three parts:
    - A Docker image
    - An execution environment
    - A standard set of instructions
- **Docker registry**: A hosted service containing repositories of images corresponding to the Registry API. The default registry could be accessed with a browser or using the Docker search command.
- **Docker hub**: A centralized resource for working with Docker and its components. It provides the following services:
    - **Docker tag**: A label applied to a Docker image in a repository.This is the way to identify the various images in a repository.
    - **Kinematic**: A desktop GUI for Docker.
    - **Virtual machine**: A program that emulates a complete computer and imitates dedicated hardware. It shares physical hardware resources but isolates the operating system. The user has the same experience as if they were on a dedicated machine.
- **Docker swarm**: A native clustering tool for Docker. It pools together several Docker hosts and exposes them as single virtual hosts. It serves the standard Docker AP; if a tool works with Docker it can transparently scale up to multiple hosts.
- **Docker trusted registry**: The enterprise grade image storage solution from Docker. We can securely store and manage the Docker images we use in our applications.
- **Docker Toolbox**: The installer for Mac and Windows environments.
- **Docker Compose**: A tool for defining and running complex applications. We can define a multi-container application in a single file and then create a single command which does everything when we run it.

- **Docker machine**: A tool which makes it easy to create Docker hosts. It creates servers, installs Docker on hosts and configures the Docker client to talk to hosts.
    - User authentication
    - Docker image hosting
    - Automated image builds and workflow (for example, build triggers and web hooks)
    - Github and Bitbucket integration

The Docker container encloses everything needed to run: code, runtime, libraries, and dependencies. The encapsulation provided warrants that the container runs in the same way independently of the environment.

The difference between a **Virtual Machine** and a **Docker Container** is shown in figure 9.1.

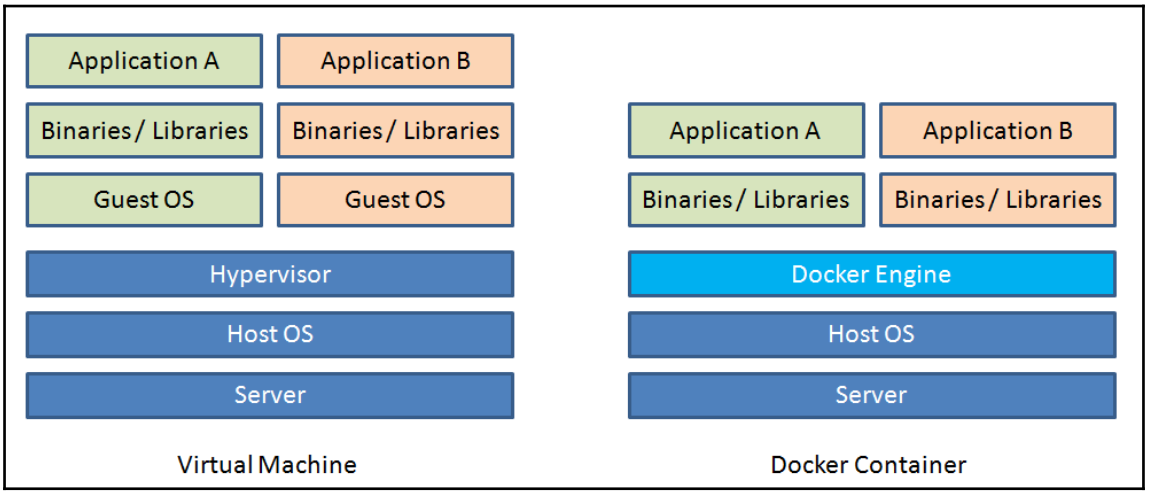

Figure 9.1. Comparison between a virtual machine and a Docker container

As we can see in figure 9.1, the Docker container is made up of only application binaries and its associated dependencies. The Docker container has resource isolation like virtual machines but it is fully portable. The Docker architecture has several open APIs such as Swarm, Compose, and Hub that support the Docker ecosystem.

Docker provides cloud portability and unties the DevOps team from specific environment tools. Docker increases efficiency by reducing storage and infrastructure costs.

Containers and containerizers

Containers are used for:

- Isolating tasks from each other
- Programmatically controlling the individual resources of tasks
- Running applications in different environments
- Breaking the application into manageable micro-services

The tasks run in containers via containerizers. The wonder is that Apache Mesos could run on Docker and also have its own container technology. At the time of writing, support for Mesos on Docker is relatively recent.

One fundamental requirement of a cluster manager is to ensure that the resource allocation on a particular framework has no impact on the framework's active jobs. One key Mesos feature is the isolation mechanism to compartmentalize tasks on slaves. The Mesos pluggable architecture uses containers for resource isolation.

The Mesos slave uses the containerizer API to run the executor and its tasks. The containerizer API supports several implementations to develop custom containerizers. When we start a slave process, we can specify which set of isolators it uses and the containerizer.

Types of containerizers

In Mesos version 1.0.0, the following containerizer options are available:

- Composing
- Docker
- Mesos (default)

The containerizer type is specified with the agent flag `-containerizers`; for example:

```
--containerizers=mesos
```

Creating containerizers

The containerizer is created by the slave with the flag `--containerizers`, if multiple containerizers are specified we use the composing containerizer. The Mesos agent uses the default task executor, and with the `--TaskInfo` flag we can specify another.

A comparison between containerization in Mesos with the **Mesos Containerizer** and **Docker Containerizer** is shown in figure 9.2

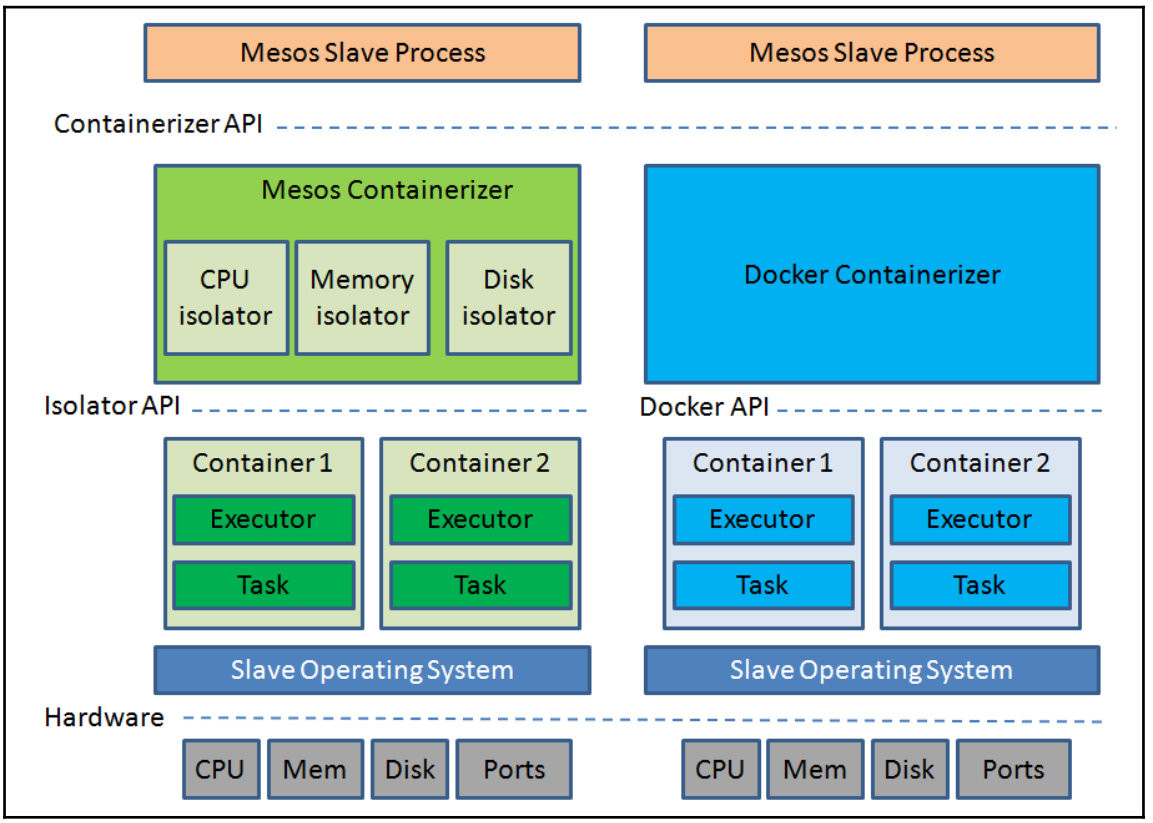

Figure 9.2. Containerization in Mesos

Mesos containerizer

For full and updated information, visit:

`http://mesos.apache.org/documentation/latest/mesos-containerizer/`

Mesos provides a default containerizer. We enable it with the configuring agent flag:

```
--containerizers=mesos
```

If a task does not specify the `ContainerInfo` it's assumed that it is handled by the Mesos containerizer.

We use this containerizer type when:

- The user does not have another container solution
- The user requires control over specific resources and is not available from another container solution
- We want a fine-grained control over the operating system
- Each task requires a custom resource isolation

Launching Mesos containerizer

As described in the Mesos chapter, isolators are responsible for creating an environment for the containers where resources such as the CPU, network, storage and memory can be separated from other containers. The launching process has the following steps:

1. Every isolator prepares the calls.
2. The launcher forks the executor, but the forked process cannot execute until the isolation is completed.
3. By calling the isolators the executor is isolated.
4. The executor is fetched.
5. The forked process is notified to execute.
6. The isolator preparation commands run.
7. The execution runs.

The Mesos containerizer states are:

- `PREPARING`: Calls are prepared on each isolator
- `ISOLATING`: Resources are isolated from other containers
- `FETCHING`: Fetches the executor

- `RUNNING`: Waiting for instructions
- `DESTROYING`: Launcher destroys a container

Architecture of Mesos containerizer

The Mesos containerizer has resource isolation and lightweight containerization to support Linux features such as namespaces and control groups. The Mesos containerizer also has the capability to enable selectively the isolators, and also has support for POSIX process.

The three isolator options are follows:

Shared filesystem

On Linux hosts, the modifications on every shared filesystem can be enabled using the Shared Filesystem isolator. The modifications are specified with the `default_container_infoagent` flag and with the framework in `ContainerInfo`.

The volumes used to map sections on the shared filesystem are specified through the `ContainerInfo`.

The path can be absolute or relative. The absolute is accessible for every container on the container path. The relative means the executor work directory. The directory is created and the permissions are copied.

This isolator is commonly used to make certain filesystem directories private among containers.

PID namespace

Isolating each container in a different PID namespace has the following advantages:

- **Visibility**: The executor and child processes running in the container can't view or interact with external processes running outside this namespace.
- **Clean termination**: If the father process in a PID namespace is terminated, all the other processes running in the namespace are terminated.

Posix disk

This isolator can be used in OS X and Linux and provides basic disk isolation. Normally it is used to reinforce disk quotas. To enable this isolator we must add the `posix/disk` to the `--isolation` flag.

The disk quota reinforcement is disabled by default, to enable it use `--enforce_container_disk_quota` when starting the slave. To retrieve disk statistics, we configure the `/monitor/statistics.json`

Disk utilization is reported by running the `du` command periodically. To configure the time interval between commands use the agent flag: `--container_disk_watch_interval`. The default value is 15 seconds.

Docker containerizers

To run Mesos tasks inside a Docker container we use this containerizer. The Docker image can also be launched as an executor or a task. We enable the Docker containerizer with the agent flag:

```
--containerizers=docker
```

We use this containerizer type when:

- The running tasks use the Docker package.
- One Mesos slave runs within a Docker container

For full and updated information visit:

`http://mesos.apache.org/documentation/latest/docker-containerizer/`

Docker containerizer setup

To enable the Docker Containerizer on a slave, the slave must be launched with "Docker" as one of the containerizer options:

```
mesos-slave --containerizers=docker
```

The Docker Command Line Interface (CLI) client must be installed on every slave where the Docker containerizer is specified.

If the `iptables` are enabled on the slave, the `iptables` must permit all traffic from the bridge interface with this command:

```
iptables –A INPUT –s 172.17.0.0/16 –i docker0 –p tcp –j ACCEPT
```

Launching the Docker containerizers

The launching process has the following steps:

1. The task is attempted only if `ContainerInfo.type` is set to `DOCKER`
2. The image is pulled from the specified repository
3. The pre-launch hook is called
4. The executor is launched in one of the following two scenarios:
5. The Mesos agent runs in a Docker container in these cases:
    - If the flag `--docker_mesos_image` is present
    - If the value of the flag `--docker_mesos_image` is considered to be the Docker image used for launching the Mesos agent
    - If an executor different from the default command executor is used to run the task
    - If the task uses `TaskInfo`, the default `mesos-dockerexecutor` is launched in a Docker container to execute commands through the Docker CLI

6. The Mesos agent does not run in a Docker container in these cases:
    - If a task uses a custom executor to run.
    - If a task uses `TaskInfo`, a sub-process to run default `mesos-docker-executor`, is forked. Shells are spawned by this executor to run Docker commands through the Docker CLI.

The Docker containerizer states are:

- `FETCHING`: Fetches the executor
- `PULLING`: Docker image pulls the image
- `RUNNING`: Waiting for instructions
- `DESTROYING`: Launcher destroys the container

Composing containerizers

This containerizers type allows the different container technologies to be combined to work together. We can enable this type with the `--containerizers` agent flag specifying the coma separated list with the containerizer names:

```
--containerizers=mesos,docker
```

The first containerizer in the list is used for task launching and will provide support for the container configuration of the task.

We use this containerizer type when we need to test tasks with different resource isolation types.

A framework can leverage this containerizer type to test a task using the controlled environment provided by the Mesos containerizer and simultaneously ensure that the task works in Docker containers.

Summary

In this study case, we've covered the Mesos API for framework development. We've also studied how to use Mesos and Docker containerizers.

The Mesos framework version 1.0.0 was released on 07/29/2016 so it's a very new technology. We could look at a complete Mesos framework creation, but it is beyond the scope of this book.

The first release of Docker was on 03/13/2013 and version 1.12.0 was released on 07/14/2016. Both technologies are still new and promise good things.

Index

A

B

C

D

E

F

G

H

I

N

O

P

R

S

T

U

V

W

Z