



Community Experience Distilled

Mastering Apache Spark

Gain expertise in processing and storing data by using advanced techniques with Apache Spark

*Foreword by Andrew Szymanski,
Cloudera Certified Hadoop Administrator/Big Data Specialist*

Mike Frampton

[PACKT] open source*
PUBLISHING community experience distilled

www.allitebooks.com

Mastering Apache Spark

Gain expertise in processing and storing data by using advanced techniques with Apache Spark

Mike Frampton



BIRMINGHAM - MUMBAI

Mastering Apache Spark

Copyright © 2015 Packt Publishing

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

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

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

First published: September 2015

Production reference: 1280915

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

ISBN 978-1-78398-714-6

www.packtpub.com

Credits

Author

Mike Frampton

Project Coordinator

Kinjal Bari

Reviewers

Andrea Mostosi

Toni Verbeiren

Lijie Xu

Proofreader

Safis Editing

Indexer

Rekha Nair

Commissioning Editor

Kunal Parikh

Graphics

Jason Monteiro

Acquisition Editor

Nadeem Bagban

Production Coordinator

Manu Joseph

Content Development Editor

Riddhi Tuljapurkar

Cover Work

Manu Joseph

Technical Editor

Rupali R. Shrawane

Copy Editor

Yesha Gangani

Foreword

Big data is getting bigger and bigger day by day. And I don't mean tera, peta, exa, zetta, and yotta bytes of data collected all over the world every day. I refer to complexity and number of components utilized in any decent and respectable big data ecosystem. Never mind the technical nitties gritties – just keeping up with terminologies, new buzzwords, and hypes popping up all the time can be a real challenge in itself. By the time you have mastered them all, and put your hard-earned knowledge to practice, you will discover that half of them are old and inefficient, and nobody uses them anymore. Spark is not one of those "here today, gone tomorrow" fads. Spark is here to stay with us for the foreseeable future, and it is well worth to get your teeth into it in order to get some value out of your data NOW, rather than in some, errr, unforeseeable future. Spark and the technologies built on top of it are the next crucial step in the big data evolution. They offer 100x faster in-memory, and 10x on disk processing speeds in comparison to the traditional Hadoop jobs.

There's no better way of getting to know Spark than by reading this book, written by Mike Frampton, a colleague of mine, whom I first met many, many years ago and have kept in touch ever since. Mike's main professional interest has always been data and in pre-big data days, he worked on data warehousing, processing, and analyzing projects for major corporations. He experienced the inefficiencies, poor value, and frustrations that the traditional methodologies of crunching the data offer first hand. So understanding big data, what it offers, where it is coming from, and where it is heading, and is intrinsically intuitive to him. Mike wholeheartedly embraced big data the moment it arrived, and has been devoted to it ever since. He practices what he preaches, and is not in it for money. He is very active in the big data community, writes books, produces presentations on SlideShare and YouTube, and is always first to test-drive the new, emerging products.

Mike's passion for big data, as you will find out, is highly infectious, and he is always one step ahead, exploring the new and innovative ways big data is used for. No wonder that in this book, he will teach you how to use Spark in conjunction with the very latest technologies; some of them are still in development stage, such as machine learning and Neural Network. But fear not, Mike will carefully guide you step by step, ensuring that you will have a direct, personal experience of the power and usefulness of these technologies, and are able to put them in practice immediately.

Andrew Szymanski

Cloudera Certified Hadoop Administrator/Big Data Specialist

About the Author

Mike Frampton is an IT contractor, blogger, and IT author with a keen interest in new technology and big data. He has worked in the IT industry since 1990 in a range of roles (tester, developer, support, and author). He has also worked in many other sectors (energy, banking, telecoms, and insurance). He now lives by the beach in Paraparaumu, New Zealand, with his wife and teenage son. Being married to a Thai national, he divides his time between Paraparaumu and their house in Roi Et, Thailand, between writing and IT consulting. He is always keen to hear about new ideas and technologies in the areas of big data, AI, IT and hardware, so look him up on LinkedIn (<http://linkedin.com/profile/view?id=73219349>) or his website (<http://www.semtech-solutions.co.nz/#!/pageHome>) to ask questions or just to say hi.

I would like to acknowledge the efforts of the open source development community who offer their time, expertise and services in order to help develop projects like Apache Spark. I have been continuously impressed by the speed with which this development takes place, which seems to exceed commercial projects. I would also like to mention the communities that grow around open source products, and people who answer technical questions and make books like this possible.

There are too many people that have helped me technically with this book and I would like to mention a few. I would like to thank Michal Malohlava at <http://h2o.ai/> for helping me with H2O, and Arsalan Tavakoli-Shiraji at <https://databricks.com/> for answering my many questions. I would also like to thank Kenny Bastani for allowing me to use his Mazerunner product.

Riddhi Tuljapurkar, at Packt, and the book reviewers have put in a sterling effort to help push this book along. Finally, I would like to thank my family who have allowed me the time develop this book through the months of 2015.

About the Reviewers

Andrea Mostosi is a technology enthusiast. Innovation lover since when he was a child, he started his professional job in the early 2000s, and has worked on several projects playing almost every role in the computer science environment. He is currently the CTO at The Fool, a company that tries to make sense of data. During his free time, he likes travelling, running, cooking, biking, reading, observing the sky, and coding.

I would like to thank my wonderful girlfriend Khadija, who lovingly supports me in everything I do. I would also thank my geek friends: Simone M, Daniele V, Luca T, Luigi P, Michele N, Luca O, Luca B, Diego C, and Fabio B. They are the smartest people I know, and comparing myself with them has always pushed me to be better.

Toni Verbeiren received his PhD in theoretical physics in 2003. He has worked on models of artificial neural networks, entailing mathematics, statistics, simulations, (lots of) data, and numerical computations. Since then, he has been active in this industry in a range of domains and roles: infrastructure management and deployment, service and IT management, and ICT/business alignment and enterprise architecture. Around 2010, he started picking up his earlier passion, which is now called Data Science. The combination of data and common sense can be a very powerful basis for making decisions and analyzing risk.

Toni is active as owner and consultant at Data Intuitive (<http://www.data-intuitive.com/>) in all the things related to (big) data science, and its applications to decision and risk management. He is currently involved in ExaScience Life (<http://www.exascience.com/>), and the Visual Data Analysis Lab (<http://vda-lab.be/>), concerning scaling up visual analysis of biological and chemical data.

I'd like to thank various employers, clients, and colleagues for the various pieces of insight and wisdom that they shared with me. I'm grateful to the Belgian and Flemish government (FWO, IWT) for financial support of the academic projects mentioned previously.

Lijie Xu is now a PhD student at Institute of Software, Chinese Academy of Sciences. His research interests focus on distributed systems and large-scale data analysis. He has both academic and industrial experience in Microsoft Research Asia, Alibaba Taobao, and Tencent. As an open source software enthusiast; he has contributed to Apache Spark, and has written a popular technical report named Spark Internals at <https://github.com/JerryLead/SparkInternals>. He believes that all things are difficult before they are easy.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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://www2.packtpub.com/books/subscription/packtlib>

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

Why subscribe?

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

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	vii
Chapter 1: Apache Spark	1
Overview	2
Spark Machine Learning	3
Spark Streaming	3
Spark SQL	4
Spark graph processing	4
Extended ecosystem	4
The future of Spark	5
Cluster design	5
Cluster management	8
Local	8
Standalone	8
Apache YARN	9
Apache Mesos	9
Amazon EC2	10
Performance	13
The cluster structure	13
The Hadoop file system	14
Data locality	14
Memory	14
Coding	15
Cloud	15
Summary	15

Chapter 2: Apache Spark MLlib	17
The environment configuration	17
Architecture	18
The development environment	19
Installing Spark	21
Classification with Naïve Bayes	25
Theory	25
Naïve Bayes in practice	26
Clustering with K-Means	36
Theory	36
K-Means in practice	36
ANN – Artificial Neural Networks	41
Theory	41
Building the Spark server	44
ANN in practice	48
Summary	59
Chapter 3: Apache Spark Streaming	61
Overview	62
Errors and recovery	63
Checkpointing	64
Streaming sources	66
TCP stream	67
File streams	69
Flume	70
Kafka	82
Summary	94
Chapter 4: Apache Spark SQL	95
The SQL context	96
Importing and saving data	97
Processing the Text files	97
Processing the JSON files	97
Processing the Parquet files	100
DataFrames	101
Using SQL	104
User-defined functions	110
Using Hive	115
Local Hive Metastore server	115
A Hive-based Metastore server	121
Summary	128

Chapter 5: Apache Spark GraphX	131
Overview	131
GraphX coding	134
Environment	134
Creating a graph	137
Example 1 – counting	138
Example 2 – filtering	139
Example 3 – PageRank	140
Example 4 – triangle counting	141
Example 5 – connected components	141
Mazerunner for Neo4j	143
Installing Docker	144
The Neo4j browser	147
The Mazerunner algorithms	149
The PageRank algorithm	150
The closeness centrality algorithm	150
The triangle count algorithm	151
The connected components algorithm	152
The strongly connected components algorithm	152
Summary	153
Chapter 6: Graph-based Storage	155
Titan	156
TinkerPop	157
Installing Titan	158
Titan with HBase	159
The HBase cluster	159
The Gremlin HBase script	160
Spark on HBase	164
Accessing HBase with Spark	165
Titan with Cassandra	169
Installing Cassandra	169
The Gremlin Cassandra script	172
The Spark Cassandra connector	173
Accessing Cassandra with Spark	174
Accessing Titan with Spark	177
Gremlin and Groovy	179
TinkerPop's Hadoop Gremlin	181
Alternative Groovy configuration	183
Using Cassandra	184
Using HBase	185

Using the file system	185
Summary	187
Chapter 7: Extending Spark with H2O	189
Overview	190
The processing environment	190
Installing H2O	191
The build environment	192
Architecture	195
Sourcing the data	198
Data quality	199
Performance tuning	200
Deep learning	200
The example code – income	202
The example code – MNIST	207
H2O Flow	208
Summary	220
Chapter 8: Spark Databricks	221
Overview	222
Installing Databricks	222
AWS billing	224
Databricks menus	225
Account management	226
Cluster management	228
Notebooks and folders	231
Jobs and libraries	235
Development environments	240
Databricks tables	240
Data import	241
External tables	244
The DbUtils package	248
Databricks file system	250
Dbutils fsutils	250
The DbUtils cache	252
The DbUtils mount	252
Summary	253
Chapter 9: Databricks Visualization	255
Data visualization	255
Dashboards	261
An RDD-based report	263
A stream-based report	264

REST interface	275
Configuration	276
Cluster management	276
The execution context	277
Command execution	277
Libraries	278
Moving data	279
The table data	279
Folder import	281
Library import	282
Further reading	282
Summary	283
Index	285

Preface

Having already written an introductory book on the Hadoop ecosystem, I was pleased to be asked by Packt to write a book on Apache Spark. Being a practical person with a support and maintenance background, I am drawn to system builds and integration. So, I always ask the questions "how can the systems be used?", "how do they fit together?" and "what do they integrate with?" In this book, I will describe each module of Spark, and explain how they can be used with practical examples. I will also show how the functionality of Spark can be extended with extra libraries like H2O from <http://h2o.ai/>.

I will show how Apache Spark's Graph processing module can be used in conjunction with the Titan graph database from Aurelius (now DataStax). This provides a coupling of graph-based processing and storage by grouping together Spark GraphX and Titan. The streaming chapter will show how data can be passed to Spark streams using tools like Apache Flume and Kafka.

Given that in the last few years there has been a large-scale migration to cloud-based services, I will examine the Spark cloud service available at <https://databricks.com/>. I will do so from a practical viewpoint, this book does not attempt to answer the question "server or cloud", as I believe it to be a subject of a separate book; it just examines the service that is available.

What this book covers

Chapter 1, Apache Spark, will give a complete overview of Spark, functionalities of its modules, and the tools available for processing and storage. This chapter will briefly give the details of SQL, Streaming, GraphX, MLlib, Databricks, and Hive on Spark.

Chapter 2, Apache Spark MLlib, covers the MLlib module, where MLlib stands for Machine Learning Library. This describes the Apache Hadoop and Spark cluster that I will be using during this book, as well as the operating system that is involved – CentOS. It also describes the development environment that is being used: Scala and SBT. It provides examples of both installing and building Apache Spark. A worked example of classification using the Naïve Bayes algorithm is explained, as is clustering with KMeans. Finally, an example build is used to extend Spark to include some Artificial Neural Network (ANN) work by Bert Greevenbosch (www.bertgreevenbosch.nl). I have always been interested in neural nets, and being able to use Bert's work (with his permission) in this chapter was enjoyable. So, the final topic in this chapter classifies some small images including distorted images using a simple ANN. The results and the resulting score are quite good!

Chapter 3, Apache Spark Streaming, covers the comparison of Apache Spark to Storm and especially Spark Streaming, but I think that Spark offers much more functionality. For instance, the data used in one Spark module can be passed to and used in another. Also, as shown in this chapter, Spark streaming integrates easily with big data movement technologies like Flume and Kafka.

So, the streaming chapter starts by giving an overview of checkpointing, and explains when you might want to use it. It gives Scala code examples of how it can be used, and shows the data can be stored on HDFS. It then moves on to give practical examples in Scala, as well as execution examples of TCP, File, Flume, and the Kafka streaming. The last two options are shown by processing an RSS data stream and finally storing it on HDFS.

Chapter 4, Apache Spark SQL, explains the Spark SQL context in Scala code terms. It explains File I/O as text, Parquet, and JSON formats. Using Apache Spark 1.3 it explains the use of data frames by example, and shows the methods that they make available for data analytics. It also introduces Spark SQL by Scala-based example, showing how temporary tables can be created, and how the SQL-based operations can be used against them.

Next, the Hive context is introduced. Initially, a local context is created and the Hive QL operations are then executed against it. Then, a method is introduced to integrate an existing distributed CDH 5.3 Hive installation to a Spark Hive context. Operations against this context are then shown to update a Hive database on the cluster. In this way, the Spark applications can be created and scheduled so that the Hive operations are driven by the real-time Spark engine.

Finally, the ability to create user-defined functions (UDFs) is introduced, and the UDFs that are created are then used in the SQL calls against the temporary tables.

Chapter 5, Apache Spark GraphX, introduces the Apache Spark GraphX module and the graph processing module. It works through a series of graph functions by example from based counting to triangle processing. It then introduces Kenny Bastani's Mazerunner work which integrates the Neo4j NoSQL database with Apache Spark. This work has been introduced with Kenny's permission; take a look at www.kennybastani.com.

This chapter works through the introduction of Docker, then Neo4j, and then it gives an introduction to the Neo4j interface. Finally, it works through some of the Mazerunner supplied functionality via the supplied REST interface.

Chapter 6, Graph-based Storage, examines graph-based storage as Apache Spark Graph processing was introduced in this book. I looked for a product that could integrate with Hadoop, was open sourced, could scale to a very high degree, and could integrate with Apache Spark.

Although it is still a relatively young product both in terms of community support and development, I think that Titan from Aurelius (now DataStax) fits the bill. The 0.9.x releases that are available, as I write now, use Apache TinkerPop for graph processing.

This chapter provides worked examples of graph creation and storage using Gremlin shell and Titan. It shows how both HBase and Cassandra can be used for backend Titan storage.

Chapter 7, Extending Spark with H2O, talks about the H2O library set developed at <http://h2o.ai/>, which is a machine learning library system that can be used to extend the functionality of Apache Spark. In this chapter, I examine the sourcing and installation of H2O, as well as the Flow interface for data analytics. The architecture of Sparkling Water is examined, as is data quality and performance tuning.

Finally, a worked example of deep learning is created and executed. *Chapter 2, Spark MLlib*, used a simple ANN for neural classification. This chapter uses a highly configurable and tunable H2O deep learning neural network for classification. The result is a fast and accurate trained neural model, as you will see.

Chapter 8, Spark Databricks, introduces the <https://databricks.com/> AWS cloud-based Apache Spark cluster system. It offers a step-by-step process of setting up both an AWS account and the Databricks account. It then steps through the <https://databricks.com/> account functionality in terms of Notebooks, Folders, Jobs, Libraries, development environments, and more.

It examines the table-based storage and processing in Databricks, and also introduces the DBUtils package for Databricks utilities functionality. This is all done by example to give you a good understanding of how this cloud-based system can be used.

Chapter 9, Databricks Visualization, extends the Databricks coverage by concentrating on data visualization and dashboards. It then examines the Databricks REST interface, showing how clusters can be managed remotely using various example REST API calls. Finally, it looks at data movement in terms of table's folders and libraries.

The cluster management section of this chapter shows that it is possible to launch Apache Spark on AWS EC2 using scripts supplied with the Spark release. The <https://databricks.com/> service takes this functionality a step further by providing a method to easily create and resize multiple EC2-based Spark clusters. It provides extra functionality for cluster management and usage, as well as user access and security as these two chapters show. Given that the people who brought us Apache Spark have created this service, it must be worth considering and examining.

What you need for this book

The practical examples in this book use Scala and SBT for Apache Spark-based code development and compilation. A Cloudera CDH 5.3 Hadoop cluster on CentOS 6.5 Linux servers is also used. Linux Bash shell and Perl scripts are used both, to assist in Spark applications and provide data feeds. Hadoop administration commands are used to move and examine data during Spark applications tests.

Given the skill overview previously, it would be useful for the reader to have a basic understanding of Linux, Apache Hadoop, and Spark. Having said that, and given that there is an abundant amount of information available on the internet today, I would not want to stop an intrepid reader from just having a go. I believe that it is possible to learn more from mistakes than successes.

Who this book is for

This book is for anyone interested in Apache Hadoop and Spark who would like to learn more about Spark. It is for the user who would like to learn how the usage of Spark can be extended with systems like H2O. It is for the user who is interested in graph processing but would like to learn more about graph storage. If the reader wants to know about Apache Spark in the cloud then he/she can learn about <https://databricks.com/>, the cloud-based system developed by the people who brought them Spark. If you are a developer with some experience with Spark and want to strengthen your knowledge of how to get around in the world of Spark, then this book is ideal for you. Basic knowledge of Linux, Hadoop, and Spark is required to understand this book; reasonable knowledge of Scala is also expected.

Conventions

In this book, you will find a number of text styles 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:

"The first step is to ensure that a Cloudera repository file exists under the `/etc/yum.repos.d` directory, on the server `hc2nn` and all of the other Hadoop cluster servers."

A block of code is set as follows:

```
export AWS_ACCESS_KEY_ID="QQp18Exxx"
export AWS_SECRET_ACCESS_KEY="0HFzqt4xxx"

./spark-ec2 \
  --key-pair=pairname \
  --identity-file=awskey.pem \
  --region=us-west-1 \
  --zone=us-west-1a \
  launch cluster1
```

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

```
[hadoop@hc2nn ec2]$ pwd
```

```
/usr/local/spark/ec2
```

```
[hadoop@hc2nn ec2]$ ls
```

```
deploy.generic  README  spark-ec2  spark_ec2.py
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Select the **User Actions** option, and then select **Manage Access Keys**."



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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you 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.

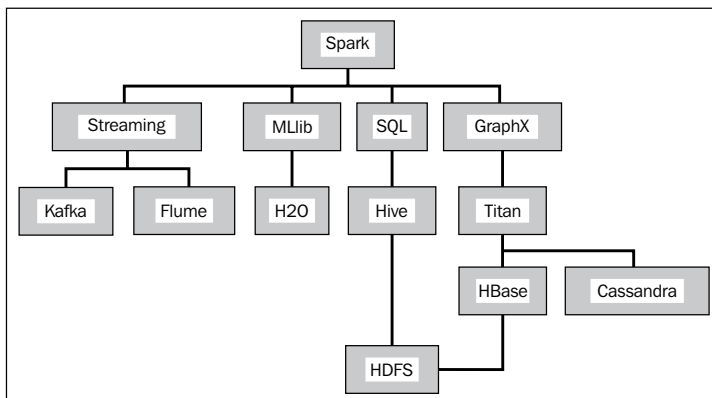
1

Apache Spark

Apache Spark is a distributed and highly scalable in-memory data analytics system, providing the ability to develop applications in Java, Scala, Python, as well as languages like R. It has one of the highest contribution/involvement rates among the Apache top level projects at this time. Apache systems, such as Mahout, now use it as a processing engine instead of MapReduce. Also, as will be shown in *Chapter 4, Apache Spark SQL*, it is possible to use a Hive context to have the Spark applications process data directly to and from Apache Hive.

Apache Spark provides four main submodules, which are SQL, MLlib, GraphX, and Streaming. They will all be explained in their own chapters, but a simple overview would be useful here. The modules are interoperable, so data can be passed between them. For instance, streamed data can be passed to SQL, and a temporary table can be created.

The following figure explains how this book will address Apache Spark and its modules. The top two rows show Apache Spark, and its four submodules described earlier. However, wherever possible, I always try to show by giving an example how the functionality may be extended using the extra tools:



For instance, the data streaming module explained in *Chapter 3, Apache Spark Streaming*, will have worked examples, showing how data movement is performed using Apache **Kafka** and **Flume**. The **MLlib** or the machine learning module will have its functionality examined in terms of the data processing functions that are available, but it will also be extended using the H2O system and deep learning.

The previous figure is, of course, simplified. It represents the system relationships presented in this book. For instance, there are many more routes between Apache Spark modules and HDFS than the ones shown in the preceding diagram.

The Spark SQL chapter will also show how Spark can use a Hive Context. So, a Spark application can be developed to create Hive-based objects, and run Hive QL against Hive tables, stored in HDFS.

Chapter 5, Apache Spark GraphX, and *Chapter 6, Graph-based Storage*, will show how the Spark GraphX module can be used to process big data scale graphs, and how they can be stored using the Titan graph database. It will be shown that Titan will allow big data scale graphs to be stored, and queried as graphs. It will show, by an example, that Titan can use both, **HBase** and **Cassandra** as a storage mechanism. When using HBase, it will be shown that implicitly, Titan uses HDFS as a cheap and reliable distributed storage mechanism.

So, I think that this section has explained that Spark is an in-memory processing system. When used at scale, it cannot exist alone—the data must reside somewhere. It will probably be used along with the Hadoop tool set, and the associated ecosystem. Luckily, Hadoop stack providers, such as Cloudera, provide the CDH Hadoop stack and cluster manager, which integrates with Apache Spark, Hadoop, and most of the current stable tool set. During this book, I will use a small CDH 5.3 cluster installed on CentOS 6.5 64 bit servers. You can use an alternative configuration, but I find that CDH provides most of the tools that I need, and automates the configuration, leaving me more time for development.

Having mentioned the Spark modules and the software that will be introduced in this book, the next section will describe the possible design of a big data cluster.

Overview

In this section, I wish to provide an overview of the functionality that will be introduced in this book in terms of Apache Spark, and the systems that will be used to extend it. I will also try to examine the future of Apache Spark, as it integrates with cloud storage.

When you examine the documentation on the Apache Spark website (<http://spark.apache.org/>), you will see that there are topics that cover SparkR and Bagel. Although I will cover the four main Spark modules in this book, I will not cover these two topics. I have limited time and scope in this book so I will leave these topics for reader investigation or for a future date.

Spark Machine Learning

The Spark MLlib module offers machine learning functionality over a number of domains. The documentation available at the Spark website introduces the data types used (for example, vectors and the LabeledPoint structure). This module offers functionality that includes:

- Statistics
- Classification
- Regression
- Collaborative Filtering
- Clustering
- Dimensionality Reduction
- Feature Extraction
- Frequent Pattern Mining
- Optimization

The Scala-based practical examples of KMeans, Naïve Bayes, and Artificial Neural Networks have been introduced and discussed in *Chapter 2, Apache Spark MLlib* of this book.

Spark Streaming

Stream processing is another big and popular topic for Apache Spark. It involves the processing of data in Spark as streams, and covers topics such as input and output operations, transformations, persistence, and checkpointing among others.

Chapter 3, Apache Spark Streaming, covers this area of processing, and provides practical examples of different types of stream processing. It discusses batch and window stream configuration, and provides a practical example of checkpointing. It also covers different examples of stream processing, including Kafka and Flume.

There are many more ways in which stream data can be used. Other Spark module functionality (for example, SQL, MLlib, and GraphX) can be used to process the stream. You can use Spark streaming with systems such as Kinesis or ZeroMQ. You can even create custom receivers for your own user-defined data sources.

Spark SQL

From Spark version 1.3 data frames have been introduced into Apache Spark so that Spark data can be processed in a tabular form and tabular functions (like select, filter, groupBy) can be used to process data. The Spark SQL module integrates with Parquet and JSON formats to allow data to be stored in formats that better represent data. This also offers more options to integrate with external systems.

The idea of integrating Apache Spark into the Hadoop Hive big data database can also be introduced. Hive context-based Spark applications can be used to manipulate Hive-based table data. This brings Spark's fast in-memory distributed processing to Hive's big data storage capabilities. It effectively lets Hive use Spark as a processing engine.

Spark graph processing

The Apache Spark GraphX module allows Spark to offer fast, big data in memory graph processing. A graph is represented by a list of vertices and edges (the lines that connect the vertices). GraphX is able to create and manipulate graphs using the property, structural, join, aggregation, cache, and uncache operators.

It introduces two new data types to support graph processing in Spark: VertexRDD and EdgeRDD to represent graph vertexes and edges. It also introduces graph processing example functions, such as PageRank and triangle processing. Many of these functions will be examined in *Chapter 5, Apache Spark GraphX*.

Extended ecosystem

When examining big data processing systems, I think it is important to look at not just the system itself, but also how it can be extended, and how it integrates with external systems, so that greater levels of functionality can be offered. In a book of this size, I cannot cover every option, but hopefully by introducing a topic, I can stimulate the reader's interest, so that they can investigate further.

I have used the H2O machine learning library system to extend Apache Spark's machine learning module. By using an H2O deep learning Scala-based example, I have shown how neural processing can be introduced to Apache Spark. I am, however, aware that I have just scratched the surface of H2O's functionality. I have only used a small neural cluster and a single type of classification functionality. Also, there is a lot more to H2O than deep learning.

As graph processing becomes more accepted and used in the coming years, so will graph based storage. I have investigated the use of Spark with the NoSQL database Neo4J, using the Mazerunner prototype application. I have also investigated the use of the Aurelius (Datastax) Titan database for graph-based storage. Again, Titan is a database in its infancy, which needs both community support and further development. But I wanted to examine the future options for Apache Spark integration.

The future of Spark

The next section will show that the Apache Spark release contains scripts to allow a Spark cluster to be created on AWS EC2 storage. There are a range of options available that allow the cluster creator to define attributes such as cluster size and storage type. But this type of cluster is difficult to resize, which makes it difficult to manage changing requirements. If the data volume changes or grows over time a larger cluster maybe required with more memory.

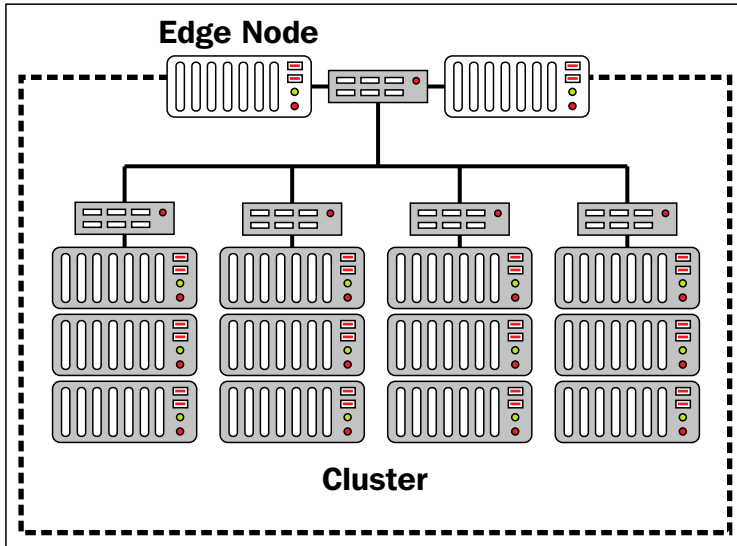
Luckily, the people that developed Apache Spark have created a new start-up called Databricks <https://databricks.com/>, which offers web console-based Spark cluster management, plus a lot of other functionality. It offers the idea of work organized by notebooks, user access control, security, and a mass of other functionality. It is described at the end of this book.

It is a service in its infancy, currently only offering cloud-based storage on Amazon AWS, but it will probably extend to Google and Microsoft Azure in the future. The other cloud-based providers, that is, Google and Microsoft Azure, are also extending their services, so that they can offer Apache Spark processing in the cloud.

Cluster design

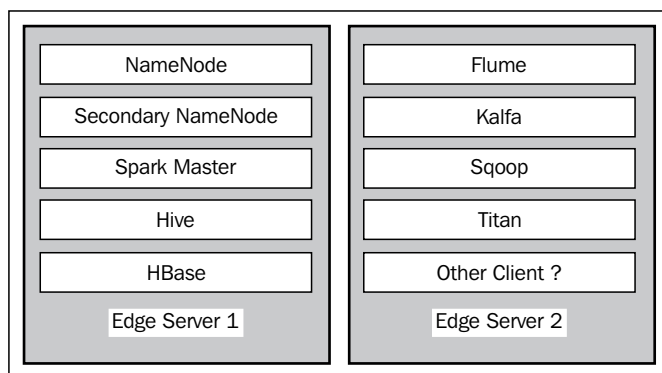
As I already mentioned, Apache Spark is a distributed, in-memory, parallel processing system, which needs an associated storage mechanism. So, when you build a big data cluster, you will probably use a distributed storage system such as Hadoop, as well as tools to move data like Sqoop, Flume, and Kafka.

I wanted to introduce the idea of edge nodes in a big data cluster. Those nodes in the cluster will be client facing, on which reside the client facing components like the Hadoop NameNode or perhaps the Spark master. The majority of the big data cluster might be behind a firewall. The edge nodes would then reduce the complexity caused by the firewall, as they would be the only nodes that would be accessible. The following figure shows a simplified big data cluster:



It shows four simplified cluster racks with switches and edge node computers, facing the client across the firewall. This is, of course, stylized and simplified, but you get the idea. The general processing nodes are hidden behind a firewall (the dotted line), and are available for general processing, in terms of Hadoop, Apache Spark, Zookeeper, Flume, and/or Kafka. The following figure represents a couple of big data cluster edge nodes, and attempts to show what applications might reside on them.

The edge node applications will be the master applications similar to the Hadoop NameNode, or the Apache Spark master server. It will be the components that are bringing the data into and out of the cluster such as Flume, Sqoop, and Kafka. It can be any component that makes a user interface available to the client user similar to Hive:



Generally, firewalls, while adding security to the cluster, also increase the complexity. Ports between system components need to be opened up, so that they can talk to each other. For instance, Zookeeper is used by many components for configuration. Apache Kafka, the publish subscribe messaging system, uses Zookeeper for configuring its topics, groups, consumers, and producers. So client ports to Zookeeper, potentially across the firewall, need to be open.

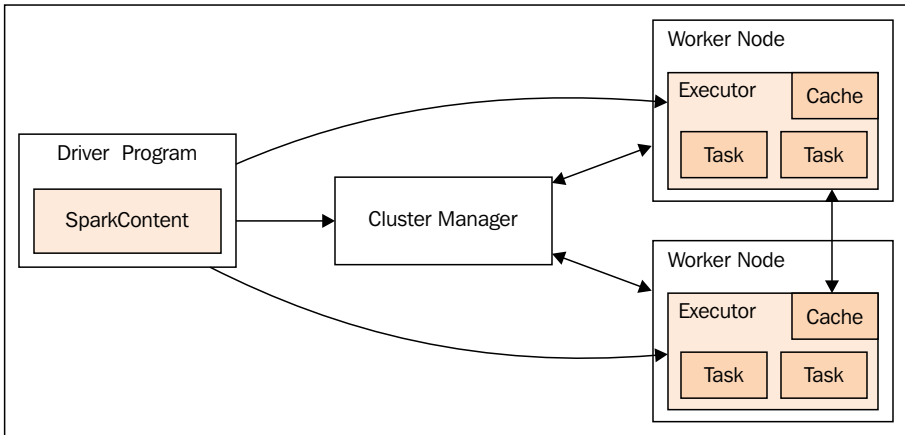
Finally, the allocation of systems to cluster nodes needs to be considered. For instance, if Apache Spark uses Flume or Kafka, then in-memory channels will be used. The size of these channels, and the memory used, caused by the data flow, need to be considered. Apache Spark should not be competing with other Apache components for memory usage. Depending upon your data flows and memory usage, it might be necessary to have the Spark, Hadoop, Zookeeper, Flume, and other tools on distinct cluster nodes.

Generally, the edge nodes that act as cluster NameNode servers, or Spark master servers, will need greater resources than the cluster processing nodes within the firewall. For instance, a CDH cluster node manager server will need extra memory, as will the Spark master server. You should monitor edge nodes for resource usage, and adjust in terms of resources and/or application location as necessary.

This section has briefly set the scene for the big data cluster in terms of Apache Spark, Hadoop, and other tools. However, how might the Apache Spark cluster itself, within the big data cluster, be configured? For instance, it is possible to have many types of Spark cluster manager. The next section will examine this, and describe each type of Apache Spark cluster manager.

Cluster management

The following diagram, borrowed from the spark.apache.org website, demonstrates the role of the Apache Spark cluster manager in terms of the master, slave (worker), executor, and Spark client applications:



The Spark context, as you will see from many of the examples in this book, can be defined via a Spark configuration object, and a Spark URL. The Spark context connects to the Spark cluster manager, which then allocates resources across the worker nodes for the application. The cluster manager allocates executors across the cluster worker nodes. It copies the application jar file to the workers, and finally it allocates tasks.

The following subsections describe the possible Apache Spark cluster manager options available at this time.

Local

By specifying a Spark configuration local URL, it is possible to have the application run locally. By specifying `local[n]`, it is possible to have Spark use `<n>` threads to run the application locally. This is a useful development and test option.

Standalone

Standalone mode uses a basic cluster manager that is supplied with Apache Spark. The spark master URL will be as follows:

Spark://<hostname>:7077

Here, `<hostname>` is the name of the host on which the Spark master is running. I have specified `7077` as the port, which is the default value, but it is configurable. This simple cluster manager, currently, only supports FIFO (first in first out) scheduling. You can contrive to allow concurrent application scheduling by setting the resource configuration options for each application. For instance, using `spark.core.max` to share cores between applications.

Apache YARN

At a larger scale when integrating with Hadoop YARN, the Apache Spark cluster manager can be YARN, and the application can run in one of two modes. If the Spark master value is set as `yarn-cluster`, then the application can be submitted to the cluster, and then terminated. The cluster will take care of allocating resources and running tasks. However, if the application master is submitted as `yarn-client`, then the application stays alive during the life cycle of processing, and requests resources from YARN.

Apache Mesos

Apache Mesos is an open source system for resource sharing across a cluster. It allows multiple frameworks to share a cluster by managing and scheduling resources. It is a cluster manager, which provides isolation using Linux containers, allowing multiple systems, like Hadoop, Spark, Kafka, Storm, and more to share a cluster safely. It is highly scalable to thousands of nodes. It is a master slave-based system, and is fault tolerant, using Zookeeper for configuration management.

For a single master node Mesos cluster, the Spark master URL will be in this form:

Mesos: `//<hostname>:5050`

Where `<hostname>` is the host name of the Mesos master server, the port is defined as `5050`, which is the default Mesos master port (this is configurable). If there are multiple Mesos master servers in a large scale high availability Mesos cluster, then the Spark master URL would look like this:

Mesos: `//zk://<hostname>:2181`

So, the election of the Mesos master server will be controlled by Zookeeper. The `<hostname>` will be the name of a host in the Zookeeper quorum. Also, the port number `2181` is the default master port for Zookeeper.

Amazon EC2

The Apache Spark release contains scripts for running Spark in the cloud against Amazon AWS EC2-based servers. The following listing, as an example, shows Spark 1.3.1 installed on a Linux CentOS server, under the directory called `/usr/local/spark/`. The EC2 resources are available in the Spark release EC2 subdirectory:

```
[hadoop@hc2nn ec2]$ pwd
```

```
/usr/local/spark/ec2
```

```
[hadoop@hc2nn ec2]$ ls
```

```
deploy.generic  README  spark-ec2  spark_ec2.py
```

In order to use Apache Spark on EC2, you will need to set up an Amazon AWS account. You can set up an initial free account to try it out here: <http://aws.amazon.com/free/>.

If you take a look at *Chapter 8, Spark Databricks* you will see that such an account has been set up, and is used to access <https://databricks.com/>. The next thing that you will need to do is access your AWS IAM Console, and select the **Users** option. You either create or select a user. Select the **User Actions** option, and then select **Manage Access Keys**. Then, select **Create Access Key**, and then **Download Credentials**. Make sure that your downloaded key file is secure, assuming that you are on Linux `chmod` the file with permissions = `600` for user-only access.

You will now have your **Access Key ID**, **Secret Access Key**, key file, and key pair name. You can now create a Spark EC2 cluster using the `spark-ec2` script as follows:

```
export AWS_ACCESS_KEY_ID="QQp18Exxx"
export AWS_SECRET_ACCESS_KEY="0HFzqt4xxx"
```

```
./spark-ec2 \
  --key-pair=pairname \
  --identity-file=awskey.pem \
  --region=us-west-1 \
  --zone=us-west-1a \
  launch cluster1
```

Here, `<pairname>` is the key pair name that you gave when your access details were created; `<awskey.pem>` is the file that you downloaded. The name of the cluster that you are going to create is called `<cluster1>`. The region chosen here is in the western USA, `us-west-1`. If you live in the Pacific, as I do, it might be wiser to choose a nearer region like `ap-southeast-2`. However, if you encounter allowance access issues, then you will need to try another zone. Remember also that using cloud-based Spark clustering like this will have higher latency and poorer I/O in general. You share your cluster hosts with multiple users, and your cluster maybe in a remote region.

You can use a series of options to this basic command to configure the cloud-based Spark cluster that you create. The `-s` option can be used:

```
-s <slaves>
```

This allows you to define how many worker nodes to create in your Spark EC2 cluster, that is, `-s 5` for a six node cluster, one master, and five slave workers. You can define the version of Spark that your cluster runs, rather than the default latest version. The following option starts a cluster with Spark version 1.3.1:

```
--spark-version=1.3.1
```

The instance type used to create the cluster will define how much memory is used, and how many cores are available. For instance, the following option will set the instance type to be `m3.large`:

```
--instance-type=m3.large
```

The current instance types for Amazon AWS can be found at: <http://aws.amazon.com/ec2/instance-types/>.

The following figure shows the current (as of July 2015) AWS M3 instance types, model details, cores, memory, and storage. There are many instance types available at this time; for instance, T2, M4, M3, C4, C3, R3, and more. Examine the current availability and choose appropriately:

M3

This family includes the M3 instance types and provides a balance of compute, memory, and network resources, and it is a good choice for many applications.

Features:

- High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge) Processors*
- SSD-based instance storage for fast I/O performance
- Balance of compute, memory, and network resources

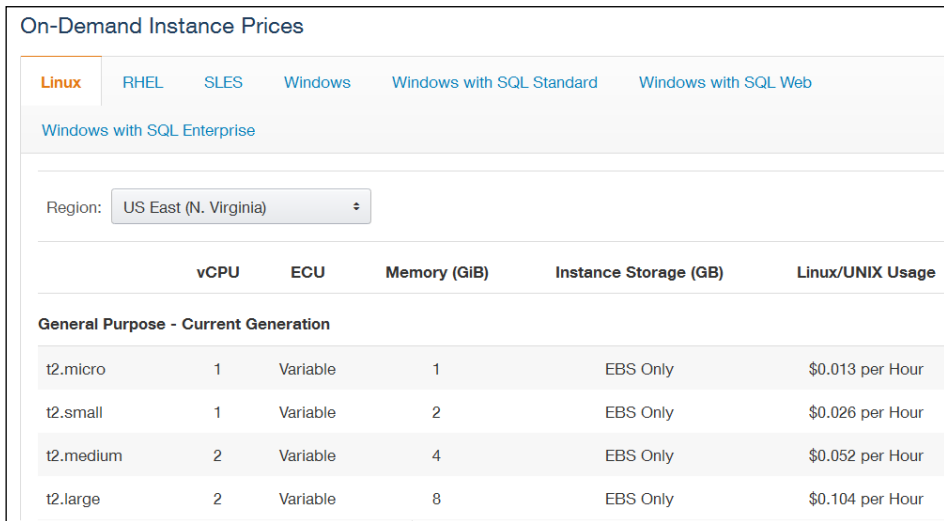
Model	vCPU	Mem (GiB)	SSD Storage (GB)
m3.medium	1	3.75	1 x 4
m3.large	2	7.5	1 x 32
m3.xlarge	4	15	2 x 40
m3.2xlarge	8	30	2 x 80

Pricing is also very important. The current AWS storage type prices can be found at: <http://aws.amazon.com/ec2/pricing/>.

The prices are shown by region with a drop-down menu, and a price by hour. Remember that each storage type is defined by cores, memory, and physical storage. The prices are also defined by operating system type, that is, Linux, RHEL, and Windows. Just select the OS via a top-level menu.

The following figure shows an example of pricing at the time of writing (July 2015); it is just provided to give an idea. Prices will differ over time, and by service provider. They will differ by the size of storage that you need, and the length of time that you are willing to commit to.

Be aware also of the costs of moving your data off of any storage platform. Try to think long term. Check whether you will need to move all, or some of your cloud-based data to the next system in, say, five years. Check the process to move data, and include that cost in your planning.



Linux	RHEL	SLES	Windows	Windows with SQL Standard	Windows with SQL Web
Windows with SQL Enterprise					
Region: US East (N. Virginia)					
	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
General Purpose - Current Generation					
t2.micro	1	Variable	1	EBS Only	\$0.013 per Hour
t2.small	1	Variable	2	EBS Only	\$0.026 per Hour
t2.medium	2	Variable	4	EBS Only	\$0.052 per Hour
t2.large	2	Variable	8	EBS Only	\$0.104 per Hour

As described, the preceding figure shows the costs of AWS storage types by operating system, region, storage type, and hour. The costs are measured per unit hour, so systems such as <https://databricks.com/> do not terminate EC2 instances, until a full hour has elapsed. These costs will change with time and need to be monitored via (for AWS) the AWS billing console.

You may also have problems when wanting to resize your Spark EC2 cluster, so you will need to be sure of the master slave configuration before you start. Be sure how many workers you are going to require, and how much memory you need. If you feel that your requirements are going to change over time, then you might consider using <https://databricks.com/>, if you definitely wish to work with Spark in the cloud. Go to *Chapter 8, Spark Databricks* and see how you can set up, and use <https://databricks.com/>.

In the next section, I will examine Apache Spark cluster performance, and the issues that might impact it.

Performance

Before moving on to the rest of the chapters covering functional areas of Apache Spark and extensions to it, I wanted to examine the area of performance. What issues and areas need to be considered? What might impact Spark application performance starting at the cluster level, and finishing with actual Scala code? I don't want to just repeat what the Spark website says, so have a look at the following URL: <http://spark.apache.org/docs/<version>/tuning.html>.

Here, `<version>` relates to the version of Spark that you are using, that is, latest, or 1.3.1 for a specific version. So, having looked at that page, I will briefly mention some of the topic areas. I am going to list some general points in this section without implying an order of importance.

The cluster structure

The size and structure of your big data cluster is going to affect performance. If you have a cloud-based cluster, your IO and latency will suffer in comparison to an unshared hardware cluster. You will be sharing the underlying hardware with multiple customers, and that the cluster hardware maybe remote.

Also, the positioning of cluster components on servers may cause resource contention. For instance, if possible, think carefully about locating Hadoop NameNodes, Spark servers, Zookeeper, Flume, and Kafka servers in large clusters. With high workloads, you might consider segregating servers to individual systems. You might also consider using an Apache system such as Mesos in order to share resources.

Also, consider potential parallelism. The greater the number of workers in your Spark cluster for large data sets, the greater the opportunity for parallelism.

The Hadoop file system

You might consider using an alternative to HDFS, depending upon your cluster requirements. For instance, MapR has the MapR-FS NFS-based read write file system for improved performance. This file system has a full read write capability, whereas HDFS is designed as a write once, read many file system. It offers an improvement in performance over HDFS. It also integrates with Hadoop and the Spark cluster tools. Bruce Penn, an architect at MapR, has written an interesting article describing its features at: <https://www.mapr.com/blog/author/bruce-penn>.

Just look for the blog post entitled *Comparing MapR-FS and HDFS NFS and Snapshots*. The links in the article describe the MapR architecture, and possible performance gains.

Data locality

Data locality or the location of the data being processed is going to affect latency and Spark processing. Is the data sourced from AWS S3, HDFS, the local file system/network, or a remote source?

As the previous tuning link mentions, if the data is remote, then functionality and data must be brought together for processing. Spark will try to use the best data locality level possible for task processing.

Memory

In order to avoid **OOM (Out of Memory)** messages for the tasks, on your Apache Spark cluster, you can consider a number of areas:

- Consider the level of physical memory available on your Spark worker nodes. Can it be increased?
- Consider data partitioning. Can you increase the number of partitions in the data used by your Spark application code?
- Can you increase the storage fraction, the memory used by the JVM for storage and caching of RDD's?
- Consider tuning data structures used to reduce memory.
- Consider serializing your RDD storage to reduce the memory usage.

Coding

Try to tune your code to improve Spark application performance. For instance, filter your application-based data early in your ETL cycle. Tune your degree of parallelism, try to find the resource-expensive parts of your code, and find alternatives.

Cloud

Although, most of this book will concentrate on examples of Apache Spark installed on physically server-based clusters (with the exception of <https://databricks.com/>), I wanted to make the point that there are multiple cloud-based options out there. There are cloud-based systems that use Apache Spark as an integrated component, and cloud-based systems that offer Spark as a service. Even though this book cannot cover all of them in depth, I thought that it would be useful to mention some of them:

- Databricks is covered in two chapters in this book. It offers a Spark cloud-based service, currently using AWS EC2. There are plans to extend the service to other cloud suppliers (<https://databricks.com/>).
- At the time of writing (July 2015) this book, Microsoft Azure has been extended to offer Spark support.
- Apache Spark and Hadoop can be installed on Google Cloud.
- The Oryx system has been built at the top of Spark and Kafka for real-time, large-scale machine learning (<http://oryx.io/>).
- The velox system for serving machine learning prediction is based upon Spark and KeystoneML (<https://github.com/amplab/velox-modelserver>).
- PredictionIO is an open source machine learning service built on Spark, HBase, and Spray (<https://prediction.io/>).
- SeldonIO is an open source predictive analytics platform, based upon Spark, Kafka, and Hadoop (<http://www.seldon.io/>).

Summary

In closing this chapter, I would invite you to work your way through each of the Scala code-based examples in the following chapters. I have been impressed by the rate at which Apache Spark has evolved, and I am also impressed at the frequency of the releases. So, even though at the time of writing, Spark has reached 1.4, I am sure that you will be using a later version. If you encounter problems, tackle them logically. Try approaching the Spark user group for assistance (user@spark.apache.org), or check the Spark website at <http://spark.apache.org/>.

I am always interested to hear from people, and connect with people on sites such as LinkedIn. I am keen to hear about the projects that people are involved with and new opportunities. I am interested to hear about Apache Spark, the ways that you use it and the systems that you build being used at scale. I can be contacted on LinkedIn at: [linkedin.com/profile/view?id=73219349](https://www.linkedin.com/profile/view?id=73219349).

Or, I can be contacted via my website at <http://semtech-solutions.co.nz/>, or finally, by email at: info@semtech-solutions.co.nz.

2

Apache Spark MLlib

MLlib is the machine learning library that is provided with Apache Spark, the in memory cluster based open source data processing system. In this chapter, I will examine the functionality, provided within the MLlib library in terms of areas such as regression, classification, and neural processing. I will examine the theory behind each algorithm before providing working examples that tackle real problems. The example code and documentation on the web can be sparse and confusing. I will take a step-by-step approach in describing how the following algorithms can be used, and what they are capable of doing:

- Classification with Naïve Bayes
- Clustering with K-Means
- Neural processing with ANN

Having decided to learn about Apache Spark, I am assuming that you are familiar with Hadoop. Before I proceed, I will explain a little about my environment. My Hadoop cluster is installed on a set of Centos 6.5 Linux 64 bit servers. The following section will describe the architecture in detail.

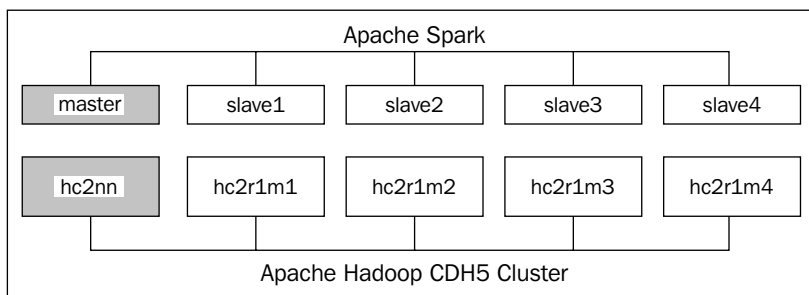
The environment configuration

Before delving into the Apache Spark modules, I wanted to explain the structure and version of Hadoop and Spark clusters that I will use in this book. I will be using the Cloudera CDH 5.1.3 version of Hadoop for storage and I will be using two versions of Spark: 1.0 and 1.3 in this chapter.

The earlier version is compatible with Cloudera software, and has been tested and packaged by them. It is installed as a set of Linux services from the Cloudera repository using the `yum` command. Because I want to examine the Neural Net technology that has not been released yet, I will also download and run the development version of Spark 1.3 from GitHub. This will be explained later in the chapter.

Architecture

The following diagram explains the structure of the small Hadoop cluster that I will use in this chapter:



The previous diagram shows a five-node Hadoop cluster with a NameNode called **hc2nn**, and DataNodes **hc2r1m1** to **hc2r1m4**. It also shows an Apache Spark cluster with a master node and four slave nodes. The Hadoop cluster provides the physical Centos 6 Linux machines while the Spark cluster runs on the same hosts. For instance, the Spark master server runs on the Hadoop Name Node machine **hc2nn**, whereas the Spark **slave1** worker runs on the host **hc2r1m1**.

The Linux server naming standard used higher up should be explained. For instance the Hadoop NameNode server is called **hc2nn**. The **h** in this server name means Hadoop, the **c** means cluster, and the **nn** means NameNode. So, **hc2nn** means Hadoop cluster 2 NameNode. Similarly, for the server **hc2r1m1**, the **h** means Hadoop the **c** means cluster the **r** means rack and the **m** means machine. So, the name stands for Hadoop cluster 2 rack 1 machine 1. In a large Hadoop cluster, the machines will be organized into racks, so this naming standard means that the servers will be easy to locate.

You can arrange your Spark and Hadoop clusters as you see fit, they don't need to be on the same hosts. For the purpose of writing this book, I have limited machines available so it makes sense to co-locate the Hadoop and Spark clusters. You can use entirely separate machines for each cluster, as long as Spark is able to access Hadoop (if you want to use it for distributed storage).

Remember that although Spark is used for the speed of its in-memory distributed processing, it doesn't provide storage. You can use the Host file system to read and write your data, but if your data volumes are big enough to be described as big data, then it makes sense to use a distributed storage system like Hadoop.

Remember also that Apache Spark may only be the processing step in your **ETL (Extract, Transform, Load)** chain. It doesn't provide the rich tool set that the Hadoop ecosystem contains. You may still need Nutch/Gora/Solr for data acquisition; Sqoop and Flume for moving data; Oozie for scheduling; and HBase, or Hive for storage. The point that I am making is that although Apache Spark is a very powerful processing system, it should be considered a part of the wider Hadoop ecosystem.

Having described the environment that will be used in this chapter, I will move on to describe the functionality of the Apache Spark **MLlib (Machine Learning library)**.

The development environment

The Scala language will be used for coding samples in this book. This is because as a scripting language, it produces less code than Java. It can also be used for the Spark shell, as well as compiled with Apache Spark applications. I will be using the `sbt` tool to compile the Scala code, which I have installed as follows:

```
[hadoop@hc2nn ~]# su -
[root@hc2nn ~]# cd /tmp
[root@hc2nn ~]# wget http://repo.scala-sbt.org/scalasbt/sbt-native-
packages/org/scala-sbt/sbt/0.13.1/sbt.rpm
[root@hc2nn ~]# rpm -ivh sbt.rpm
```

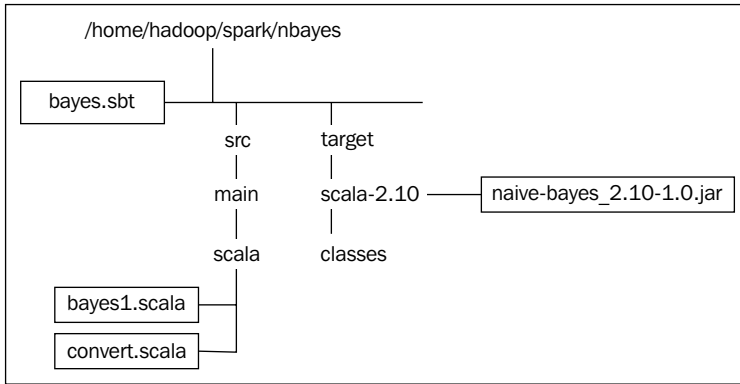
For convenience while writing this book, I have used the generic Linux account called **hadoop** on the Hadoop NameNode server `hc2nn`. As the previous commands show that I need to install `sbt` as the root account, which I have accessed via `su` (switch user). I have then downloaded the `sbt.rpm` file, to the `/tmp` directory, from the web-based server called `repo.scala-sbt.org` using `wget`. Finally, I have installed the `rpm` file using the `rpm` command with the options `i` for install, `v` for verify, and `h` to print the hash marks while the package is being installed.



Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. 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.

I have developed all of the Scala code for Apache Spark, in this chapter, on the Linux server `hc2nn`, using the Linux `hadoop` account. I have placed each set of code within a sub directory under `/home/hadoop/spark`. For instance, the following sbt structure diagram shows that the MLlib Naïve Bayes code is stored within a subdirectory called `nbayes`, under the `spark` directory. What the diagram also shows is that the Scala code is developed within a subdirectory structure named `src/main/scala`, under the `nbayes` directory. The files called `bayes1.scala` and `convert.scala` contain the Naïve Bayes code that will be used in the next section:



The `bayes.sbt` file is a configuration file used by the sbt tool, which describes how to compile the Scala files within the `Scala` directory (also note that if you were developing in Java, you would use a path of the form `nbayes/src/main/java`). The contents of the `bayes.sbt` file are shown next. The `pwd` and `cat` Linux commands remind you of the file location, and they also remind you to dump the file contents.

The `name`, `version`, and `scalaVersion` options set the details of the project, and the `libraryDependencies` options define where the Hadoop and Spark libraries can be located. In this case, CDH5 has been installed using the Cloudera parcels, and the packages libraries can be located in the standard locations, that is, `/usr/lib/hadoop` for Hadoop and `/usr/lib/spark` for Spark. The resolver's option specifies the location for the Cloudera repository for other dependencies:

```
[hadoop@hc2nn nbayes]$ pwd
/home/hadoop/spark/nbayes
[hadoop@hc2nn nbayes]$ cat bayes.sbt
```

```
name := "Naive Bayes"
```

```
version := "1.0"
```

```
scalaVersion := "2.10.4"
```

```
libraryDependencies += "org.apache.hadoop" % "hadoop-client" % "2.3.0"
```

```
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.0.0"
```

```
libraryDependencies += "org.apache.spark" %% "spark-mllib" % "1.0.0"
```

```
// If using CDH, also add Cloudera repo
```

```
resolvers += "Cloudera Repository" at  
https://repository.cloudera.com/artifactory/cloudera-repos/
```

The Scala nbayes project code can be compiled from the nbayes sub directory using this command:

```
[hadoop@hc2nn nbayes]$ sbt compile
```

The `sbt compile` command is used to compile the code into classes. The classes are then placed in the nbayes/target/scala-2.10/classes directory. The compiled classes can be packaged into a JAR file with this command:

```
[hadoop@hc2nn nbayes]$ sbt package
```

The `sbt package` command will create a JAR file under the directory nbayes/target/scala-2.10. As the example in the *sbt structure diagram* shows the JAR file named naive-bayes_2.10-1.0.jar has been created after a successful compile and package. This JAR file, and the classes that it contains, can then be used in a `spark-submit` command. This will be described later as the functionality in the Apache Spark MLLib module is explored.

Installing Spark

Finally, when describing the environment used for this book, I wanted to touch on the approach to installing and running Apache Spark. I won't elaborate on the Hadoop CDH5 install, except to say that I installed it using the Cloudera parcels. However, I manually installed version 1.0 of Apache Spark from the Cloudera repository, using the Linux `yum` commands. I installed the service-based packages, because I wanted the flexibility that would enable me to install multiple versions of Spark as services from Cloudera, as I needed.

When preparing a CDH Hadoop release, Cloudera takes the code that has been developed by the Apache Spark team, and the code released by the Apache Bigtop project. They perform an integration test so that it is guaranteed to work as a code stack. They also reorganize the code and binaries into services and parcels. This means that libraries, logs, and binaries can be located in defined locations under Linux, that is, `/var/log/spark`, `/usr/lib/spark`. It also means that, in the case of services, the components can be installed using the Linux `yum` command, and managed via the Linux `service` command.

Although, in the case of the Neural Network code described later in this chapter, a different approach was used. This is how Apache Spark 1.0 was installed for use with Hadoop CDH5:

```
[root@hc2nn ~]# cd /etc/yum.repos.d
[root@hc2nn yum.repos.d]# cat cloudera-cdh5.repo

[cloudera-cdh5]
# Packages for Cloudera's Distribution for Hadoop, Version 5, on RedHat
or CentOS 6 x86_64
name=Cloudera's Distribution for Hadoop, Version 5
baseurl=http://archive.cloudera.com/cdh5/redhat/6/x86_64/cdh/5/
gpgkey = http://archive.cloudera.com/cdh5/redhat/6/x86_64/cdh/RPM-GPG-
KEY-cloudera
gpgcheck = 1
```

The first step is to ensure that a Cloudera repository file exists under the `/etc/yum.repos.d` directory, on the server `hc2nn` and all of the other Hadoop cluster servers. The file is called `cloudera-cdh5.repo`, and specifies where the `yum` command can locate software for the Hadoop CDH5 cluster. On all the Hadoop cluster nodes, I use the Linux `yum` command, as `root`, to install the Apache Spark components `core`, `master`, `worker`, `history-server`, and `python`:

```
[root@hc2nn ~]# yum install spark-core spark-master spark-worker
spark-history-server spark-python
```

This gives me the flexibility to configure Spark in any way that I want in the future. Note that I have installed the master component on all the nodes, even though I only plan to use it from the Name Node at this time. Now, the Spark install needs to be configured on all the nodes. The configuration files are stored under `/etc/spark/conf`. The first thing to do, will be to set up a `slaves` file, which specifies on which hosts Spark will run its worker components:

```
[root@hc2nn ~]# cd /etc/spark/conf
```

```
[root@hc2nn conf]# cat slaves
```

A Spark Worker will be started on each of the machines listed below.

```
hc2r1m1
hc2r1m2
hc2r1m3
hc2r1m4
```

As you can see from the contents of the `slaves` file above Spark, it will run four workers on the Hadoop CDH5 cluster, Data Nodes, from `hc2r1m1` to `hc2r1m4`. Next, it will alter the contents of the `spark-env.sh` file to specify the Spark environment options. The `SPARK_MASTER_IP` values are defined as the full server name:

```
export STANDALONE_SPARK_MASTER_HOST=hc2nn.semtech-solutions.co.nz
export SPARK_MASTER_IP=$STANDALONE_SPARK_MASTER_HOST
```

```
export SPARK_MASTER_WEBUI_PORT=18080
export SPARK_MASTER_PORT=7077
export SPARK_WORKER_PORT=7078
export SPARK_WORKER_WEBUI_PORT=18081
```

The web user interface port numbers are specified for the master and worker processes, as well as the operational port numbers. The Spark service can then be started as root from the Name Node server. I use the following script:

```
echo "hc2r1m1 - start worker"
ssh hc2r1m1 'service spark-worker start'

echo "hc2r1m2 - start worker"
ssh hc2r1m2 'service spark-worker start'

echo "hc2r1m3 - start worker"
ssh hc2r1m3 'service spark-worker start'

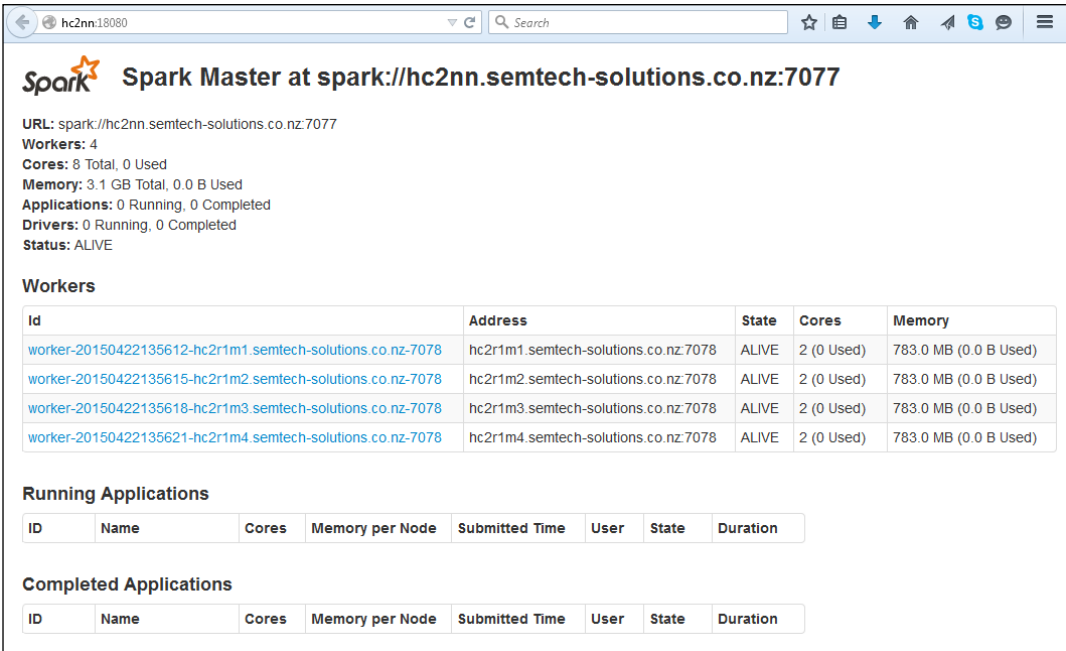
echo "hc2r1m4 - start worker"
ssh hc2r1m4 'service spark-worker start'

echo "hc2nn - start master server"
service spark-master start
service spark-history-server start
```

This starts the Spark worker service on all of the slaves, and the master and history server on the Name Node hc2nn. So now, the Spark user interface can be accessed using the `http://hc2nn:18080` URL.

The following figure shows an example of the Spark 1.0 master web user interface. It shows details about the Spark install, the workers, and the applications that are running or completed. The statuses of the master and workers are given. In this case, all are alive. Memory used and availability is given in total and by worker. Although, there are no applications running at the moment, each worker link can be selected to view the executor processes' running on each worker node, as the work volume for each application run is spread across the spark cluster.

Note also the Spark URL, `spark://hc2nn.semtech-solutions.co.nz:7077`, will be used when running the Spark applications like `spark-shell` and `spark-submit`. Using this URL, it is possible to ensure that the shell or application is run against this Spark cluster.



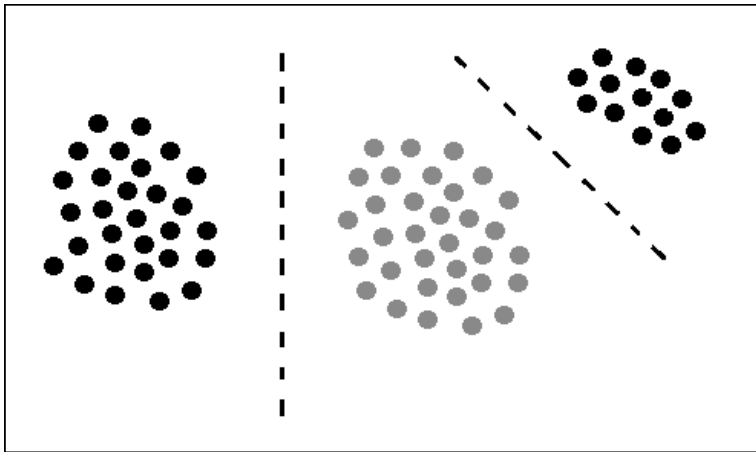
This gives a quick overview of the Apache Spark installation using services, its configuration, how to start it, and how to monitor it. Now, it is time to tackle the first of the MLlib functional areas, which is classification using the Naïve Bayes algorithm. The use of Spark will become clearer as Scala scripts are developed, and the resulting applications are monitored.

Classification with Naïve Bayes

This section will provide a working example of the Apache Spark MLlib Naïve Bayes algorithm. It will describe the theory behind the algorithm, and will provide a step-by-step example in Scala to show how the algorithm may be used.

Theory

In order to use the Naïve Bayes algorithm to classify a data set, the data must be linearly divisible, that is, the classes within the data must be linearly divisible by class boundaries. The following figure visually explains this with three data sets, and two class boundaries shown via the dotted lines:



Naïve Bayes assumes that the features (or dimensions) within a data set are independent of one another, that is, they have no effect on each other. An example for Naïve Bayes is supplied with the help of Hernan Amiune at <http://hernan.amiune.com/>. The following example considers the classification of emails as spam. If you have 100 e-mails then perform the following:

60% of emails are spam

80% of spam emails contain the word buy

20% of spam emails don't contain the word buy

40% of emails are not spam

10% of non spam emails contain the word buy

90% of non spam emails don't contain the word buy

Thus, convert this example into probabilities, so that a Naïve Bayes equation can be created.

$P(\text{Spam})$ = the probability that an email is spam = 0.6

$P(\text{Not Spam})$ = the probability that an email is not spam = 0.4

$P(\text{Buy}|\text{Spam})$ = the probability that an email that is spam has the word buy = 0.8

$P(\text{Buy}|\text{Not Spam})$ = the probability that an email that is not spam has the word buy = 0.1

So, what is the probability that an e-mail that contains the word buy is spam? Well, this would be written as **$P(\text{Spam}|\text{Buy})$** . Naïve Bayes says that it is described by the equation in the following figure:

$P(\text{Spam} \text{Buy}) = \frac{P(\text{Buy} \text{Spam}) * P(\text{Spam})}{P(\text{Buy} \text{Spam}) * P(\text{Spam}) + P(\text{Buy} \text{Not Spam}) * P(\text{Not Spam})}$
--

So, using the previous percentage figures, we get the following:

$$\begin{aligned} P(\text{Spam}|\text{Buy}) &= (0.8 * 0.6) / ((0.8 * 0.6) + (0.1 * 0.4)) = (\\ &.48) / (.48 + .04) \\ &= .48 / .52 = .923 \end{aligned}$$

This means that it is 92 percent more likely that an e-mail that contains the word buy is spam. That was a look at the theory; now, it's time to try a real world example using the Apache Spark MLlib Naïve Bayes algorithm.

Naïve Bayes in practice

The first step is to choose some data that will be used for classification. I have chosen some data from the UK government data web site, available at: <http://data.gov.uk/dataset/road-accidents-safety-data>.

The data set is called "Road Safety - Digital Breath Test Data 2013," which downloads a zipped text file called `DigitalBreathTestData2013.txt`. This file contains around half a million rows. The data looks like this:

```
Reason,Month,Year,WeekType,TimeBand,BreathAlcohol,AgeBand,Gender
Suspicion of Alcohol,Jan,2013,Weekday,12am-4am,75,30-39,Male
Moving Traffic Violation,Jan,2013,Weekday,12am-4am,0,20-24,Male
Road Traffic Collision,Jan,2013,Weekend,12pm-4pm,0,20-24,Female
```

In order to classify the data, I have modified both the column layout, and the number of columns. I have simply used Excel to give the data volume. However, if my data size had been in the big data range, I would have had to use Scala, or perhaps a tool like Apache Pig. As the following commands show, the data now resides on HDFS, in the directory named `/data/spark/nbayer`. The file name is called `DigitalBreathTestData2013-MALE2.csv`. Also, the line count from the Linux `wc` command shows that there are 467,000 rows. Finally, the following data sample shows that I have selected the columns: Gender, Reason, WeekType, TimeBand, BreathAlcohol, and AgeBand to classify. I will try and classify on the Gender column using the other columns as features:

```
[hadoop@hc2nn ~]$ hdfs dfs -cat /data/spark/nbayer/
DigitalBreathTestData2013-MALE2.csv | wc -l
467054
```

```
[hadoop@hc2nn ~]$ hdfs dfs -cat /data/spark/nbayer/
DigitalBreathTestData2013-MALE2.csv | head -5
Male,Suspicion of Alcohol,Weekday,12am-4am,75,30-39
Male,Moving Traffic Violation,Weekday,12am-4am,0,20-24
Male,Suspicion of Alcohol,Weekend,4am-8am,12,40-49
Male,Suspicion of Alcohol,Weekday,12am-4am,0,50-59
Female,Road Traffic Collision,Weekend,12pm-4pm,0,20-24
```

The Apache Spark MLlib classification functions use a data structure called `LabeledPoint`, which is a general purpose data representation defined at: <http://spark.apache.org/docs/1.0.0/api/scala/index.html#org.apache.spark.mllib.regression.LabeledPoint>.

This structure only accepts Double values, which means the text values in the previous data need to be classified numerically. Luckily, all of the columns in the data will convert to numeric categories, and I have provided two programs in the software package with this book, under the directory `chapter2\naive bayes` to do just that. The first is called `convTestData.pl`, and is a Perl script to convert the previous text file into Linux. The second file, which will be examined here is called `convert.scala`. It takes the contents of the `DigitalBreathTestData2013-MALE2.csv` file and converts each record into a Double vector.

The directory structure and files for an sbt Scala-based development environment have already been described earlier. I am developing my Scala code on the Linux server `hc2nn` using the Linux account `hadoop`. Next, the Linux `pwd` and `ls` commands show my top level `nbayes` development directory with the `bayes.sbt` configuration file, whose contents have already been examined:

```
[hadoop@hc2nn nbayes]$ pwd
/home/hadoop/spark/nbayes
[hadoop@hc2nn nbayes]$ ls
bayes.sbt      target      project     src
```

The Scala code to run the Naïve Bayes example is shown next, in the `src/main/scala` subdirectory, under the `nbayes` directory:

```
[hadoop@hc2nn scala]$ pwd
/home/hadoop/spark/nbayes/src/main/scala
[hadoop@hc2nn scala]$ ls
bayes1.scala  convert.scala
```

We will examine the `bayes1.scala` file later, but first, the text-based data on HDFS must be converted into the numeric `Double` values. This is where the `convert.scala` file is used. The code looks like this:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

These lines import classes for Spark context, the connection to the Apache Spark cluster, and the Spark configuration. The object that is being created is called `convert1`. It is an application, as it extends the class `App`:

```
object convert1 extends App
{
```

The next line creates a function called `enumerateCsvRecord`. It has a parameter called `colData`, which is an array of strings, and returns a string:

```
def enumerateCsvRecord( colData:Array[String]): String =
{
```

The function then enumerates the text values in each column, so for an instance, `Male` becomes `0`. These numeric values are stored in values like `colVal1`:

```
    val colVal1 =
        colData(0) match
```

```
{
    case "Male"                => 0
    case "Female"              => 1
    case "Unknown"             => 2
    case _                      => 99
}

val colVal2 =
  colData(1) match
  {
    case "Moving Traffic Violation"  => 0
    case "Other"                     => 1
    case "Road Traffic Collision"    => 2
    case "Suspicion of Alcohol"      => 3
    case _                           => 99
  }

val colVal3 =
  colData(2) match
  {
    case "Weekday"                => 0
    case "Weekend"                => 0
    case _                        => 99
  }

val colVal4 =
  colData(3) match
  {
    case "12am-4am"               => 0
    case "4am-8am"                => 1
    case "8am-12pm"               => 2
    case "12pm-4pm"               => 3
    case "4pm-8pm"                => 4
    case "8pm-12pm"               => 5
    case _                        => 99
  }
```

```

    }

    val colVal5 = colData(4)

    val colVal6 =
      colData(5) match
      {
        case "16-19"           => 0
        case "20-24"           => 1
        case "25-29"           => 2
        case "30-39"           => 3
        case "40-49"           => 4
        case "50-59"           => 5
        case "60-69"           => 6
        case "70-98"           => 7
        case "Other"           => 8
        case _                  => 99
      }

```

A comma separated string called `lineString` is created from the numeric column values, and is then returned. The function closes with the final brace character `}`. Note that the data line created next starts with a label value at column one, and is followed by a vector, which represents the data. The vector is space separated while the label is separated from the vector by a comma. Using these two separator types allows me to process both: the label and the vector in two simple steps later:

```

    val lineString = colVal1+","+colVal2+" "+colVal3+" "+colVal4+"
"+colVal5+" "+colVal6

    return lineString
}

```

The main script defines the HDFS server name and path. It defines the input file, and the output path in terms of these values. It uses the Spark URL and application name to create a new configuration. It then creates a new context or connection to Spark using these details:

```

val hdfsServer = "hdfs://hc2nn.semtech-solutions.co.nz:8020"
val hdfsPath   = "/data/spark/nbayes/"

```

```

val inDataFile = hdfsServer + hdfsPath + "DigitalBreathTestData2013-
MALE2.csv"
val outDataFile = hdfsServer + hdfsPath + "result"

val sparkMaster = "spark://hc2nn.semtech-solutions.co.nz:7077"
val appName = "Convert 1"
val sparkConf = new SparkConf()

sparkConf.setMaster(sparkMaster)
sparkConf.setAppName(appName)

val sparkCxt = new SparkContext(sparkConf)

```

The CSV-based raw data file is loaded from HDFS using the Spark context `textFile` method. Then, a data row count is printed:

```

val csvData = sparkCxt.textFile(inDataFile)
println("Records in : " + csvData.count() )

```

The CSV raw data is passed line by line to the `enumerateCsvRecord` function. The returned string-based numeric data is stored in the `enumRddData` variable:

```

val enumRddData = csvData.map
{
    csvLine =>
        val colData = csvLine.split(',')

        enumerateCsvRecord(colData)
}

```

Finally, the number of records in the `enumRddData` variable is printed, and the enumerated data is saved to HDFS:

```

println("Records out : " + enumRddData.count() )

enumRddData.saveAsTextFile(outDataFile)

} // end object

```

In order to run this script as an application against Spark, it must be compiled. This is carried out with the `sbt package` command, which also compiles the code. The following command was run from the `nbayes` directory:

```
[hadoop@hc2nn nbayes]$ sbt package
Loading /usr/share/sbt/bin/sbt-launch-lib.bash
....
[info] Done packaging.
[success] Total time: 37 s, completed Feb 19, 2015 1:23:55 PM
```

This causes the compiled classes that are created to be packaged into a JAR library, as shown here:

```
[hadoop@hc2nn nbayes]$ pwd
/home/hadoop/spark/nbayes
[hadoop@hc2nn nbayes]$ ls -l target/scala-2.10
total 24
drwxrwxr-x 2 hadoop hadoop 4096 Feb 19 13:23 classes
-rw-rw-r-- 1 hadoop hadoop 17609 Feb 19 13:23 naive-bayes_2.10-1.0.jar
```

The application `convert1` can now be run against Spark using the application name, the Spark URL, and the full path to the JAR file that was created. Some extra parameters specify memory and maximum cores that are supposed to be used:

```
spark-submit \
  --class convert1 \
  --master spark://hc2nn.semtech-solutions.co.nz:7077 \
  --executor-memory 700M \
  --total-executor-cores 100 \
  /home/hadoop/spark/nbayes/target/scala-2.10/naive-bayes_2.10-1.0.jar
```

This creates a data directory on HDFS called the `/data/spark/nbayes/` followed by the result, which contains part files, containing the processed data:

```
[hadoop@hc2nn nbayes]$ hdfs dfs -ls /data/spark/nbayes
Found 2 items
-rw-r--r-- 3 hadoop supergroup 24645166 2015-01-29 21:27 /data/spark/nbayes/DigitalBreathTestData2013-MALE2.csv
```

```
drwxr-xr-x - hadoop supergroup          0 2015-02-19 13:36 /data/spark/
nbayes/result
```

```
[hadoop@hc2nn nbayes]$ hdfs dfs -ls /data/spark/nbayes/result
```

```
Found 3 items
```

```
-rw-r--r--  3 hadoop supergroup          0 2015-02-19 13:36 /data/spark/
nbayes/result/_SUCCESS
-rw-r--r--  3 hadoop supergroup    2828727 2015-02-19 13:36 /data/spark/
nbayes/result/part-00000
-rw-r--r--  3 hadoop supergroup    2865499 2015-02-19 13:36 /data/spark/
nbayes/result/part-00001
```

In the following HDFS `cat` command, I have concatenated the part file data into a file called `DigitalBreathTestData2013-MALE2a.csv`. I have then examined the top five lines of the file using the `head` command to show that it is numeric. Finally, I have loaded it into HDFS with the `put` command:

```
[hadoop@hc2nn nbayes]$ hdfs dfs -cat /data/spark/nbayes/result/part* > ./
DigitalBreathTestData2013-MALE2a.csv
```

```
[hadoop@hc2nn nbayes]$ head -5 DigitalBreathTestData2013-MALE2a.csv
```

```
0,3 0 0 75 3
0,0 0 0 0 1
0,3 0 1 12 4
0,3 0 0 0 5
1,2 0 3 0 1
```

```
[hadoop@hc2nn nbayes]$ hdfs dfs -put ./DigitalBreathTestData2013-MALE2a.
csv /data/spark/nbayes
```

The following HDFS `ls` command now shows the numeric data file stored on HDFS, in the `nbayes` directory:

```
[hadoop@hc2nn nbayes]$ hdfs dfs -ls /data/spark/nbayes
```

```
Found 3 items
```

```
-rw-r--r--  3 hadoop supergroup    24645166 2015-01-29 21:27 /data/spark/
nbayes/DigitalBreathTestData2013-MALE2.csv
-rw-r--r--  3 hadoop supergroup    5694226 2015-02-19 13:39 /data/spark/
nbayes/DigitalBreathTestData2013-MALE2a.csv
drwxr-xr-x - hadoop supergroup          0 2015-02-19 13:36 /data/spark/
nbayes/result
```

Now that the data has been converted into a numeric form, it can be processed with the MLlib Naïve Bayes algorithm; this is what the Scala file `bayes1.scala` does. This file imports the same configuration and context classes as before. It also imports MLlib classes for Naïve Bayes, vectors, and the `LabeledPoint` structure. The application class that is created this time is called `bayes1`:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.mllib.classification.NaiveBayes
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
```

```
object bayes1 extends App
{
```

Again, the HDFS data file is defined, and a Spark context is created as before:

```
  val hdfsServer = "hdfs://hc2nn.semtech-solutions.co.nz:8020"
  val hdfsPath   = "/data/spark/nbayes/"

  val dataFile = hdfsServer+hdfsPath+"DigitalBreathTestData2013-MALE2a.csv"

  val sparkMaster = "spark://hc2nn.semtech-solutions.co.nz:7077"
  val appName = "Naive Bayes 1"
  val conf = new SparkConf()
  conf.setMaster(sparkMaster)
  conf.setAppName(appName)

  val sparkCxt = new SparkContext(conf)
```

The raw CSV data is loaded and split by the separator characters. The first column becomes the label (`Male/Female`) that the data will be classified upon. The final columns separated by spaces become the classification features:

```
  val csvData = sparkCxt.textFile(dataFile)

  val ArrayData = csvData.map
```

```
{
  csvLine =>
    val colData = csvLine.split(',')
    LabeledPoint(colData(0).toDouble, Vectors.dense(colData(1).split('')
.map(_.toDouble)))
}
```

The data is then randomly divided into training (70%) and testing (30%) data sets:

```
val divData = ArrayData.randomSplit(Array(0.7, 0.3), seed = 13L)

val trainDataSet = divData(0)
val testDataSet  = divData(1)
```

The Naïve Bayes MLlib function can now be trained using the previous training set. The trained Naïve Bayes model, held in the variable `nbTrained`, can then be used to predict the Male/Female result labels against the testing data:

```
val nbTrained = NaiveBayes.train(trainDataSet)
val nbPredict = nbTrained.predict(testDataSet.map(_.features))
```

Given that all of the data already contained labels, the original and predicted labels for the test data can be compared. An accuracy figure can then be computed to determine how accurate the predictions were, by comparing the original labels with the prediction values:

```
val predictionAndLabel = nbPredict.zip(testDataSet.map(_.label))
val accuracy = 100.0 * predictionAndLabel.filter(x => x._1 == x._2).
count() / testDataSet.count()
println( "Accuracy : " + accuracy );
}
```

So this explains the Scala Naïve Bayes code example. It's now time to run the compiled `bayes1` application using `spark-submit`, and to determine the classification accuracy. The parameters are the same. It's just the class name that has changed:

```
spark-submit \
  --class bayes1 \
  --master spark://hc2nn.semtech-solutions.co.nz:7077 \
  --executor-memory 700M \
  --total-executor-cores 100 \
  /home/hadoop/spark/nbayes/target/scala-2.10/naive-bayes_2.10-1.0.jar
```

The resulting accuracy given by the Spark cluster is just 43 percent, which seems to imply that this data is not suitable for Naïve Bayes:

Accuracy: 43.30

In the next example, I will use K-Means to try and determine what clusters exist within the data. Remember, Naïve Bayes needs the data classes to be linearly divisible along the class boundaries. With K-Means, it will be possible to determine both: the membership and centroid location of the clusters within the data.

Clustering with K-Means

This example will use the same test data from the previous example, but will attempt to find clusters in the data using the MLlib K-Means algorithm.

Theory

The K-Means algorithm iteratively attempts to determine clusters within the test data by minimizing the distance between the mean value of cluster center vectors, and the new candidate cluster member vectors. The following equation assumes data set members that range from X_1 to X_n ; it also assumes K cluster sets that range from S_1 to S_k where $K \leq n$.

$$\arg \min_s \sum_{i=1}^K \sum_{x \in S_i} \|x - B_i\|^2$$

where B_i is the mean of members of S_i

K-Means in practice

Again, the K-Means MLlib functionality uses the LabeledPoint structure to process its data and so, it needs numeric input data. As the same data from the last section is being reused, I will not re-explain the data conversion. The only change that has been made in data terms, in this section, is that processing under HDFS will now take place under the `/data/spark/kmeans/` directory. Also, the conversion Scala script for the K-Means example produces a record that is all comma separated.

The development and processing for the K-Means example has taken place under the `/home/hadoop/spark/kmeans` directory, to separate the work from other development. The sbt configuration file is now called `kmeans.sbt`, and is identical to the last example, except for the project name:

```
name := "K-Means"
```

The code for this section can be found in the software package under `chapter2\K-Means`. So, looking at the code for `kmeans1.scala`, which is stored under `kmeans/src/main/scala`, some similar actions occur. The import statements refer to Spark context and configuration. This time, however, the K-Means functionality is also being imported from MLlib. Also, the application class name has been changed for this example to `kmeans1`:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}

object kmeans1 extends App
{
```

The same actions are being taken as the last example to define the data file—define the Spark configuration and create a Spark context:

```
    val hdfsServer = "hdfs://hc2nn.semtech-solutions.co.nz:8020"
    val hdfsPath   = "/data/spark/kmeans/"

    val dataFile   = hdfsServer + hdfsPath + "DigitalBreathTestData2013-
MALE2a.csv"

    val sparkMaster = "spark://hc2nn.semtech-solutions.co.nz:7077"
    val appName     = "K-Means 1"
    val conf        = new SparkConf()

    conf.setMaster(sparkMaster)
    conf.setAppName(appName)

    val sparkCxt = new SparkContext(conf)
```

Next, the CSV data is loaded from the data file, and is split by comma characters into the variable `VectorData`:

```
    val csvData = sparkCxt.textFile(dataFile)
    val VectorData = csvData.map
```

```
{  
  csvLine =>  
    Vectors.dense( csvLine.split(',').map(_toDouble))  
}
```

A K-Means object is initialized, and the parameters are set to define the number of clusters, and the maximum number of iterations to determine them:

```
val kMeans = new KMeans  
val numClusters = 3  
val maxIterations = 50
```

Some default values are defined for initialization mode, the number of runs, and Epsilon, which I needed for the K-Means call, but did not vary for processing. Finally, these parameters were set against the K-Means object:

```
val initializationMode = KMeans.K_MEANS_PARALLEL  
val numRuns = 1  
val numEpsilon = 1e-4  
  
kMeans.setK( numClusters )  
kMeans.setMaxIterations( maxIterations )  
kMeans.setInitializationMode( initializationMode )  
kMeans.setRuns( numRuns )  
kMeans.setEpsilon( numEpsilon )
```

I cached the training vector data to improve the performance, and trained the K-Means object using the Vector Data to create a trained K-Means model:

```
VectorData.cache  
val kMeansModel = kMeans.run( VectorData )
```

I have computed the K-Means cost, the number of input data rows, and output the results via print line statements. The cost value indicates how tightly the clusters are packed, and how separated clusters are:

```
val kMeansCost = kMeansModel.computeCost( VectorData )  
  
println( "Input data rows : " + VectorData.count() )  
println( "K-Means Cost      : " + kMeansCost )
```

Next, I have used the K-Means Model to print the cluster centers as vectors for each of the three clusters that were computed:

```
kMeansModel.clusterCenters.foreach{ println }
```

Finally, I have used the K-Means Model `predict` function to create a list of cluster membership predictions. I have then counted these predictions by value to give a count of the data points in each cluster. This shows which clusters are bigger, and if there really are three clusters:

```
val clusterRddInt = kMeansModel.predict( VectorData )
```

```
val clusterCount = clusterRddInt.countByValue
```

```
clusterCount.toList.foreach{ println }
```

```
} // end object kmeans1
```

So, in order to run this application, it must be compiled and packaged from the `kmeans` subdirectory as the Linux `pwd` command shows here:

```
[hadoop@hc2nn kmeans]$ pwd
```

```
/home/hadoop/spark/kmeans
```

```
[hadoop@hc2nn kmeans]$ sbt package
```

```
Loading /usr/share/sbt/bin/sbt-launch-lib.bash
```

```
[info] Set current project to K-Means (in build file:/home/hadoop/spark/kmeans/)
```

```
[info] Compiling 2 Scala sources to /home/hadoop/spark/kmeans/target/scala-2.10/classes...
```

```
[info] Packaging /home/hadoop/spark/kmeans/target/scala-2.10/kmeans_2.10-1.0.jar ...
```

```
[info] Done packaging.
```

```
[success] Total time: 20 s, completed Feb 19, 2015 5:02:07 PM
```

Once this packaging is successful, I check HDFS to ensure that the test data is ready. As in the last example, I converted my data to numeric form using the `convert.scala` file, provided in the software package. I will process the data file `DigitalBreathTestData2013-MALE2a.csv` in the HDFS directory `/data/spark/kmeans` shown here:

```
[hadoop@hc2nn nbayes]$ hdfs dfs -ls /data/spark/kmeans
```

```
Found 3 items
```

```
-rw-r--r--    3 hadoop supergroup    24645166 2015-02-05 21:11 /data/spark/
kmeans/DigitalBreathTestData2013-MALE2.csv
-rw-r--r--    3 hadoop supergroup    5694226 2015-02-05 21:48 /data/spark/
kmeans/DigitalBreathTestData2013-MALE2a.csv
drwxr-xr-x    - hadoop supergroup          0 2015-02-05 21:46 /data/spark/
kmeans/result
```

The spark-submit tool is used to run the K-Means application. The only change in this command, as shown here, is that the class is now `kmeans1`:

```
spark-submit \
  --class kmeans1 \
  --master spark://hc2nn.semtech-solutions.co.nz:7077 \
  --executor-memory 700M \
  --total-executor-cores 100 \
  /home/hadoop/spark/kmeans/target/scala-2.10/k-means_2.10-1.0.jar
```

The output from the Spark cluster run is shown to be as follows:

```
Input data rows : 467054
K-Means Cost    : 5.40312223450789E7
```

The previous output shows the input data volume, which looks correct, plus it also shows the K-Means cost value. Next comes the three vectors, which describe the data cluster centers with the correct number of dimensions. Remember that these cluster centroid vectors will have the same number of columns as the original vector data:

```
[0.24698249738061878,1.3015883142472253,0.005830116872250263,2.9173747788
555207,1.156645130895448,3.4400290524342454]
```

```
[0.3321793984152627,1.784137241326256,0.007615970459266097,2.583198707592
8917,119.58366028156011,3.8379106085083468]
```

```
[0.25247226760684494,1.702510963969387,0.006384899819416975,2.23140424800
0688,52.202897927594805,3.551509158139135]
```

Finally, cluster membership is given for clusters 1 to 3 with cluster 1 (index 0) having the largest membership at 407,539 member vectors.

```
(0,407539)
(1,12999)
(2,46516)
```

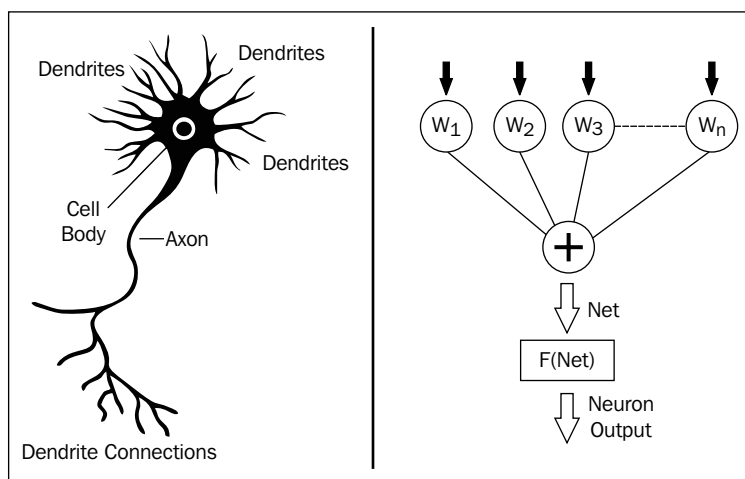
So, these two examples show how data can be classified and clustered using Naïve Bayes and K-Means. But what if I want to classify images or more complex patterns, and use a black box approach to classification? The next section examines Spark-based classification using ANN's, or **Artificial Neural Network's**. In order to do this, I need to download the latest Spark code, and build a server for Spark 1.3, as it has not yet been formally released (at the time of writing).

ANN – Artificial Neural Networks

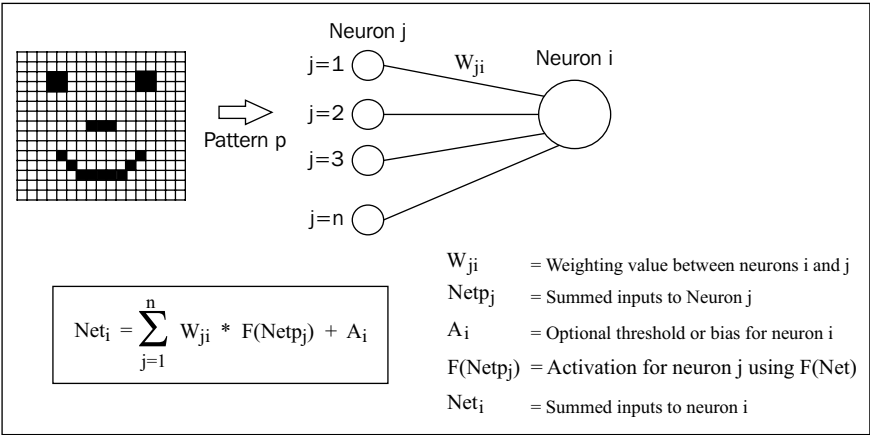
In order to examine the ANN (Artificial Neural Network) functionality in Apache Spark, I will need to obtain the latest source code from the GitHub website. The ANN functionality has been developed by Bert Greevenbosch (<http://www.bertgreevenbosch.nl/>), and is set to be released in Apache Spark 1.3. At the time of writing the current Spark release is 1.2.1, and CDH 5.x ships with Spark 1.0. So, in order to examine this unreleased ANN functionality, the source code will need to be sourced and built into a Spark server. This is what I will do after explaining a little on the theory behind ANN.

Theory

The following figure shows a simple biological neuron to the left. The neuron has dendrites that receive signals from other neurons. A cell body controls activation, and an axon carries an electrical impulse to the dendrites of other neurons. The artificial neuron to the right has a series of weighted inputs: a summing function that groups the inputs, and a firing mechanism (**F(Net)**), which decides whether the inputs have reached a threshold, and if so, the neuron will fire:

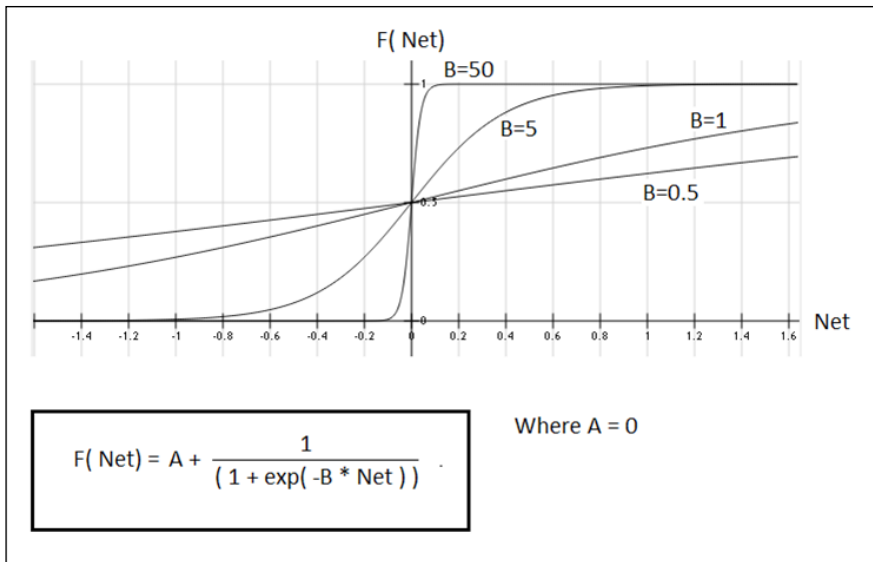


Neural networks are tolerant of noisy images and distortion, and so are useful when a black box classification method is needed for potentially degraded images. The next area to consider is the summation function for the neuron inputs. The following diagram shows the summation function called **Net** for neuron **i**. The connections between the neurons that have the weighting values, contain the stored knowledge of the network. Generally, a network will have an input layer, an output layer, and a number of hidden layers. A neuron will fire if the sum of its inputs exceeds a threshold.



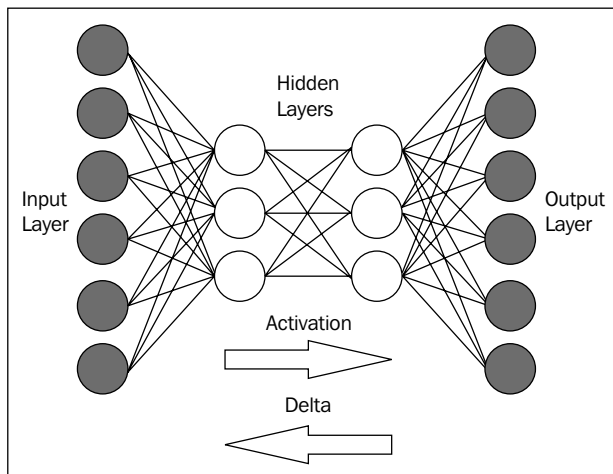
In the previous equation, the diagram and the key show that the input values from a pattern **P** are passed to neurons in the input layer of a network. These values become the input layer neuron activation values; they are a special case. The inputs to neuron **i** are the sum of the weighting value for neuron connection **i-j**, multiplied by the activation from neuron **j**. The activation at neuron **j** (if it is not an input layer neuron) is given by **F(Net)**, the squashing function, which will be described next.

A simulated neuron needs a firing mechanism, which decides whether the inputs to the neuron have reached a threshold. And then, it fires to create the activation value for that neuron. This firing or squashing function can be described by the generalized sigmoid function shown in the following figure:



This function has two constants: **A** and **B**; **B** affects the shape of the activation curve as shown in the previous graph. The bigger the value, the more similar a function becomes to an on/off step. The value of **A** sets a minimum for the returned activation. In the previous graph it is zero.

So, this provides a mechanism for simulating a neuron, creating weighting matrices as the neuron connections, and managing the neuron activation. But how are the networks organized? The next diagram shows a suggested neuron architecture—the neural network has an input layer of neurons, an output layer, and one or more hidden layers. All neurons in each layer are connected to each neuron in the adjacent layers.



During the training, activation passes from the input layer through the network to the output layer. Then, the error or difference between the expected or actual output causes error deltas to be passed back through the network, altering the weighting matrix values. Once the desired output layer vector is achieved, then the knowledge is stored in the weighting matrices, and the network can be further trained or used for classification.

So, the theory behind neural networks has been described in terms of back propagation. Now is the time to obtain the development version of the Apache Spark code, and build the Spark server, so that the ANN Scala code can be run.

Building the Spark server

I would not normally advise that Apache Spark code be downloaded and used before it has been released by Spark, or packaged by Cloudera (for use with CDH), but the desire to examine ANN functionality, along with the time scale allowed for this book, mean that I need to do so. I extracted the full Spark code tree from this path:

```
https://github.com/apache/spark/pull/1290.
```

I stored this code on the Linux server `hc2nn`, under the directory `/home/hadoop/spark/spark`. I then obtained the ANN code from Bert Greevenbosch's GitHub development area:

```
https://github.com/bgreeven/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/ann/ArtificialNeuralNetwork.scala
```

```
https://github.com/bgreeven/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/classification/ANNClassifier.scala
```

The `ANNClassifier.scala` file contains the public functions that will be called. The `ArtificialNeuralNetwork.scala` file contains the private MLlib ANN functions that `ANNClassifier.scala` calls. I already have Java open JDK installed on my server, so the next step is to set up the `spark-env.sh` environment configuration file under `/home/hadoop/spark/spark/conf`. My file looks like this:

```
export STANDALONE_SPARK_MASTER_HOST=hc2nn.semtech-solutions.co.nz
export SPARK_MASTER_IP=$STANDALONE_SPARK_MASTER_HOST
export SPARK_HOME=/home/hadoop/spark/spark
export SPARK_LAUNCH_WITH_SCALA=0
export SPARK_MASTER_WEBUI_PORT=19080
export SPARK_MASTER_PORT=8077
export SPARK_WORKER_PORT=8078
```

```

export SPARK_WORKER_WEBUI_PORT=19081
export SPARK_WORKER_DIR=/var/run/spark/work
export SPARK_LOG_DIR=/var/log/spark
export SPARK_HISTORY_SERVER_LOG_DIR=/var/log/spark
export SPARK_PID_DIR=/var/run/spark/
export HADOOP_CONF_DIR=/etc/hadoop/conf
export SPARK_JAR_PATH=${SPARK_HOME}/assembly/target/scala-2.10/
export SPARK_JAR=${SPARK_JAR_PATH}/spark-assembly-1.3.0-SNAPSHOT-
hadoop2.3.0-cdh5.1.2.jar
export JAVA_HOME=/usr/lib/jvm/java-1.7.0
export SPARK_LOCAL_IP=192.168.1.103

```

The `SPARK_MASTER_IP` variable tells the cluster which server is the master. The port variables define the master, the worker web, and the operating port values. There are some log and JAR file paths defined, as well as `JAVA_HOME` and the local server IP address. Details for building Spark with Apache Maven can be found at:

<http://spark.apache.org/docs/latest/building-spark.html>

The slaves file in the same directory will be set up as before with the names of the four workers servers from `hc2r1m1` to `hc2r1m4`.

In order to build using Apache Maven, I had to install `mvn` on to my Linux server `hc2nn`, where I will run the Spark build. I did this as the root user, obtaining a Maven repository file by first using `wget`:

```

wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-
maven.repo -O /etc/yum.repos.d/epel-apache-maven.repo

```

Then, checking that the new repository file is in place with `ls` long listing.

```

[root@hc2nn ~]# ls -l /etc/yum.repos.d/epel-apache-maven.repo
-rw-r--r-- 1 root root 445 Mar  4 2014 /etc/yum.repos.d/epel-apache-
maven.repo

```

Then Maven can be installed using the Linux `yum` command, the examples below show the install command and a check via `ls` that the `mvn` command exists.

```

[root@hc2nn ~]# yum install apache-maven
[root@hc2nn ~]# ls -l /usr/share/apache-maven/bin/mvn
-rwxr-xr-x 1 root root 6185 Dec 15 06:30 /usr/share/apache-maven/bin/mvn

```

The commands that I have used to build the Spark source tree are shown here along with the successful output. First, the environment is set up, and then the build is started with the `mvn` command. Options are added to build for Hadoop 2.3/yarn, and the tests are skipped. The build uses the `clean` and `package` options to remove the old build files each time, and then create JAR files. Finally, the build output is copied via the `tee` command to a file named `build.log`:

```
cd /home/hadoop/spark/spark/conf ; . ./spark-env.sh ; cd ..
```

```
mvn -Pyarn -Phadoop-2.3 -Dhadoop.version=2.3.0-cdh5.1.2 -DskipTests  
clean package | tee build.log 2>&1
```

```
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 44:20 min  
[INFO] Finished at: 2015-02-16T12:20:28+13:00  
[INFO] Final Memory: 76M/925M  
[INFO] -----
```

The actual build command that you use will depend upon whether you have Hadoop, and the version of it. Check the previous *building spark* for details, the build takes around 40 minutes on my servers.

Given that this build will be packaged and copied to the other servers in the Spark cluster, it is important that all the servers use the same version of Java, else errors such as these will occur:

```
15/02/15 12:41:41 ERROR executor.Executor: Exception in task 0.1 in stage  
0.0 (TID 2)  
  
java.lang.VerifyError: class org.apache.hadoop.hdfs.protocol.proto.Cli  
entNamenodeProtocolProtos$GetBlockLocationsRequestProto overrides final  
method getUnknownFields.()Lcom/google/protobuf/UnknownFieldSet;  
    at java.lang.ClassLoader.defineClass1(Native Method)
```

Given that the source tree has been built, it now needs to be bundled up and released to each of the servers in the Spark cluster. Given that these servers are also the members of the CDH cluster, and have password-less SSH access set up, I can use the `scp` command to release the built software. The following commands show the `spark` directory under the `/home/hadoop/spark` path being packaged into a tar file called `spark_bld.tar`. The Linux `scp` command is then used to copy the tar file to each slave server; the following example shows `hc2r1m1`:

```
[hadoop@hc2nn spark]$ cd /home/hadoop/spark
```

```
[hadoop@hc2nn spark]$ tar cvf spark_bld.tar spark
[hadoop@hc2nn spark]$ scp ./spark_bld.tar hadoop@hc2r1m1:/home/hadoop/
spark/spark_bld.tar
```

Now that the tarred Spark build is on the slave node, it needs to be unpacked. The following command shows the process for the server `hc2r1m1`. The tar file is unpacked to the same directory as the build server `hc2nn`, that is, `/home/hadoop/spark`:

```
[hadoop@hc2r1m1 ~]$ mkdir spark ; mv spark_bld.tar spark
[hadoop@hc2r1m1 ~]$ cd spark ; ls
spark_bld.tar
[hadoop@hc2r1m1 spark]$ tar xvf spark_bld.tar
```

Once the build has been run successfully, and the built code has been released to the slave servers, the built version of Spark can be started from the master server **hc2nn**. Note that I have chosen different port numbers from the Spark version 1.0, installed on these servers. Also note that I will start Spark as root, because the Spark 1.0 install is managed as Linux services under the root account. As the two installs will share facilities like logging and `.pid` file locations, root user will ensure access. This is the script that I have used to start Apache Spark 1.3:

```
cd /home/hadoop/spark/spark/conf ; . ./spark-env.sh ; cd ../sbin
echo "hc2nn - start master server"
./start-master.sh
echo "sleep 5000 ms"
sleep 5
echo "hc2nn - start history server"
./start-history-server.sh
echo "Start Spark slaves workers"
./start-slaves.sh
```

It executes the `spark-env.sh` file to set up the environment, and then uses the scripts in the Spark `sbin` directory to start the services. It starts the master and the history server first on `hc2nn`, and then it starts the slaves. I added a delay before starting the slaves, as I found that they were trying to connect to the master before it was ready. The Spark 1.3 web user interface can now be accessed via this URL:

```
http://hc2nn.semtech-solutions.co.nz:19080/
```

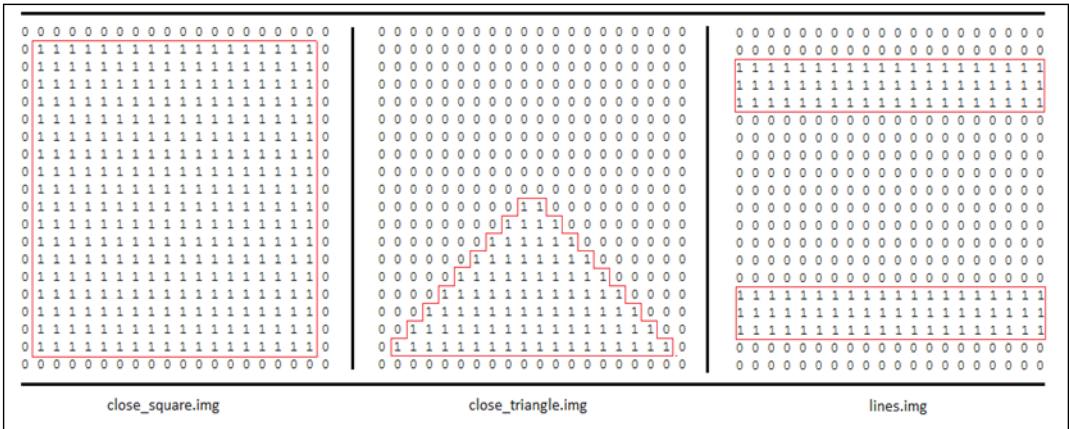
The Spark URL, which allows applications to connect to Spark is this:

```
Spark Master at spark://hc2nn.semtech-solutions.co.nz:8077
```

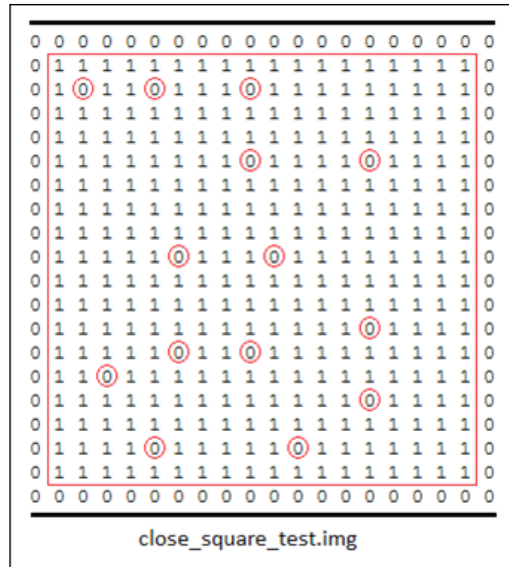
As defined by the port numbers in the spark environment configuration file, Spark is now available to be used with ANN functionality. The next section will present the ANN Scala scripts and data to show how this Spark-based functionality can be used.

ANN in practice

In order to begin ANN training, test data is needed. Given that this type of classification method is supposed to be good at classifying distorted or noisy images, I have decided to attempt to classify the images here:



They are hand-crafted text files that contain shaped blocks, created from the characters 1 and 0. When they are stored on HDFS, the carriage return characters are removed, so that the image is presented as a single line vector. So, the ANN will be classifying a series of shape images, and then it will be tested against the same images with noise added to determine whether the classification will still work. There are six training images, and they will each be given an arbitrary training label from 0.1 to 0.6. So, if the ANN is presented with a closed square, it should return a label of 0.1. The following image shows an example of a testing image with noise added. The noise, created by adding extra zero (0) characters within the image, has been highlighted:



Because the Apache Spark server has changed from the previous examples, and the Spark library locations have also changed, the `sbt` configuration file used for compiling the example ANN Scala code must also be changed. As before, the ANN code is being developed using the Linux `hadoop` account in a subdirectory called `spark/ann`. The `ann.sbt` file exists within the `ann` directory:

```
[hadoop@hc2nn ann]$ pwd
/home/hadoop/spark/ann

[hadoop@hc2nn ann]$ ls
ann.sbt  project  src      target
```

The contents of the `ann.sbt` file have been changed to use full paths of JAR library files for the Spark dependencies. This is because the new Apache Spark code for build 1.3 now resides under `/home/hadoop/spark/spark`. Also, the project name has been changed to `ANN`:

```
name := "ANN"
version := "1.0"
scalaVersion := "2.10.4"
```

```
libraryDependencies += "org.apache.hadoop" % "hadoop-client" % "2.3.0"
libraryDependencies += "org.apache.spark" % "spark-core" % "1.3.0" from
"file:///home/hadoop/spark/spark/core/target/spark-core_2.10-1.3.0-
SNAPSHOT.jar"
libraryDependencies += "org.apache.spark" % "spark-mllib" % "1.3.0" from
"file:///home/hadoop/spark/spark/mllib/target/spark-mllib_2.10-1.3.0-
SNAPSHOT.jar"
libraryDependencies += "org.apache.spark" % "akka" % "1.3.0" from
"file:///home/hadoop/spark/spark/assembly/target/scala-2.10/spark-
assembly-1.3.0-SNAPSHOT-hadoop2.3.0-cdh5.1.2.jar"
```

As in the previous examples, the actual Scala code to be compiled exists in a subdirectory named `src/main/scala` as shown next. I have created two Scala programs. The first trains using the input data, and then tests the ANN model with the same input data. The second tests the trained model with noisy data, to the test distorted data classification:

```
[hadoop@hc2nn scala]$ pwd
/home/hadoop/spark/ann/src/main/scala
```

```
[hadoop@hc2nn scala]$ ls
test_ann1.scala  test_ann2.scala
```

I will examine the first Scala file entirely, and then I will just show the extra features of the second file, as the two examples are very similar up to the point of training the ANN. The code examples shown here can be found in the software package provided with this book, under the path `chapter2\ANN`. So, to examine the first Scala example, the import statements are similar to the previous examples. The Spark context, configuration, vectors, and `LabeledPoint` are being imported. The RDD class for RDD processing is being imported this time, along with the new ANN class `ANNClassifier`. Note that the `MLlib/classification` routines widely use the `LabeledPoint` structure for input data, which will contain the features and labels that are supposed to be trained against:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

import org.apache.spark.mllib.classification.ANNClassifier
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
```

```
import org.apache.spark.mllib.linalg._
import org.apache.spark.rdd.RDD
```

```
object testann1 extends App
{
```

The application class in this example has been called `testann1`. The HDFS files to be processed have been defined in terms of the HDFS server, path, and file name:

```
val server = "hdfs://hc2nn.semtech-solutions.co.nz:8020"
val path   = "/data/spark/ann/"

val data1 = server + path + "close_square.img"
val data2 = server + path + "close_triangle.img"
val data3 = server + path + "lines.img"
val data4 = server + path + "open_square.img"
val data5 = server + path + "open_triangle.img"
val data6 = server + path + "plus.img"
```

The Spark context has been created with the URL for the Spark instance, which now has a different port number—8077. The application name is `ANN 1`. This will appear on the Spark web UI when the application is run:

```
val sparkMaster = "spark://hc2nn.semtech-solutions.co.nz:8077"
val appName     = "ANN 1"
val conf        = new SparkConf()

conf.setMaster(sparkMaster)
conf.setAppName(appName)

val sparkCxt = new SparkContext(conf)
```

The HDFS-based input training and test data files are loaded. The values on each line are split by space characters, and the numeric values have been converted into Doubles. The variables that contain this data are then stored in an array called `inputs`. At the same time, an array called `outputs` is created, containing the labels from 0.1 to 0.6. These values will be used to classify the input patterns:

```
val rData1 = sparkCxt.textFile(data1).map(_.split(" ").map(_.  
toDouble)).collect
```

```
val rData2 = sparkCxt.textFile(data2).map(_.split(" ").map(_.toDouble)).collect
val rData3 = sparkCxt.textFile(data3).map(_.split(" ").map(_.toDouble)).collect
val rData4 = sparkCxt.textFile(data4).map(_.split(" ").map(_.toDouble)).collect
val rData5 = sparkCxt.textFile(data5).map(_.split(" ").map(_.toDouble)).collect
val rData6 = sparkCxt.textFile(data6).map(_.split(" ").map(_.toDouble)).collect

val inputs = Array[Array[Double]] (
    rData1(0), rData2(0), rData3(0), rData4(0), rData5(0), rData6(0) )

val outputs = Array[Double]( 0.1, 0.2, 0.3, 0.4, 0.5, 0.6 )
```

The input and output data, representing the input data features and labels, are then combined and converted into a `LabeledPoint` structure. Finally, the data is parallelized in order to partition it for the optimal parallel processing:

```
val ioData = inputs.zip( outputs )
val lpData = ioData.map{ case(features,label) =>

    LabeledPoint( label, Vectors.dense(features) )
}

val rddData = sparkCxt.parallelize( lpData )
```

Variables are created to define the hidden layer topology of the ANN. In this case, I have chosen to have two hidden layers, each with 100 neurons. The maximum numbers of iterations are defined, as well as a batch size (six patterns) and convergence tolerance. The tolerance refers to how big the training error can get before we can consider training to have worked. Then, an ANN model is created using these configuration parameters and the input data:

```
val hiddenTopology : Array[Int] = Array( 100, 100 )
val maxNumIterations = 1000
val convTolerance    = 1e-4
val batchSize        = 6

val annModel = ANNClassifier.train(rddData,
                                   batchSize,
```

```

        hiddenTopology,
        maxNumIterations,
        convTolerance)

```

In order to test the trained ANN model, the same input training data is used as testing data used to obtain prediction labels. First, an input data variable is created called `rPredictData`. Then, the data is partitioned and finally, the predictions are obtained using the trained ANN model. For this model to work, it must output the labels 0.1 to 0.6:

```

val rPredictData = inputs.map{ case(features) =>

    ( Vectors.dense(features) )
}
val rddPredictData = sparkCxt.parallelize( rPredictData )
val predictions = annModel.predict( rddPredictData )

```

The label predictions are printed, and the script closes with a closing bracket:

```

    predictions.toArray().foreach( value => println( "prediction > " +
value ) )
} // end ann1

```

So, in order to run this code sample, it must first be compiled and packaged. By now, you must be familiar with the `sbt` command, executed from the `ann` sub directory:

```

[hadoop@hc2nn ann]$ pwd
/home/hadoop/spark/ann
[hadoop@hc2nn ann]$ sbt package

```

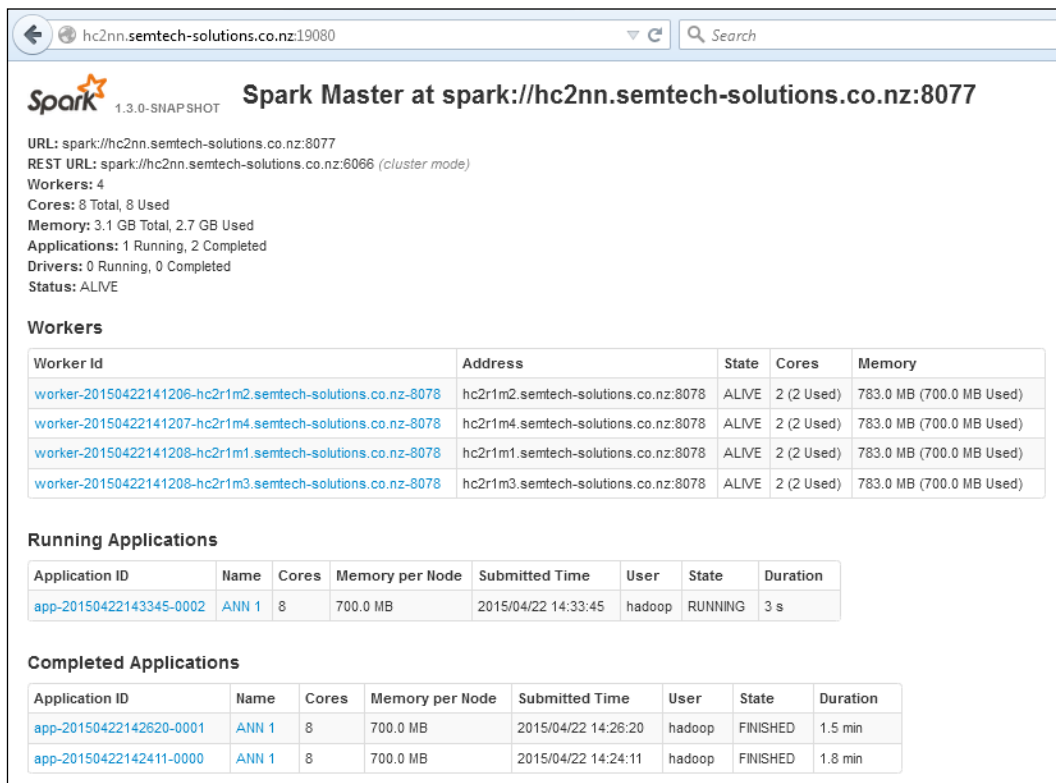
The `spark-submit` command is then used from within the new `spark/spark` path using the new Spark-based URL at port 8077 to run the application `testann1`:

```

/home/hadoop/spark/spark/bin/spark-submit \
--class testann1 \
--master spark://hc2nn.semtech-solutions.co.nz:8077 \
--executor-memory 700M \
--total-executor-cores 100 \
/home/hadoop/spark/ann/target/scala-2.10/a-n-n_2.10-1.0.jar

```

By checking the Apache Spark web URL at <http://hc2nn.semtech-solutions.co.nz:19080/>, it is now possible to see the application running. The following figure shows the application ANN 1 running, as well as the previous completed executions:



The screenshot shows the Apache Spark web interface at <http://hc2nn.semtech-solutions.co.nz:19080/>. The page displays the Spark Master status and a list of workers.

Spark Master at spark://hc2nn.semtech-solutions.co.nz:8077

1.3.0-SNAPSHOT

URL: spark://hc2nn.semtech-solutions.co.nz:8077
 REST URL: spark://hc2nn.semtech-solutions.co.nz:8066 (cluster mode)

Workers: 4
 Cores: 8 Total, 8 Used
 Memory: 3.1 GB Total, 2.7 GB Used
 Applications: 1 Running, 2 Completed
 Drivers: 0 Running, 0 Completed
 Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20150422141206-hc2r1m2.semtech-solutions.co.nz-8078	hc2r1m2.semtech-solutions.co.nz:8078	ALIVE	2 (2 Used)	783.0 MB (700.0 MB Used)
worker-20150422141207-hc2r1m4.semtech-solutions.co.nz-8078	hc2r1m4.semtech-solutions.co.nz:8078	ALIVE	2 (2 Used)	783.0 MB (700.0 MB Used)
worker-20150422141208-hc2r1m1.semtech-solutions.co.nz-8078	hc2r1m1.semtech-solutions.co.nz:8078	ALIVE	2 (2 Used)	783.0 MB (700.0 MB Used)
worker-20150422141208-hc2r1m3.semtech-solutions.co.nz-8078	hc2r1m3.semtech-solutions.co.nz:8078	ALIVE	2 (2 Used)	783.0 MB (700.0 MB Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20150422143345-0002	ANN 1	8	700.0 MB	2015/04/22 14:33:45	hadoop	RUNNING	3 s

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20150422142620-0001	ANN 1	8	700.0 MB	2015/04/22 14:26:20	hadoop	FINISHED	1.5 min
app-20150422142411-0000	ANN 1	8	700.0 MB	2015/04/22 14:24:11	hadoop	FINISHED	1.8 min

By selecting one of the cluster host worker instances, it is possible to see a list of executors that actually carry out cluster processing for that worker:


Application: ANN 1
 1.3.0-SNAPSHOT

ID: app-20150422143345-0002
 Name: ANN 1
 User: hadoop
 Cores: 100 (8 granted, 92 left)
 Executor Memory: 700.0 MB
 Submit Date: Wed Apr 22 14:33:45 NZST 2015
 State: RUNNING
[Application Detail UI](#)

Executor Summary

ExecutorID	Worker	Cores	Memory	State	Logs
2	worker-20150422141208-hc2r1m1.semtech-solutions.co.nz-8078	2	700	RUNNING	stdout stderr
1	worker-20150422141208-hc2r1m3.semtech-solutions.co.nz-8078	2	700	RUNNING	stdout stderr
3	worker-20150422141206-hc2r1m2.semtech-solutions.co.nz-8078	2	700	RUNNING	stdout stderr
0	worker-20150422141207-hc2r1m4.semtech-solutions.co.nz-8078	2	700	RUNNING	stdout stderr

Finally, by selecting one of the executors, it is possible to see its history and configuration, as well as the links to the log file, and error information. At this level, with the log information provided, debugging is possible. These log files can be checked for processing error messages.

hc2r1m2.semtech-solutions.co.nz:19081

Spark 1.3.0-SNAPSHOT Spark Worker at hc2r1m2.semtech-solutions.co.nz:8078

ID: worker-20150222113812-hc2r1m2.semtech-solutions.co.nz-8078

Master URL: spark://hc2nn.semtech-solutions.co.nz:8077

Cores: 2 (2 Used)

Memory: 783.0 MB (700.0 MB Used)

[Back to Master](#)

Running Executors (1)

ExecutorID	Cores	State	Memory	Job Details	Logs
2	2	LOADING	700.0 MB	ID: app-20150222134613-0002 Name: ANN 1 User: root	stdout stderr

Finished Executors (2)

ExecutorID	Cores	State	Memory	Job Details	Logs
2	2	KILLED	700.0 MB	ID: app-20150222125002-0000 Name: ANN 1 User: root	stdout stderr
2	2	KILLED	700.0 MB	ID: app-20150222125935-0001 Name: ANN 1 User: root	stdout stderr

The ANN 1 application provides the following output to show that it has reclassified the same input data correctly. The reclassification has been successful, as each of the input patterns has been given the same label as it was trained with:

```
prediction > 0.1
prediction > 0.2
prediction > 0.3
prediction > 0.4
prediction > 0.5
prediction > 0.6
```

So, this shows that ANN training and test prediction will work with the same data. Now, I will train with the same data, but test with distorted or noisy data, an example of which I already demonstrated. This example can be found in the file called `test_ann2.scala`, in your software package. It is very similar to the first example, so I will just demonstrate the changed code. The application is now called `testann2`:

```
object testann2 extends App
```

An extra set of testing data is created, after the ANN model has been created using the training data. This testing data contains noise:

```
val tData1 = server + path + "close_square_test.img"
val tData2 = server + path + "close_triangle_test.img"
val tData3 = server + path + "lines_test.img"
val tData4 = server + path + "open_square_test.img"
val tData5 = server + path + "open_triangle_test.img"
val tData6 = server + path + "plus_test.img"
```

This data is processed into input arrays, and is partitioned for cluster processing:

```
val rtData1 = sparkCxt.textFile(tData1).map(_.split(" ").map(_.toDouble)).collect
val rtData2 = sparkCxt.textFile(tData2).map(_.split(" ").map(_.toDouble)).collect
val rtData3 = sparkCxt.textFile(tData3).map(_.split(" ").map(_.toDouble)).collect
val rtData4 = sparkCxt.textFile(tData4).map(_.split(" ").map(_.toDouble)).collect
val rtData5 = sparkCxt.textFile(tData5).map(_.split(" ").map(_.toDouble)).collect
val rtData6 = sparkCxt.textFile(tData6).map(_.split(" ").map(_.toDouble)).collect

val tInputs = Array[Array[Double]] (
    rtData1(0), rtData2(0), rtData3(0), rtData4(0), rtData5(0),
    rtData6(0) )

val rTestPredictData = tInputs.map{ case(features) => ( Vectors.dense(features) ) }
val rddTestPredictData = sparkCxt.parallelize( rTestPredictData )
```

It is then used to generate label predictions in the same way as the first example. If the model classifies the data correctly, then the same label values should be printed from 0.1 to 0.6:

```
val testPredictions = annModel.predict( rddTestPredictData )
testPredictions.toArray().foreach( value => println( "test
prediction > " + value ) )
```

The code has already been compiled, so it can be run using the `spark-submit` command:

```
/home/hadoop/spark/spark/bin/spark-submit \
--class testann2 \
--master spark://hc2nn.semtech-solutions.co.nz:8077 \
--executor-memory 700M \
--total-executor-cores 100 \
/home/hadoop/spark/ann/target/scala-2.10/a-n-n_2.10-1.0.jar
```

Here is the cluster output from this script, which shows a successful classification using a trained ANN model, and some noisy test data. The noisy data has been classified correctly. For instance, if the trained model had become confused, it might have given a value of 0.15 for the noisy `close_square_test.img` test image in position one, instead of returning 0.1 as it did:

```
test prediction > 0.1
test prediction > 0.2
test prediction > 0.3
test prediction > 0.4
test prediction > 0.5
test prediction > 0.6
```

Summary

This chapter has attempted to provide you with an overview of some of the functionality available within the Apache Spark MLlib module. It has also shown the functionality that will soon be available in terms of ANN, or artificial neural networks, which is intended for release in Spark 1.3. It has not been possible to cover all the areas of MLlib, due to the time and space allowed for this chapter.

You have been shown how to develop Scala-based examples for Naïve Bayes classification, K-Means clustering, and ANN or artificial neural networks. You have been shown how to prepare test data for these Spark MLlib routines. You have also been shown that they all accept the LabeledPoint structure, which contains features and labels. Also, each approach takes a training and prediction approach to training and testing a model using different data sets. Using the approach shown in this chapter, you can now investigate the remaining functionality in the MLlib library. You should refer to the <http://spark.apache.org/> website, and ensure that when checking documentation that you refer to the correct version, that is, <http://spark.apache.org/docs/1.0.0/> for version 1.0.0.

Having examined the Apache Spark MLlib machine learning library, in this chapter, it is now time to consider Apache Spark's stream processing capability. The next chapter will examine stream processing using the Spark and Scala-based example code.

3

Apache Spark Streaming

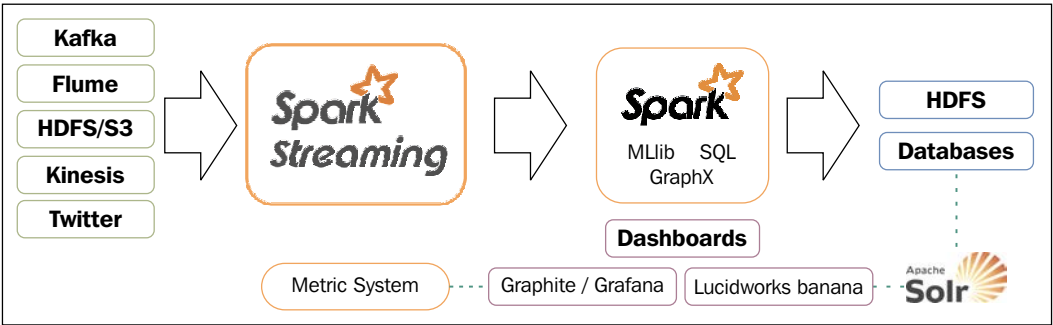
The Apache Streaming module is a stream processing-based module within Apache Spark. It uses the Spark cluster to offer the ability to scale to a high degree. Being based on Spark, it is also highly fault tolerant, having the ability to rerun failed tasks by checkpointing the data stream that is being processed. The following areas will be covered in this chapter after an initial section, which will provide a practical overview of how Apache Spark processes stream-based data:

- Error recovery and checkpointing
- TCP-based Stream Processing
- File Streams
- Flume Stream source
- Kafka Stream source

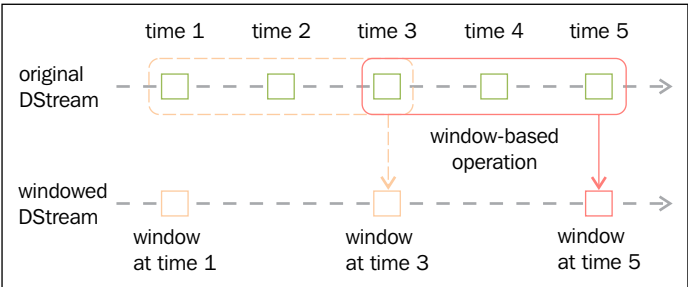
For each topic, I will provide a worked example in Scala, and will show how the stream-based architecture can be set up and tested.

Overview

When giving an overview of the Apache Spark streaming module, I would advise you to check the <http://spark.apache.org/> website for up-to-date information, as well as the Spark-based user groups such as user@spark.apache.org. My reason for saying this is because these are the primary places where Spark information is available. Also the extremely fast (and increasing) pace of change means that by the time you read this new Spark functionality and versions, will be available. So, in the light of this, when giving an overview, I will try to generalize.



The previous figure shows potential data sources for Apache Streaming, such as **Kafka**, **Flume**, and **HDFS**. These feed into the Spark Streaming module, and are processed as discrete streams. The diagram also shows that other Spark module functionality, such as machine learning, can be used to process the stream-based data. The fully processed data can then be an output for **HDFS**, **databases**, or **dashboards**. This diagram is based on the one at the Spark streaming website, but I wanted to extend it for both—expressing the Spark module functionality, and for dashboarding options. The previous diagram shows a MetricSystems feed being fed from Spark to Graphite. Also, it is possible to feed Solr-based data to Lucidworks banana (a port of kabana). It is also worth mentioning here that Databricks (see *Chapter 8, Spark Databricks* and *Chapter 9, Databricks Visualization*) can also present the Spark stream data as a dashboard.



When discussing Spark discrete streams, the previous figure, again taken from the Spark website at <http://spark.apache.org/>, is the diagram I like to use. The green boxes in the previous figure show the continuous data stream sent to Spark, being broken down into a **discrete stream (DStream)**. The size of each element in the stream is then based on a batch time, which might be two seconds. It is also possible to create a window, expressed as the previous red box, over the DStream. For instance, when carrying out trend analysis in real time, it might be necessary to determine the top ten Twitter-based Hashtags over a ten minute window.

So, given that Spark can be used for Stream processing, how is a Stream created? The following Scala-based code shows how a Twitter stream can be created. This example is simplified because Twitter authorization has not been included, but you get the idea (the full example code is in the *Checkpointing* section). The Spark stream context, called `ssc`, is created using the spark context `sc`. A batch time is specified when it is created; in this case, five seconds. A Twitter-based DStream, called `stream`, is then created from the `StreamingContext` using a window of 60 seconds:

```
val ssc    = new StreamingContext(sc, Seconds(5) )
val stream = TwitterUtils.createStream(ssc,None).window( Seconds(60) )
```

The stream processing can be started with the stream context `start` method (shown next), and the `awaitTermination` method indicates that it should process until stopped. So, if this code is embedded in a library-based application, it will run until the session is terminated, perhaps with a `Ctrl + C`:

```
ssc.start()
ssc.awaitTermination()
```

This explains what Spark streaming is, and what it does, but it does not explain error handling, or what to do if your stream-based application fails. The next section will examine Spark streaming error management and recovery.

Errors and recovery

Generally, the question that needs to be asked for your application is; is it critical that you receive and process all the data? If not, then on failure you might just be able to restart the application and discard the missing or lost data. If this is not the case, then you will need to use checkpointing, which will be described in the next section.

It is also worth noting that your application's error management should be robust and self-sufficient. What I mean by this is that; if an exception is non-critical, then manage the exception, perhaps log it, and continue processing. For instance, when a task reaches the maximum number of failures (specified by `spark.task.maxFailures`), it will terminate processing.

Checkpointing

It is possible to set up an HDFS-based checkpoint directory to store Apache Spark-based streaming information. In this Scala example, data will be stored in HDFS, under `/data/spark/checkpoint`. The following HDFS file system `ls` command shows that before starting, the directory does not exist:

```
[hadoop@hc2nn stream]$ hdfs dfs -ls /data/spark/checkpoint
ls: `/data/spark/checkpoint': No such file or directory
```

The Twitter-based Scala code sample given next, starts by defining a package name for the application, and by importing Spark, streaming, context, and Twitter-based functionality. It then defines an application object named `stream1`:

```
package nz.co.semtechsolutions

import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.streaming._
import org.apache.spark.streaming.twitter._
import org.apache.spark.streaming.StreamingContext._

object stream1 {
```

Next, a method is defined called `createContext`, which will be used to create both the spark, and streaming contexts. It will also checkpoint the stream to the HDFS-based directory using the streaming context checkpoint method, which takes a directory path as a parameter. The directory path being the value (`cpDir`) that was passed into the `createContext` method:

```
def createContext( cpDir : String ) : StreamingContext = {

    val appName = "Stream example 1"
    val conf     = new SparkConf()

    conf.setAppName( appName )

    val sc = new SparkContext( conf )

    val ssc = new StreamingContext( sc, Seconds( 5 ) )

    ssc.checkpoint( cpDir )
```

```

        ssc
    }

```

Now, the main method is defined, as is the HDFS directory, as well as Twitter access authority and parameters. The Spark streaming context `ssc` is either retrieved or created using the HDFS checkpoint directory via the `StreamingContext` method — `getOrCreate`. If the directory doesn't exist, then the previous method called `createContext` is called, which will create the context and checkpoint. Obviously, I have truncated my own Twitter auth. keys in this example for security reasons:

```

def main(args: Array[String]) {

    val hdfsDir = "/data/spark/checkpoint"

    val consumerKey      = "QQpxx"
    val consumerSecret   = "0HFzxx"
    val accessToken      = "323xx"
    val accessTokenSecret = "IlQxx"

    System.setProperty("twitter4j.oauth.consumerKey", consumerKey)
    System.setProperty("twitter4j.oauth.consumerSecret", consumerSecret)
    System.setProperty("twitter4j.oauth.accessToken", accessToken)
    System.setProperty("twitter4j.oauth.accessTokenSecret",
        accessTokenSecret)

    val ssc = StreamingContext.getOrCreate(hdfsDir,
        () => { createContext( hdfsDir ) })

    val stream = TwitterUtils.createStream(ssc,None).window(
        Seconds(60) )

    // do some processing

    ssc.start()
    ssc.awaitTermination()

} // end main

```

Having run this code, which has no actual processing, the HDFS checkpoint directory can be checked again. This time it is apparent that the checkpoint directory has been created, and the data has been stored:

```
[hadoop@hc2nn stream]$ hdfs dfs -ls /data/spark/checkpoint
Found 1 items
drwxr-xr-x  - hadoop supergroup          0 2015-07-02 13:41
/data/spark/checkpoint/0fc3d94e-6f53-40fb-910d-1eef044b12e9
```

This example, taken from the Apache Spark website, shows how checkpoint storage can be set up and used. But how often is checkpointing carried out? The Meta data is stored during each stream batch. The actual data is stored with a period, which is the maximum of the batch interval, or ten seconds. This might not be ideal for you, so you can reset the value using the method:

```
DStream.checkpoint( newRequiredInterval )
```

Where `newRequiredInterval` is the new checkpoint interval value that you require, generally you should aim for a value which is five to ten times your batch interval.

Checkpointing saves both the stream batch and metadata (data about the data). If the application fails, then when it restarts, the checkpointed data is used when processing is started. The batch data that was being processed at the time of failure is reprocessed, along with the batched data since the failure.

Remember to monitor the HDFS disk space being used for check pointing. In the next section, I will begin to examine the streaming sources, and will provide some examples of each type.

Streaming sources

I will not be able to cover all the stream types with practical examples in this section, but where this chapter is too small to include code, I will at least provide a description. In this chapter, I will cover the TCP and file streams, and the Flume, Kafka, and Twitter streams. I will start with a practical TCP-based example.

This chapter examines stream processing architecture. For instance, what happens in cases where the stream data delivery rate exceeds the potential data processing rate? Systems like Kafka provide the possibility of solving this issue by providing the ability to use multiple data topics and consumers.

TCP stream

There is a possibility of using the Spark streaming context method called `socketTextStream` to stream data via TCP/IP, by specifying a hostname and a port number. The Scala-based code example in this section will receive data on port 10777 that was supplied using the `netcat` Linux command. The code sample starts by defining the package name, and importing Spark, the context, and the streaming classes. The object class named `stream2` is defined, as it is the main method with arguments:

```
package nz.co.semtechsolutions

import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._

object stream2 {

    def main(args: Array[String]) {
```

The number of arguments passed to the class is checked to ensure that it is the hostname and the port number. A Spark configuration object is created with an application name defined. The Spark and streaming contexts are then created. Then, a streaming batch time of 10 seconds is set:

```
    if ( args.length < 2 )
    {
        System.err.println("Usage: stream2 <host> <port>")
        System.exit(1)
    }

    val hostname = args(0).trim
    val portnum  = args(1).toInt

    val appName = "Stream example 2"
    val conf    = new SparkConf()

    conf.setAppName(appName)

    val sc = new SparkContext(conf)
    val ssc = new StreamingContext(sc, Seconds(10) )
```

A DStream called `rawDstream` is created by calling the `socketTextStream` method of the streaming context using the host and port name parameters.

```
val rawDstream = ssc.socketTextStream( hostname, portnum )
```

A top-ten word count is created from the raw stream data by splitting words by spacing. Then a (key,value) pair is created as `(word,1)`, which is reduced by the key value, this being the word. So now, there is a list of words and their associated counts. Now, the key and value are swapped, so the list becomes `(count and word)`. Then, a sort is done on the key, which is now the count. Finally, the top 10 items in the rdd, within the DStream, are taken and printed out:

```
val wordCount = rawDstream
    .flatMap(line => line.split(" "))
    .map(word => (word,1))
    .reduceByKey(_+_ )
    .map(item => item.swap)
    .transform(rdd => rdd.sortByKey(false))
    .foreachRDD( rdd =>
        { rdd.take(10).foreach(x=>println("List : " + x))
    })
```

The code closes with the Spark Streaming `start`, and `awaitTermination` methods being called to start the stream processing and await process termination:

```
ssc.start()
ssc.awaitTermination()

} // end main

} // end stream2
```

The data for this application is provided, as I stated previously, by the Linux `netcat` (`nc`) command. The Linux `cat` command dumps the contents of a log file, which is piped to `nc`. The `lk` options force `netcat` to listen for connections, and keep on listening if the connection is lost. This example shows that the port being used is 10777:

```
[root@hc2nn log]# pwd
/var/log
[root@hc2nn log]# cat ./anaconda.storage.log | nc -lk 10777
```

The output from this TCP-based stream processing is shown here. The actual output is not as important as the method demonstrated. However, the data shows, as expected, a list of 10 log file words in descending count order. Note that the top word is empty because the stream was not filtered for empty words:

```
List : (17104,)
List : (2333,=)
List : (1656,:)
List : (1603,;)
List : (1557,DEBUG)
List : (564,True)
List : (495,False)
List : (411,None)
List : (356,at)
List : (335,object)
```

This is interesting if you want to stream data using Apache Spark streaming, based upon TCP/IP from a host and port. But what about more exotic methods? What if you wish to stream data from a messaging system, or via memory-based channels? What if you want to use some of the big data tools available today like Flume and Kafka? The next sections will examine these options, but first I will demonstrate how streams can be based upon files.

File streams

I have modified the Scala-based code example in the last section, to monitor an HDFS-based directory, by calling the Spark streaming context method called `textFileStream`. I will not display all of the code, given this small change. The application class is now called `stream3`, which takes a single parameter—the HDFS directory. The directory path could be on NFS or AWS S3 (all the code samples will be available with this book):

```
val rawDstream = ssc.textFileStream( directory )
```

The stream processing is the same as before. The stream is split into words, and the top-ten word list is printed. The only difference this time is that the data must be put into the HDFS directory while the application is running. This is achieved with the HDFS file system `put` command here:

```
[root@hc2nn log]# hdfs dfs -put ./anaconda.storage.log /data/spark/stream
```

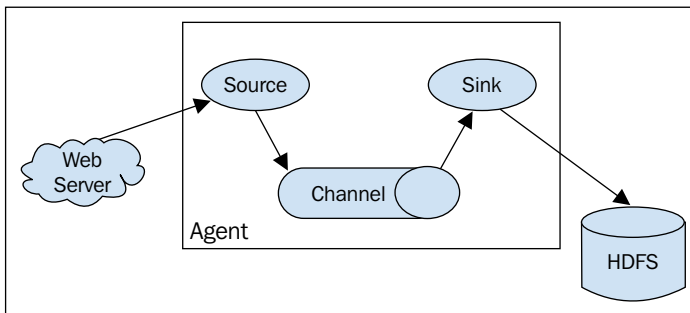
As you can see, the HDFS directory used is `/data/spark/stream/`, and the text-based source log file is `anaconda.storage.log` (under `/var/log/`). As expected, the same word list and count is printed:

```
List : (17104,)
List : (2333,=)
.....
List : (564,True)
List : (495,False)
List : (411,None)
List : (356,at)
List : (335,object)
```

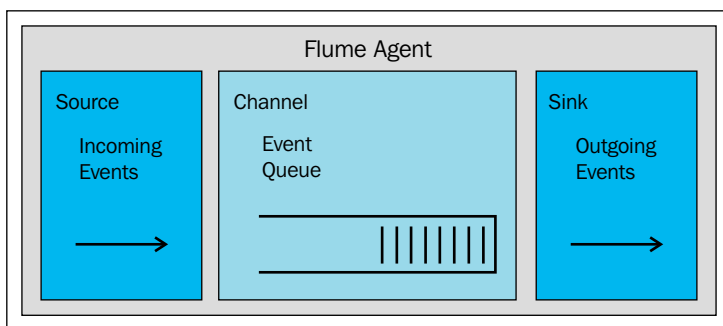
These are simple streaming methods based on TCP, and file system data. But what if I want to use some of the built-in streaming functionality within Spark streaming? This will be examined next. The Spark streaming Flume library will be used as an example.

Flume

Flume is an Apache open source project and product, which is designed to move large amounts of data at a big data scale. It is highly scalable, distributed, and reliable, working on the basis of data source, data sink, and data channels, as the diagram here, taken from the <http://flume.apache.org/> website, shows:



Flume uses agents to process data streams. As can be seen in the previous figure, an agent has a data source, a data processing channel, and a data sink. A clearer way to describe this is via the following figure. The channel acts as a queue for the sourced data and the sink passes the data to the next link in the chain.



Flume agents can form Flume architectures; the output of one agent's sink can be the input to a second agent. Apache Spark allows two approaches to using Apache Flume. The first is an Avro push-based in-memory approach, whereas the second one, still based on Avro, is a pull-based system, using a custom Spark sink library.

I installed Flume via the Cloudera CDH 5.3 cluster manager, which installs a single agent. Checking the Linux command line, I can see that Flume version 1.5 is now available:

```
[root@hc2nn ~]# flume-ng version
```

```
Flume 1.5.0-cdh5.3.3
```

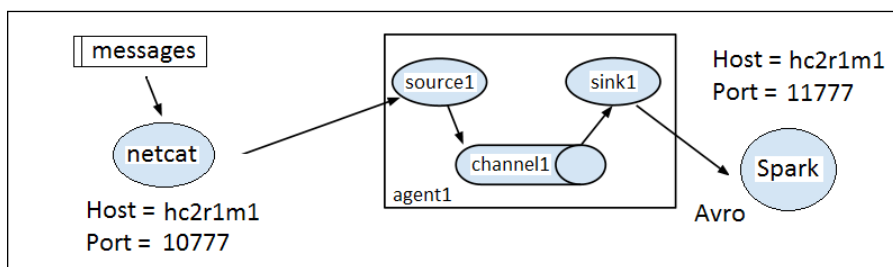
```
Source code repository: https://git-wip-us.apache.org/repos/asf/flume.git
```

```
Revision: b88celfd016bc873d817343779dfff6aeea07706
```

```
Compiled by jenkins on Wed Apr 8 14:57:43 PDT 2015
```

```
From source with checksum 389d91c718e03341a2367bf4ef12428e
```

The Flume-based Spark example that I will initially implement here, is the Flume-based push approach, where Spark acts as a receiver, and Flume pushes the data to Spark. The following figure represents the structure that I will implement on a single node:



The message data will be sent to port 10777 on a host called `hc2r1m1` using the Linux `netcat (nc)` command. This will act as a source (`source1`) for the Flume agent (`agent1`), which will have an in-memory channel called `channel1`. The sink used by `agent1` will be Apache Avro based, again on a host called `hc2r1m1`, but this time, the port number will be 11777. The Apache Spark Flume application `stream4` (which I will describe shortly) will listen for Flume stream data on this port.

I start the streaming process by executing the `netcat (nc)` command next, against the 10777 port. Now, when I type text into this window, it will be used as a Flume source, and the data will be sent to the Spark application:

```
[hadoop@hc2nn ~]$ nc hc2r1m1.semtech-solutions.co.nz 10777
```

In order to run my Flume agent, `agent1`, I have created a Flume configuration file called `agent1.flume.cfg`, which describes the agent's source, channel, and sink. The contents of the file are as follows. The first section defines the `agent1` source, channel, and sink names.

```
agent1.sources = source1
agent1.channels = channel1
agent1.sinks = sink1
```

The next section defines `source1` to be netcat based, running on the host called `hc2r1m1`, and 10777 port:

```
agent1.sources.source1.channels=channel1
agent1.sources.source1.type=netcat
agent1.sources.source1.bind=hc2r1m1.semtech-solutions.co.nz
agent1.sources.source1.port=10777
```

The `agent1` channel, `channel1`, is defined as a memory-based channel with a maximum event capacity of 1000 events:

```
agent1.channels.channel1.type=memory
agent1.channels.channel1.capacity=1000
```

Finally, the `agent1` sink, `sink1`, is defined as an Apache Avro sink on the host called `hc2r1m1`, and 11777 port:

```
agent1.sinks.sink1.type=avro
agent1.sinks.sink1.hostname=hc2r1m1.semtech-solutions.co.nz
agent1.sinks.sink1.port=11777
agent1.sinks.sink1.channel=channel1
```

I have created a Bash script called `flume.bash` to run the Flume agent, `agent1`. It looks like this:

```
[hadoop@hc2r1m1 stream]$ more flume.bash

#!/bin/bash

# run the bash agent

flume-ng agent \
  --conf /etc/flume-ng/conf \
  --conf-file ./agent1.flume.cfg \
  -Dflume.root.logger=DEBUG,INFO,console \
  -name agent1
```

The script calls the Flume executable `flume-ng`, passing the `agent1` configuration file. The call specifies the agent named `agent1`. It also specifies the Flume configuration directory to be `/etc/flume-ng/conf/`, the default value. Initially, I will use a `netcat` Flume source with a Scala-based example to show how data can be sent to an Apache Spark application. Then, I will show how an `RSS`-based data feed can be processed in a similar way. So initially, the Scala code that will receive the `netcat` data looks like this. The class package name and the application class name are defined. The necessary classes for Spark and Flume are imported. Finally, the main method is defined:

```
package nz.co.semtechsolutions

import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.flume._

object stream4 {

  def main(args: Array[String]) {
```

The host and port name arguments for the data stream are checked and extracted:

```
if ( args.length < 2 )
{
    System.err.println("Usage: stream4 <host> <port>")
    System.exit(1)
}

val hostname = args(0).trim
val portnum  = args(1).toInt

println("hostname : " + hostname)
println("portnum  : " + portnum)
```

The Spark and streaming contexts are created. Then, the Flume-based data stream is created using the stream context host and port number. The Flume-based class `FlumeUtils` has been used to do this by calling its `createStream` method:

```
val appName = "Stream example 4"
val conf    = new SparkConf()

conf.setAppName(appName)

val sc = new SparkContext(conf)
val ssc = new StreamingContext(sc, Seconds(10) )

val rawDstream = FlumeUtils.createStream(ssc,hostname,portnum)
```

Finally, a stream event count is printed, and (for debug purposes while we test the stream) the stream content is dumped. After this, the stream context is started and configured to run until terminated via the application:

```
rawDstream.count()
    .map(cnt => ">>>> Received events : " + cnt )
    .print()

rawDstream.map(e => new String(e.event.getBody.array() ))
    .print

ssc.start()
```

```
ssc.awaitTermination()
```

```
} // end main
```

```
} // end stream4
```

Having compiled it, I will run this application using `spark-submit`. In the other chapters of this book, I will use a Bash-based script called `run_stream.bash` to execute the job. The script looks like this:

```
[hadoop@hc2r1m1 stream]$ more run_stream.bash
```

```
#!/bin/bash
```

```
SPARK_HOME=/usr/local/spark
```

```
SPARK_BIN=$SPARK_HOME/bin
```

```
SPARK_SBIN=$SPARK_HOME/sbin
```

```
JAR_PATH=/home/hadoop/spark/stream/target/scala-2.10/streaming_2.10-1.0.jar
```

```
CLASS_VAL=$1
```

```
CLASS_PARAMS="${*:2}"
```

```
STREAM_JAR=/usr/local/spark/lib/spark-examples-1.3.1-hadoop2.3.0.jar
```

```
cd $SPARK_BIN
```

```
./spark-submit \
```

```
--class $CLASS_VAL \
```

```
--master spark://hc2nn.semtech-solutions.co.nz:7077 \
```

```
--executor-memory 100M \
```

```
--total-executor-cores 50 \
```

```
--jars $STREAM_JAR \
```

```
$JAR_PATH \
```

```
$CLASS_PARAMS
```

So, this script sets some Spark-based variables, and a JAR library path for this job. It takes which Spark class to run, as its first parameter. It passes all the other variables, as parameters, to the Spark application class job. So, the execution of the application looks like this:

```
[hadoop@hc2r1m1 stream]$ ./run_stream.bash \
    nz.co.semtechsolutions.stream4 \
    hc2r1m1.semtech-solutions.co.nz \
    11777
```

This means that the Spark application is ready, and is running as a Flume sink on port 11777. The Flume input is ready, running as a netcat task on port 10777. Now, the Flume agent, `agent1`, can be started using the Flume script called `flume.bash` to send the netcat source-based data to the Apache Spark Flume-based sink:

```
[hadoop@hc2r1m1 stream]$ ./flume.bash
```

Now, when the text is passed to the netcat session, it should flow through Flume, and be processed as a stream by Spark. Let's try it:

```
[hadoop@hc2nn ~]$ nc hc2r1m1.semtech-solutions.co.nz 10777
```

```
I hope that Apache Spark will print this
```

```
OK
```

```
I hope that Apache Spark will print this
```

```
OK
```

```
I hope that Apache Spark will print this
```

```
OK
```

Three simple pieces of text have been added to the netcat session, and have been acknowledged with an `OK`, so that they can be passed to Flume. The debug output in the Flume session shows that the events (one per line) have been received and processed:

```
2015-07-06 18:13:18,699 (netcat-handler-0) [DEBUG - org.apache.flume.
source.NetcatSource$NetcatSocketHandler.run(NetcatSource.java:318)] Chars
read = 41
```

```
2015-07-06 18:13:18,700 (netcat-handler-0) [DEBUG - org.apache.flume.
source.NetcatSource$NetcatSocketHandler.run(NetcatSource.java:322)]
Events processed = 1
```

```
2015-07-06 18:13:18,990 (netcat-handler-0) [DEBUG - org.apache.flume.
source.NetcatSource$NetcatSocketHandler.run(NetcatSource.java:318)] Chars
read = 41
```

```

2015-07-06 18:13:18,991 (netcat-handler-0) [DEBUG - org.apache.flume.
source.NetcatSource$NetcatSocketHandler.run(NetcatSource.java:322)]
Events processed = 1
2015-07-06 18:13:19,270 (netcat-handler-0) [DEBUG - org.apache.flume.
source.NetcatSource$NetcatSocketHandler.run(NetcatSource.java:318)] Chars
read = 41
2015-07-06 18:13:19,271 (netcat-handler-0) [DEBUG - org.apache.flume.
source.NetcatSource$NetcatSocketHandler.run(NetcatSource.java:322)]
Events processed = 1

```

Finally, in the Spark `stream4` application session, three events have been received and processed. In this case, dumped to the session to prove the point that the data arrived. Of course, this is not what you would normally do, but I wanted to prove data transit through this configuration:

```

-----
Time: 1436163210000 ms
-----
>>> Received events : 3
-----
Time: 1436163210000 ms
-----
I hope that Apache Spark will print this
I hope that Apache Spark will print this
I hope that Apache Spark will print this

```

This is interesting, but it is not really a production-worthy example of Spark Flume data processing. So, in order to demonstrate a potentially real data processing approach, I will change the Flume configuration file source details so that it uses a Perl script, which is executable as follows:

```

agent1.sources.source1.type=exec
agent1.sources.source1.command=./rss.perl

```

The Perl script, which is referenced previously, `rss.perl`, just acts as a source of Reuters science news. It receives the news as XML, and converts it into JSON format. It also cleans the data of unwanted noise. First, it imports packages like LWP and XML::XPath to enable XML processing. Then, it specifies a science-based Reuters news data source, and creates a new LWP agent to process the data, similar to this:

```
#!/usr/bin/perl
```

```
use strict;
```

```
use LWP::UserAgent;
use XML::XPath;

my $urlsource="http://feeds.reuters.com/reuters/scienceNews" ;

my $agent = LWP::UserAgent->new;
```

Then an infinite while loop is opened, and an HTTP GET request is carried out against the URL. The request is configured, and the agent makes the request via a call to the request method:

```
while()
{
    my $req = HTTP::Request->new(GET => ($urlsource));

    $req->header('content-type' => 'application/json');
    $req->header('Accept'      => 'application/json');

    my $resp = $agent->request($req);
```

If the request is successful, then the XML data returned, is defined as the decoded content of the request. Title information is extracted from the XML, via an XPath call using the path called `/rss/channel/item/title`:

```
if ( $resp->is_success )
{
    my $xmlpage = $resp -> decoded_content;

    my $xp = XML::XPath->new( xml => $xmlpage );
    my $nodeset = $xp->find( '/rss/channel/item/title' );

    my @titles = ( ) ;
    my $index = 0 ;
```

For each node in the extracted title data title XML string, data is extracted. It is cleaned of unwanted XML tags, and added to a Perl-based array called `titles`:

```
foreach my $node ($nodeset->get_nodelist)
{
    my $xmlstring = XML::XPath::XMLParser::as_string($node) ;

    $xmlstring =~ s/<title>//g;
```

```

$xmlstring =~ s/<\\title>\\/g;
$xmlstring =~ s/"\\/g;
$xmlstring =~ s/,\\/g;

$titles[$index] = $xmlstring ;
$index = $index + 1 ;

} # foreach find node

```

The same process is carried out for description-based data in the request response XML. The XPath value used this time is `/rss/channel/item/description/`. There are many more tags to be cleaned from the description data, so there are many more Perl searches, and line replacements that act on this data (`s///g`):

```

my $nodeset = $xp->find( '/rss/channel/item/description' );

my @desc = ( ) ;
$index = 0 ;

foreach my $node ($nodeset->get_nodelist)
{
    my $xmlstring = XML::XPath::XMLParser::as_string($node) ;

    $xmlstring =~ s/<img.+\\img>\\/g;
    $xmlstring =~ s/href="."\\/g;
    $xmlstring =~ s/src="."\\/g;
    $xmlstring =~ s/src='.'\\/g;
    $xmlstring =~ s/<br.+\\>\\/g;
    $xmlstring =~ s/<\\div>\\/g;
    $xmlstring =~ s/<\\a>\\/g;
    $xmlstring =~ s/<a >\\n\\/g;
    $xmlstring =~ s/<img >\\/g;
    $xmlstring =~ s/<img \\>\\/g;
    $xmlstring =~ s/<div.+>\\/g;
    $xmlstring =~ s/<title>\\/g;
    $xmlstring =~ s/<\\title>\\/g;
    $xmlstring =~ s/<description>\\/g;
    $xmlstring =~ s/<\\description>\\/g;

```

```

$xmlstring =~ s/&lt;.+>//g;
$xmlstring =~ s/"//g;
$xmlstring =~ s/,//g;
$xmlstring =~ s/\r|\n//g;

$desc[$index] = $xmlstring ;
$index = $index + 1 ;

} # foreach find node

```

Finally, the XML-based title and description data is output in the RSS JSON format using a `print` command. The script then sleeps for 30 seconds, and requests more RSS news information to process:

```

my $newsitems = $index ;
$index = 0 ;

for ($index=0; $index < $newsitems; $index++) {

    print "{\"category\": \"science\", \"
        . \" \"title\": \"\" . $titles[$index] . "\", \"
        . \" \"summary\": \"\" . $desc[$index] . \"\"
        . \"}\n\";

} # for rss items

} # success ?

sleep(30) ;

} # while

```

I have created a second Scala-based stream processing code example called `stream5`. It is similar to the `stream4` example, but it now processes the `rss` item data from the stream. A case class is defined next to process the category, title, and summary from the XML `rss` information. An HTML location is defined to store the resulting data that comes from the Flume channel:

```

case class RSSItem(category : String, title : String, summary :
String)

```

```
val now: Long = System.currentTimeMillis
```

```
val hdfsdir = "hdfs://hc2nn:8020/data/spark/flume/rss/"
```

The `rss` stream data from the Flume-based event is converted into a string. It is then formatted using the case class called `RSSItem`. If there is event data, it is then written to an HDFS directory using the previous `hdfsdir` path:

```
rawDstream.map(record => {
  implicit val formats = DefaultFormats
  read[RSSItem](new String(record.event.getBody().array()))
})

.foreachRDD(rdd => {
  if (rdd.count() > 0) {
    rdd.map(item => {
      implicit val formats = DefaultFormats
      write(item)
    }).saveAsTextFile(hdfsdir+"file_"+now.toString())
  }
})
```

Running this code sample, it is possible to see that the Perl `rss` script is producing data, because the Flume script output indicates that 80 events have been accepted and received:

```
2015-07-07 14:14:24,017 (agent-shutdown-hook) [DEBUG - org.apache.
flume.source.ExecSource.stop(ExecSource.java:219)] Exec source with
command:./news_rss_collector.py stopped. Metrics:SOURCE:source1{src.
events.accepted=80, src.events.received=80, src.append.accepted=0, src.
append-batch.accepted=0, src.open-connection.count=0, src.append-batch.
received=0, src.append.received=0}
```

The Scala Spark application `stream5` has processed 80 events in two batches:

```
>>>> Received events : 73
>>>> Received events : 7
```

And the events have been stored on HDFS, under the expected directory, as the Hadoop file system `ls` command shows here:

```
[hadoop@hc2r1m1 stream]$ hdfs dfs -ls /data/spark/flume/rss/
Found 2 items
```

```
drwxr-xr-x - hadoop supergroup          0 2015-07-07 14:09 /data/spark/
flume/rss/file_1436234439794
drwxr-xr-x - hadoop supergroup          0 2015-07-07 14:14 /data/spark/
flume/rss/file_1436235208370
```

Also, using the Hadoop file system `cat` command, it is possible to prove that the files on HDFS contain RSS feed news-based data as shown here:

```
[hadoop@hc2rlm1 stream]$ hdfs dfs -cat /data/spark/flume/rss/
file_1436235208370/part-00000 | head -1
```

```
{"category":"healthcare","title":"BRIEF-Aetna CEO says has not had
specific conversations with DOJ on Humana - CNBC","summary":"* Aetna CEO
Says Has Not Had Specific Conversations With Doj About Humana Acquisition
- CNBC"}
```

This Spark stream-based example has used Apache Flume to transmit data from an RSS source, through Flume, to HDFS via a Spark consumer. This is a good example, but what if you want to publish data to a group of consumers? In the next section, I will examine Apache Kafka—a publish subscribe messaging system, and determine how it can be used with Spark.

Kafka

Apache Kafka (<http://kafka.apache.org/>) is a top level open-source project in Apache. It is a big data publish/subscribe messaging system that is fast and highly scalable. It uses message brokers for data management, and ZooKeeper for configuration, so that data can be organized into consumer groups and topics. Data in Kafka is split into partitions. In this example, I will demonstrate a receiver-less Spark-based Kafka consumer, so that I don't need to worry about configuring Spark data partitions when compared to my Kafka data.

In order to demonstrate Kafka-based message production and consumption, I will use the Perl RSS script from the last section as a data source. The data passing into Kafka and onto Spark will be Reuters RSS news data in the JSON format.

As topic messages are created by message producers, they are then placed in partitions in message order sequence. The messages in the partitions are retained for a configurable time period. Kafka then stores the offset value for each consumer, which is that consumer's position (in terms of message consumption) in that partition.

I am currently using Cloudera's CDH 5.3 Hadoop cluster. In order to install Kafka, I need to download a Kafka JAR library file from: <http://archive.cloudera.com/csdh/kafka/>.

Having downloaded the file, and given that I am using CDH cluster manager, I then need to copy the file to the `/opt/cloudera/csd/` directory on my NameNode CentOS server, so that it will be visible to install:

```
[root@hc2nn csd]# pwd
```

```
/opt/cloudera/csd
```

```
[root@hc2nn csd]# ls -l KAFKA-1.2.0.jar
```

```
-rw-r--r-- 1 hadoop hadoop 5670 Jul 11 14:56 KAFKA-1.2.0.jar
```

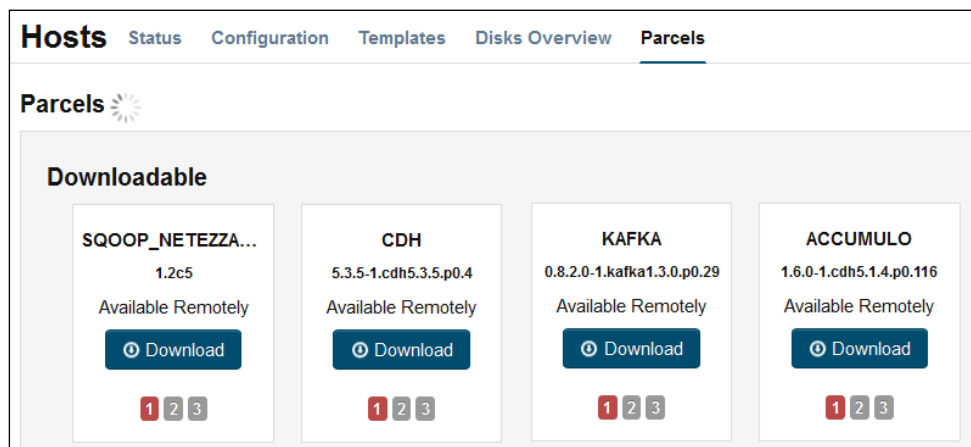
I then need to restart the Cloudera cluster manager server on my NameNode, or master server, so that the change will be recognized. This was done as root using the service command, which is as follows:

```
[root@hc2nn hadoop]# service cloudera-scm-server restart
```

```
Stopping cloudera-scm-server: [ OK ]
```

```
Starting cloudera-scm-server: [ OK ]
```

Now, the Kafka parcel should be visible within the CDH manager under **Hosts** | **Parcels**, as shown in the following figure. You can follow the usual download, distribution, and activate cycle for the CDH parcel installation:



I have installed Kafka message brokers on each Data Node, or Spark Slave machine in my cluster. I then set the Kafka broker ID values for each Kafka broker server, giving them a `broker.id` number of 1 through 4. As Kafka uses ZooKeeper for cluster data configuration, I wanted to keep all the Kafka data in a top level node called `kafka` in ZooKeeper. In order to do this, I set the Kafka ZooKeeper root value, called `zookeeper.chroot`, to `/kafka`. After making these changes, I restarted the CDH Kafka servers for the changes to take effect.

With Kafka installed, I can check the scripts available for testing. The following listing shows Kafka-based scripts for message producers and consumers, as well as scripts for managing topics, and checking consumer offsets. These scripts will be used in this section in order to demonstrate Kafka functionality:

```
[hadoop@hc2nn ~]$ ls /usr/bin/kafka*
```

```
/usr/bin/kafka-console-consumer      /usr/bin/kafka-run-class
/usr/bin/kafka-console-producer      /usr/bin/kafka-topics
/usr/bin/kafka-consumer-offset-checker
```

In order to run the installed Kafka servers, I need to have the broker server ID's (broker.id) values set, else an error will occur. Once Kafka is installed and running, I will need to prepare a message producer script. The simple Bash script given next, called `kafka.bash`, defines a comma-separated broker list of hosts and ports. It also defines a topic called `rss`. It then calls the Perl script `rss.perl` to generate the RSS-based data. This data is then piped into the Kafka producer script called `kafka-console-producer` to be sent to Kafka.

```
[hadoop@hc2r1m1 stream]$ more kafka.bash
```

```
#!/bin/bash
```

```
BROKER_LIST="hc2r1m1:9092,hc2r1m2:9092,hc2r1m3:9092,hc2r1m4:9092"
```

```
TOPIC="rss"
```

```
./rss.perl | /usr/bin/kafka-console-producer --broker-list $BROKER_LIST
--topic $TOPIC
```

Notice that I have not mentioned Kafka topics at this point. When a topic is created in Kafka, the number of partitions can be specified. In the following example, the `kafka-topics` script has been called with the `create` option. The number of partitions have been set to 5, and the data replication factor has been set to 3. The ZooKeeper server string has been defined as `hc2r1m2-4` with a port number of 2181. Also note that the top level ZooKeeper Kafka node has been defined as `/kafka` in the ZooKeeper string:

```
/usr/bin/kafka-topics \
  --create \
  --zookeeper hc2r1m2:2181,hc2r1m3:2181,hc2r1m4:2181/kafka \
  --replication-factor 3 \
  --partitions 5 \
  --topic rss
```

I have also created a Bash script called `kafka_list.bash` for use during testing, which checks all the Kafka topics that have been created, and also the Kafka consumer offsets. It calls the `kafka-topics` commands with a `list` option, and a ZooKeeper string to get a list of created topics. It then calls the Kafka script called `kafka-consumer-offset-checker` with a ZooKeeper string—the topic name and a group name to get a list of consumer offset values. Using this script, I can check that my topics are created, and the topic data is being consumed correctly:

```
[hadoop@hc2r1m1 stream]$ cat kafka_list.bash
```

```
#!/bin/bash

ZOOKEEPER="hc2r1m2:2181,hc2r1m3:2181,hc2r1m4:2181/kafka"
TOPIC="rss"
GROUP="group1"

echo ""
echo "===== "
echo " Kafka Topics "
echo "===== "

/usr/bin/kafka-topics --list --zookeeper $ZOOKEEPER

echo ""
echo "===== "
echo " Kafka Offsets "
echo "===== "

/usr/bin/kafka-consumer-offset-checker \
  --group $GROUP \
  --topic $TOPIC \
  --zookeeper $ZOOKEEPER
```

Next, I need to create the Apache Spark Scala-based Kafka consumer code. As I said, I will create a receiver-less example, so that the Kafka data partitions match in both, Kafka and Spark. The example is called `stream6`. First, the package is defined, and the classes are imported for Kafka, spark, context, and streaming. Then, the object class called `stream6`, and the main method are defined. The code looks like this:

```
package nz.co.semtechsolutions

import kafka.serializer.StringDecoder

import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.kafka._

object stream6 {

    def main(args: Array[String]) {
```

Next, the class parameters (broker's string, group ID, and topic) are checked and processed. If the class parameters are incorrect, then an error is printed, and execution stops, else the parameter variables are defined:

```
    if ( args.length < 3 )
    {
        System.err.println("Usage: stream6 <brokers> <groupid> <topics>\n")
        System.err.println("<brokers> = host1:port1,host2:port2\n")
        System.err.println("<groupid> = group1\n")
        System.err.println("<topics> = topic1,topic2\n")
        System.exit(1)
    }

    val brokers = args(0).trim
    val groupid = args(1).trim
    val topics  = args(2).trim

    println("brokers : " + brokers)
    println("groupid : " + groupid)
    println("topics  : " + topics)
```



```
        if (rdd.count() > 0) {  
            rdd.saveAsTextFile(hdfsdir+"file_"+now.toString())  
        }  
    })
```

Finally, the Scala script closes by starting the stream processing, and setting the application class to run until terminated with `awaitTermination`:

```
    ssc.start()  
    ssc.awaitTermination()  
  
} // end main  
  
} // end stream6
```

With all of the scripts explained and the Kafka CDH brokers running, it is time to examine the Kafka configuration, which if you remember is maintained by Apache ZooKeeper (all of the code samples that have been described so far will be released with the book). I will use the `zookeeper-client` tool, and connect to the zookeeper server on the host called `hc2r1m2` on the 2181 port. As you can see here, I have received a connected message from the client session:

```
[hadoop@hc2r1m1 stream]$ /usr/bin/zookeeper-client -server hc2r1m2:2181
```

```
[zk: hc2r1m2:2181(CONNECTED) 0]
```

If you remember, I specified the top level ZooKeeper directory for Kafka to be `/kafka`. If I examine this now via a client session, I can see the Kafka ZooKeeper structure. I will be interested in `brokers` (the CDH Kafka broker servers), and `consumers` (the previous Spark Scala code). The ZooKeeper `ls` commands show that the four Kafka servers have registered with ZooKeeper, and are listed by their `broker.id` configuration values one to four:

```
[zk: hc2r1m2:2181(CONNECTED) 2] ls /kafka  
[consumers, config, controller, admin, brokers, controller_epoch]
```

```
[zk: hc2r1m2:2181(CONNECTED) 3] ls /kafka/brokers  
[topics, ids]
```

```
[zk: hc2r1m2:2181(CONNECTED) 4] ls /kafka/brokers/ids  
[3, 2, 1, 4]
```

I will create the topic that I want to use for this test using the Kafka script `kafka-topics` with a `create` flag. I do this manually, because I can demonstrate the definition of the data partitions while I do it. Note that I have set the partitions in the Kafka topic `rss` to five as shown in the following piece of code. Note also that the ZooKeeper connection string for the command has a comma-separated list of ZooKeeper servers, terminated by the top level ZooKeeper Kafka directory called `/kafka`. This means that the command puts the new topic in the proper place:

```
[hadoop@hc2nn ~]$ /usr/bin/kafka-topics \
> --create \
> --zookeeper hc2r1m2:2181,hc2r1m3:2181,hc2r1m4:2181/kafka \
> --replication-factor 3 \
> --partitions 5 \
> --topic rss
```

Created topic "rss".

Now, when I use the ZooKeeper client to check the Kafka topic configuration, I can see the correct topic name, and the expected number of the partitions:

```
[zk: hc2r1m2:2181(CONNECTED) 5] ls /kafka/brokers/topics
[rss]
```

```
[zk: hc2r1m2:2181(CONNECTED) 6] ls /kafka/brokers/topics/rss
[partitions]
```

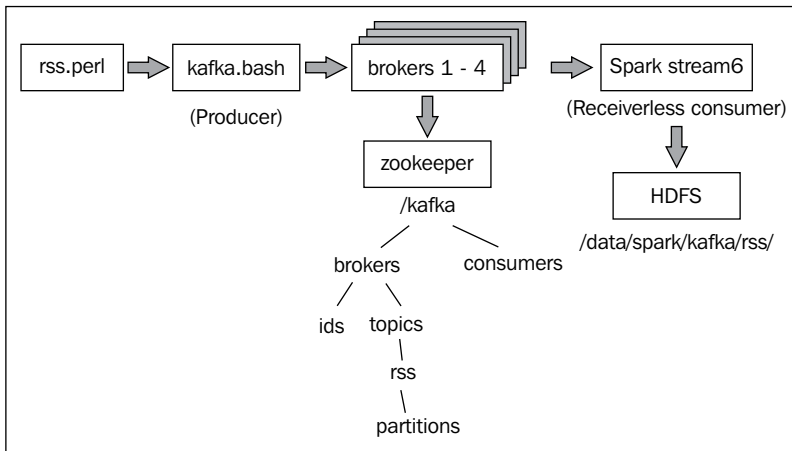
```
[zk: hc2r1m2:2181(CONNECTED) 7] ls /kafka/brokers/topics/rss/partitions
[3, 2, 1, 0, 4]
```

This describes the configuration for the Kafka broker servers in ZooKeeper, but what about the data consumers? Well, the following listing shows where the data will be held. Remember though, at this time, there is no consumer running, so it is not represented in ZooKeeper:

```
[zk: hc2r1m2:2181(CONNECTED) 9] ls /kafka/consumers
[]
[zk: hc2r1m2:2181(CONNECTED) 10] quit
```

In order to start this test, I will run my Kafka data producer, and consumer scripts. I will also check the output of the Spark application class and need to check the Kafka partition offsets and HDFS to make sure that the data has arrived. This is quite complicated, so I will add a diagram here in the following figure to explain the test architecture.

The Perl script called `rss.perl` will be used to provide a data source for a Kafka data producer, which will feed data into the CDH Kafka broker servers. The data will be stored in ZooKeeper, in the structure that has just been examined, under the top level node called `/kafka`. The Apache Spark Scala-based application will then act as a Kafka consumer, and read the data that it will store under HDFS.



In order to try and explain the complexity here, I will also examine my method of running the Apache Spark class. It will be started via the `spark-submit` command. Remember again that all of these scripts will be released with this book, so that you can examine them in your own time. I always use scripts for server test management, so that I encapsulate complexity, and command execution is quickly repeatable. The script, `run_stream.bash`, is like many example scripts that have already been used in this chapter, and this book. It accepts a class name and the class parameters, and runs the class via `spark-submit`:

```
[hadoop@hc2r1m1 stream]$ more run_stream.bash
```

```
#!/bin/bash
```

```
SPARK_HOME=/usr/local/spark
```

```
SPARK_BIN=$SPARK_HOME/bin
```

```

SPARK_SBIN=$SPARK_HOME/sbin

JAR_PATH=/home/hadoop/spark/stream/target/scala-2.10/streaming_2.10-1.0.jar
CLASS_VAL=$1
CLASS_PARAMS="${*:2}"

STREAM_JAR=/usr/local/spark/lib/spark-examples-1.3.1-hadoop2.3.0.jar
cd $SPARK_BIN

./spark-submit \
  --class $CLASS_VAL \
  --master spark://hc2nn.semtech-solutions.co.nz:7077 \
  --executor-memory 100M \
  --total-executor-cores 50 \
  --jars $STREAM_JAR \
  $JAR_PATH \
  $CLASS_PARAMS

```

I then used a second script, which calls the `run_kafka_example.bash` script to execute the Kafka consumer code in the previous `stream6` application class. Note that this script sets up the full application class name—the broker server list. It also sets up the topic name, called `rss`, to use for data consumption. Finally, it defines a consumer group called `group1`. Remember that Kafka is a publish/subscribe message brokering system. There may be many producers and consumers organized by topic, group, and partition:

```

[hadoop@hc2r1m1 stream]$ more run_kafka_example.bash

#!/bin/bash

RUN_CLASS=nz.co.semtechsolutions.stream6
BROKERS="hc2r1m1:9092,hc2r1m2:9092,hc2r1m3:9092,hc2r1m4:9092"
GROUPID=group1
TOPICS=rss

# run the Apache Spark Kafka example

./run_stream.bash $RUN_CLASS \

```

```
$BROKERS \
$GROUPID \
$TOPICS
```

So, I will start the Kafka consumer by running the `run_kafka_example.bash` script, which in turn will run the previous `stream6` Scala code using `spark-submit`. While monitoring Kafka data consumption using the script called `kafka_list.bash`, I was able to get the `kafka-consumer-offset-checker` script to list the Kafka-based topics, but for some reason, it will not check the correct path (under `/kafka` in ZooKeeper) when checking the offsets as shown here:

```
[hadoop@hc2r1m1 stream]$ ./kafka list.bash
```

```
=====
Kafka Topics
=====
__consumer_offsets
rss
```

```
=====
Kafka Offsets
=====
```

```
Exiting due to: org.apache.zookeeper.KeeperException$NoNodeException:
KeeperErrorCode = NoNode for /consumers/group1/offsets/rss/4.
```

By starting the Kafka producer RSS feed using the script `kafka.bash`, I can now start feeding the rss-based data through Kafka into Spark, and then into HDFS. Periodically checking the `spark-submit` session output it can be seen that events are passing through the Spark-based Kafka DStream. The following output comes from the stream count in the Scala code, and shows that at that point, 28 events were processed:

```
-----  
Time: 1436834440000 ms  
-----  
  
>>>>>>>>>>> Received events : 28
```

By checking HDFS under the `/data/spark/kafka/rss/` directory, via the Hadoop file system `ls` command, it can be seen that there is now data stored on HDFS:

```
[hadoop@hc2r1m1 stream]$ hdfs dfs -ls /data/spark/kafka/rss
Found 1 items
```

```
drwxr-xr-x - hadoop supergroup          0 2015-07-14 12:40 /data/spark/
kafka/rss/file_1436833769907
```

By checking the contents of this directory, it can be seen that an HDFS part data file exists, which should contain the RSS-based data from Reuters:

```
[hadoop@hc2r1m1 stream]$ hdfs dfs -ls /data/spark/kafka/rss/
file_1436833769907
```

Found 2 items

```
-rw-r--r--  3 hadoop supergroup          0 2015-07-14 12:40 /data/spark/
kafka/rss/file_1436833769907/_SUCCESS
-rw-r--r--  3 hadoop supergroup      8205 2015-07-14 12:40 /data/spark/
kafka/rss/file_1436833769907/part-00001
```

Using the Hadoop file system `cat` command below, I can dump the contents of this HDFS-based file to check its contents. I have used the Linux `head` command to limit the data to save space. Clearly this is RSS Reuters science based information that the Perl script `rss.perl` has converted from XML to RSS JSON format.

```
[hadoop@hc2r1m1 stream]$ hdfs dfs -cat /data/spark/kafka/rss/
file_1436833769907/part-00001 | head -2
```

```
{"category": "science", "title": "Bear necessities: low metabolism lets
pandas survive on bamboo", "summary": "WASHINGTON (Reuters) - Giant
pandas eat vegetables even though their bodies are better equipped to
eat meat. So how do these black-and-white bears from the remote misty
mountains of central China survive on a diet almost exclusively of a low-
nutrient food like bamboo?"}
```

```
{"category": "science", "title": "PlanetiQ tests sensor for commercial
weather satellites", "summary": "CAPE CANAVERAL (Reuters) - PlanetiQ a
privately owned company is beginning a key test intended to pave the way
for the first commercial weather satellites."}
```

This ends this Kafka example. It can be seen that Kafka brokers have been installed and configured. It shows that an RSS data-based Kafka producer has fed data into the brokers. It has been proved, using the ZooKeeper client, that the Kafka architecture, matching the brokers, topics, and partitions has been set up in ZooKeeper. Finally, it has been shown using the Apache Spark-based Scala code, in the `stream6` application, that the Kafka data has been consumed and saved to HDFS.

Summary

I could have provided streaming examples for systems like Kinesis, as well as queuing systems, but there was not room in this chapter. Twitter streaming has been examined by example in the checkpointing section.

This chapter has provided practical examples of data recovery via checkpointing in Spark streaming. It has also touched on the performance limitations of checkpointing and shown that the checkpointing interval should be set at five to ten times the Spark stream batch interval. Checkpointing provides a stream-based recovery mechanism in the case of Spark application failure.

This chapter has provided some stream-based worked examples for TCP, File, Flume, and Kafka-based Spark stream coding. All the examples here are based on Scala, and are compiled with `sbt`. All of the code will be released with this book. Where the example architecture has become over-complicated, I have provided an architecture diagram (I'm thinking of the Kafka example here).

It is clear to me that the Apache Spark streaming module contains a rich source of functionality that should meet most of your needs, and will grow as future releases of Spark are delivered. Remember to check the Apache Spark website (<http://spark.apache.org/>), and join the Spark user list via user@spark.apache.org. Don't be afraid to ask questions, or make mistakes, as it seems to me that mistakes teach more than success.

The next chapter will examine the Spark SQL module, and will provide worked examples of SQL, data frames, and accessing Hive among other topics.

4

Apache Spark SQL

In this chapter, I would like to examine Apache Spark SQL, the use of Apache Hive with Spark, and DataFrames. DataFrames have been introduced in Spark 1.3, and are columnar data storage structures, roughly equivalent to relational database tables. The chapters in this book have not been developed in sequence, so the earlier chapters might use older versions of Spark than the later ones. I also want to examine user-defined functions for Spark SQL. A good place to find information about the Spark class API is: `spark.apache.org/docs/<version>/api/scala/index.html`.

I prefer to use Scala, but the API information is also available in Java and Python formats. The `<version>` value refers to the release of Spark that you will be using—1.3.1. This chapter will cover the following topics:

- SQL context
- Importing and saving data
- DataFrames
- Using SQL
- User-defined functions
- Using Hive

Before moving straight into SQL and DataFrames, I will give an overview of the SQL context.

The SQL context

The SQL context is the starting point for working with columnar data in Apache Spark. It is created from the Spark context, and provides the means for loading and saving data files of different types, using DataFrames, and manipulating columnar data with SQL, among other things. It can be used for the following:

- Executing SQL via the SQL method
- Registering user-defined functions via the UDF method
- Caching
- Configuration
- DataFrames
- Data source access
- DDL operations

I am sure that there are other areas, but you get the idea. The examples in this chapter are written in Scala, just because I prefer the language, but you can develop in Python and Java as well. As shown previously, the SQL context is created from the Spark context. Importing the SQL context implicitly allows you to implicitly convert RDDs into DataFrames:

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext.implicits._
```

For instance, using the previous `implicits` call, allows you to import a CSV file and split it by separator characters. It can then convert the RDD that contains the data into a data frame using the `toDF` method.

It is also possible to define a Hive context for the access and manipulation of Apache Hive database table data (Hive is the Apache data warehouse that is part of the Hadoop eco-system, and it uses HDFS for storage). The Hive context allows a superset of SQL functionality when compared to the Spark context. The use of Hive with Spark will be covered in a later section in this chapter.

Next, I will examine some of the supported file formats available for importing and saving data.

Importing and saving data

I wanted to add this section about importing and saving data here, even though it is not purely about Spark SQL, so I could introduce concepts such as **Parquet** and **JSON** file formats. This section also allows me to cover how to access and save data in loose text; as well as the CSV, Parquet and JSON formats, conveniently, in one place.

Processing the Text files

Using the Spark context, it is possible to load a text file into an RDD using the `textFile` method. Also, the `wholeTextFile` method can read the contents of a directory into an RDD. The following examples show how a file, based on the local file system (`file://`), or HDFS (`hdfs://`) can be read into a Spark RDD. These examples show that the data will be partitioned into six parts for increased performance. The first two examples are the same, as they both manipulate a file on the Linux file system:

```
sc.textFile("/data/spark/tweets.txt", 6)
sc.textFile("file:///data/spark/tweets.txt", 6)
sc.textFile("hdfs://server1:4014/data/spark/tweets.txt", 6)
```

Processing the JSON files

JSON is a data interchange format, developed from Javascript. **JSON** actually stands for **JavaScript Object Notation**. It is a text-based format, and can be expressed, for instance, as XML. The following example uses the SQL context method called `jsonFile` to load the HDFS-based JSON data file named `device.json`. The resulting data is created as a data frame:

```
val dframe = sqlContext.jsonFile("hdfs:///data/spark/device.json")
```

Data can be saved in JSON format using the data frame `toJSON` method, as shown by the following example. First, the Apache Spark and Spark SQL classes are imported:

```
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.sql.Row;
import org.apache.spark.sql.types.{StructType, StructField, StringType};
```

Next, the object class called `sql1` is defined as is a main method with parameters. A configuration object is defined that is used to create a spark context. The master Spark URL is left as the default value, so Spark expects local mode, the local host, and the 7077 port:

```
object sql1 {  
  
  def main(args: Array[String]) {  
  
    val appName = "sql example 1"  
    val conf    = new SparkConf()  
  
    conf.setAppName(appName)  
  
    val sc = new SparkContext(conf)
```

An SQL context is created from the Spark context, and a raw text file is loaded in CSV format called `adult.test.data_1x`, using the `textFile` method. A schema string is then created, which contains the data column names and the schema created from it by splitting the string by its spacing, and using the `StructType` and `StructField` methods to define each schema column as a string value:

```
    val sqlContext = new org.apache.spark.sql.SQLContext(sc)  
  
    val rawRdd = sc.textFile("hdfs:///data/spark/sql/adult.test.data_1x")  
  
    val schemaString = "age workclass fnlwtg education " +  
"educational-num marital-status occupation relationship " +  
"race gender capital-gain capital-loss hours-per-week " +  
"native-country income"  
  
    val schema =  
      StructType(  
        schemaString.split(" ").map(fieldName => StructField(fieldName,  
StringType, true)))
```

Each data row is then created from the raw CSV data by splitting it with the help of a comma as a line divider, and then the elements are added to a `Row()` structure. A data frame is created from the schema, and the row data which is then converted into JSON format using the `toJSON` method. Finally, the data is saved to HDFS using the `saveAsTextFile` method:

```
val rowRDD = rawRdd.map(_.split(","))
    .map(p => Row( p(0),p(1),p(2),p(3),p(4),p(5),p(6),p(7),p(8),
                  p(9),p(10),p(11),p(12),p(13),p(14) ))

val adultDataFrame = sqlContext.createDataFrame(rowRDD, schema)

val jsonData = adultDataFrame.toJSON

jsonData.saveAsTextFile("hdfs:///data/spark/sql/adult.json")

} // end main

} // end sql1
```

So the resulting data can be seen on HDFS, the Hadoop file system `ls` command below shows that the data resides in the `target` directory as a success file and two part files.

```
[hadoop@hc2nn sql]$ hdfs dfs -ls /data/spark/sql/adult.json
```

Found 3 items

```
-rw-r--r--  3 hadoop supergroup          0 2015-06-20 17:17 /data/spark/
sql/adult.json/_SUCCESS
-rw-r--r--  3 hadoop supergroup       1731 2015-06-20 17:17 /data/spark/
sql/adult.json/part-00000
-rw-r--r--  3 hadoop supergroup       1724 2015-06-20 17:17 /data/spark/
sql/adult.json/part-00001
```

Using the Hadoop file system's `cat` command, it is possible to display the contents of the JSON data. I will just show a sample to save space:

```
[hadoop@hc2nn sql]$ hdfs dfs -cat /data/spark/sql/adult.json/part-00000 |
more
```

```
{"age":"25","workclass":" Private","fnlwgt":" 226802","education":"
11th","educational-num":"
```

```
7","marital-status":" Never-married","occupation":" Machine-op-  
inspct","relationship":" Own-  
child","race":" Black","gender":" Male","capital-gain":" 0","capital-  
loss":" 0","hours-per-we  
ek":" 40","native-country":" United-States","income":" <=50K"}
```

Processing the Parquet data is very similar, as I will show next.

Processing the Parquet files

Apache Parquet is another columnar-based data format used by many tools in the Hadoop tool set for file I/O, such as Hive, Pig, and Impala. It increases performance by using efficient compression and encoding routines.

The Parquet processing example is very similar to the JSON Scala code. The `DataFrame` is created, and then saved in a Parquet format using the `save` method with a type of Parquet:

```
val adultDataFrame = sqlContext.createDataFrame(rowRDD, schema)  
adultDataFrame.save("hdfs:///data/spark/sql/adult.parquet", "parquet")  
  
} // end main  
  
} // end sql2
```

This results in an HDFS-based directory, which contains three Parquet-based files: a common Metadata file, a Metadata file, and a temporary file:

```
[hadoop@hc2nn sql]$ hdfs dfs -ls /data/spark/sql/adult.parquet  
Found 3 items  
-rw-r--r--    3 hadoop supergroup      1412 2015-06-21 13:17 /data/spark/  
sql/adult.parquet/_common_metadata  
-rw-r--r--    3 hadoop supergroup      1412 2015-06-21 13:17 /data/spark/  
sql/adult.parquet/_metadata  
drwxr-xr-x    - hadoop supergroup        0 2015-06-21 13:17 /data/spark/  
sql/adult.parquet/_temporary
```

Listing the contents of the metadata file, using the Hadoop file system's `cat` command, gives an idea of the data format. However the Parquet header is binary, and so, it does not display with `more` and `cat`:

```
[hadoop@hc2nn sql]$ hdfs dfs -cat /data/spark/sql/adult.parquet/_metadata  
| more  
s%
```

```
ct", "fields": [{ "name": "age", "type": "string", "nullable": true, "metadata": {}
}, { "name": "workclass
", "type": "string", "nullable": true, "metadata": {} }, { "name": "fnlwgt", "type":
"string", "nullable":
true, "metadata": {} },
```

For more information about possible Spark and SQL context methods, check the contents of the classes called `org.apache.spark.SparkContext`, and `org.apache.spark.sql.SQLContext`, using the Apache Spark API path here for the specific <version> of Spark that you are interested in:

spark.apache.org/docs/<version>/api/scala/index.html

In the next section, I will examine Apache Spark DataFrames, introduced in Spark 1.3.

DataFrames

I have already mentioned that a DataFrame is based on a columnar format. Temporary tables can be created from it, but I will expand on this in the next section. There are many methods available to the data frame that allow data manipulation, and processing. I have based the Scala code used here, on the code in the last section, so I will just show you the working lines and the output. It is possible to display a data frame schema as shown here:

```
adultDataFrame.printSchema()
```

```
root
```

```
|-- age: string (nullable = true)
|-- workclass: string (nullable = true)
|-- fnlwgt: string (nullable = true)
|-- education: string (nullable = true)
|-- educational-num: string (nullable = true)
|-- marital-status: string (nullable = true)
|-- occupation: string (nullable = true)
|-- relationship: string (nullable = true)
|-- race: string (nullable = true)
|-- gender: string (nullable = true)
|-- capital-gain: string (nullable = true)
|-- capital-loss: string (nullable = true)
|-- hours-per-week: string (nullable = true)
|-- native-country: string (nullable = true)
|-- income: string (nullable = true)
```

It is possible to use the `select` method to filter columns from the data. I have limited the output here, in terms of rows, but you get the idea:

```
adultDataFrame.select("workclass","age","education","income").show()
```

workclass	age	education	income
Private	25	11th	<=50K
Private	38	HS-grad	<=50K
Local-gov	28	Assoc-acdm	>50K
Private	44	Some-college	>50K
none	18	Some-college	<=50K
Private	34	10th	<=50K
none	29	HS-grad	<=50K
Self-emp-not-inc	63	Prof-school	>50K
Private	24	Some-college	<=50K
Private	55	7th-8th	<=50K

It is possible to filter the data returned from the DataFrame using the `filter` method. Here, I have added the occupation column to the output, and filtered on the worker age:

```
adultDataFrame
  .select("workclass","age","education","occupation","income")
  .filter( adultDataFrame("age") > 30 )
  .show()
```

workclass	age	education	occupation	income
Private	38	HS-grad	Farming-fishing	<=50K
Private	44	Some-college	Machine-op-inspct	>50K
Private	34	10th	Other-service	<=50K
Self-emp-not-inc	63	Prof-school	Prof-specialty	>50K
Private	55	7th-8th	Craft-repair	<=50K

There is also a `group by` method for determining volume counts within a data set. As this is an income-based dataset, I think that volumes within the wage brackets would be interesting. I have also used a bigger dataset to give more meaningful results:

```
adultDataFrame
  .groupBy("income")
  .count()
```

```
.show()
```

```
income count
<=50K 24720
>50K 7841
```

This is interesting, but what if I want to compare income brackets with occupation, and sort the results for a better understanding? The following example shows how this can be done, and gives the example data volumes. It shows that there is a high volume of managerial roles compared to other occupations. This example also sorts the output by the occupation column:

```
adultDataFrame
    .groupBy("income","occupation")
    .count()
    .sort("occupation")
    .show()
```

```
income occupation      count
>50K   Adm-clerical      507
<=50K   Adm-clerical    3263
<=50K   Armed-Forces      8
>50K   Armed-Forces      1
<=50K   Craft-repair    3170
>50K   Craft-repair      929
<=50K   Exec-managerial  2098
>50K   Exec-managerial  1968
<=50K   Farming-fishing  879
>50K   Farming-fishing   115
<=50K   Handlers-cleaners 1284
>50K   Handlers-cleaners  86
>50K   Machine-op-inspct 250
<=50K   Machine-op-inspct 1752
>50K   Other-service     137
<=50K   Other-service    3158
>50K   Priv-house-serv    1
<=50K   Priv-house-serv  148
>50K   Prof-specialty    1859
<=50K   Prof-specialty   2281
```

So, SQL-like actions can be carried out against DataFrames, including `select`, `filter`, `sort group by`, and `print`. The next section shows how tables can be created from the DataFrames, and how the SQL-based actions are carried out against them.

Using SQL

After using the previous Scala example to create a data frame, from a CSV based-data input file on HDFS, I can now define a temporary table, based on the data frame, and run SQL against it. The following example shows the temporary table called `adult` being defined, and a row count being created using `COUNT(*)`:

```
adultDataFrame.registerTempTable("adult")

val resRDD = sqlContext.sql("SELECT COUNT(*) FROM adult")

resRDD.map(t => "Count - " + t(0)).collect().foreach(println)
```

This gives a row count of over 32,000 rows:

```
Count - 32561
```

It is also possible to limit the volume of the data selected from the table using the `LIMIT SQL` option, which is shown in the following example. The first 10 rows have been selected from the data, this is useful if I just want to check data types and quality:

```
val resRDD = sqlContext.sql("SELECT * FROM adult LIMIT 10")

resRDD.map(t => t(0)  + " " + t(1)  + " " + t(2)  + " " + t(3)  + " "
+
                t(4)  + " " + t(5)  + " " + t(6)  + " " + t(7)  + " "
+
                t(8)  + " " + t(9)  + " " + t(10) + " " + t(11) + " "
+
                t(12) + " " + t(13) + " " + t(14)
        )
    .collect().foreach(println)
```

A sample of the data looks like the following:

```
50 Private 283676 Some-college 10 Married-civ-spouse Craft-repair
Husband White Male 0 0 40 United-States >50K
```

When the schema for this data was created in the Scala-based data frame example in the last section, all the columns were created as strings. However, if I want to filter the data in SQL using `WHERE` clauses, it would be useful to have proper data types. For instance, if an age column stores integer values, it should be stored as an integer so that I can execute numeric comparisons against it. I have changed my Scala code to include all the possible types:

```
import org.apache.spark.sql.types._
```

I have also now defined my schema using different types, to better match the data, and I have defined the row data in terms of the actual data types, converting raw data string values into integer values, where necessary:

```
val schema =
  StructType(
    StructField("age",           IntegerType, false) ::
    StructField("workclass",     StringType,  false) ::
    StructField("fnlwgt",       IntegerType, false) ::
    StructField("education",    StringType,  false) ::
    StructField("educational-num", IntegerType, false) ::
    StructField("marital-status", StringType,  false) ::
    StructField("occupation",    StringType,  false) ::
    StructField("relationship",  StringType,  false) ::
    StructField("race",          StringType,  false) ::
    StructField("gender",        StringType,  false) ::
    StructField("capital-gain",   IntegerType, false) ::
    StructField("capital-loss",   IntegerType, false) ::
    StructField("hours-per-week", IntegerType, false) ::
    StructField("native-country", StringType,  false) ::
    StructField("income",        StringType,  false) ::
    Nil)

val rowRDD = rawRdd.map(_.split(","))
  .map(p => Row( p(0).trim.toInt,p(1),p(2).trim.toInt,p(3),
    p(4).trim.toInt,p(5),p(6),p(7),p(8),
    p(9),p(10).trim.toInt,p(11).trim.toInt,
    p(12).trim.toInt,p(13),p(14) ))
```

The SQL can now use numeric filters in the `WHERE` clause correctly. If the `age` column were a string, this would not work. You can now see that the data has been filtered to give age values below 60 years:

```
val resRDD = sqlContext.sql("SELECT COUNT(*) FROM adult WHERE age < 60")
resRDD.map(t => "Count - " + t(0)).collect().foreach(println)
```

This gives a row count of around 30,000 rows:

```
Count - 29917
```

It is possible to use Boolean logic in the `WHERE`-based filter clauses. The following example specifies an age range for the data. Note that I have used variables to describe the `select` and `filter` components of the SQL statement. This allows me to break down the statement into different parts as they become larger:

```
val selectClause = "SELECT COUNT(*) FROM adult "
val filterClause = "WHERE age > 25 AND age < 60"
val resRDD = sqlContext.sql( selectClause + filterClause )
resRDD.map(t => "Count - " + t(0)).collect().foreach(println)
```

Giving a data count of around 23,000 rows:

```
Count - 23506
```

I can create compound filter clauses using the Boolean terms, such as `AND`, `OR`, as well as parentheses:

```
val selectClause = "SELECT COUNT(*) FROM adult "
val filterClause =
"WHERE ( age > 15 AND age < 25 ) OR ( age > 30 AND age < 45 ) "

val resRDD = sqlContext.sql( selectClause + filterClause )
resRDD.map(t => "Count - " + t(0)).collect().foreach(println)
```

This gives me a row count of 17,000 rows, and represents a count of two age ranges in the data:

```
Count - 17198
```

It is also possible to use subqueries in Apache Spark SQL. You can see in the following example that I have created a subquery called `t1` by selecting three columns; age, education, and occupation from the table `adult`. I have then used the table called `t1` to create a row count. I have also added a filter clause acting on the age column from the table `t1`. Notice also that I have added `group by` and `order by` clauses, even though they are empty currently, to my SQL:

```
val selectClause = "SELECT COUNT(*) FROM "
val tableClause = " ( SELECT age,education,occupation from adult) t1
"

val filterClause = "WHERE ( t1.age > 25 ) "
val groupClause = ""
val orderClause = ""

val resRDD = sqlContext.sql( selectClause + tableClause +
                             filterClause +
                             groupClause + orderClause
                           )

resRDD.map(t => "Count - " + t(0)).collect().foreach(println)
```

In order to examine the table joins, I have created a version of the adult CSV data file called `adult.train.data2`, which only differs from the original by the fact that it has an added first column called `idx`, which is a unique index. The Hadoop file system's `cat` command here shows a sample of the data. The output from the file has been limited using the Linux `head` command:

```
[hadoop@hc2nn sql]$ hdfs dfs -cat /data/spark/sql/adult.train.data2 |
head -2
```

```
1,39, State-gov, 77516, Bachelors, 13, Never-married, Adm-clerical, Not-
in-family, White, Male, 2174, 0, 40, United-States, <=50K
2,50, Self-emp-not-inc, 83311, Bachelors, 13, Married-civ-spouse, Exec-
managerial, Husband, White, Male, 0, 0, 13, United-States, <=50K
```

The schema has now been redefined to have an integer-based first column called `idx` for an index, as shown here:

```
val schema =
  StructType(
    StructField("idx",
                 IntegerType, false) ::
    StructField("age",
                 IntegerType, false) ::
```

```

    StructField("workclass",      StringType, false) ::
    StructField("fnlwtg",         IntegerType, false) ::
    StructField("education",      StringType, false) ::
    StructField("educational-num", IntegerType, false) ::
    StructField("marital-status", StringType, false) ::
    StructField("occupation",     StringType, false) ::
    StructField("relationship",   StringType, false) ::
    StructField("race",           StringType, false) ::
    StructField("gender",         StringType, false) ::
    StructField("capital-gain",    IntegerType, false) ::
    StructField("capital-loss",    IntegerType, false) ::
    StructField("hours-per-week",  IntegerType, false) ::
    StructField("native-country",  StringType, false) ::
    StructField("income",         StringType, false) ::
    Nil)

```

And the raw row RDD in the Scala example now processes the new initial column, and converts the string value into an integer:

```

val rowRDD = rawRdd.map(_.split(","))
  .map(p => Row( p(0).trim.toInt,
                p(1).trim.toInt,
                p(2),
                p(3).trim.toInt,
                p(4),
                p(5).trim.toInt,
                p(6),
                p(7),
                p(8),
                p(9),
                p(10),
                p(11).trim.toInt,
                p(12).trim.toInt,
                p(13).trim.toInt,
                p(14),
                p(15)
              ))

```

```

val adultDataFrame = sqlContext.createDataFrame(rowRDD, schema)

```

We have looked at subqueries. Now, I would like to consider table joins. The next example will use the index that was just created. It uses it to join two derived tables. The example is somewhat contrived, given that it joins two data sets from the same underlying table, but you get the idea. Two derived tables are created as subqueries, and are joined at a common index column.

The SQL for a table join now looks like this. Two derived tables have been created from the temporary table `adult` called `t1` and `t2` as subqueries. The new row index column called `idx` has been used to join the data in tables `t1` and `t2`. The major `SELECT` statement outputs all seven columns from the compound data set. I have added a `LIMIT` clause to minimize the data output:

```
val selectClause = "SELECT t1.idx,age,education,occupation,workclass,rac
e,gender FROM "
val tableClause1 = " ( SELECT idx,age,education,occupation FROM adult)
t1 JOIN "
val tableClause2 = " ( SELECT idx,workclass,race,gender FROM adult) t2 "
val joinClause = " ON (t1.idx=t2.idx) "
val limitClause = " LIMIT 10"

val resRDD = sqlContext.sql( selectClause +
                             tableClause1 + tableClause2 +
                             joinClause   + limitClause
                             )

resRDD.map(t => t(0) + " " + t(1) + " " + t(2) + " " +
             t(3) + " " + t(4) + " " + t(5) + " " + t(6)
        )
        .collect().foreach(println)
```

Note that in the major `SELECT` statement, I have to define where the index column comes from, so I use `t1.idx`. All the other columns are unique to the `t1` and `t2` datasets, so I don't need to use an alias to refer to them (that is, `t1.age`). So, the data that is output now looks like the following:

```
33 45 Bachelors Exec-managerial Private White Male
233 25 Some-college Adm-clerical Private White Male
433 40 Bachelors Prof-specialty Self-emp-not-inc White Female
633 43 Some-college Craft-repair Private White Male
833 26 Some-college Handlers-cleaners Private White Male
1033 27 Some-college Sales Private White Male
```

```
1233 27 Bachelors Adm-clerical Private White Female
1433 32 Assoc-voc Sales Private White Male
1633 40 Assoc-acdm Adm-clerical State-gov White Male
1833 46 Some-college Prof-specialty Local-gov White Male
```

This gives some idea of the SQL-based functionality within Apache Spark, but what if I find that the method that I need is not available? Perhaps, I need a new function. This is where the **user-defined functions (UDFs)** are useful. I will cover them in the next section.

User-defined functions

In order to create some user-defined functions in Scala, I need to examine my data in the previous adult dataset. I plan to create a UDF that will enumerate the education column, so that I can convert the column into an integer value. This will be useful if I need to use the data for machine learning, and so create a LabelPoint structure. The vector used, which represents each record, will need to be numeric. I will first determine what kind of unique education values exist, then I will create a function to enumerate them, and finally use it in SQL.

I have created some Scala code to display a sorted list of the education values. The `DISTINCT` keyword ensures that there is only one instance of each value. I have selected the data as a subtable, using an alias called `edu_dist` for the data column to ensure that the `ORDER BY` clause works:

```
val selectClause = "SELECT t1.edu_dist FROM "
val tableClause  = " ( SELECT DISTINCT education AS edu_dist FROM
adult ) t1 "
val orderClause  = " ORDER BY t1.edu_dist "

val resRDD = sqlContext.sql( selectClause + tableClause +
orderClause )

resRDD.map(t => t(0)).collect().foreach(println)
```

The data looks like the following. I have removed some values to save space, but you get the idea:

```
10th
11th
12th
```

```

1st-4th
.....
Preschool
Prof-school
Some-college

```

I have defined a method in Scala to accept the string-based education value, and return an enumerated integer value that represents it. If no value is recognized, then a special value called 9999 is returned:

```

def enumEdu( education:String ) : Int =
{
    var enumval = 9999

        if ( education == "10th" )           { enumval = 0 }
    else if ( education == "11th" )          { enumval = 1 }
    else if ( education == "12th" )          { enumval = 2 }
    else if ( education == "1st-4th" )       { enumval = 3 }
    else if ( education == "5th-6th" )       { enumval = 4 }
    else if ( education == "7th-8th" )       { enumval = 5 }
    else if ( education == "9th" )           { enumval = 6 }
    else if ( education == "Assoc-acdm" )    { enumval = 7 }
    else if ( education == "Assoc-voc" )     { enumval = 8 }
    else if ( education == "Bachelors" )     { enumval = 9 }
    else if ( education == "Doctorate" )     { enumval = 10 }
    else if ( education == "HS-grad" )       { enumval = 11 }
    else if ( education == "Masters" )       { enumval = 12 }
    else if ( education == "Preschool" )     { enumval = 13 }
    else if ( education == "Prof-school" )   { enumval = 14 }
    else if ( education == "Some-college" ) { enumval = 15 }

    return enumval
}

```

I can now register this function using the SQL context in Scala, so that it can be used in an SQL statement:

```

sqlContext.udf.register( "enumEdu", enumEdu _ )

```

The SQL, and the Scala code to enumerate the data then look like this. The newly registered function called `enumEdu` is used in the `SELECT` statement. It takes the education type as a parameter, and returns the integer enumeration. The column that this value forms is aliased to the name `idx`:

```
val selectClause = "SELECT enumEdu(t1.edu_dist) as idx,t1.edu_dist
FROM "

val tableClause  = " ( SELECT DISTINCT education AS edu_dist FROM
adult ) t1 "

val orderClause  = " ORDER BY t1.edu_dist "

val resRDD = sqlContext.sql( selectClause + tableClause +
orderClause )

resRDD.map(t => t(0) + " " + t(1) ).collect().foreach(println)
```

The resulting data output, as a list of education values and their enumerations, looks like the following:

```
0  10th
1  11th
2  12th
3  1st-4th
4  5th-6th
5  7th-8th
6  9th
7  Assoc-acdm
8  Assoc-voc
9  Bachelors
10  Doctorate
11  HS-grad
12  Masters
13  Preschool
14  Prof-school
15  Some-college
```

Another example function called `ageBracket` takes the adult integer age value, and returns an enumerated age bracket:

```
def ageBracket( age:Int ) : Int =
{
```

```

var bracket = 9999

    if ( age >= 0  && age < 20 ) { bracket = 0 }
else if ( age >= 20 && age < 40 ) { bracket = 1 }
else if ( age >= 40 && age < 60 ) { bracket = 2 }
else if ( age >= 60 && age < 80 ) { bracket = 3 }
else if ( age >= 80 && age < 100 ) { bracket = 4 }
else if ( age > 100 )                { bracket = 5 }

return bracket
}

```

Again, the function is registered using the SQL context so that it can be used in an SQL statement:

```
sqlContext.udf.register( "ageBracket", ageBracket _ )
```

Then, the Scala-based SQL uses it to select the age, age bracket, and education value from the adult dataset:

```

val selectClause = "SELECT age, ageBracket(age) as bracket, education
FROM "
val tableClause  = " adult "
val limitClause  = " LIMIT 10 "

val resRDD = sqlContext.sql( selectClause + tableClause +
                             limitClause )

resRDD.map(t => t(0) + " " + t(1) + " " + t(2) ).collect().
foreach(println)

```

The resulting data then looks like this, given that I have used the `LIMIT` clause to limit the output to 10 rows:

```

39 1  Bachelors
50 2  Bachelors
38 1  HS-grad
53 2  11th
28 1  Bachelors
37 1  Masters
49 2  9th

```

```
52 2  HS-grad
31 1  Masters
42 2  Bachelors
```

It is also possible to define functions for use in SQL, inline, during the UDF registration using the SQL context. The following example defines a function called `dblAge`, which just multiplies the adult's age by two. The registration looks like this. It takes integer parameters (`age`), and returns twice its value:

```
sqlContext.udf.register( "dblAge", (a:Int) => 2*a )
```

And the SQL that uses it, now selects the age, and the double of the age value called `dblAge(age)`:

```
val selectClause = "SELECT age,dblAge(age) FROM "
val tableClause  = " adult "
val limitClause  = " LIMIT 10 "

val resRDD = sqlContext.sql( selectClause + tableClause +
limitClause )

resRDD.map(t => t(0) + " " + t(1) ).collect().foreach(println)
```

The two columns of the output data, which now contain the age and its doubled value, now look like this:

```
39 78
50 100
38 76
53 106
28 56
37 74
49 98
52 104
31 62
42 84
```

So far, DataFrames, SQL, and user-defined functions have been examined, but what if, as in my case, you are using a Hadoop stack cluster, and have Apache Hive available? The adult table that I have defined so far is a temporary table, but if I access Hive using Apache Spark SQL, I can access the static database tables. The next section will examine the steps needed to do this.

Using Hive

If you have a business intelligence-type workload with low latency requirements and multiple users, then you might consider using Impala for your database access. Apache Spark on Hive is for batch processing and ETL chains. This section will be used to show how to connect Spark to Hive, and how to use this configuration. First, I will develop an application that uses a local Hive Metastore, and show that it does not store and persist table data in Hive itself. I will then set up Apache Spark to connect to the Hive Metastore server, and store tables and data within Hive. I will start with the local Metastore server.

Local Hive Metastore server

The following example Scala code shows how to create a Hive context, and create a Hive-based table using Apache Spark. First, the Spark configuration, context, SQL, and Hive classes are imported. Then, an object class called `hive_ex1`, and the main method are defined. The application name is defined, and a Spark configuration object is created. The Spark context is then created from the configuration object:

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql._
import org.apache.spark.sql.hive.HiveContext
```

```
object hive_ex1 {

  def main(args: Array[String]) {

    val appName = "Hive Spark Ex 1"
    val conf    = new SparkConf()

    conf.setAppName(appName)

    val sc = new SparkContext(conf)
```

Next, I create a new Hive context from the Spark context, and import the Hive implicits, and the Hive context SQL. The `implicits` allow for implicit conversions, and the `SQL` include allows me to run Hive context-based SQL:

```
val hiveContext = new HiveContext(sc)

import hiveContext.implicits._
import hiveContext.sql
```

The next statement creates an empty table called `adult2` in Hive. You will recognize the schema from the adult data that has already been used in this chapter:

```
hiveContext.sql( "
    CREATE TABLE IF NOT EXISTS adult2
    (
        idx            INT,
        age            INT,
        workclass       STRING,
        fnlwgt          INT,
        education       STRING,
        educationnum    INT,
        maritalstatus   STRING,
        occupation      STRING,
        relationship    STRING,
        race            STRING,
        gender          STRING,
        capitalgain     INT,
        capitalloss     INT,
        nativecountry   STRING,
        income          STRING
    )

    ")
```

Next, a row count is taken from the table called `adult2` via a `COUNT(*)`, and the output value is printed:

```
val resRDD = hiveContext.sql("SELECT COUNT(*) FROM adult2")

resRDD.map(t => "Count : " + t(0) ).collect().foreach(println)
```

As expected, there are no rows in the table.

```
Count : 0
```

It is also possible to create Hive-based external tables in Apache Spark Hive. The following HDFS file listing shows that the CSV file called `adult.train.data2` exists in the HDFS directory called `/data/spark/hive`, and it contains data:

```
[hadoop@hc2nn hive]$ hdfs dfs -ls /data/spark/hive
```

Found 1 items

```
-rw-r--r--  3 hadoop supergroup    4171350 2015-06-24 15:18 /data/spark/
hive/adult.train.data2
```

Now, I adjust my Scala-based Hive SQL to create an external table called `adult3` (if it does not exist), which has the same structure as the previous table. The row format in this table-create statement specifies a comma as a row column delimiter, as would be expected for CSV data. The location option in this statement specifies the `/data/spark/hive` directory on HDFS for data. So, there can be multiple files on HDFS, in this location, to populate this table. Each file would need to have the same data structure matching this table structure:

```
hiveContext.sql("

CREATE EXTERNAL TABLE IF NOT EXISTS adult3
(
    idx            INT,
    age            INT,
    workclass      STRING,
    fnlwgt         INT,
    education      STRING,
    educationnum   INT,
    maritalstatus  STRING,
    occupation     STRING,
    relationship   STRING,
    race           STRING,
    gender         STRING,
    capitalgain    INT,
    capitalloss    INT,
    nativecountry  STRING,
    income         STRING
)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/data/spark/hive'

")
```

A row count is then taken against the `adult3` table, and the count is printed:

```
val resRDD = hiveContext.sql("SELECT COUNT(*) FROM adult3")

resRDD.map(t => "Count : " + t(0) ).collect().foreach(println)
```

As you can see, the table now contains around 32,000 rows. Since this is an external table, the HDFS-based data has not been moved, and the row calculation has been derived from the underlying CSV-based data.

```
Count : 32561
```

It occurs to me that I want to start stripping dimension data out of the raw CSV-based data in the external `adult3` table. After all, Hive is a data warehouse, so a part of a general ETL chain using the raw CSV-based data would strip dimensions and objects from the data, and create new tables. If I consider the education dimension, and try to determine what unique values exist, then for instance, the SQL would be as follows:

```
val resRDD = hiveContext.sql("

    SELECT DISTINCT education AS edu FROM adult3
    ORDER BY edu

    ")

resRDD.map(t => t(0) ).collect().foreach(println)
```

And the ordered data matches the values that were derived earlier in this chapter using Spark SQL:

```
10th
11th
12th
1st-4th
5th-6th
7th-8th
9th
Assoc-acdm
Assoc-voc
Bachelors
Doctorate
```

```

HS-grad
Masters
Preschool
Prof-school
Some-college

```

This is useful, but what if I want to create dimension values, and then assign integer index values to each of the previous education dimension values. For instance, 10th would be 0, and 11th would be 1. I have set up a dimension CSV file for the education dimension on HDFS, as shown here. The contents just contain the list of unique values, and an index:

```

[hadoop@hc2nn hive]$ hdfs dfs -ls /data/spark/dim1/
Found 1 items
-rw-r--r--  3 hadoop supergroup          174 2015-06-25 14:08 /data/spark/
dim1/education.csv
[hadoop@hc2nn hive]$ hdfs dfs -cat /data/spark/dim1/education.csv
1,10th
2,11th
3,12th

```

Now, I can run some Hive QL in my Apache application to create an education dimension table. First, I drop the education table if it already exists, then I create the table by parsing the HDFS CSV file:

```

hiveContext.sql(" DROP TABLE IF EXISTS education ")
hiveContext.sql("

CREATE TABLE IF NOT EXISTS  education
(
    idx          INT,
    name         STRING
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/data/spark/dim1/'

")

```

I can then select the contents of the new education table to ensure that it looks correct.

```

val resRDD = hiveContext.sql(" SELECT * FROM education ")
resRDD.map( t => t(0)+" "+t(1) ).collect().foreach(println)

```

This gives the expected list of indexes and the education dimension values:

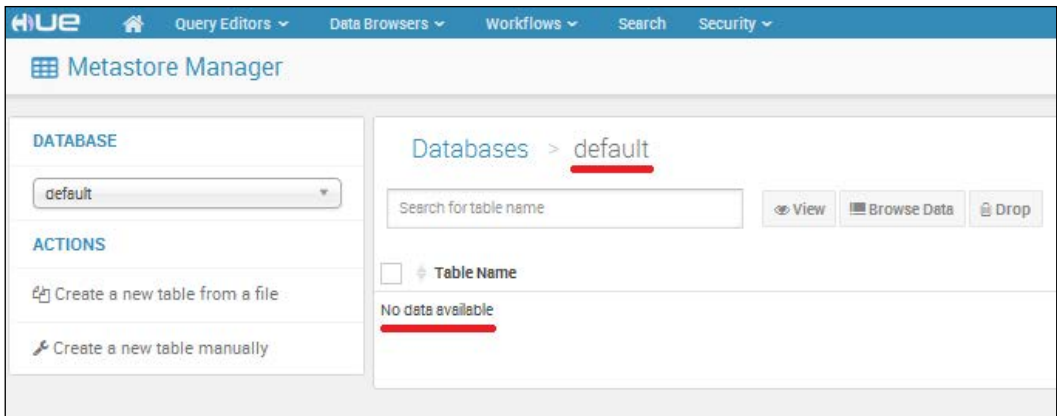
```
1 10th
2 11th
3 12th
.....
16 Some-college
```

So, I have the beginnings of an ETL pipeline. The raw CSV data is being used as external tables, and the dimension tables are being created, which could then be used to convert the dimensions in the raw data to numeric indexes. I have now successfully created a Spark application, which uses a Hive context to connect to a Hive Metastore server, which allows me to create and populate tables.

I have the Hadoop stack Cloudera CDH 5.3 installed on my Linux servers. I am using it for HDFS access while writing this book, and I also have Hive and Hue installed and running (CDH install information can be found at the Cloudera website at <http://cloudera.com/content/cloudera/en/documentation.html>). When I check HDFS for the `adult3` table, which should have been created under `/user/hive/warehouse`, I see the following:

```
[hadoop@hc2nn hive]$ hdfs dfs -ls /user/hive/warehouse/adult3
ls: `/user/hive/warehouse/adult3': No such file or directory
```

The Hive-based table does not exist in the expected place for Hive. I can confirm this by checking the Hue Metastore manager to see what tables exist in the default database. The following figure shows that my default database is currently empty. I have added red lines to show that I am currently looking at the default database, and that there is no data. Clearly, when I run an Apache Spark-based application, with a Hive context, I am connecting to a Hive Metastore server. I know this because the log indicates that this is the case and also, my tables created in this way persist when Apache Spark is restarted.



The Hive context within the application that was just run has used a local Hive Metastore server, and has stored data to a local location; actually in this case under `/tmp` on HDFS. I now want to use the Hive-based Metastore server, so that I can create tables and data in Hive directly. The next section will show how this can be done.

A Hive-based Metastore server

I already mentioned that I am using Cloudera's CDH 5.3 Hadoop stack. I have Hive, HDFS, Hue, and Zookeeper running. I am using Apache Spark 1.3.1 installed under `/usr/local/spark`, in order to create and run applications (I know that CDH 5.3 is released with Spark 1.2, but I wanted to use DataFrames in this instance, which were available in Spark 1.3.x.).

The first thing that I need to do to configure Apache Spark to connect to Hive, is to drop the Hive configuration file called `hive-site.xml` into the Spark configuration directory on all servers where Spark is installed:

```
[hadoop@hc2nn bin]# cp /var/run/cloudera-scm-agent/process/1237-hive-
HIVEMETASTORE/hive-site.xml /usr/local/spark/conf
```

Then, given that I have installed Apache Hive via the CDH Manager to be able to use PostgreSQL, I need to install a PostgreSQL connector JAR for Spark, else it won't know how to connect to Hive, and errors like this will occur:

```
15/06/25 16:32:24 WARN DataNucleus.Connection: BoneCP specified but not
present in CLASSPATH (s)
Caused by: java.lang.RuntimeException: Unable to instantiate org.apache.
hadoop.hive.metastore.
Caused by: java.lang.reflect.InvocationTargetException
```

```
Caused by: javax.jdo.JDOFatalInternalException: Error creating
transactional connection factor
```

```
Caused by: org.datanucleus.exceptions.NucleusException: Attempt to invoke
the "dbcp-builtin" pnectionPool gave an
```

```
error : The specified datastore driver ("org.postgresql.Driver") was not
f. Please check your CLASSPATH
```

```
specification, and the name of the driver.
```

```
Caused by: org.datanucleus.store.rdbms.connectionpool.
DatastoreDriverNotFoundException: The spver
```

```
("org.postgresql.Driver") was not found in the CLASSPATH. Please check
your CLASSPATH specme of the driver.
```

I have stripped that error message down to just the pertinent parts, otherwise it would have been many pages long. I have determined the version of PostgreSQL that I have installed, as follows. It appears to be of version 9.0, determined from the Cloudera parcel-based jar file:

```
[root@hc2nn jars]# pwd ; ls postgresql*
/opt/cloudera/parcels/CDH/jars
postgresql-9.0-801.jdbc4.jar
```

Next, I have used the <https://jdbc.postgresql.org/> website to download the necessary PostgreSQL connector library. I have determined my Java version to be 1.7, as shown here, which affects which version of library to use:

```
[hadoop@hc2nn spark]$ java -version
java version "1.7.0_75"
OpenJDK Runtime Environment (rhel-2.5.4.0.el6_6-x86_64 u75-b13)
OpenJDK 64-Bit Server VM (build 24.75-b04, mixed mode)
```

The site says that if you are using Java 1.7 or 1.8, then you should use the JDBC41 version of the library. So, I have sourced the `postgresql-9.4-1201.jdbc41.jar` file. The next step is to copy this file to the Apache Spark install `lib` directory, as shown here:

```
[hadoop@hc2nn lib]$ pwd ; ls -l postgresql*
/usr/local/spark/lib
-rw-r--r-- 1 hadoop hadoop 648487 Jun 26 13:20 postgresql-9.4-1201.
jdbc41.jar
```

Now, the PostgreSQL library must be added to the Spark `CLASSPATH`, by adding an entry to the file called `compute-classpath.sh`, in the Spark bin directory, as shown here:

```
[hadoop@hc2nn bin]$ pwd ; tail compute-classpath.sh
/usr/local/spark/bin

# add postgresql connector to classpath
appendToClasspath "${assembly_folder}"/postgresql-9.4-1201.jdbc41.jar

echo "$CLASSPATH"
```

In my case, I encountered an error regarding Hive versions between CDH 5.3 Hive and Apache Spark as shown here. I thought that the versions were so close that I should be able to ignore this error:

```
Caused by: MetaException(message:Hive Schema version 0.13.1aa does not
match metastore's schema version 0.13.0)
```

Metastore is not upgraded or corrupt)

I decided, in this case, to switch off schema verification in my Spark version of the `hive-site.xml` file. This had to be done in all the Spark-based instances of this file, and then Spark restarted. The change is shown here; the value is set to `false`:

```
<property>
  <name>hive.metastore.schema.verification</name>
  <value>false</value>
</property>
```

Now, when I run the same set of application-based SQL as the last section, I can create objects in the Apache Hive default database. First, I will create the empty table called `adult2` using the Spark-based Hive context:

```
hiveContext.sql( "

CREATE TABLE IF NOT EXISTS adult2
(
  idx          INT,
  age          INT,
  workclass    STRING,
  fnlwgt       INT,
```

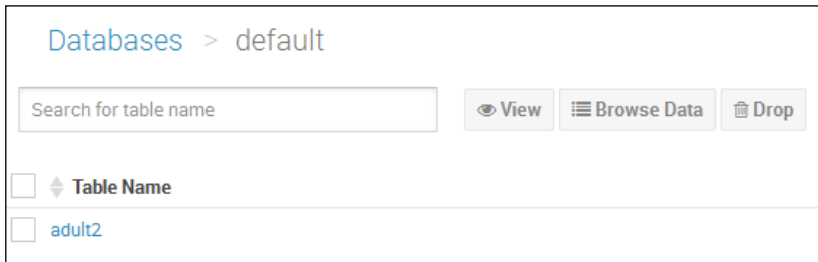
```

education      STRING,
educationnum   INT,
maritalstatus  STRING,
occupation     STRING,
relationship   STRING,
race           STRING,
gender         STRING,
capitalgain    INT,
capitalloss    INT,
nativecountry  STRING,
income        STRING
)

```

")

As you can see, when I run the application and check the Hue Metastore browser, the table `adult2` now exists:



I have shown the table entry previously, and it's structure is obtained by selecting the table entry called `adult2`, in the Hue default database browser:

	Columns	Sample	Properties
	Name	Type	
0	idx	int	
1	age	int	
2	workclass	string	
3	fnlwgt	int	
4	education	string	
5	educationnum	int	
6	maritalstatus	string	
7	occupation	string	
8	relationship	string	
9	race	string	
10	gender	string	
11	capitalgain	int	
12	capitalloss	int	
13	nativecountry	string	
14	income	string	

Now the external table `adult3` Spark based Hive QL can be executed and data access confirmed from Hue. In the last section, the necessary Hive QL was as follows:

```
hiveContext.sql ("
```

```
CREATE EXTERNAL TABLE IF NOT EXISTS adult3
(
    idx            INT,
    age            INT,
    workclass      STRING,
    fnlwgt         INT,
    education      STRING,
    educationnum   INT,
    maritalstatus  STRING,
```

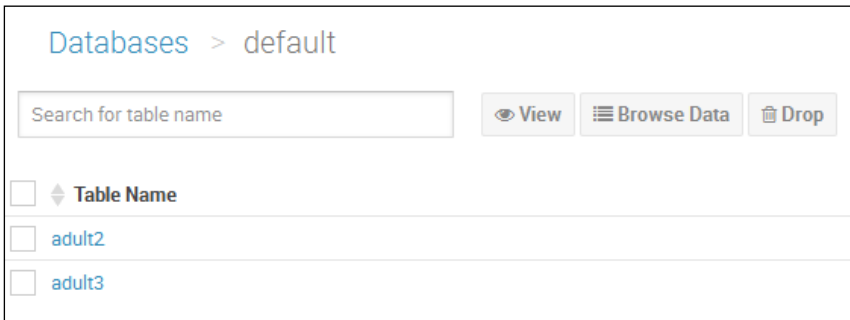
```

        occupation      STRING,
        relationship    STRING,
        race            STRING,
        gender          STRING,
        capitalgain     INT,
        capitalloss     INT,
        nativecountry   STRING,
        income          STRING
    )
    ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
    LOCATION '/data/spark/hive'

")

```

As you can now see, the Hive-based table called `adult3` has been created in the default database by Spark. The following figure is again generated from the Hue Metastore browser:



The following Hive QL has been executed from the Hue Hive query editor. It shows that the `adult3` table is accessible from Hive. I have limited the rows to make the image presentable. I am not worried about the data, only the fact that I can access it:

1 <code>SELECT * FROM default.adult3 LIMIT 5</code>						
Execute Save Save as... Explain or create a New query						
...						
Recent queries Query Log Columns Results Chart						
adult3.idx adult3.age adult3.workclass adult3.fnlwgt adult3.education adult3.educationnum						
0	1	39	State-gov	NULL	Bachelors	NULL
1	2	50	Self-emp-not-inc	NULL	Bachelors	NULL
2	3	38	Private	NULL	HS-grad	NULL
3	4	53	Private	NULL	11th	NULL
4	5	28	Private	NULL	Bachelors	NULL
'''						

The last thing that I will mention in this section which will be useful when using Hive QL from Spark against Hive, will be user-defined functions or UDF's. As an example, I will consider the `row_sequence` function, which is used in the following Scala-based code:

```
hiveContext.sql("
```

```
ADD JAR /opt/cloudera/parcels/CDH-5.3.3-1.cd5.3.3.p0.5/jars/hive-
contrib-0.13.1-cdh5.3.3.jar
```

")

```
hiveContext.sql("
```

```
CREATE TEMPORARY FUNCTION row_sequence as 'org.apache.hadoop.hive.
contrib.udf.UDFRowSequence';
```

")

```
val resRDD = hiveContext.sql("
```

```
SELECT row_sequence(),t1.edu FROM
```

```
( SELECT DISTINCT education AS edu FROM adult3 ) t1
ORDER BY t1.edu

")
```

Either existing, or your own, JAR-based libraries can be made available to your Spark Hive session via the `ADD JAR` command. Then, the functionality within that library can be registered as a temporary function with `CREATE TEMPORARY FUNCTION` using the package-based class name. Then, the new function name can be incorporated in Hive QL statements.

This chapter has managed to connect an Apache Spark-based application to Hive, and run Hive QL against Hive, so that table and data changes persist in Hive. But why is this important? Well, Spark is an in-memory parallel processing system. It is an order faster than Hadoop-based Map Reduce in processing speed. Apache Spark can now be used as a processing engine, whereas the Hive data warehouse can be used for storage. Fast in-memory Spark-based processing speed coupled with big data scale structured data warehouse storage available in Hive.

Summary

This chapter started by explaining the Spark SQL context, and file I/O methods. It then showed that Spark and HDFS-based data could be manipulated, as both `DataFrames` with SQL-like methods and with Spark SQL by registering temporary tables. Next, user-defined functions were introduced to show that the functionality of Spark SQL could be extended by creating new functions to suit your needs, registering them as UDF's, and then calling them in SQL to process data.

Finally, the Hive context was introduced for use in Apache Spark. Remember that the Hive context in Spark offers a super set of the functionality of the SQL context. I understand that over time, the SQL context is going to be extended to match the Hive Context functionality. Hive QL data processing in Spark using a Hive context was shown using both, a local Hive, and a Hive-based Metastore server. I believe that the latter configuration is better, as the tables are created, and data changes persist in your Hive instance.

In my case, I used Cloudera CDH 5.3, which used Hive 0.13, PostgreSQL, ZooKeeper, and Hue. I also used Apache Spark version 1.3.1. The configuration setup that I have shown you is purely for this configuration. If you wanted to use MySQL, for instance, you would need to research the necessary changes. A good place to start would be the `user@spark.apache.org` mailing list.

Finally, I would say that Apache Spark Hive context configuration, with Hive-based storage, is very useful. It allows you to use Hive as a big data scale data warehouse, with Apache Spark for fast in-memory processing. It offers you the ability to manipulate your data with not only the Spark-based modules (MLlib, SQL, GraphX, and Stream), but also other Hadoop-based tools, making it easier to create ETL chains.

The next chapter will examine the Spark graph processing module, GraphX, it will also investigate the Neo4J graph database, and the MazeRunner application.

5

Apache Spark GraphX

In this chapter, I want to examine the Apache Spark GraphX module, and graph processing in general. I also want to briefly examine graph-based storage by looking at the graph database called Neo4j. So, this chapter will cover the following topics:

- GraphX coding
- Mazerunner for Neo4j

The GraphX coding section, written in Scala, will provide a series of graph coding examples. The work carried out on the experimental Mazerunner product by Kenny Bastani, which I will also examine, ties the two topics together in one practical example. It provides an example prototype-based on Docker to replicate data between Apache Spark GraphX, and Neo4j storage.

Before writing code in Scala to use the Spark GraphX module, I think it would be useful to provide an overview of what a graph actually is in terms of graph processing. The following section provides a brief introduction using a couple of simple graphs as examples.

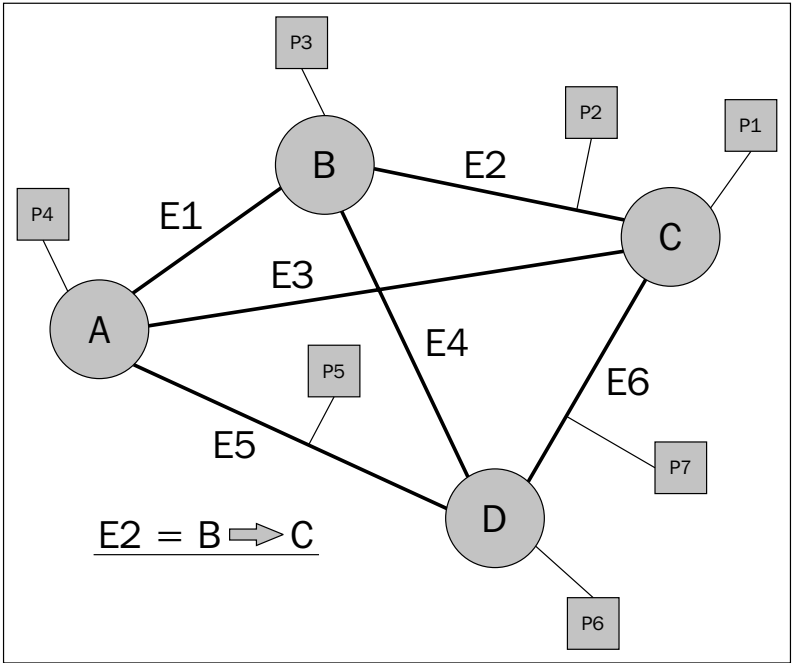
Overview

A graph can be considered to be a data structure, which consists of a group of vertices, and edges that connect them. The vertices or nodes in the graph can be objects or perhaps, people, and the edges are the relationships between them. The edges can be directional, meaning that the relationship operates from one node to the next. For instance, node A is the father of node B.

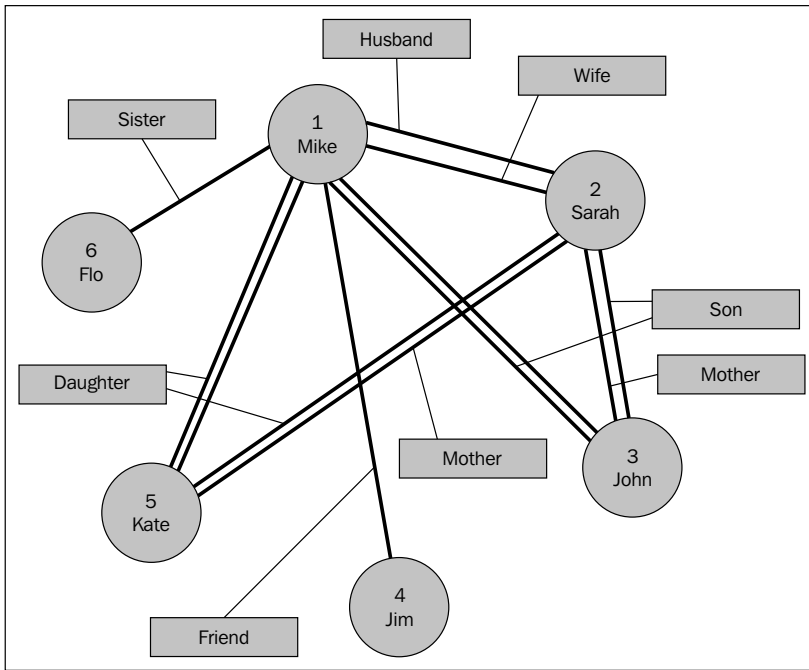
In the following diagram, the circles represent the vertices or nodes (**A** to **D**), whereas the thick lines represent the edges, or relationships between them (**E1** to **E6**). Each node, or edge may have properties, and these values are represented by the associated grey squares (**P1** to **P7**).

So, if a graph represented a physical route map for route finding, then the edges might represent minor roads or motorways. The nodes would be motorway junctions, or road intersections. The node and edge properties might be the road type, speed limit, distance, and the cost and grid locations.

There are many types of graph implementation, but some examples are fraud modeling, financial currency transaction modeling, social modeling (as in friend-to-friend connections on Facebook), map processing, web processing, and page ranking.



The previous diagram shows a generic example of a graph with associated properties. It also shows that the edge relationships can be directional, that is, the **E2** edge acts from node **B** to node **C**. However, the following example uses family members, and the relationships between them to create a graph. Note that there can be multiple edges between two nodes or vertices. For instance, the husband-and-wife relationships between **Mike** and **Sarah**. Also, it is possible that there could be multiple properties on a node or edge.



So, in the previous example, the **Sister** property acts from node 6 **Flo**, to node 1, **Mike**. These are simple graphs to explain the structure of a graph, and the element nature. Real graph applications can reach extreme sizes, and require both, distributed processing, and storage to enable them to be manipulated. Facebook is able to process graphs, containing over 1 trillion edges using **Apache Giraph** (source: Avery Ching-Facebook). Giraph is an Apache Hadoop eco-system tool for graph processing, which has historically based its processing on Map Reduce, but now uses TinkerPop, which will be introduced in *Chapter 6, Graph-based Storage*. Although this book concentrates on Apache Spark, the number of edges provides a very impressive indicator of the size that a graph can reach.

In the next section, I will examine the use of the Apache Spark GraphX module using Scala.

GraphX coding

This section will examine Apache Spark GraphX programming in Scala, using the family relationship graph data sample, which was shown in the last section. This data will be stored on HDFS, and will be accessed as a list of vertices and edges. Although this data set is small, the graphs that you build in this way could be very large. I have used HDFS for storage, because if your graph scales to the big data scale, then you will need some type of distributed and redundant storage. As this chapter shows by way of example, that could be HDFS. Using the Apache Spark SQL module, the storage could also be Apache Hive; see *Chapter 4, Apache Spark SQL*, for details.

Environment

I have used the `hadoop` Linux account on the server `hc2nn` to develop the Scala-based GraphX code. The structure for SBT compilation follows the same pattern as the previous examples, with the code tree existing in a subdirectory named `graphx`, where an `sbt` configuration file called `graph.sbt` resides:

```
[hadoop@hc2nn graphx]$ pwd
/home/hadoop/spark/graphx
```

```
[hadoop@hc2nn graphx]$ ls
src      graph.sbt      project      target
```

The source code lives, as expected, under a subtree of this level called `src/main/scala`, and contains five code samples:

```
[hadoop@hc2nn scala]$ pwd
/home/hadoop/spark/graphx/src/main/scala
```

```
[hadoop@hc2nn scala]$ ls
graph1.scala  graph2.scala  graph3.scala  graph4.scala  graph5.scala
```

In each graph-based example, the Scala file uses the same code to load data from HDFS, and to create a graph; but then, each file provides a different facet of GraphX-based graph processing. As a different Spark module is being used in this chapter, the `sbt` configuration file `graph.sbt` has been changed to support this work:

```
[hadoop@hc2nn graphx]$ more graph.sbt
```

```
name := "Graph X"
```

```

version := "1.0"

scalaVersion := "2.10.4"

libraryDependencies += "org.apache.hadoop" % "hadoop-client" % "2.3.0"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.0.0"

libraryDependencies += "org.apache.spark" %% "spark-graphx" % "1.0.0"

// If using CDH, also add Cloudera repo
resolvers += "Cloudera Repository" at https://repository.cloudera.com/artifactory/cloudera-repos/

```

The contents of the `graph.sbt` file are shown previously, via the Linux `more` command. There are only two changes here to note from previous examples—the value of `name` has changed to represent the content. Also, more importantly, the Spark GraphX 1.0.0 library has been added as a library dependency.

Two data files have been placed on HDFS, under the `/data/spark/graphx/` directory. They contain the data that will be used for this section in terms of the vertices, and edges that make up a graph. As the Hadoop file system `ls` command shows next, the files are called `graph1_edges.csv` and `graph1_vertex.csv`:

```

[hadoop@hc2nn scala]$ hdfs dfs -ls /data/spark/graphx
Found 2 items
-rw-r--r--   3 hadoop supergroup      129 2015-03-01 13:52 /data/spark/graphx/graph1_edges.csv
-rw-r--r--   3 hadoop supergroup       59 2015-03-01 13:52 /data/spark/graphx/graph1_vertex.csv

```

The `vertex` file, shown next, via a Hadoop file system `cat` command, contains just six lines, representing the graph used in the last section. Each vertex represents a person, and has a vertex ID number, a name and an age value:

```

[hadoop@hc2nn scala]$ hdfs dfs -cat /data/spark/graphx/graph1_vertex.csv
1,Mike,48
2,Sarah,45
3,John,25
4,Jim,53
5,Kate,22
6,Flo,52

```

The edge file contains a set of directed edge values in the form of source vertex ID, destination vertex ID, and relationship. So, record one forms a Sister relationship between Flo and Mike:

```
[hadoop@hc2nn scala]$ hdfs dfs -cat /data/spark/graphx/graph1_edges.csv
6,1,Sister
1,2,Husband
2,1,Wife
5,1,Daughter
5,2,Daughter
3,1,Son
3,2,Son
4,1,Friend
1,5,Father
1,3,Father
2,5,Mother
2,3,Mother
```

Having explained the sbt environment, and the HDFS-based data, we are now ready to examine some of the GraphX code samples. As in the previous examples, the code can be compiled, and packaged as follows from the `graphx` subdirectory. This creates a JAR called `graph-x_2.10-1.0.jar` from which the example applications can be run:

```
[hadoop@hc2nn graphx]$ pwd
/home/hadoop/spark/graphx
```

```
[hadoop@hc2nn graphx]$ sbt package
```

```
Loading /usr/share/sbt/bin/sbt-launch-lib.bash
[info] Set current project to Graph X (in build file:/home/hadoop/spark/graphx/)
[info] Compiling 5 Scala sources to /home/hadoop/spark/graphx/target/scala-2.10/classes...
[info] Packaging /home/hadoop/spark/graphx/target/scala-2.10/graph-x_2.10-1.0.jar ...
[info] Done packaging.
[success] Total time: 30 s, completed Mar 3, 2015 5:27:10 PM
```

Creating a graph

This section will explain the generic Scala code, up to the point of creating a GraphX graph, from the HDFS-based data. This will save time, as the same code is reused in each example. Once this is explained, I will concentrate on the actual graph-based manipulation in each code example:

The generic code starts by importing the Spark context, graphx, and RDD functionality for use in the Scala code:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
```

Then, an application is defined, which extends the App class, and the application name changes, for each example, from graph1 to graph5. This application name will be used when running the application using spark-submit:

```
object graph1 extends App
{
```

The data files are defined in terms of the HDFS server and port, the path that they reside under in HDFS and their file names. As already mentioned, there are two data files that contain the vertex and edge information:

```
val hdfsServer = "hdfs://hc2nn.semtech-solutions.co.nz:8020"
val hdfsPath   = "/data/spark/graphx/"
val vertexFile = hdfsServer + hdfsPath + "graph1_vertex.csv"
val edgeFile   = hdfsServer + hdfsPath + "graph1_edges.csv"
```

The Spark Master URL is defined, as is the application name, which will appear in the Spark user interface when the application runs. A new Spark configuration object is created, and the URL and name are assigned to it:

```
val sparkMaster = "spark://hc2nn.semtech-solutions.co.nz:7077"
val appName     = "Graph 1"
val conf        = new SparkConf()
conf.setMaster(sparkMaster)
conf.setAppName(appName)
```

A new Spark context is created using the configuration that was just defined:

```
val sparkCxt = new SparkContext(conf)
```

The vertex information from the HDFS-based file is then loaded into an RDD-based structure called `vertices` using the `sparkCxt.textFile` method. The data is stored as a long `VertexId`, and strings to represent the person's name and age. The data lines are split by commas as this is CSV based-data:

```
val vertices: RDD[(VertexId, (String, String))] =
  sparkCxt.textFile(vertexFile).map { line =>
    val fields = line.split(",")
    ( fields(0).toLong, ( fields(1), fields(2) ) )
  }
```

Similarly, the HDFS-based edge data is loaded into an RDD-based data structure called `edges`. The CSV-based data is again split by comma values. The first two data values are converted into Long values, as they represent the source and destination vertex ID's. The final value, representing the relationship of the edge, is left as a string. Note that each record in the RDD structure `edges` is actually now an `Edge` record:

```
val edges: RDD[Edge[String]] =
  sparkCxt.textFile(edgeFile).map { line =>
    val fields = line.split(",")
    Edge(fields(0).toLong, fields(1).toLong, fields(2))
  }
```

A default value is defined in case a connection, or a vertex is missing, then the graph is constructed from the RDD-based structures—`vertices`, `edges`, and the default record:

```
val default = ("Unknown", "Missing")
val graph = Graph(vertices, edges, default)
```

This creates a `GraphX`-based structure called `graph`, which can now be used for each of the examples. Remember that although these data samples are small, you can create extremely large graphs using this approach. Many of these algorithms are iterative applications, for instance, PageRank and Triangle Count, and as a result, the programs will generate many iterative Spark jobs.

Example 1 – counting

The graph has been loaded, and we know the data volumes in the data files, but what about the data content in terms of vertices, and edges in the actual graph itself? It is very simple to extract this information by using the `vertices`, and the `edges` count function as shown here:

```
println( "vertices : " + graph.vertices.count )
println( "edges      : " + graph.edges.count )
```

Running the `graph1` example, using the example name and the JAR file created previously, will provide the count information. The master URL is supplied to connect to the Spark cluster, and some default parameters are supplied for the executor memory, and the total executor cores:

```
spark-submit \
  --class graph1 \
  --master spark://hc2nn.semtech-solutions.co.nz:7077 \
  --executor-memory 700M \
  --total-executor-cores 100 \
  /home/hadoop/spark/graphx/target/scala-2.10/graph-x_2.10-1.0.jar
```

The Spark cluster job called `graph1` provides the following output, which is as expected and also, it matches the data files:

```
vertices : 6
edges    : 12
```

Example 2 – filtering

What happens if we need to create a subgraph from the main graph, and filter by the person's age or relationships? The example code from the second example Scala file, `graph2`, shows how this can be done:

```
val c1 = graph.vertices.filter { case (id, (name, age)) => age.
  toLong > 40 }.count

val c2 = graph.edges.filter { case Edge(from, to, property)
  => property == "Father" | property == "Mother" }.count

println( "Vertices count : " + c1 )
println( "Edges      count : " + c2 )
```

The two example counts have been created from the main graph. The first filters the person-based vertices on the age, only taking those people who are greater than 40 years old. Notice that the age value, which was stored as a string, has been converted into a long for comparison. The previous second example filters the edges on the relationship property of `Mother` or `Father`. The two count values: `c1` and `c2` are created, and printed as the Spark output shows here:

```
Vertices count : 4
Edges      count : 4
```

Example 3 – PageRank

The PageRank algorithm provides a ranking value for each of the vertices in a graph. It makes the assumption that the vertices that are connected to the most edges are the most important ones. Search engines use PageRank to provide ordering for the page display during a web search:

```
val tolerance = 0.0001
val ranking = graph.pageRank(tolerance).vertices
val rankByPerson = vertices.join(ranking).map {
  case (id, ( (person,age) , rank )) => (rank, id, person)
}
```

The previous example code creates a `tolerance` value, and calls the `graph pageRank` method using it. The vertices are then ranked into a new value `ranking`. In order to make the ranking more meaningful the ranking values are joined with the original vertices RDD. The `rankByPerson` value then contains the rank, vertex ID, and person's name.

The PageRank result, held in `rankByPerson`, is then printed record by record, using a case statement to identify the record contents, and a format statement to print the contents. I did this, because I wanted to define the format of the rank value which can vary:

```
rankByPerson.collect().foreach {
  case (rank, id, person) =>
    println ( f"Rank $rank%1.2f id $id person $person")
}
```

The output from the application is then shown here. As expected, Mike and Sarah have the highest rank, as they have the most relationships:

```
Rank 0.15 id 4 person Jim
Rank 0.15 id 6 person Flo
Rank 1.62 id 2 person Sarah
Rank 1.82 id 1 person Mike
Rank 1.13 id 3 person John
Rank 1.13 id 5 person Kate
```

Example 4 – triangle counting

The triangle count algorithm provides a vertex-based count of the number of triangles, associated with this vertex. For instance, vertex *Mike* (1) is connected to *Kate* (5), who is connected to *Sarah* (2); *Sarah* is connected to *Mike* (1) and so, a triangle is formed. This can be useful for route finding, where minimum, triangle-free, spanning tree graphs need to be generated for route planning.

The code to execute a triangle count, and print it, is simple, as shown next. The graph `triangleCount` method is executed for the graph vertices. The result is saved in the value `tCount`, and then printed:

```
val tCount = graph.triangleCount().vertices
println( tCount.collect().mkString("\n") )
```

The results of the application job show that the vertices called, *Flo* (4) and *Jim* (6), have no triangles, whereas *Mike* (1) and *Sarah* (2) have the most, as expected, as they have the most relationships:

```
(4,0)
(6,0)
(2,4)
(1,4)
(3,2)
(5,2)
```

Example 5 – connected components

When a large graph is created from the data, it might contain unconnected subgraphs, that is, subgraphs that are isolated from each other, and contain no bridging or connecting edges between them. This algorithm provides a measure of this connectivity. It might be important, depending upon your processing, to know that all the vertices are connected.

The Scala code, for this example, calls two graph methods: `connectedComponents`, and `stronglyConnectedComponents`. The strong method required a maximum iteration count, which has been set to 1000. These counts are acting on the graph vertices:

```
val iterations = 1000
val connected = graph.connectedComponents().vertices
val connectedS =
graph.stronglyConnectedComponents(iterations).vertices
```

The vertex counts are then joined with the original vertex records, so that the connection counts can be associated with the vertex information, such as the person's name:

```
val connByPerson = vertices.join(connected).map {  
  case (id, ( (person,age) , conn )) => (conn, id, person)  
}  
  
val connByPersonS = vertices.join(connectedS).map {  
  case (id, ( (person,age) , conn )) => (conn, id, person)  
}
```

The results are then output using a case statement, and formatted printing:

```
connByPerson.collect().foreach {  
  case (conn, id, person) =>  
    println ( f"Weak $conn $id $person" )  
}
```

As expected for the `connectedComponents` algorithm, the results show that for each vertex, there is only one component. This means that all the vertices are the members of a single graph, as the graph diagram earlier in the chapter showed:

```
Weak 1 4 Jim  
Weak 1 6 Flo  
Weak 1 2 Sarah  
Weak 1 1 Mike  
Weak 1 3 John  
Weak 1 5 Kate
```

The `stronglyConnectedComponents` method gives a measure of the connectivity in a graph, taking into account the direction of the relationships between them. The results for the `stronglyConnectedComponents` algorithm output is as follows:

```
connByPersonS.collect().foreach {  
  case (conn, id, person) =>  
    println ( f"Strong $conn $id $person" )  
}
```

You might notice from the graph that the relationships, `Sister` and `Friend`, act from vertices `Flo` (6) and `Jim` (4), to `Mike` (1) as the edge and vertex data shows here:

```
6,1,Sister  
4,1,Friend
```

1, Mike, 48

4, Jim, 53

6, Flo, 52

So, the strong method output shows that for most vertices, there is only one graph component signified by the 1 in the second column. However, vertices 4 and 6 are not reachable due to the direction of their relationship, and so they have a vertex ID instead of a component ID:

Strong 4 4 Jim

Strong 6 6 Flo

Strong 1 2 Sarah

Strong 1 1 Mike

Strong 1 3 John

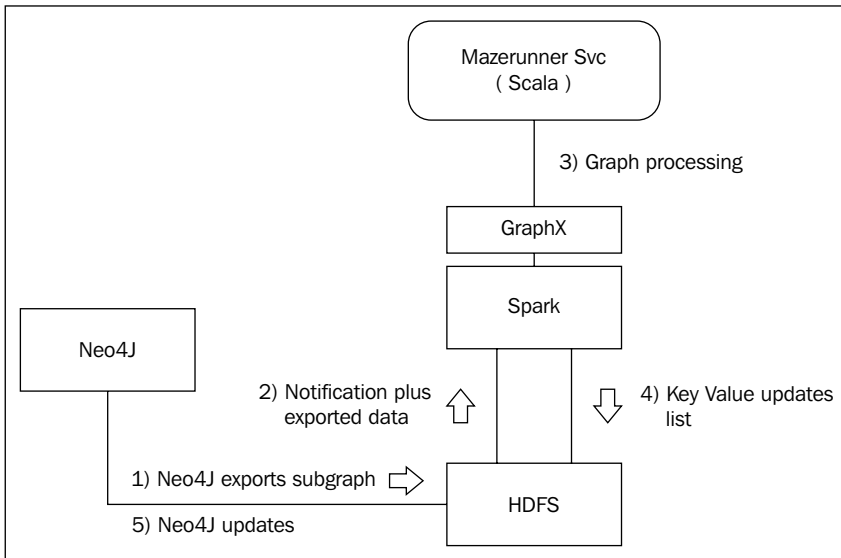
Strong 1 5 Kate

Mazerunner for Neo4j

In the previous sections, you have been shown how to write Apache Spark graphx code in Scala to process the HDFS-based graph data. You have been able to execute the graph-based algorithms, such as PageRank, and triangle counting. However, this approach has a limitation. Spark does not have storage, and storing graph-based data in the flat files on HDFS does not allow you to manipulate it in its place of storage. For instance, if you had data stored in a relational database, you could use SQL to interrogate it in place. Databases such as Neo4j are graph databases. This means that their storage mechanisms and data access language act on graphs. In this section, I want to take a look at the work done on Mazerunner, created as a GraphX Neo4j processing prototype by Kenny Bastani.

The following figure describes the Mazerunner architecture. It shows that data in Neo4j is exported to HDFS, and processed by GraphX via a notification process. The GraphX data updates are then saved back to HDFS as a list of key value updates. These changes are then propagated to Neo4j to be stored. The algorithms in this prototype architecture are accessed via a Rest based HTTP URL, which will be shown later. The point here though, is that algorithms can be run via processing in graphx, but the data changes can be checked via Neo4j database cypher language queries. Kenny's work and further details can be found at: <http://www.kennybastani.com/2014/11/using-apache-spark-and-neo4j-for-big.html>.

This section will be dedicated to explaining the Mazerunner architecture, and will show, with the help of an example, how it can be used. This architecture provides a unique example of GraphX-based processing, coupled with graph-based storage.



Installing Docker

The process for installing the Mazerunner example code is described via <https://github.com/kbastani/neo4j-mazerunner>.

I have used the 64 bit Linux Centos 6.5 machine `hc1r1m1` for the install. The Mazerunner example uses the Docker tool, which creates virtual containers with a small foot print for running HDFS, Neo4j, and Mazerunner in this example. First, I must install Docker. I have done this, as follows, using the Linux root user via `yum` commands. The first command installs the `docker-io` module (the docker name was already used for CentOS 6.5 by another application):

```
[root@hc1r1m1 bin]# yum -y install docker-io
```

I needed to enable the `public_ol6_latest` repository, and install the `device-mapper-event-libs` package, as I found that my current `lib-device-mapper`, which I had installed, wasn't exporting the symbol `Base` that Docker needed. I executed the following commands as root:

```
[root@hc1r1m1 ~]# yum-config-manager --enable public_ol6_latest
```

```
[root@hc1r1m1 ~]# yum install device-mapper-event-libs
```

The actual error that I encountered was as follows:

```
/usr/bin/docker: relocation error: /usr/bin/docker: symbol dm_task_get_info_with_deferred_remove, version Base not defined in file libdevmapper.so.1.02 with link time reference
```

I can then check that Docker will run by checking the Docker version number with the following call:

```
[root@hclrlml ~]# docker version
Client version: 1.4.1
Client API version: 1.16
Go version (client): go1.3.3
Git commit (client): 5bc2ff8/1.4.1
OS/Arch (client): linux/amd64
Server version: 1.4.1
Server API version: 1.16
Go version (server): go1.3.3
Git commit (server): 5bc2ff8/1.4.1
```

I can start the Linux docker service using the following service command. I can also force Docker to start on Linux server startup using the following `chkconfig` command:

```
[root@hclrlml bin]# service docker start
[root@hclrlml bin]# chkconfig docker on
```

The three Docker images (HDFS, Mazerunner, and Neo4j) can then be downloaded. They are large, so this may take some time:

```
[root@hclrlml ~]# docker pull sequenceiq/hadoop-docker:2.4.1
Status: Downloaded newer image for sequenceiq/hadoop-docker:2.4.1
```

```
[root@hclrlml ~]# docker pull kbastani/docker-neo4j:latest
Status: Downloaded newer image for kbastani/docker-neo4j:latest
```

```
[root@hclrlml ~]# docker pull kbastani/neo4j-graph-analytics:latest
Status: Downloaded newer image for kbastani/neo4j-graph-analytics:latest
```

Once downloaded, the Docker containers can be started in the order; HDFS, Mazerunner, and then Neo4j. The default Neo4j movie database will be loaded and the Mazerunner algorithms run using this data. The HDFS container starts as follows:

```
[root@hclrlml ~]# docker run -i -t --name hdfs sequenceiq/hadoop-docker:2.4.1 /etc/bootstrap.sh -bash
```

```
Starting sshd: [ OK ]
Starting namenodes on [26d939395e84]
26d939395e84: starting namenode, logging to /usr/local/hadoop/logs/hadoop-root-namenode-26d939395e84.out
localhost: starting datanode, logging to /usr/local/hadoop/logs/hadoop-root-datanode-26d939395e84.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop/logs/hadoop-root-secondarynamenode-26d939395e84.out
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop/logs/yarn--resourcemanager-26d939395e84.out
localhost: starting nodemanager, logging to /usr/local/hadoop/logs/yarn-root-nodemanager-26d939395e84.out
```

The Mazerunner service container starts as follows:

```
[root@hclrlml ~]# docker run -i -t --name mazerunner --link hdfs:hdfs kbastani/neo4j-graph-analytics
```

The output is long, so I will not include it all here, but you will see no errors. There also comes a line, which states that the install is waiting for messages:

```
[*] Waiting for messages. To exit press CTRL+C
```

In order to start the Neo4j container, I need the install to create a new Neo4j database for me, as this is a first time install. Otherwise on restart, I would just supply the path of the database directory. Using the `link` command, the Neo4j container is linked to the HDFS and Mazerunner containers:

```
[root@hclrlml ~]# docker run -d -P -v /home/hadoop/neo4j/data:/opt/data --name graphdb --link mazerunner:mazerunner --link hdfs:hdfs kbastani/docker-neo4j
```

By checking the `neo4j/data` path, I can now see that a database directory, named `graph.db` has been created:

```
[root@hc1r1m1 data]# pwd
/home/hadoop/neo4j/data
```

```
[root@hc1r1m1 data]# ls
graph.db
```

I can then use the following `docker inspect` command, which the container-based IP address and the Docker-based Neo4j container is making available. The `inspect` command supplies me with the local IP address that I will need to access the Neo4j container. The `curl` command, along with the port number, which I know from Kenny's website, will default to 7474, shows me that the Rest interface is running:

```
[root@hc1r1m1 data]# docker inspect --format="{ {.NetworkSettings.
IPAddress}}" graphdb
172.17.0.5
```

```
[root@hc1r1m1 data]# curl 172.17.0.5:7474
{
  "management" : "http://172.17.0.5:7474/db/manage/",
  "data" : "http://172.17.0.5:7474/db/data/"
}
```

The Neo4j browser

The rest of the work in this section will now be carried out using the Neo4j browser URL, which is as follows:

```
http://172.17.0.5:7474/browser.
```

This is a local, Docker-based IP address that will be accessible from the `hc1r1m1` server. It will not be visible on the rest of the local intranet without further network configuration.

This will show the default Neo4j browser page. The Movie graph can be installed by following the movie link here, selecting the Cypher query, and executing it.



2.1.7

Getting started

by learning about Neo4j...

- ▶ [Intro](#) - a guided tour
- ▶ [Concepts](#) - GraphDB 101
- ▶ [Cypher](#) - query language

Jump into code

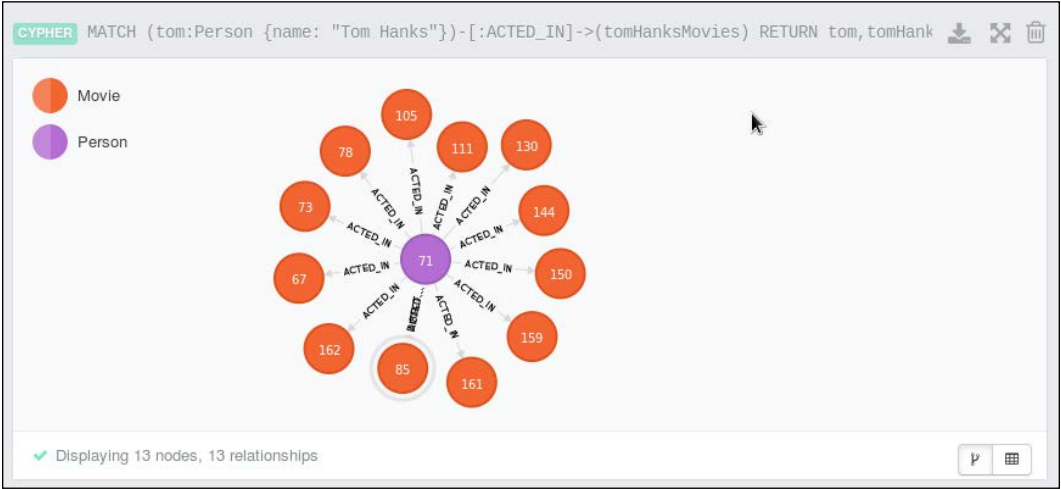
with a mini graph app...

- ▶ [The Movie Graph](#)

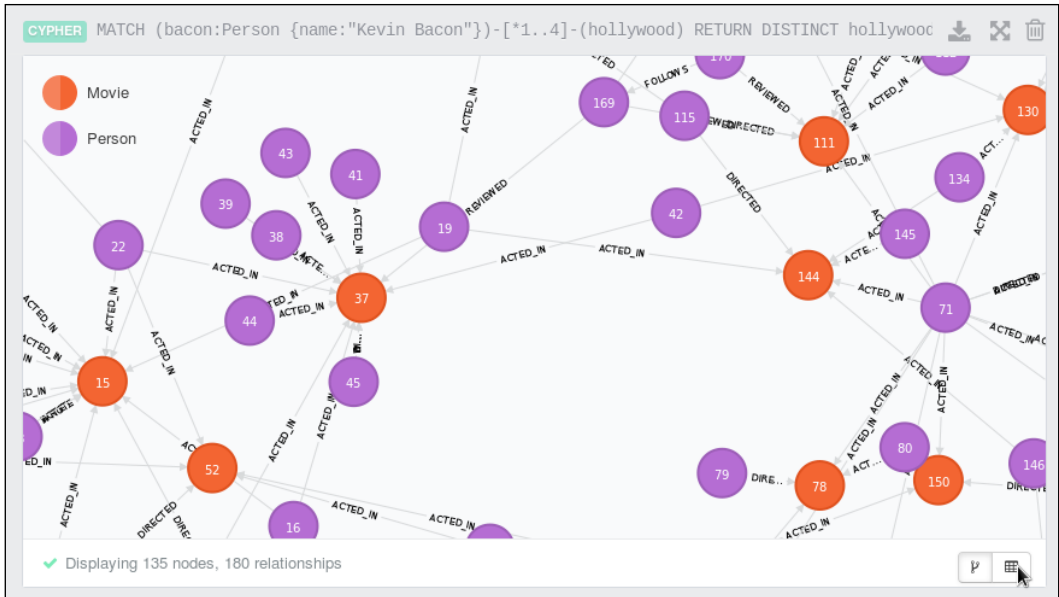
📌 We love that you use Neo4j! Talk to us at GraphConnect Europe in London: 30% off with NEO4JUSER

Copyright © Neo Technology 2002-2015

The data can then be interrogated using Cypher queries, which will be examined in more depth in the next chapter. The following figures are supplied along with their associated Cypher queries, in order to show that the data can be accessed as graphs that are displayed visually. The first graph shows a simple Person to Movie relationship, with the relationship details displayed on the connecting edges.



The second graph, provided as a visual example of the power of Neo4j, shows a far more complex cypher query, and resulting graph. This graph states that it contains 135 nodes and 180 relationships. These are relatively small numbers in processing terms, but it is clear that the graph is becoming complex.



The following figures show the Mazerunner example algorithms being called via an HTTP Rest URL. The call is defined by the algorithm to be called, and the attribute that it is going to act upon within the graph:

```
http://localhost:7474/service/mazerunner/analysis/{algorithm}/
{attribute}.
```

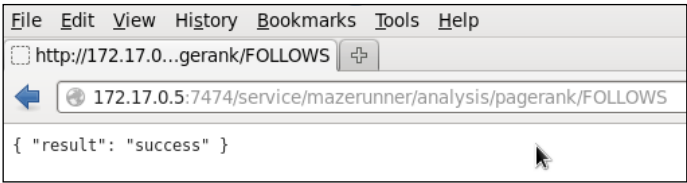
So for instance, as the next section will show, this generic URL can be used to run the PageRank algorithm by setting `algorithm=pagerank`. The algorithm will operate on the `follows` relationship by setting `attribute=FOLLOWS`. The next section will show how each Mazerunner algorithm can be run along with an example of the Cypher output.

The Mazerunner algorithms

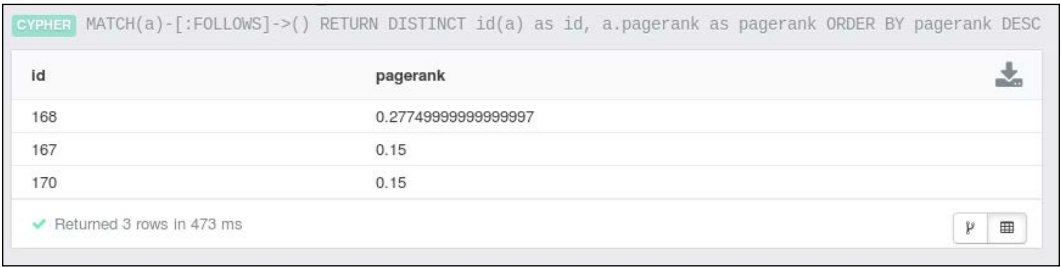
This section shows how the Mazerunner example algorithms may be run using the Rest based HTTP URL, which was shown in the last section. Many of these algorithms have already been examined, and coded in this chapter. Remember that the interesting thing occurring in this section is that data starts in Neo4j, it is processed on Spark with GraphX, and then is updated back into Neo4j. It looks simple, but there are underlying processes doing all of the work. In each example, the attribute that the algorithm has added to the graph is interrogated via a Cypher query. So, each example isn't so much about the query, but that the data update to Neo4j has occurred.

The PageRank algorithm

The first call shows the PageRank algorithm, and the PageRank attribute being added to the movie graph. As before, the PageRank algorithm gives a rank to each vertex, depending on how many edge connections it has. In this case, it is using the `FOLLOWS` relationship for processing.

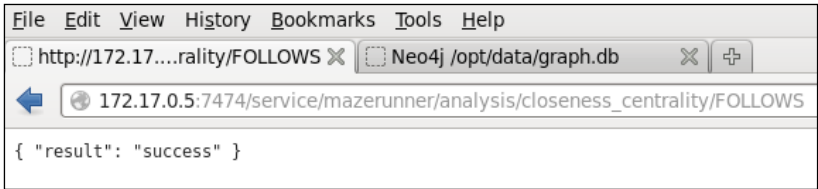


The following image shows a screenshot of the PageRank algorithm result. The text at the top of the image (starting with `MATCH`) shows the cypher query, which proves that the PageRank property has been added to the graph.



The closeness centrality algorithm

The closeness algorithm attempts to determine the most important vertices in the graph. In this case, the `closeness` attribute has been added to the graph.



The following image shows a screenshot of the closeness algorithm result. The text at the top of the image (starting with `MATCH`) shows the Cypher query, which proves that the `closeness_centrality` property has been added to the graph. Note that an alias called `closeness` has been used in this Cypher query, to represent the `closeness_centrality` property, and so the output is more presentable.



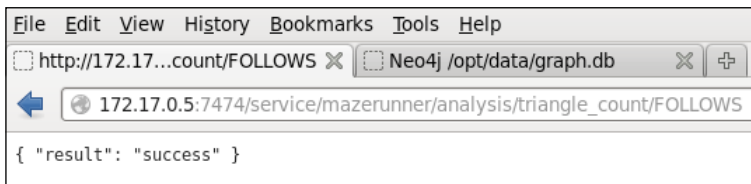
CYPHER MATCH(a)-[:FOLLOWS]->() RETURN DISTINCT id(a) as id, a.closeness_centrality as closeness ORDER BY closeness DESC

id	closeness
168	2
167	0
170	0

Returned 3 rows in 482 ms

The triangle count algorithm

The `triangle_count` algorithm has been used to count triangles associated with vertices. The `FOLLOWS` relationship has been used, and the `triangle_count` attribute has been added to the graph.



The following image shows a screenshot of the triangle algorithm result. The text at the top of the image (starting with `MATCH`) shows the cypher query, which proves that the `triangle_count` property has been added to the graph. Note that an alias called `tcount` has been used in this cypher query, to represent the `triangle_count` property, and so the output is more presentable.



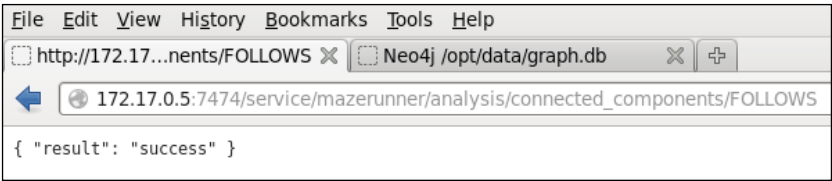
CYPHER MATCH(a)-[:FOLLOWS]->() RETURN DISTINCT id(a) as id, a.triangle_count as tcount ORDER BY tcount DESC

id	tcount
167	0
168	0
170	0

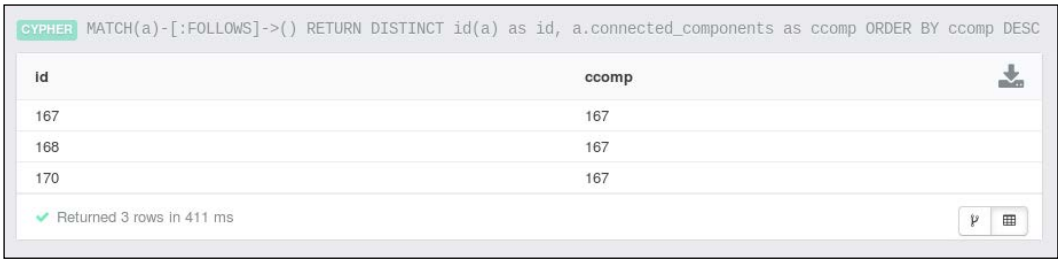
Returned 3 rows in 444 ms

The connected components algorithm

The connected components algorithm is a measure of how many actual components exist in the graph data. For instance, the data might contain two subgraphs with no routes between them. In this case, the `connected_components` attribute has been added to the graph.

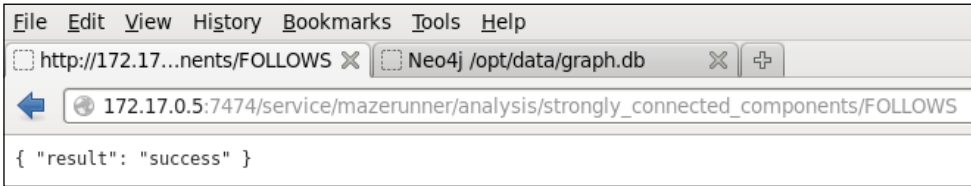


The following image shows a screenshot of the connected component algorithm result. The text at the top of the image (starting with `MATCH`) shows the cypher query, which proves that the `connected_components` property has been added to the graph. Note that an alias called `ccomp` has been used in this cypher query, to represent the `connected_components` property, and so the output is more presentable.



The strongly connected components algorithm

The strongly connected components algorithm is very similar to the connected components algorithm. Subgraphs are created from the graph data using the directional `FOLLOWS` relationship. Multiple subgraphs are created until all the graph components are used. These subgraphs form the strongly connected components. As seen here, a `strongly_connected_components` attribute has been added to the graph:



The following image shows a screenshot of the strongly connected component algorithm result. The text at the top of the image (starting with `MATCH`) shows the cypher query, which proves that the `strongly_connected_components` connected component property has been added to the graph. Note that an alias called **sccomp** has been used in this cypher query, to represent the `strongly_connected_components` property, and so the output is more presentable.



The screenshot shows a Cypher query interface with the following query: `MATCH(a)-[:FOLLOWS]->( ) RETURN DISTINCT id(a) as id, a.strongly_connected_components as sccomp ORDER BY sccomp DESC`. The result is a table with two columns: `id` and `sccomp`. It contains three rows of data. At the bottom, a status bar indicates 'Returned 3 rows in 619 ms'.

id	sccomp
170	170
168	168
167	167

Summary

This chapter has shown, with the help of examples, how the Scala-based code can be used to call GraphX algorithms in Apache Spark. Scala has been used, because it requires less code to develop the examples, which saves time. A Scala-based shell can be used, and the code can be compiled into Spark applications. Examples of the application compilation and configuration have been supplied using the SBT tool. The configuration and the code examples from this chapter will also be available for download with the book.

Finally, the Mazerunner example architecture (developed by Kenny Bastani while at Neo) for Neo4j and Apache Spark has been introduced. Why is Mazerunner important? It provides an example of how a graph-based database can be used for graph storage, while Apache Spark is used for graph processing. I am not suggesting that Mazerunner be used in a production scenario at this time. Clearly, more work needs to be done to make this architecture ready for release. However, graph-based storage, when associated with the graph-based processing within a distributed environment, offers the option to interrogate the data using a query language such as Cypher from Neo4j.

I hope that you have found this chapter useful. The next chapter will delve into graph-based storage in more depth. You can now delve into further GraphX coding, try to run the examples provided, and try modifying the code, so that you become familiar with the development process.

6

Graph-based Storage

Processing with Apache Spark and especially GraphX provides the ability to use in memory cluster-based, real-time processing for graphs. However, Apache Spark does not provide storage; the graph-based data must come from somewhere and after processing, probably there will be a need for storage. In this chapter, I will examine graph-based storage using the Titan graph database as an example. This chapter will cover the following topics:

- An overview of Titan
- An overview of TinkerPop
- Installing Titan
- Using Titan with HBase
- Using Titan with Cassandra
- Using Titan with Spark

The young age of this field of processing means that the storage integration between Apache Spark, and the graph-based storage system Titan is not yet mature.

In the previous chapter, the Neo4j Mazerunner architecture was examined, which showed how the Spark-based transactions could be replicated to Neo4j. This chapter deals with Titan not because of the functionality that it shows today, but due to the future promise that it offers for the field of the graph-based storage when used with Apache Spark.

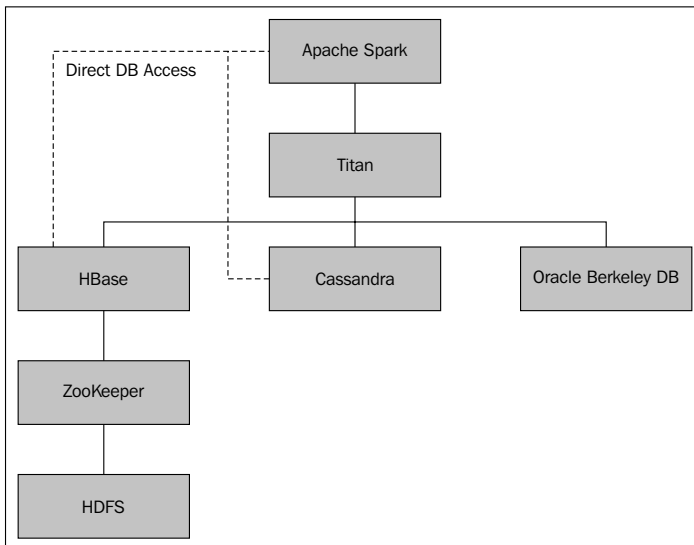
Titan

Titan is a graph database that was developed by Aurelius (<http://thinkaurelius.com/>). The application source and binaries can be downloaded from GitHub (<http://thinkaurelius.github.io/titan/>), and this location also contains the Titan documentation. Titan has been released as an open source application under an Apache 2 license. At the time of writing this book, Aurelius has been acquired by DataStax, although Titan releases should go ahead.

Titan offers a number of storage options, but I will concentrate only on two, HBase—the Hadoop NoSQL database, and Cassandra—the non-Hadoop NoSQL database. Using these underlying storage mechanisms, Titan is able to provide a graph-based storage in the big data range.

The TinkerPop3-based Titan release 0.9.0-M2 was released in June 2015, which will enable greater integration with Apache Spark (TinkerPop will be explained in the next section). It is this release that I will use in this chapter. It is TinkerPop that the Titan database now uses for graph manipulation. This Titan release is an experimental development release but hopefully, future releases should consolidate Titan functionality.

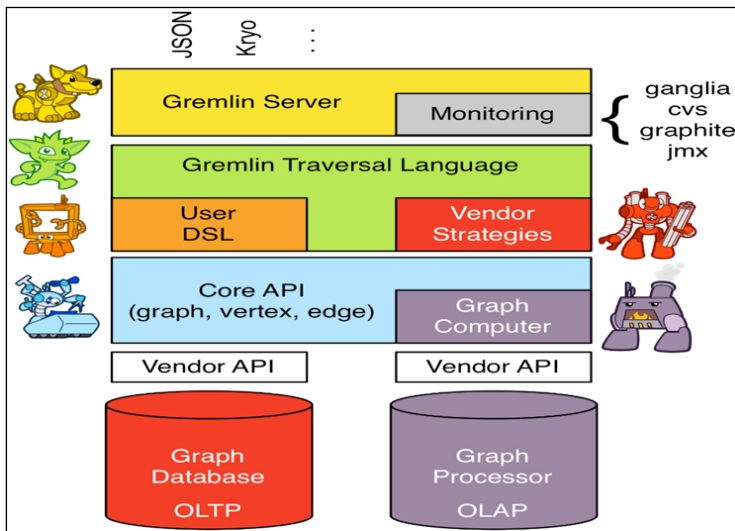
This chapter concentrates on the Titan database rather than an alternative graph database, such as Neo4j, because Titan can use Hadoop-based storage. Also, Titan offers the future promise of integration with Apache Spark for a big data scale, in memory graph-based processing. The following diagram shows the architecture being discussed in this chapter. The dotted line shows direct Spark database access, whereas the solid lines represent Spark access to the data through Titan classes.



The Spark interface doesn't officially exist yet (it is only available in the M2 development release), but it is just added for reference. Although Titan offers the option of using Oracle for storage, it will not be covered in this chapter. I will initially examine the Titan to the HBase and Cassandra architectures, and consider the Apache Spark integration later. When considering (distributed) HBase, ZooKeeper is required as well for integration. Given that I am using an existing CDH5 cluster, HBase and ZooKeeper are already installed.

TinkerPop

TinkerPop, currently at version 3 as of July 2015, is an Apache incubator project, and can be found at <http://tinkerpop.incubator.apache.org/>. It enables both graph databases (like Titan) and graph analytic systems (like Giraph) to use it as a subsystem for graph processing rather than creating their own graph processing modules.



The previous figure (borrowed from the TinkerPop website) shows the TinkerPop architecture. The blue layer shows the Core TinkerPop API, which offers the graph processing API for graph, vertex, and edge processing. The **Vendor API** boxes show the APIs that the vendors will implement to integrate their systems. The diagram shows that there are two possible APIs: one for the **OLTP** database systems, and another for the **OLAP** analytics systems.

The diagram also shows that the **Gremlin** language is used to create and manage graphs for TinkerPop, and so for Titan. Finally, the Gremlin server sits at the top of the architecture, and allows integration to monitoring systems like Ganglia.

Installing Titan

As Titan is required throughout this chapter, I will install it now, and show how it can be acquired, installed, and configured. I have downloaded the latest prebuilt version (0.9.0-M2) of Titan at: s3.thinkaurelius.com/downloads/titan/titan-0.9.0-M2-hadoop1.zip.

I have downloaded the zipped release to a temporary directory, as shown next. Carry out the following steps to ensure that Titan is installed on each node in the cluster:

```
[hadoop@hc2nn tmp]$ ls -lh titan-0.9.0-M2-hadoop1.zip
-rw-r--r-- 1 hadoop hadoop 153M Jul 22 15:13 titan-0.9.0-M2-hadoop1.zip
```

Using the Linux unzip command, unpack the zipped Titan release file:

```
[hadoop@hc2nn tmp]$ unzip titan-0.9.0-M2-hadoop1.zip

[hadoop@hc2nn tmp]$ ls -l
total 155752
drwxr-xr-x 10 hadoop hadoop      4096 Jun  9 00:56 titan-0.9.0-M2-hadoop1
-rw-r--r--  1 hadoop hadoop 159482381 Jul 22 15:13 titan-0.9.0-M2-hadoop1.zip
```

Now, use the Linux `su` (switch user) command to change to the `root` account, and move the install to the `/usr/local/` location. Change the file and group membership of the install to the `hadoop` user, and create a symbolic link called `titan` so that the current Titan release can be referred to as the simplified path called `/usr/local/titan`:

```
[hadoop@hc2nn ~]$ su -
[root@hc2nn ~]# cd /home/hadoop/tmp
[root@hc2nn titan]# mv titan-0.9.0-M2-hadoop1 /usr/local
[root@hc2nn titan]# cd /usr/local
[root@hc2nn local]# chown -R hadoop:hadoop titan-0.9.0-M2-hadoop1
[root@hc2nn local]# ln -s titan-0.9.0-M2-hadoop1 titan
[root@hc2nn local]# ls -ld *titan*
lrwxrwxrwx 1 root root      19 Mar 13 14:10 titan -> titan-0.9.0-M2-hadoop1
drwxr-xr-x 10 hadoop hadoop 4096 Feb 14 13:30 titan-0.9.0-M2-hadoop1
```

Using a Titan Gremlin shell that will be demonstrated later, Titan is now available for use. This version of Titan needs Java 8; make sure that you have it installed.

Titan with HBase

As the previous diagram shows, HBase depends upon ZooKeeper. Given that I have a working ZooKeeper quorum on my CDH5 cluster (running on the `hc2r1m2`, `hc2r1m3`, and `hc2r1m4` nodes), I only need to ensure that HBase is installed and working on my Hadoop cluster.

The HBase cluster

I will install a distributed version of HBase using the Cloudera CDH cluster manager. Using the manager console, it is a simple task to install HBase. The only decision required is where to locate the HBase servers on the cluster. The following figure shows the **View By Host** form from the CDH HBase installation. The HBase components are shown to the right as **Added Roles**.

I have chosen to add the HBase region servers (RS) to the `hc2r1m2`, `hc2r1m3`, and `hc2r1m4` nodes. I have installed the HBase master (M), the HBase REST server (HBREST), and HBase Thrift server (HBTS) on the `hc2r1m1` host.

View By Host

This table is grouped by hosts having the same roles assigned to them.

Hosts	Count	Existing Roles	Added Roles
hc2nn	1	A B NN SNN G HMS HS2 HS	
hc2r1m1	1	DN HFS G WHCS K HS W NM	M HBREST HBTS
hc2r1m2	1	DN G K AM W NM S	RS
hc2r1m[3-4]	2	DN G K W NM S	RS

Close

I have manually installed and configured many Hadoop-based components in the past, and I find that this simple manager-based installation and configuration of components is both quick and reliable. It saves me time so that I can concentrate on other systems, such as Titan.

Once HBase is installed, and has been started from the CDH manager console, it needs to be checked to ensure that it is working. I will do this using the HBase shell command shown here:

```
[hadoop@hc2r1m2 ~]$ hbase shell
Version 0.98.6-cdh5.3.2, rUnknown, Tue Feb 24 12:56:59 PST 2015
hbase(main):001:0>
```

As you can see from the previous commands, I run the HBase shell as the Linux user `hadoop`. The HBase version 0.98.6 has been installed; this version number will become important later when we start using Titan:

```
hbase(main):001:0> create 'table2', 'cf1'
hbase(main):002:0> put 'table2', 'row1', 'cf1:1', 'value1'
hbase(main):003:0> put 'table2', 'row2', 'cf1:1', 'value2'
```

I have created a simple table called `table2` with a column family of `cf1`. I have then added two rows with two different values. This table has been created from the `hc2r1m2` node, and will now be checked from an alternate node called `hc2r1m4` in the HBase cluster:

```
[hadoop@hc2r1m4 ~]$ hbase shell
```

```
hbase(main):001:0> scan 'table2'
```

```
ROW                                COLUMN+CELL
  row1                             column=cf1:1, timestamp=1437968514021,
value=value1
  row2                             column=cf1:1, timestamp=1437968520664,
value=value2
2 row(s) in 0.3870 seconds
```

As you can see, the two data rows are visible in `table2` from a different host, so HBase is installed and working. It is now time to try and create a graph in Titan using HBase and the Titan Gremlin shell.

The Gremlin HBase script

I have checked my Java version to make sure that I am on version 8, otherwise Titan 0.9.0-M2 will not work:

```
[hadoop@hc2r1m2 ~]$ java -version
openjdk version "1.8.0_51"
```

If you do not set your Java version correctly, you will get errors like this, which don't seem to be meaningful until you Google them:

```
Exception in thread "main" java.lang.UnsupportedClassVersionError: org/
apache/tinkerpop/gremlin/groovy/plugin/RemoteAcceptor :
Unsupported major.minor version 52.0
```

The interactive Titan Gremlin shell can be found within the bin directory of the Titan install, as shown here. Once started, it offers a Gremlin prompt:

```
[hadoop@hc2r1m2 bin]$ pwd
/usr/local/titan/

[hadoop@hc2r1m2 titan]$ bin/gremlin.sh
gremlin>
```

The following script will be entered using the Gremlin shell. The first section of the script defines the configuration in terms of the storage (HBase), the ZooKeeper servers used, the ZooKeeper port number, and the HBase table name that is to be used:

```
hBaseConf = new BaseConfiguration();
hBaseConf.setProperty("storage.backend", "hbase");
hBaseConf.setProperty("storage.hostname", "hc2r1m2, hc2r1m3, hc2r1m4");
hBaseConf.setProperty("storage.hbase.ext.hbase.zookeeper.property.
clientPort", "2181")
hBaseConf.setProperty("storage.hbase.table", "titan")
```

```
titanGraph = TitanFactory.open(hBaseConf);
```

The next section defines the generic vertex properties' name and age for the graph to be created using the Management System. It then commits the management system changes:

```
manageSys = titanGraph.openManagement();
nameProp = manageSys.makePropertyKey('name').dataType(String.class).
make();
ageProp = manageSys.makePropertyKey('age').dataType(String.class).
make();
manageSys.buildIndex('nameIdx', Vertex.class).addKey(nameProp).
buildCompositeIndex();
manageSys.buildIndex('ageIdx', Vertex.class).addKey(ageProp).
buildCompositeIndex();

manageSys.commit();
```

Now, six vertices are added to the graph. Each one is given a numeric label to represent its identity. Each vertex is given an age and name value:

```
v1=titanGraph.addVertex(label, '1');
v1.property('name', 'Mike');
v1.property('age', '48');

v2=titanGraph.addVertex(label, '2');
v2.property('name', 'Sarah');
v2.property('age', '45');

v3=titanGraph.addVertex(label, '3');
v3.property('name', 'John');
v3.property('age', '25');

v4=titanGraph.addVertex(label, '4');
v4.property('name', 'Jim');
v4.property('age', '53');

v5=titanGraph.addVertex(label, '5');
v5.property('name', 'Kate');
v5.property('age', '22');

v6=titanGraph.addVertex(label, '6');
v6.property('name', 'Flo');
v6.property('age', '52');
```

Finally, the graph edges are added to join the vertices together. Each edge has a relationship value. Once created, the changes are committed to store them to Titan, and therefore HBase:

```
v6.addEdge("Sister", v1)
v1.addEdge("Husband", v2)
v2.addEdge("Wife", v1)
v5.addEdge("Daughter", v1)
v5.addEdge("Daughter", v2)
v3.addEdge("Son", v1)
v3.addEdge("Son", v2)
```

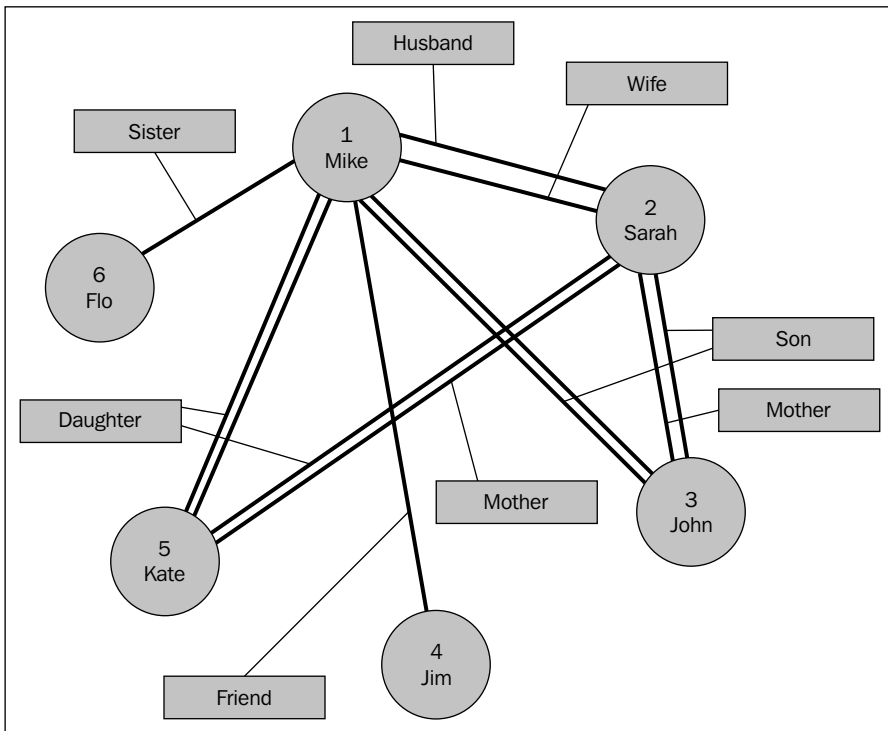
```

v4.addEdge("Friend", v1)
v1.addEdge("Father", v5)
v1.addEdge("Father", v3)
v2.addEdge("Mother", v5)
v2.addEdge("Mother", v3)

titanGraph.tx().commit();

```

This results in a simple person-based graph, shown in the following figure, which was also used in the previous chapter:



This graph can then be tested in Titan via the Gremlin shell using a similar script to the previous one. Just enter the following script at the `gremlin>` prompt, as was shown previously. It uses the same initial six lines to create the `titanGraph` configuration, but it then creates a graph traversal variable `g`:

```

hBaseConf = new BaseConfiguration();
hBaseConf.setProperty("storage.backend", "hbase");
hBaseConf.setProperty("storage.hostname", "hc2r1m2,hc2r1m3,hc2r1m4");

```

```
hBaseConf.setProperty("storage.hbase.ext.hbase.zookeeper.property.  
clientPort","2181")  
hBaseConf.setProperty("storage.hbase.table","titan")
```

```
titanGraph = TitanFactory.open(hBaseConf);
```

```
gremlin> g = titanGraph.traversal()
```

Now, the graph traversal variable can be used to check the graph contents. Using the ValueMap option, it is possible to search for the graph nodes called Mike and Flo. They have been successfully found here:

```
gremlin> g.V().has('name','Mike').valueMap();  
==>[name:[Mike], age:[48]]
```

```
gremlin> g.V().has('name','Flo').valueMap();  
==>[name:[Flo], age:[52]]
```

So, the graph has been created and checked in Titan using the Gremlin shell, but we can also check the storage in HBase using the HBase shell, and check the contents of the Titan table. The following scan shows that the table exists, and contains 72 rows of the data for this small graph:

```
[hadoop@hc2r1m2 ~]$ hbase shell  
hbase(main):002:0> scan 'titan'  
72 row(s) in 0.8310 seconds
```

Now that the graph has been created, and I am confident that it has been stored in HBase, I will attempt to access the data using apache Spark. I have already started Apache Spark on all the nodes as shown in the previous chapter. This will be a direct access from Apache Spark 1.3 to the HBase storage. I won't at this stage be attempting to use Titan to interpret the HBase stored graph.

Spark on HBase

In order to access HBase from Spark, I will be using Cloudera's SparkOnHBase module, which can be downloaded from <https://github.com/cloudera-labs/SparkOnHBase>.

The downloaded file is in a zipped format, and needs to be unzipped. I have done this using the Linux unzip command in a temporary directory:

```
[hadoop@hc2r1m2 tmp]$ ls -l SparkOnHBase-cdh5-0.0.2.zip
```

```
-rw-r--r-- 1 hadoop hadoop 370439 Jul 27 13:39 SparkOnHBase-cdh5-0.0.2.zip
```

```
[hadoop@hc2r1m2 tmp]$ unzip SparkOnHBase-cdh5-0.0.2.zip
```

```
[hadoop@hc2r1m2 tmp]$ ls
```

```
SparkOnHBase-cdh5-0.0.2  SparkOnHBase-cdh5-0.0.2.zip
```

I have then moved into the unpacked module, and used the Maven command `mvn` to build the JAR file:

```
[hadoop@hc2r1m2 tmp]$ cd SparkOnHBase-cdh5-0.0.2
```

```
[hadoop@hc2r1m2 SparkOnHBase-cdh5-0.0.2]$ mvn clean package
```

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 13:17 min
[INFO] Finished at: 2015-07-27T14:05:55+12:00
[INFO] Final Memory: 50M/191M
[INFO] -----
```

Finally, I moved the built component to my development area to keep things tidy, so that I could use this module in my Spark HBase code:

```
[hadoop@hc2r1m2 SparkOnHBase-cdh5-0.0.2]$ cd ..
```

```
[hadoop@hc2r1m2 tmp]$ mv SparkOnHBase-cdh5-0.0.2 /home/hadoop/spark
```

Accessing HBase with Spark

As in previous chapters, I will be using SBT and Scala to compile my Spark-based scripts into applications. Then, I will use `spark-submit` to run these applications on the Spark cluster. My SBT configuration file looks like this. It contains the Hadoop, Spark, and HBase libraries:

```
[hadoop@hc2r1m2 titan_hbase]$ pwd
/home/hadoop/spark/titan_hbase
```

```
[hadoop@hc2r1m2 titan_hbase]$ more titan.sbt
```

```
name := "T i t a n"
```

```
version := "1.0"
```

```
scalaVersion := "2.10.4"
```

```
libraryDependencies += "org.apache.hadoop" % "hadoop-client" % "2.3.0"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.3.1"
libraryDependencies += "com.cloudera.spark" % "hbase" % "5-0.0.2" from
"file:///home/hadoop/spark/SparkOnHBase-cdh5-0.0.2/target/SparkHBase.jar"
libraryDependencies += "org.apache.hadoop.hbase" % "client" % "5-
0.0.2" from "file:///home/hadoop/spark/SparkOnHBase-cdh5-0.0.2/target/
SparkHBase.jar"
resolvers += "Cloudera Repository" at "https://repository.cloudera.com/
artifactory/clouder
a-repos/"
```

Notice that I am running this application on the `hc2r1m2` server, using the Linux `hadoop` account, under the directory `/home/hadoop/spark/titan_hbase`. I have created a Bash shell script called `run_titan.bash.hbase`, which allows me to run any application that is created and compiled under the `src/main/scala` subdirectory:

```
[hadoop@hc2r1m2 titan_hbase]$ pwd ; more run_titan.bash.hbase
/home/hadoop/spark/titan_hbase
```

```
#!/bin/bash
```

```
SPARK_HOME=/usr/local/spark
SPARK_BIN=$SPARK_HOME/bin
SPARK_SBIN=$SPARK_HOME/sbin
```

```
JAR_PATH=/home/hadoop/spark/titan_hbase/target/scala-2.10/t-i-t-a-n_2.10-
1.0.jar
CLASS_VAL=$1
```

```
CDH_JAR_HOME=/opt/cloudera/parcels/CDH/lib/hbase/
CONN_HOME=/home/hadoop/spark/SparkOnHBase-cdh5-0.0.2/target/
```

```
HBASE_JAR1=$CDH_JAR_HOME/hbase-common-0.98.6-cdh5.3.3.jar
HBASE_JAR2=$CONN_HOME/SparkHBase.jar
```

```
cd $SPARK_BIN
```

```
./spark-submit \
  --jars $HBASE_JAR1 \
```

```
--jars $HBASE_JAR2 \
--class $CLASS_VAL \
--master spark://hc2nn.semtech-solutions.co.nz:7077 \
--executor-memory 100M \
--total-executor-cores 50 \
$JAR_PATH
```

The Bash script is held within the same `titan_hbase` directory, and takes a single parameter of the application class name. The parameters to the `spark-submit` call are the same as the previous examples. In this case, there is only a single script under `src/main/scala`, called `spark3_hbase2.scala`:

```
[hadoop@hc2r1m2 scala]$ pwd
/home/hadoop/spark/titan_hbase/src/main/scala
```

```
[hadoop@hc2r1m2 scala]$ ls
spark3_hbase2.scala
```

The Scala script starts by defining the package name to which the application class will belong. It then imports the Spark, Hadoop, and HBase classes:

```
package nz.co.semtechsolutions

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

import org.apache.hadoop.hbase._
import org.apache.hadoop.fs.Path
import com.cloudera.spark.hbase.HBaseContext
import org.apache.hadoop.hbase.client.Scan
```

The application class name is defined as well as the main method. A configuration object is then created in terms of the application name, and the Spark URL. Finally, a Spark context is created from the configuration:

```
object spark3_hbase2
{

  def main(args: Array[String]) {

    val sparkMaster = "spark://hc2nn.semtech-solutions.co.nz:7077"
    val appName = "Spark HBase 2"
```

```
val conf = new SparkConf()

conf.setMaster(sparkMaster)
conf.setAppName(appName)

val sparkCxt = new SparkContext(conf)
```

Next, an HBase configuration object is created, and a Cloudera CDH `hbase-site.xml` file-based resource is added:

```
val jobConf = HBaseConfiguration.create()

val hbasePath="/opt/cloudera/parcels/CDH/etc/hbase/conf.dist/"

jobConf.addResource(new Path(hbasePath+"hbase-site.xml"))
```

An HBase context object is created using the Spark context and the HBase configuration object. The scan and cache configurations are also defined:

```
val hbaseContext = new HBaseContext(sparkCxt, jobConf)

var scan = new Scan()
scan.setCaching(100)
```

Finally, the data from the HBase Titan table is retrieved using the `hbaseRDD` HBase context method, and the scan object. The RDD count is printed, and then the script closes:

```
var hbaseRdd = hbaseContext.hbaseRDD("titan", scan)

println( "Rows in Titan hbase table : " + hbaseRdd.count() )

println( " >>>> Script Finished <<<< " )

} // end main

} // end spark3_hbase2
```

I am only printing the count of the data retrieved because Titan compresses the data in GZ format. So, it would make little sense in trying to manipulate it directly.

Using the `run_titan.bash.hbase` script, the Spark application called `spark3_hbase2` is run. It outputs an RDD row count of 72, matching the Titan table row count that was previously found. This proves that Apache Spark has been able to access the raw Titan HBase stored graph data, but Spark has not yet used the Titan libraries to access the Titan data as a graph. This will be discussed later. And here is the code:

```
[hadoop@hc2rlm2 titan_hbase]$ ./run_titan.bash.hbase nz.co.
semtechsolutions.spark3_hbase2
```

```
Rows in Titan hbase table : 72
```

```
>>>> Script Finished <<<<<
```

Titan with Cassandra

In this section, the Cassandra NoSQL database will be used as a storage mechanism for Titan. Although it does not use Hadoop, it is a large-scale, cluster-based database in its own right, and can scale to very large cluster sizes. This section will follow the same process. As for HBase, a graph will be created, and stored in Cassandra using the Titan Gremlin shell. It will then be checked using Gremlin, and the stored data will be checked in Cassandra. The raw Titan Cassandra graph-based data will then be accessed from Spark. The first step then will be to install Cassandra on each node in the cluster.

Installing Cassandra

Create a repo file that will allow the community version of DataStax Cassandra to be installed using the Linux `yum` command. Root access will be required for this, so the `su` command has been used to switch the user to the root. Install Cassandra on all the nodes:

```
[hadoop@hc2nn lib]$ su -
[root@hc2nn ~]# vi /etc/yum.repos.d/datastax.repo

[datastax]
name= DataStax Repo for Apache Cassandra
baseurl=http://rpm.datastax.com/community
enabled=1
gpgcheck=0
```

Now, install Cassandra on each node in the cluster using the Linux `yum` command:

```
[root@hc2nn ~]# yum -y install dsc20-2.0.13-1 cassandra20-2.0.13-1
```

Set up the Cassandra configuration under `/etc/cassandra/conf` by altering the `cassandra.yaml` file:

```
[root@hc2nn ~]# cd /etc/cassandra/conf ; vi cassandra.yaml
```

I have made the following changes to specify my cluster name, the server seed IP addresses, the RPC address, and the snitch value. Seed nodes are the nodes that the other nodes will try to connect to first. In this case, the `NameNode (103)`, and `node2 (108)` have been used as seeds. The snitch method manages network topology and routing:

```
cluster_name: 'Cluster1'
seeds: "192.168.1.103,192.168.1.108"
listen_address:
rpc_address: 0.0.0.0
endpoint_snitch: GossipingPropertyFileSnitch
```

Cassandra can now be started on each node as root using the service command:

```
[root@hc2nn ~]# service cassandra start
```

Log files can be found under `/var/log/cassandra`, and the data is stored under `/var/lib/cassandra`. The `nodetool` command can be used on any Cassandra node to check the status of the Cassandra cluster:

```
[root@hc2nn cassandra]# nodetool status
```

Datacenter: DC1

=====

Status=Up/Down

|/ State=Normal/Leaving/Joining/Moving

--	Address	Load	Tokens Rack	Owns (effective)	Host ID
UN	192.168.1.105	63.96 KB	256	37.2%	f230c5d7-ff6f-43e7-821d-c7ae2b5141d3 RAC1
UN	192.168.1.110	45.86 KB	256	39.9%	fc1d80fe-6c2d-467d-9034-96a1f203c20d RAC1
UN	192.168.1.109	45.9 KB	256	40.9%	daadf2ee-f8c2-4177-ae72-683e39fd1ea0 RAC1

```
UN 192.168.1.108 50.44 KB 256 40.5% b9d796c0-5893-46bc-8e3c-187a524b1f5a RAC1
UN 192.168.1.103 70.68 KB 256 41.5% 53c2eebd-a66c-4a65-b026-96e232846243 RAC1
```

The Cassandra CQL shell command called `cqlsh` can be used to access the cluster, and create objects. The shell is invoked next, and it shows that Cassandra version 2.0.13 is installed:

```
[hadoop@hc2nn ~]$ cqlsh
Connected to Cluster1 at localhost:9160.
[cqlsh 4.1.1 | Cassandra 2.0.13 | CQL spec 3.1.1 | Thrift protocol 19.39.0]
Use HELP for help.
cqlsh>
```

The Cassandra query language next shows a key space called `keyspace1` that is being created and used via the CQL shell:

```
cqlsh> CREATE KEYSPACE keyspace1 WITH REPLICATION = { 'class' :
'SimpleStrategy', 'replication_factor' : 1 };
```

```
cqlsh> USE keyspace1;
```

```
cqlsh:keyspace1> SELECT * FROM system.schema_keyspaces;
```

```
keyspace_name | durable_writes | strategy_class
| strategy_options
-----+-----+-----+-----+-----+
keyspace1 | True | org.apache.cassandra.locator.SimpleStrategy |
{"replication_factor":"1"}
system | True | org.apache.cassandra.locator.LocalStrategy |
{}
system_traces | True | org.apache.cassandra.locator.SimpleStrategy |
{"replication_factor":"2"}
```

Since Cassandra is installed and working, it is now time to create a Titan graph using Cassandra for storage. This will be tackled in the next section using the Titan Gremlin shell. It will follow the same format as the HBase section previously.

The Gremlin Cassandra script

As with the previous Gremlin script, this Cassandra version creates the same simple graph. The difference with this script is in the configuration. The backend storage type is defined as Cassandra, and the hostnames are defined to be the Cassandra seed nodes. The key space and the port number are specified and finally, the graph is created:

```
cassConf = new BaseConfiguration();
cassConf.setProperty("storage.backend", "cassandra");
cassConf.setProperty("storage.hostname", "hc2nn,hc2r1m2");
cassConf.setProperty("storage.port", "9160")
cassConf.setProperty("storage.keyspace", "titan")
titanGraph = TitanFactory.open(cassConf);
```

From this point, the script is the same as the previous HBase example, so I will not repeat it. This script will be available in the download package as `cassandra_create.bash`. The same checks, using the previous configuration, can be carried out in the Gremlin shell to check the data. This returns the same results as the previous checks, and so proves that the graph has been stored:

```
gremlin> g = titanGraph.traversal()
```

```
gremlin> g.V().has('name','Mike').valueMap();
==>[name:[Mike], age:[48]]
```

```
gremlin> g.V().has('name','Flo').valueMap();
==>[name:[Flo], age:[52]]
```

Using the Cassandra CQL shell, and the Titan keyspace, it can be seen that a number of Titan tables have been created in Cassandra:

```
[hadoop@hc2nn ~]$ cqlsh
cqlsh> use titan;
cqlsh:titan> describe tables;
edgestore          graphindex          system_properties
      systemlog    txlog
edgestore_lock_    graphindex_lock_    system_properties_lock_  titan_ids
```

It can also be seen that the data exists in the edgestore table within Cassandra:

```
cqlsh:titan> select * from edgestore;
```

key	column1	value
0x00000000000004815 0x00011ee0	0x02	
0x00000000000004815 0xa0727425536fee1ec0	0x10c0	
.....		
0x00000000000001005 0x00800512644c1b149004a0	0x10c8	
0x00000000000001005 65782d706ff30200	0x30c9801009800c20	0x000101143c01023b0101696e64

This assures me that a Titan graph has been created in the Gremlin shell, and is stored in Cassandra. Now, I will try to access the data from Spark.

The Spark Cassandra connector

In order to access Cassandra from Spark, I will download the DataStax Spark Cassandra connector and driver libraries. Information and version matching on this can be found at <http://mvnrepository.com/artifact/com.datastax.spark/>.

The version compatibility section of this URL shows the Cassandra connector version that should be used with each Cassandra and Spark version. The version table shows that the connector version should match the Spark version that is being used. The next URL allows the libraries to be sourced at http://mvnrepository.com/artifact/com.datastax.spark/spark-cassandra-connector_2.10.

By following the previous URL, and selecting a library version, you will see a compile dependencies table associated with the library, which indicates all of the other dependent libraries, and their versions that you will need. The following libraries are those that are needed for use with Spark 1.3.1. If you use the previous URLs, you will see which version of the Cassandra connector library to use with each version of Spark. You will also see the libraries that the Cassandra connector depends upon. Be careful to choose just (and all of) those library versions that are required:

```
[hadoop@hc2rlm2 titan_cass]$ pwd ; ls *.jar
/home/hadoop/spark/titan_cass

spark-cassandra-connector_2.10-1.3.0-M1.jar
cassandra-driver-core-2.1.5.jar
```

```
cassandra-thrift-2.1.3.jar
libthrift-0.9.2.jar
cassandra-clientutil-2.1.3.jar
guava-14.0.1.jar
joda-time-2.3.jar
joda-convert-1.2.jar
```

Accessing Cassandra with Spark

Now that I have the Cassandra connector library and all of its dependencies in place, I can begin to think about the Scala code, required to connect to Cassandra. The first thing to do, given that I am using SBT as a development tool, is to set up the SBT build configuration file. Mine looks like this:

```
[hadoop@hc2rlm2 titan_cass]$ pwd ; more titan.sbt
/home/hadoop/spark/titan_cass

name := "Spark Cass"
version := "1.0"
scalaVersion := "2.10.4"
libraryDependencies += "org.apache.hadoop" % "hadoop-client" % "2.3.0"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.3.1"
libraryDependencies += "com.datastax.spark" % "spark-cassandra-connector"
% "1.3.0-M1" from
om "file:///home/hadoop/spark/titan_cass/spark-cassandra-connector_2.10-
1.3.0-M1.jar"
libraryDependencies += "com.datastax.cassandra" % "cassandra-driver-core"
% "2.1.5" from
"file:///home/hadoop/spark/titan_cass/cassandra-driver-core-2.1.5.jar"
libraryDependencies += "org.joda" % "time" % "2.3" from "file:///home/
hadoop/spark/titan_
cass/joda-time-2.3.jar"
libraryDependencies += "org.apache.cassandra" % "thrift" % "2.1.3" from
"file:///home/hado
op/spark/titan_cass/cassandra-thrift-2.1.3.jar"
libraryDependencies += "com.google.common" % "collect" % "14.0.1" from
"file:///home/hadoo
```

```
p/spark/titan_cass/guava-14.0.1.jar
resolvers += "Cloudera Repository" at "https://repository.cloudera.com/
artifactory/clouder
a-repos/"
```

The Scala script for the Cassandra connector example, called `spark3_cass.scala`, now looks like the following code. First, the package name is defined. Then, the classes are imported for Spark, and the Cassandra connector. Next, the object application class `spark3_cass` ID is defined, and so is the main method:

```
package nz.co.semtechsolutions

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

import com.datastax.spark.connector._

object spark3_cass
{

    def main(args: Array[String]) {
```

A Spark configuration object is created using a Spark URL and application name. The Cassandra connection host is added to the configuration. Then, the Spark context is created using the configuration object:

```
    val sparkMaster = "spark://hc2nn.semtech-solutions.co.nz:7077"
    val appName = "Spark Cass 1"
    val conf = new SparkConf()

    conf.setMaster(sparkMaster)
    conf.setAppName(appName)

    conf.set("spark.cassandra.connection.host", "hc2r1m2")

    val sparkCxt = new SparkContext(conf)
```

The Cassandra keyspace, and table names that are to be checked are defined. Then, the Spark context method called `cassandraTable` is used to connect to Cassandra, and obtain the contents of the `edgestore` table as an RDD. The size of this RDD is then printed, and the script exits. We won't look at this data at this time, because all that was needed was to prove that a connection to Cassandra could be made:

```
    val keySpace = "titan"
```

```
val tableName = "edgestore"

val cassRDD = sparkCxt.cassandraTable( keySpace, tableName )

println( "Cassandra Table Rows : " + cassRDD.count )

println( " >>>> Script Finished <<<< " )

} // end main

} // end spark3_cass
```

As in the previous examples, the Spark submit command has been placed in a Bash script called `run_titan.bash.cass`. This script, shown next, looks similar to many others used already. The point to note here is that there is a JARs option, which lists all of the JAR files used so that they are available at run time. The order of JAR files in this option has been determined to avoid the class exception errors:

```
[hadoop@hc2r1m2 titan_cass]$ more run_titan.bash

#!/bin/bash

SPARK_HOME=/usr/local/spark
SPARK_BIN=$SPARK_HOME/bin
SPARK_SBIN=$SPARK_HOME/sbin

JAR_PATH=/home/hadoop/spark/titan_cass/target/scala-2.10/spark-cass_2.10-1.0.jar
CLASS_VAL=$1

CASS_HOME=/home/hadoop/spark/titan_cass/

CASS_JAR1=$CASS_HOME/spark-cassandra-connector_2.10-1.3.0-M1.jar
CASS_JAR2=$CASS_HOME/cassandra-driver-core-2.1.5.jar
CASS_JAR3=$CASS_HOME/cassandra-thrift-2.1.3.jar
CASS_JAR4=$CASS_HOME/libthrift-0.9.2.jar
CASS_JAR5=$CASS_HOME/cassandra-clientutil-2.1.3.jar
CASS_JAR6=$CASS_HOME/guava-14.0.1.jar
CASS_JAR7=$CASS_HOME/joda-time-2.3.jar
```

```

CASS_JAR8=$CASS_HOME/joda-convert-1.2.jar

cd $SPARK_BIN

./spark-submit \
  --jars $CASS_JAR8,$CASS_JAR7,$CASS_JAR5,$CASS_JAR4,$CASS_JAR3,$CASS_
JAR6,$CASS_JAR2,$CASS_JAR1 \
  --class $CLASS_VAL \
  --master spark://hc2nn.semtech-solutions.co.nz:7077 \
  --executor-memory 100M \
  --total-executor-cores 50 \
  $JAR_PATH

```

This application is invoked using the previous Bash script. It connects to Cassandra, selects the data, and returns a Cassandra table data-based count of 218 rows.

```

[hadoop@hc2r1m2 titan_cass]$ ./run_titan.bash.cass nz.co.
semtechsolutions.spark3_cass

```

Cassandra Table Rows : 218

```
>>>> Script Finished <<<<<
```

This proves that the raw Cassandra-based Titan table data can be accessed from Apache Spark. However, as in the HBase example, this is raw table-based Titan data, and not the data in Titan graph form. The next step will be to use Apache Spark as a processing engine for the Titan database. This will be examined in the next section.

Accessing Titan with Spark

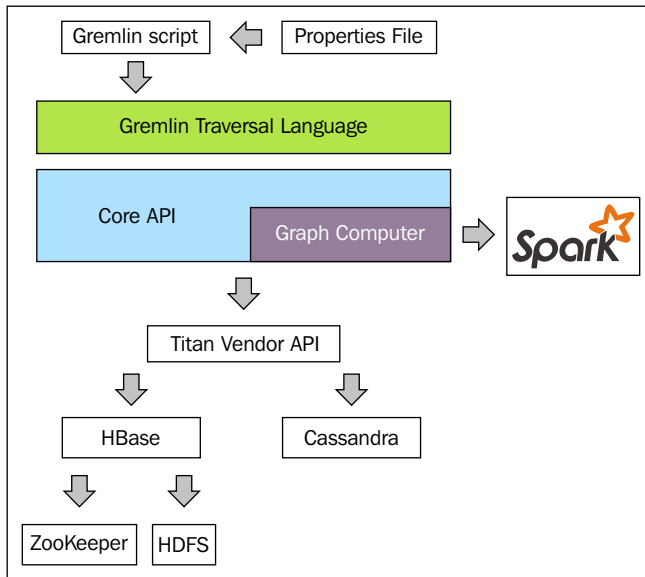
So far in this chapter, Titan 0.9.0-M2 has been installed, and the graphs have successfully been created using both HBase and Cassandra as backend storage options. These graphs have been created using Gremlin-based scripts. In this section, a properties file will be used via a Gremlin script to process a Titan-based graph using Apache Spark. The same two backend storage options, HBase and Cassandra, will be used with Titan.

The following figure, based on the TinkerPop3 diagram earlier in this chapter, shows the architecture used in this section. I have simplified the diagram, but it is basically the same as the previous TinkerPop version. I have just added the link to Apache Spark via the Graph Computer API. I have also added both HBase and Cassandra storage via the Titan vendor API. Of course, a distributed installation of HBase uses both Zookeeper for configuration, and HDFS for storage.

Titan uses TinkerPop's Hadoop-Gremlin package for graph processing OLAP processes. The link to the documentation section can be found at: <http://s3.thinkaurelius.com/docs/titan/0.9.0-M2/titan-hadoop-tp3.html>.

This section will show how the Bash shell, Groovy, and properties files can be used to configure, and run a Titan Spark-based job. It will show different methods for configuring the job, and it will also show methods for managing logging to enable error tracking. Also, different configurations of the property file will be described to give access to HBase, Cassandra, and the Linux file system.

Remember that the Titan release 0.9.0-M2, that this chapter is based on, is a development release. It is a prototype release, and is not yet ready for production. I assume that as the future Titan releases become available, the link between Titan and Spark will be more developed and stable. Currently, the work in this section is for demonstration purposes only, given the nature of the Titan release.



In the next section, I will explain the use of Gremlin, and Groovy scripts before moving onto connecting Titan to Spark using Cassandra and HBase as storage options.

Gremlin and Groovy

The Gremlin shell, which is used to execute Groovy commands against Titan, can be used in a number of ways. The first method of use just involves starting a Gremlin shell for use as an interactive session. Just execute the following:

```
cd $TITAN_HOME/bin ; ./ gremlin.sh
```

This starts the session, and automatically sets up required plug-ins such as TinkerPop and Titan (see next). Obviously, the previous `TITAN_HOME` variable is used to indicate that the bin directory in question is located within your Titan install (`TITAN_HOME`) directory:

```
plugin activated: tinkerpops.server
plugin activated: tinkerpops.utilities
plugin activated: tinkerpops.hadoop
plugin activated: tinkerpops.tinkergraph
plugin activated: aurelius.titan
```

It then provides you with a Gremlin shell prompt where you can interactively execute your shell commands against your Titan database. This shell is useful for testing scripts and running ad hoc commands against your Titan database.

```
gremlin>
```

A second method is to embed your Groovy commands inline in a script when you call the `gremlin.sh` command. In this example, the Groovy commands between the EOF markers are piped into the Gremlin shell. When the last Groovy command has executed, the Gremlin shell will terminate. This is useful when you still want to use the automated environment setup of the Gremlin shell, but you still want to be able to quickly re-execute a script. This code snippet has been executed from a Bash shell script, as can be seen in the next example. The following script uses the `titan.sh` script to manage the Gremlin server:

```
#!/bin/bash

TITAN_HOME=/usr/local/titan/

cd $TITAN_HOME

bin/titan.sh start

bin/gremlin.sh  << EOF

t = TitanFactory.open('cassandra.properties')
```

```
GraphOfTheGodsFactory.load(t)
t.close()
EOF
```

```
bin/titan.sh stop
```

A third method involves moving the Groovy commands into a separate Groovy file, and using the `-e` option with the Gremlin shell to execute the file. This method offers extra logging options for error tracking, but means that extra steps need to be taken when setting up the Gremlin environment for a Groovy script:

```
#!/bin/bash

TITAN_HOME=/usr/local/titan/
SCRIPTS_HOME=/home/hadoop/spark/gremlin
GREMLIN_LOG_FILE=$TITAN_HOME/log/gremlin_console.log

GROOVY_SCRIPT=$1

export GREMLIN_LOG_LEVEL="DEBUG"

cd $TITAN_HOME

bin/titan.sh start

bin/gremlin.sh -e $SCRIPTS_HOME/$GROOVY_SCRIPT > $GREMLIN_LOG_FILE 2>&1

bin/titan.sh stop
```

So, this script defines a Gremlin log level, which can be set to different logging levels to obtain extra information about a problem, that is, INFO, WARN, and DEBUG. It also redirects the script output to a log file (`GREMLIN_LOG_FILE`), and redirects errors to the same log file (`2>&1`). This has the benefit of allowing the log file to be continuously monitored, and provides a permanent record of the session. The Groovy script name that is to be executed is then passed to the encasing Bash shell script as a parameter (`$1`).

As I already mentioned, the Groovy scripts invoked in this way need extra environment configuration to set up the Gremlin session when compared to the previous Gremlin session options. For instance, it is necessary to import the necessary TinkerPop and Aurelius classes that will be used:

```
import com.thinkaurelius.titan.core.*
import com.thinkaurelius.titan.core.titan.*
import org.apache.tinkerpop.gremlin.*
```

Having described the script and configuration options necessary to start a Gremlin shell session, and run a Groovy script, from this point onwards I will concentrate on Groovy scripts, and the property files necessary to configure the Gremlin session.

TinkerPop's Hadoop Gremlin

As already mentioned previously in this section, it is the TinkerPop Hadoop Gremlin package within Titan that will be used to call Apache Spark as a processing engine (Hadoop Giraph can be used for processing as well). The link available at <http://s3.thinkaurelius.com/docs/titan/0.9.0-M2/titan-hadoop-tp3.html> provides documentation for Hadoop Gremlin; remember that this TinkerPop package is still being developed and is subject to change.

At this point, I will examine a properties file that can be used to connect to Cassandra as a storage backend for Titan. It contains sections for Cassandra, Apache Spark, and the Hadoop Gremlin configuration. My Cassandra properties file is called `cassandra.properties`, and it looks like this (lines beginning with a hash character (#) are comments):

```
#####
# Storage details
#####
storage.backend=cassandra
storage.hostname=hc2rlm2
storage.port=9160
storage.cassandra.keyspace=dead
cassandra.input.partitionner.class=org.apache.cassandra.dht.
Murmur3Partitioner
```

The previous Cassandra-based properties describe the Cassandra host and port. This is why the storage backend type is Cassandra, the Cassandra keyspace that is to be used is called `dead` (short for grateful dead – the data that will be used in this example). Remember that the Cassandra tables are grouped within keyspaces. The previous `partitioner` class defines the Cassandra class that will be used to partition the Cassandra data. The Apache Spark configuration section contains the master URL, executor memory, and the data serializer class that is to be used:

```
#####
# Spark
#####
spark.master=spark://hc2nn.semtech-solutions.co.nz:6077
spark.executor.memory=400M
spark.serializer=org.apache.spark.serializer.KryoSerializer
```

Finally, the Hadoop Gremlin section of the properties file, which defines the classes to be used for graph and non-graph input and output is shown here. It also defines the data input and output locations, as well as the flags for caching JAR files, and deriving memory:

```
#####
# Hadoop Gremlin
#####
gremlin.graph=org.apache.tinkerpop.gremlin.hadoop.structure.HadoopGraph
gremlin.hadoop.graphInputFormat=com.thinkaurelius.titan.hadoop.formats.
cassandra.CassandraInputFormat
gremlin.hadoop.graphOutputFormat=org.apache.tinkerpop.gremlin.hadoop.
structure.io.gryo.GryoOutputFormat
gremlin.hadoop.memoryOutputFormat=org.apache.hadoop.mapreduce.lib.output.
SequenceFileOutputFormat

gremlin.hadoop.deriveMemory=false
gremlin.hadoop.jarsInDistributedCache=true
gremlin.hadoop.inputLocation=none
gremlin.hadoop.outputLocation=output
```

Blueprints is the TinkerPop property graph model interface. Titan releases its own implementation of blueprints, so instead of seeing `blueprints.graph` in the preceding properties, you see `gremlin.graph`. This defines the class, used to define the graph that is supposed to be used. If this option were omitted, then the graph type would default to the following:

```
com.thinkaurelius.titan.core.TitanFactory
```

The `CassandraInputFormat` class defines that the data is being retrieved from the Cassandra database. The graph output serialization class is defined to be `GryoOutputFormat`. The memory output format class is defined to use the Hadoop Map Reduce class `SequenceFileOutputFormat`.

The `jarsInDistributedCache` value has been defined to be true so that the JAR files are copied to the memory, enabling Apache Spark to source them. Given more time, I would investigate ways to make the Titan classes visible to Spark, on the class path, to avoid excessive memory usage.

Given that the TinkerPop Hadoop Gremlin module is only available as a development prototype release, currently the documentation is minimal. There are very limited coding examples, and there does not seem to be documentation available describing each of the previous properties.

Before I delve into the examples of Groovy scripts, I thought that I would show you an alternative method for configuring your Groovy jobs using a configuration object.

Alternative Groovy configuration

A configuration object can be created using the `BaseConfiguration` method. In this example, I have created a Cassandra configuration called `cassConf`:

```
cassConf = new BaseConfiguration();

cassConf.setProperty("storage.backend", "cassandra");
cassConf.setProperty("storage.hostname", "hc2r1m2");
cassConf.setProperty("storage.port", "9160")
cassConf.setProperty("storage.cassandra.keyspace", "titan")

titanGraph = TitanFactory.open(cassConf);
```

The `setProperty` method is then used to define Cassandra connection properties, such as backend type, host, port, and keyspace. Finally, a Titan graph is created called `titanGraph` using the `open` method. As will be shown later, a Titan graph can be created using a configuration object or a path to a properties file. The properties that have been set match those that were defined in the Cassandra properties file described previously.

The next few sections will show how graphs can be created, and traversed. They will show how Cassandra, HBase, and the file system can be used for storage. Given that I have gone to such lengths to describe the Bash scripts, and the properties files, I will just describe those properties that need to be changed in each instance. I will also provide simple Groovy script snippets in each instance.

Using Cassandra

The Cassandra-based properties file called `cassandra.properties` has already been described, so I will not repeat the details here. This example Groovy script creates a sample graph, and stores it in Cassandra. It has been executed using the **end of file markers** (EOF) to pipe the script to the Gremlin shell:

```
t1 = TitanFactory.open('/home/hadoop/spark/gremlin/cassandra.properties')
GraphOfTheGodsFactory.load(t1)

t1.traversal().V().count()

t1.traversal().V().valueMap()

t1.close()
```

A Titan graph has been created using the `TitanFactory.open` method and the Cassandra properties file. It is called `t1`. The graph of the Gods, an example graph provided with Titan, has been loaded into the graph `t1` using the method `GraphOfTheGodsFactory.load`. A count of vertices (`V()`) has then been generated along with a `ValueMap` to display the contents of the graph. The output looks like this:

```
==>12

==>[name:[jupiter], age:[5000]]
==>[name:[hydra]]
==>[name:[nemean]]
==>[name:[tartarus]]
==>[name:[saturn], age:[10000]]
==>[name:[sky]]
==>[name:[pluto], age:[4000]]
==>[name:[alcmene], age:[45]]
==>[name:[hercules], age:[30]]
==>[name:[sea]]
==>[name:[cerberus]]
==>[name:[neptune], age:[4500]]
```

So, there are 12 vertices in the graph, each has a name and age element shown in the previous data. Having successfully created a graph, it is now possible to configure the previous graph traversal Gremlin command to use Apache Spark for processing. This is simply achieved by specifying the `SparkGraphComputer` in the traversal command. See the full *TinkerPop* diagram at the top of this chapter for architectural details. When this command is executed, you will see the task appear on the Spark cluster user interface:

```
t1.traversal(computer(SparkGraphComputer)).V().count()
```

Using HBase

When using HBase, the properties file needs to change. The following values have been taken from my `hbase.properties` file:

```
gremlin.hadoop.graphInputFormat=com.thinkaurelius.titan.hadoop.formats.  
hbase.HBaseInputFormat
```

```
input.conf.storage.backend=hbase  
input.conf.storage.hostname=hc2r1m2  
input.conf.storage.port=2181  
input.conf.storage.hbase.table=titan  
input.conf.storage.hbase.ext.zookeeper.znode.parent=/hbase
```

Remember that HBase uses Zookeeper for configuration purposes. So, the port number, and server for connection now becomes a zookeeper server, and zookeeper master port 2181. The `znode` parent value in Zookeeper is also defined as the top level node `/hbase`. Of course, the backend type is now defined to be `hbase`.

Also, the `GraphInputFormat` class has been changed to `HBaseInputFormat` to describe HBase as an input source. A Titan graph can now be created using this properties file, as shown in the last section. I won't repeat the graph creation here, as it will be the same as the last section. Next, I will move on to filesystem storage.

Using the filesystem

In order to run this example, I used a basic Gremlin shell (`bin/gremlin.sh`). Within the data directory of the Titan release, there are many example data file formats that can be loaded to create graphs. In this example, I will use the file called `grateful-dead.kryo`. So this time, the data will be loaded straight from the file to a graph without specifying a storage backend, such as Cassandra. The properties file that I will use only contains the following entries:

```
gremlin.graph=org.apache.tinkerpop.gremlin.hadoop.structure.HadoopGraph
```

```
gremlin.hadoop.graphInputFormat=org.apache.tinkerpop.gremlin.hadoop.  
structure.io.gryo.GryoInputFormat  
gremlin.hadoop.graphOutputFormat=org.apache.tinkerpop.gremlin.hadoop.  
structure.io.gryo.GryoOutputFormat  
gremlin.hadoop.jarsInDistributedCache=true  
gremlin.hadoop.deriveMemory=true  
  
gremlin.hadoop.inputLocation=/usr/local/titan/data/grateful-dead.kryo  
gremlin.hadoop.outputLocation=output
```

Again, it uses the Hadoop Gremlin package but this time the graph input and output formats are defined as `GryoInputFormat` and `GryoOutputFormat`. The input location is specified to be the actual kyro-based file. So, the source for input and output is the file. So now, the Groovy script looks like this. First, the graph is created using the properties file. Then, a graph traversal is created, so that we can count vertices and see the structure:

```
graph = GraphFactory.open('/home/hadoop/spark/gremlin/hadoop-gryo.  
properties')  
g1 = graph.traversal()
```

Next, a vertex count is executed, which shows that there are over 800 vertices; and finally, a value map shows the structure of the data, which I have obviously clipped to save the space. But you can see the song name, type, and the performance details:

```
g1.V().count()  
==>808  
g1.V().valueMap()  
==>[name:[MIGHT AS WELL], songType:[original], performances:[111]]  
==>[name:[BROWN EYED WOMEN], songType:[original], performances:[347]]
```

This gives you a basic idea of the available functionality. I am sure that if you search the web, you will find more complex ways of using Spark with Titan. Take this for example:

```
r = graph.compute(SparkGraphComputer.class).  
program(PageRankVertexProgram.build().create()).submit().get()
```

The previous example specifies the use of the `SparkGraphComputer` class using the `compute` method. It also shows how the page rank vertex program, supplied with Titan, can be executed using the `program` method. This would modify your graph by adding page ranks to each vertex. I provide this as an example, as I am not convinced that it will work with Spark at this time.

Summary

This chapter has introduced the Titan graph database from Aurelius. It has shown how it can be installed and configured on a Linux cluster. Using a Titan Gremlin shell example, the graphs have been created, and stored in both HBase and Cassandra NoSQL databases. The choice of Titan storage option required will depend upon your project requirements; HBase HDFS based storage or Cassandra non HDFS based storage. This chapter has also shown that you can use the Gremlin shell both interactively to develop the graph scripts, and with Bash shell scripts so that you can run scheduled jobs with associated logging.

Simple Spark Scala code has been provided, which shows that Apache Spark can access the underlying tables that Titan creates on both HBase and Cassandra. This has been achieved by using the database connector modules provided by Cloudera (for HBase), and DataStax (for Cassandra). All example code and build scripts have been described along with the example output. I have included this Scala-based section to show you that the graph-based data can be accessed in Scala. The previous section processed data from the Gremlin shell, and used Spark as a processing backend. This section uses Spark as the main processing engine, and accesses Titan data from Spark. If the Gremlin shell was not suitable for your requirements, you might consider this approach. As Titan matures, so will the ways in which you can integrate Titan with Spark via Scala.

Finally, Titan's Gremlin shell has been used along with Apache Spark to demonstrate simple methods for creating, and accessing Titan-based graphs. Data has been stored on the file system, Cassandra, and HBase to do this.

Google groups are available for Aurelius and Gremlin users via the URLs at <https://groups.google.com/forum/#!forum/aureliusgraphs> and <https://groups.google.com/forum/#!forum/gremlin-users>.

Although the community seems smaller than other Apache projects, posting volume can be somewhat light, and it can be difficult to get a response to posts.

DataStax, the people who created Cassandra, acquired Aurelius, the creators of Titan this year. The creators of Titan are now involved in the development of DataStax's DSE graph database, which may have a knock-on effect on Titan's development. Having said that, the 0.9.x Titan release has been created, and a 1.0 release is expected.

So, having shown some of the Titan functionality with the help of an example with both Scala and Gremlin, I will close the chapter here. I wanted to show the pairing of Spark-based graph processing, and a graph storage system. I like open source systems for their speed of development and accessibility. I am not saying that Titan is the database for you, but it is a good example. If its future can be assured, and its community grows, then as it matures, it could offer a valuable resource.

Note that two versions of Spark have been used in this chapter: 1.3 and 1.2.1. The earlier version was required, because it was apparently the only version that would work with Titan's `SparkGraphComputer`, and so avoids Kyro serialization errors.

In the next chapter, extensions to the Apache Spark MLlib machine learning library will be examined in terms of the <http://h2o.ai/> H2O product. A neural-based deep learning example will be developed in Scala to demonstrate its potential functionality.

7

Extending Spark with H2O

H2O is an open source system, developed in Java by <http://h2o.ai/> for machine learning. It offers a rich set of machine learning algorithms, and a web-based data processing user interface. It offers the ability to develop in a range of languages: Java, Scala, Python, and R. It also has the ability to interface to Spark, HDFS, Amazon S3, SQL, and NoSQL databases. This chapter will concentrate on H2O's integration with Apache Spark using the **Sparkling Water** component of H2O. A simple example, developed in Scala, will be used, based on real data to create a deep-learning model. This chapter will:

- Examine the H2O functionality
- Consider the necessary Spark H2O environment
- Examine the Sparkling Water architecture
- Introduce and use the H2O Flow interface
- Introduce deep learning with an example
- Consider performance tuning
- Examine data quality

The next step will be to provide an overview of the H2O functionality, and the Sparkling Water architecture that will be used in this chapter.

Overview

Since it is only possible to examine, and use, a small amount of H2O's functionality in this chapter, I thought that it would be useful to provide a list of all of the functional areas that it covers. This list is taken from <http://h2o.ai/> website at <http://h2o.ai/product/algorithms/> and is based upon munging/wrangling data, modeling using the data, and scoring the resulting models:

Process	Model	The score tool
Data profiling	Generalized Linear Models (GLM)	Predict
Summary statistics	Decision trees	Confusion Matrix
Aggregate, filter, bin, and derive columns	Gradient Boosting (GBM)	AUC
Slice, log transform, and anonymize	K-Means	Hit Ratio
Variable creation	Anomaly detection	PCA Score
PCA	Deep learning	Multi Model Scoring
Training and validation sampling plan	Naïve Bayes	
	Grid search	

The following section will explain the environment used for the Spark and H2O examples in this chapter and it will also explain some of the problems encountered.

The processing environment

If any of you have examined my web-based blogs, or read my first book, *Big Data Made Easy*, you will see that I am interested in Big Data integration, and how the big data tools connect. None of these systems exist in isolation. The data will start upstream, be processed in Spark plus H2O, and then the result will be stored, or moved to the next step in the ETL chain. Given this idea in this example, I will use Cloudera CDH HDFS for storage, and source my data from there. I could just as easily use S3, an SQL or NoSQL database.

At the point of starting the development work for this chapter, I had a Cloudera CDH 4.1.3 cluster installed and working. I also had various Spark versions installed, and available for use. They are as follows:

- Spark 1.0 installed as CentOS services
- Spark 1.2 binary downloaded and installed
- Spark 1.3 built from a source snapshot

I thought that I would experiment to see which combinations of Spark, and Hadoop I could get to work together. I downloaded Sparkling water at <http://h2o-release.s3.amazonaws.com/sparkling-water/master/98/index.html> and used the 0.2.12-95 version. I found that the 1.0 Spark version worked with H2O, but the Spark libraries were missing. Some of the functionality that was used in many of the Sparkling Water-based examples was available. Spark versions 1.2 and 1.3 caused the following error to occur:

```
15/04/25 17:43:06 ERROR netty.NettyTransport: failed to bind to
/192.168.1.103:0, shutting down Netty transport
15/04/25 17:43:06 WARN util.Utils: Service 'sparkDriver' could not bind
on port 0. Attempting port 1.
```

The Spark master port number, although correctly configured in Spark, was not being picked up, and so the H2O-based application could not connect to Spark. After discussing the issue with the guys at H2O, I decided to upgrade to an H2O certified version of both Hadoop and Spark. The recommended system versions that should be used are available at <http://h2o.ai/product/recommended-systems-for-h2o/>.

I upgraded my CDH cluster from version 5.1.3 to version 5.3 using the Cloudera Manager interface parcels page. This automatically provided Spark 1.2—the version that has been integrated into the CDH cluster. This solved all the H2O-related issues, and provided me with an H2O-certified Hadoop and Spark environment.

Installing H2O

For completeness, I will show you how I downloaded, installed, and used H2O. Although, I finally settled on version 0.2.12-95, I first downloaded and used 0.2.12-92. This section is based on the earlier install, but the approach used to source the software is the same. The download link changes over time so follow the Sparkling Water download option at <http://h2o.ai/download/>.

This will source the zipped Sparkling water release, as shown by the CentOS Linux long file listing here:

```
[hadoop@hc2r1m2 h2o]$ pwd ; ls -l
/home/hadoop/h2o
total 15892
-rw-r--r-- 1 hadoop hadoop 16272364 Apr 11 12:37 sparkling-
water-0.2.12-92.zip
```

This zipped release file is unpacked using the Linux `unzip` command, and it results in a sparkling water release file tree:

```
[hadoop@hc2r1m2 h2o]$ unzip sparkling-water-0.2.12-92.zip
```

```
[hadoop@hc2r1m2 h2o]$ ls -d sparkling-water*
```

```
sparkling-water-0.2.12-92  sparkling-water-0.2.12-92.zip
```

I have moved the release tree to the `/usr/local/` area using the root account, and created a simple symbolic link to the release called `h2o`. This means that my H2O-based build can refer to this link, and it doesn't need to change as new versions of sparkling water are sourced. I have also made sure, using the Linux `chmod` command, that my development account, `hadoop`, has access to the release:

```
[hadoop@hc2r1m2 h2o]$ su -
```

```
[root@hc2r1m2 ~]# cd /home/hadoop/h2o
```

```
[root@hc2r1m2 h2o]# mv sparkling-water-0.2.12-92 /usr/local
```

```
[root@hc2r1m2 h2o]# cd /usr/local
```

```
[root@hc2r1m2 local]# chown -R hadoop:hadoop sparkling-water-0.2.12-92
```

```
[root@hc2r1m2 local]# ln -s sparkling-water-0.2.12-92 h2o
```

```
[root@hc2r1m2 local]# ls -lrt | grep sparkling
```

```
total 52
```

```
drwxr-xr-x   6 hadoop hadoop 4096 Mar 28 02:27 sparkling-water-0.2.12-92
lrwxrwxrwx   1 root   root    25 Apr 11 12:43 h2o -> sparkling-
water-0.2.12-92
```

The release has been installed on all the nodes of my Hadoop CDH clusters.

The build environment

From past examples, you will know that I favor SBT as a build tool for developing Scala source examples. I have created a development environment on the Linux CentOS 6.5 server called `hc2r1m2` using the `hadoop` development account. The development directory is called `h2o_spark_1_2`:

```
[hadoop@hc2r1m2 h2o_spark_1_2]$ pwd
```

```
/home/hadoop/spark/h2o_spark_1_2
```

My SBT build configuration file named `h2o.sbt` is located here; it contains the following:

```
[hadoop@hc2r1m2 h2o_spark_1_2]$ more h2o.sbt

name := "H 2 O"

version := "1.0"

scalaVersion := "2.10.4"

libraryDependencies += "org.apache.hadoop" % "hadoop-client" % "2.3.0"

libraryDependencies += "org.apache.spark" % "spark-core" % "1.2.0" from
"file:///opt/cloudera/parcels/CDH-5.3.3-1.cdh5.3.3.p0.5/jars/spark-
assembly-1.2.0-cdh5.3.3-hadoop2.5.0-cdh5.3.3.jar"

libraryDependencies += "org.apache.spark" % "mllib" % "1.2.0" from
"file:///opt/cloudera/parcels/CDH-5.3.3-1.cdh5.3.3.p0.5/jars/spark-
assembly-1.2.0-cdh5.3.3-hadoop2.5.0-cdh5.3.3.jar"

libraryDependencies += "org.apache.spark" % "sql" % "1.2.0" from
"file:///opt/cloudera/parcels/CDH-5.3.3-1.cdh5.3.3.p0.5/jars/spark-
assembly-1.2.0-cdh5.3.3-hadoop2.5.0-cdh5.3.3.jar"

libraryDependencies += "org.apache.spark" % "h2o" % "0.2.12-95" from
"file:///usr/local/h2o/assembly/build/libs/sparkling-water-assembly-
0.2.12-95-all.jar"

libraryDependencies += "hex.deeplearning" % "DeepLearningModel" %
"0.2.12-95" from "file:///usr/local/h2o/assembly/build/libs/sparkling-
water-assembly-0.2.12-95-all.jar"

libraryDependencies += "hex" % "ModelMetricsBinomial" % "0.2.12-95"
from "file:///usr/local/h2o/assembly/build/libs/sparkling-water-assembly-
0.2.12-95-all.jar"

libraryDependencies += "water" % "Key" % "0.2.12-95" from "file:///usr/
local/h2o/assembly/build/libs/sparkling-water-assembly-0.2.12-95-all.jar"

libraryDependencies += "water" % "fvec" % "0.2.12-95" from "file:///usr/
local/h2o/assembly/build/libs/sparkling-water-assembly-0.2.12-95-all.jar"
```

I have provided SBT configuration examples in the previous chapters, so I won't go into the line-by-line-detail here. I have used the file-based URLs to define the library dependencies, and have sourced the Hadoop JAR files from the Cloudera parcel path for the CDH install. The Sparkling Water JAR path is defined as `/usr/local/h2o/` that was just created.

I use a Bash script called `run_h2o.bash` within this development directory to execute my H2O-based example code. It takes the application class name as a parameter, and is shown below:

```
[hadoop@hc2r1m2 h2o_spark_1_2]$ more run_h2o.bash
```

```
#!/bin/bash
```

```
SPARK_HOME=/opt/cloudera/parcels/CDH
```

```
SPARK_LIB=$SPARK_HOME/lib
```

```
SPARK_BIN=$SPARK_HOME/bin
```

```
SPARK_SBIN=$SPARK_HOME/sbin
```

```
SPARK_JAR=$SPARK_LIB/spark-assembly-1.2.0-cdh5.3.3-hadoop2.5.0-  
cdh5.3.3.jar
```

```
H2O_PATH=/usr/local/h2o/assembly/build/libs
```

```
H2O_JAR=$H2O_PATH/sparkling-water-assembly-0.2.12-95-all.jar
```

```
PATH=$SPARK_BIN:$PATH
```

```
PATH=$SPARK_SBIN:$PATH
```

```
export PATH
```

```
cd $SPARK_BIN
```

```
./spark-submit \
```

```
--class $1 \
```

```
--master spark://hc2nn.semtech-solutions.co.nz:7077 \
```

```
--executor-memory 85m \
```

```
--total-executor-cores 50 \
```

```
--jars $H2O_JAR \
```

```
/home/hadoop/spark/h2o_spark_1_2/target/scala-2.10/h-2-o_2.10-1.0.jar
```

This example of Spark application submission has already been covered, so again, I won't get into the detail. Setting the executor memory at a correct value was critical to avoiding out-of-memory issues and performance problems. This will be examined in the *Performance Tuning* section.

As in the previous examples, the application Scala code is located in the `src/main/scala` subdirectory, under the `development` directory level. The next section will examine the Apache Spark, and the H2O architecture.

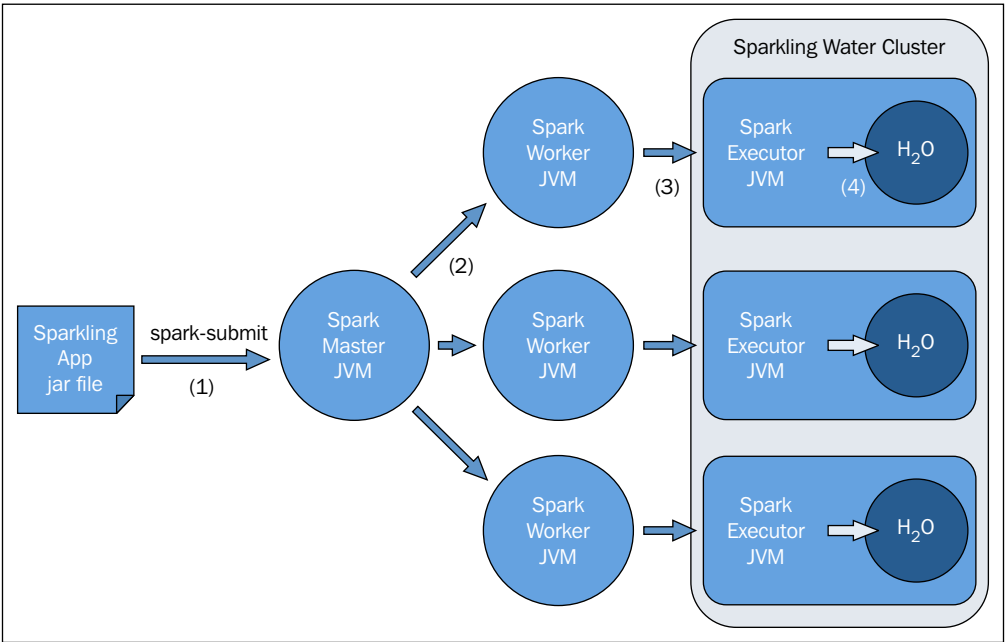
Architecture

The diagrams in this section have been sourced from the <http://h2o.ai/> web site at <http://h2o.ai/blog/2014/09/how-sparkling-water-brings-h2o-to-spark/> to provide a clear method of describing the way in which H2O Sparkling Water can be used to extend the functionality of Apache Spark. Both, H2O and Spark are open source systems. Spark MLlib contains a great deal of functionality, while H2O extends this with a wide range of extra functionality, including deep learning. It offers tools to *munge* (transform), model, and score the data. It also offers a web-based user interface to interact with.

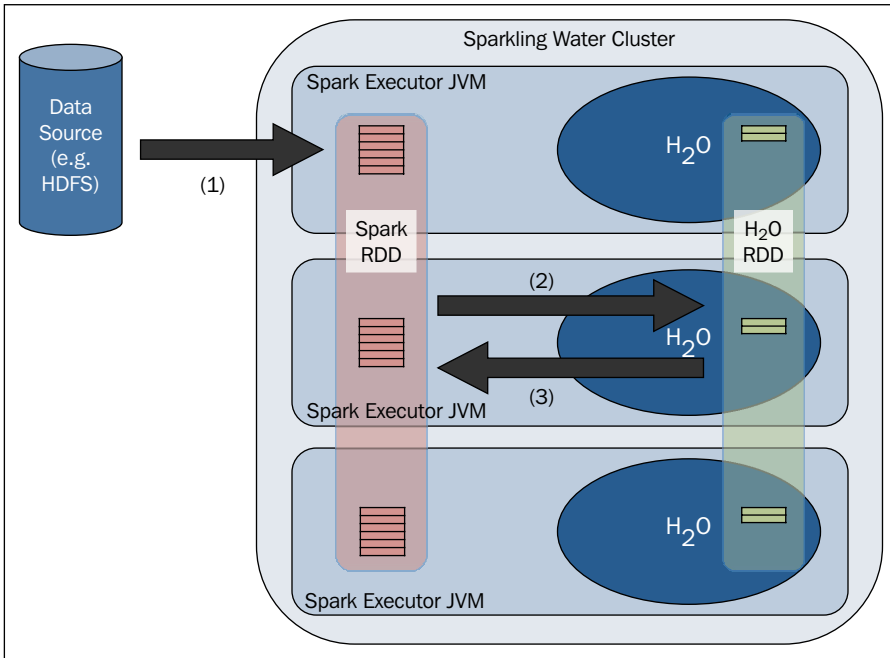
The next diagram, borrowed from <http://h2o.ai/>, shows how H2O integrates with Spark. As we already know, Spark has master and worker servers; the workers create executors to do the actual work. The following steps occur to run a Sparkling water-based application:

1. Spark's `submit` command sends the sparkling water JAR to the Spark master.
2. The Spark master starts the workers, and distributes the JAR file.
3. The Spark workers start the executor JVMs to carry out the work.
4. The Spark executor starts an H2O instance.

The H2O instance is embedded with the Executor JVM, and so it shares the JVM heap space with Spark. When all of the H2O instances have started, H2O forms a cluster, and then the H2O flow web interface is made available.



The preceding diagram explains how H2O fits into the Apache Spark architecture, and how it starts, but what about data sharing? How does data pass between Spark and H2O? The following diagram explains this:



A new H2O RDD data structure has been created for H2O and Sparkling Water. It is a layer, based at the top of an H2O frame, each column of which represents a data item, and is independently compressed to provide the best compression ratio.

In the deep learning example, Scala code presented later in this chapter you will see that a data frame has been created implicitly from a Spark schema RDD and a columnar data item, income has been enumerated. I won't dwell on this now as it will be explained later but this is a practical example of the above architecture:

```
val testFrame:DataFrame = schemaRddTest
testFrame.replace( testFrame.find("income"), testFrame.
vec("income").toEnum)
```

In the Scala-based example that will be tackled in this chapter, the following actions will take place:

1. Data is being sourced from HDFS, and is being stored in a Spark RDD.
2. Spark SQL is used to filter data.
3. The Spark schema RDD is converted into an H2O RDD.
4. The H2O-based processing and modeling occurs.
5. The results are passed back to Spark for accuracy checking.

To this point, the general architecture of H2O has been examined, and the product has been sourced for use. The development environment has been explained, and the process by which H2O and Spark integrate has been considered. Now, it is time to delve into a practical example of the use of H2O. First though, some real-world data must be sourced for modeling purposes.

Sourcing the data

Since I have already used the **Artificial Neural Net (ANN)** functionality in *Chapter 2, Apache Spark MLlib*, to classify images, it seems only fitting that I use H2O deep learning to classify data in this chapter. In order to do this, I need to source data sets that are suitable for classification. I need either image data with associated image labels, or the data containing vectors and a label that I can enumerate, so that I can force H2O to use its classification algorithm.

The MNIST test and training image data was sourced from `ann.lecun.com/exdb/mnist/`. It contains 50,000 training rows, and 10,000 rows for testing. It contains digital images of numbers 0 to 9 and associated labels.

I was not able to use this data as, at the time of writing, there was a bug in H2O Sparkling water that limited the record size to 128 elements. The MNIST data has a record size of $28 \times 28 + 1$ elements for the image plus the label:

```
15/05/14 14:05:27 WARN TaskSetManager: Lost task 0.0 in stage
9.0 (TID 256, hc2r1m4.semtech-solutions.co.nz): java.lang.
ArrayIndexOutOfBoundsException: -128
```

This issue should have been fixed and released by the time you read this, but in the short term I sourced another data set called income from `http://www.cs.toronto.edu/~delve/data/datasets.html`, which contains Canadian employee income data. The following information shows the attributes and the data volume. It also shows the list of columns in the data, and a sample row of the data:

Number of attributes: 16

Number of cases: 45,225

```
age workclass fnlwgt education educational-num marital-status occupation
relationship race gender capital-gain capital-loss hours-per-week native-
country income
```

```
39, State-gov, 77516, Bachelors, 13, Never-married, Adm-clerical, Not-in-
family, White, Male, 2174, 0, 40, United-States, <=50K
```

I will enumerate the last column in the data – the income bracket, so $\leq 50k$ will enumerate to 0. This will allow me to force the H2O deep learning algorithm to carry out classification rather than regression. I will also use Spark SQL to limit the data columns, and filter the data.

Data quality is absolutely critical when creating an H2O-based example like that described in this chapter. The next section examines the steps that can be taken to improve the data quality, and so save time.

Data Quality

When I import CSV data files from HDFS to my Spark Scala H2O example code, I can filter the incoming data. The following example code contains two filter lines; the first checks that a data line is not empty, while the second checks that the final column in each data row (income), which will be enumerated, is not empty:

```
val testRDD = rawTestData
  .filter(!_isEmpty)
  .map(_.split(","))
  .filter( rawRow => ! rawRow(14).trim.isEmpty )
```

I also needed to clean my raw data. There are two data sets, one for training and one for testing. It is important that the training and testing data have the following:

- The same number of columns
- The same data types
- The null values must be allowed for in the code
- The enumerated type values must match—especially for the labels

I encountered an error related to the enumerated label column income and the values that it contained. I found that my test data set rows were terminated with a full stop character "." When processed, this caused the training and the test data values to mismatch when enumerated.

So, I think that time and effort should be spent safeguarding the data quality, as a pre-step to training, and testing machine learning functionality so that time is not lost, and extra cost incurred.

Performance tuning

It is important to monitor the Spark application error and the standard output logs in the Spark web user interface if you see errors like the following:

```
05-15 13:55:38.176 192.168.1.105:54321 6375 Thread-10 ERROR: Out
of Memory and no swap space left from hc2r1m1.semtech-solutions.
co.nz/192.168.1.105:54321
```

If you encounter instances where application executors seem to hang without response, you may need to tune your executor memory. You need to do so if you see an error like the following in your executor log:

```
05-19 13:46:57.300 192.168.1.105:54321 10044 Thread-11 WARN: Unblock
allocations; cache emptied but memory is low: OOM but cache is emptied:
MEM_MAX = 89.5 MB, DESIRED_CACHE = 96.4 MB, CACHE = N/A, POJO = N/A, this
request bytes = 36.4 MB
```

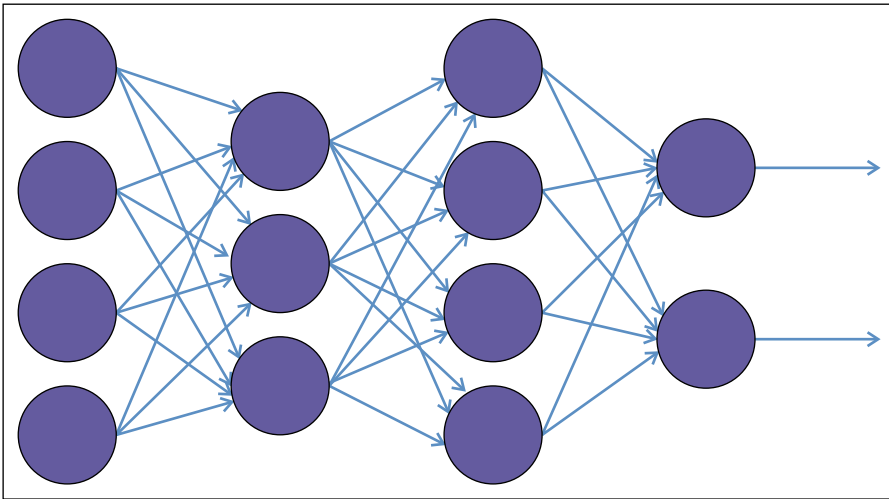
This can cause a loop, as the application requests more memory than is available, and so waits until the next iteration retries. The application can seem to hang until the executors are killed, and the tasks re-executed on alternate nodes. A short task's run time can extend considerably due to such problems.

Monitor the Spark logs for these types of error. In the previous example, changing the executor memory setting in the `spark-submit` command removes the error, and reduces the runtime substantially. The memory value requested has been reduced to a figure below that which is available.

```
--executor-memory 85m
```

Deep learning

Neural networks were introduced in *Chapter 2, Apache Spark MLlib*. This chapter builds upon this understanding by introducing deep learning, which uses deep neural networks. These are neural networks that are feature-rich, and contain extra hidden layers, so that their ability to extract data features is increased. These networks are generally feed-forward networks, where the feature characteristics are inputs to the input layer neurons. These neurons then fire and spread the activation through the hidden layer neurons to an output layer, which should present the feature label values. Errors in the output are then propagated back through the network (at least in back propagation), adjusting the neuron connection weight matrices so that classification errors are reduced during training.



The previous example image, described in the H2O booklet at <https://leanpub.com/deeplearning/read>, shows a deep learning network with four input neurons to the left, two hidden layers in the middle, and two output neurons. The arrows show both the connections between neurons and the direction that activation takes through the network.

These networks are feature-rich because they provide the following options:

- Multiple training algorithms
- Automated network configuration
- The ability to configure many options
 - Structure
 - Hidden layer structure
 - Training
 - Learning rate, annealing, and momentum

So, after giving this brief introduction to deep learning, it is now time to look at some of the sample Scala-based code. H2O provides a great deal of functionality; the classes that are needed to build and run the network have been developed for you. You just need to do the following:

- Prepare the data and parameters
- Create and train the model

- Validate the model with a second data set
- Score the validation data set output

When scoring your model, you must hope for a high value in percentage terms. Your model must be able to accurately predict and classify your data.

Example code – income

This section examines the Scala-based H2O Sparkling Water deep learning example using the previous Canadian income data source. First, the Spark (Context, Conf, mllib, and RDD), and H2O (h2o, deeplearning, and water) classes are imported:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

import hex.deeplearning.{DeepLearningModel, DeepLearning}
import hex.deeplearning.DeepLearningModel.DeepLearningParameters
import org.apache.spark.h2o._
import org.apache.spark.mllib
import org.apache.spark.mllib.feature.{IDFModel, IDF, HashingTF}
import org.apache.spark.rdd.RDD
import water.Key
```

Next an application class called `h2o_spark_dl2` is defined, the master URL is created, and then a configuration object is created, based on this URL, and the application name. The Spark context is then created using the configuration object:

```
object h2o_spark_dl2 extends App
{
  val sparkMaster = "spark://hc2nn.semtech-solutions.co.nz:7077"
  val appName = "Spark h2o ex1"
  val conf = new SparkConf()

  conf.setMaster(sparkMaster)
  conf.setAppName(appName)

  val sparkCxt = new SparkContext(conf)
```

An H2O context is created from the Spark context, and also an SQL context:

```
import org.apache.spark.h2o._
implicit val h2oContext = new org.apache.spark.h2o.
H2OContext(sparkCxt).start()

import h2oContext._
import org.apache.spark.sql._

implicit val sqlContext = new SQLContext(sparkCxt)
```

The H2O Flow user interface is started with the `openFlow` command:

```
import sqlContext._
openFlow
```

The training and testing of the data files are now defined (on HDFS) using the server URL, path, and the file names:

```
val server    = "hdfs://hc2nn.semtech-solutions.co.nz:8020"
val path      = "/data/spark/h2o/"

val train_csv = server + path + "adult.train.data" // 32,562 rows
val test_csv  = server + path + "adult.test.data"  // 16,283 rows
```

The CSV based training and testing data is loaded using the Spark context's `textFile` method:

```
val rawTrainData = sparkCxt.textFile(train_csv)
val rawTestData  = sparkCxt.textFile(test_csv)
```

Now, the schema is defined in terms of a string of attributes. Then, a schema variable is created by splitting the string using a series of `StructField`, based on each column. The data types are left as `String`, and the `true` value allows for the Null values in the data:

```
val schemaString = "age workclass fnlwgt education " +
"educationalnum maritalstatus " + "occupation relationship race
gender " + "capitalgain capitalloss " + "hourspersweek nativecountry
income"

val schema = StructType( schemaString.split(" ")
    .map(fieldName => StructField(fieldName, StringType, true)))
```

The raw CSV line training and testing data is now split by commas into columns. The data is filtered on empty lines to ensure that the last column (income) is not empty. The actual data rows are created from the fifteen (0-14) trimmed elements in the raw CSV data. Both, the training and the test data sets are processed:

```
val trainRDD = rawTrainData
    .filter(!_isEmpty)
    .map(_.split(","))
    .filter( rawRow => ! rawRow(14).trim.isEmpty )
    .map(rawRow => Row(
        rawRow(0).toString.trim, rawRow(1).toString.trim,
        rawRow(2).toString.trim, rawRow(3).toString.trim,
        rawRow(4).toString.trim, rawRow(5).toString.trim,
        rawRow(6).toString.trim, rawRow(7).toString.trim,
        rawRow(8).toString.trim, rawRow(9).toString.trim,
        rawRow(10).toString.trim, rawRow(11).toString.trim,
        rawRow(12).toString.trim, rawRow(13).toString.trim,
        rawRow(14).toString.trim
    )
)

val testRDD = rawTestData
    .filter(!_isEmpty)
    .map(_.split(","))
    .filter( rawRow => ! rawRow(14).trim.isEmpty )
    .map(rawRow => Row(
        rawRow(0).toString.trim, rawRow(1).toString.trim,
        rawRow(2).toString.trim, rawRow(3).toString.trim,
        rawRow(4).toString.trim, rawRow(5).toString.trim,
        rawRow(6).toString.trim, rawRow(7).toString.trim,
        rawRow(8).toString.trim, rawRow(9).toString.trim,
        rawRow(10).toString.trim, rawRow(11).toString.trim,
        rawRow(12).toString.trim, rawRow(13).toString.trim,
        rawRow(14).toString.trim
    )
)
```

Spark Schema RDD variables are now created for the training and test data sets by applying the schema variable, created previously for the data using the Spark context's `applySchema` method:

```
val trainSchemaRDD = sqlContext.applySchema(trainRDD, schema)
val testSchemaRDD = sqlContext.applySchema(testRDD, schema)
```

Temporary tables are created for the training and testing data:

```
trainSchemaRDD.registerTempTable("trainingTable")
testSchemaRDD.registerTempTable("testingTable")
```

Now, SQL is run against these temporary tables, both to filter the number of columns, and to potentially limit the data. I could have added a `WHERE` or `LIMIT` clause. This is a useful approach that enables me to manipulate both the column and row-based data:

```
val schemaRddTrain = sqlContext.sql(
  """SELECT
    |age,workclass,education,maritalstatus,
    |occupation,relationship,race,
    |gender,hoursperweek,nativecountry,income
    |FROM trainingTable """.stripMargin)

val schemaRddTest = sqlContext.sql(
  """SELECT
    |age,workclass,education,maritalstatus,
    |occupation,relationship,race,
    |gender,hoursperweek,nativecountry,income
    |FROM testingTable """.stripMargin)
```

The H2O data frames are now created from the data. The final column in each data set (income) is enumerated, because this is the column that will form the deep learning label for the data. Also, enumerating this column forces the deep learning model to carry out classification rather than regression:

```
val trainFrame:DataFrame = schemaRddTrain
trainFrame.replace( trainFrame.find("income"),          trainFrame.
vec("income").toEnum)
trainFrame.update(null)

val testFrame:DataFrame = schemaRddTest
testFrame.replace( testFrame.find("income"),            testFrame.
vec("income").toEnum)
testFrame.update(null)
```

The enumerated results data income column is now saved so that the values in this column can be used to score the tested model prediction values:

```
val testResArray = schemaRddTest.collect()
val sizeResults  = testResArray.length
```


The same general approach to deep learning can be taken to data processing as the previous example, apart from the actual processing of the raw CSV data. There are too many columns to process individually, and they all need to be converted into integers to represent their data type. This can be done in one of two ways. In the first example, `var args` can be used to process all the elements in the row:

```
val trainRDD = rawTrainData.map( rawRow => Row( rawRow.split(",") .
map(_ .toInt):_* ))
```

The second example uses the `fromSeq` method to process the row elements:

```
val trainRDD = rawTrainData.map(rawRow => Row.fromSeq(rawRow.
split(",") .map(_ .toInt)))
```

In the next section, the H2O Flow user interface will be examined to see how it can be used to both monitor H2O and process the data.

H2O Flow

H2O Flow is a web-based open source user interface for H2O, and given that it is being used with Spark, Sparkling Water. It is a fully functional H2O web interface for monitoring the H2O Sparkling Water cluster plus jobs, and also for manipulating data and training models. I have created some simple example code to start the H2O interface. As in the previous Scala-based code samples, all I need to do is create a Spark, an H2O context, and then call the `openFlow` command, which will start the Flow interface.

The following Scala code example just imports classes for Spark context, configuration, and H2O. It then defines the configuration in terms of the application name and the Spark cluster URL. A Spark context is then created using the configuration object:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.h2o._

object h2o_spark_ex2 extends App
{
  val sparkMaster = "spark://hc2nn.semtech-solutions.co.nz:7077"
  val appName = "Spark h2o ex2"
  val conf = new SparkConf()

  conf.setMaster(sparkMaster)
  conf.setAppName(appName)

  val sparkCxt = new SparkContext(conf)
```

An H2O context is then created, and started using the Spark context. The H2O context classes are imported, and the Flow user interface is started with the `openFlow` command:

```
implicit val h2oContext = new org.apache.spark.h2o.
H2OContext(sparkCxt).start()

import h2oContext._

// Open H2O UI

openFlow
```

Note, for the purposes of this example and to enable me to use the Flow application, I have commented out the H2O shutdown and the Spark context stop options. I would not normally do this, but I wanted to make this application long-running so that it gives me plenty of time to use the interface:

```
// shutdown h2o

// water.H2O.shutdown()
// sparkCxt.stop()

println( " >>>> Script Finished <<<< " )

} // end application
```

I use my Bash script `run_h2o.bash` with the application class name called `h2o_spark_ex2` as a parameter. This script contains a call to the `spark-submit` command, which will execute the compiled application:

```
[hadoop@hc2r1m2 h2o_spark_1_2]$ ./run_h2o.bash h2o_spark_ex2
```

When the application runs, it lists the state of the H2O cluster and provides a URL by which the H2O Flow browser can be accessed:

```
15/05/20 13:00:21 INFO H2OContext: Sparkling Water started, status of
context:
```

```
Sparkling Water Context:
```

```
* number of executors: 4
```

```
* list of used executors:
```

```
(executorId, host, port)
```

```
-----
```

```
(1,hc2r1m4.semtech-solutions.co.nz,54321)
```

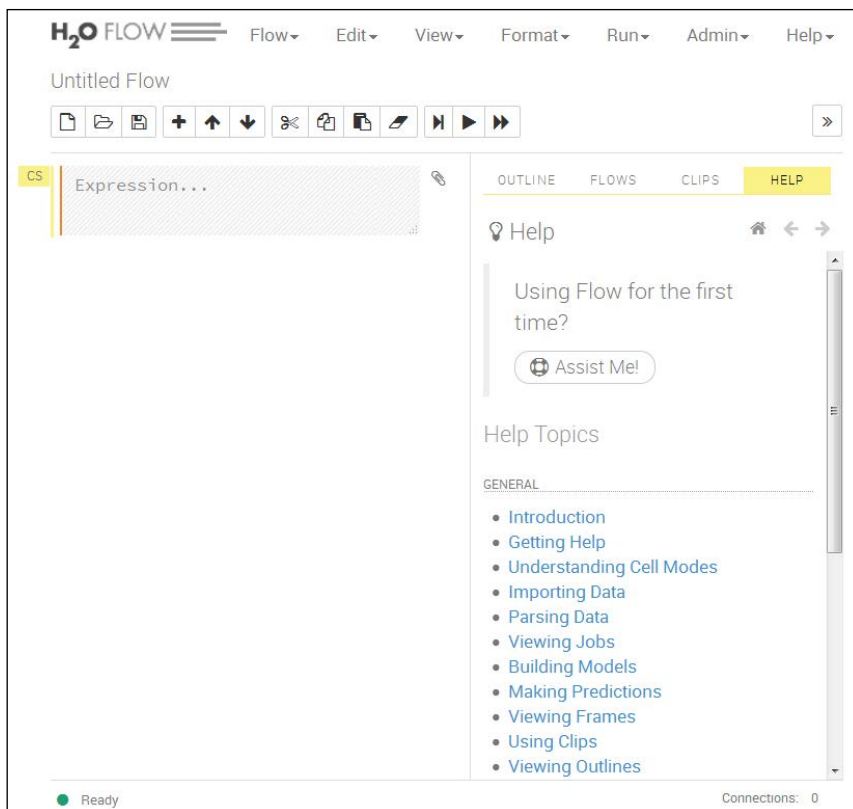
```
(3, hc2r1m2.semtech-solutions.co.nz, 54321)
(0, hc2r1m3.semtech-solutions.co.nz, 54321)
(2, hc2r1m1.semtech-solutions.co.nz, 54321)
-----
```

Open H2O Flow in browser: <http://192.168.1.108:54323> (CMD + click in Mac OSX)

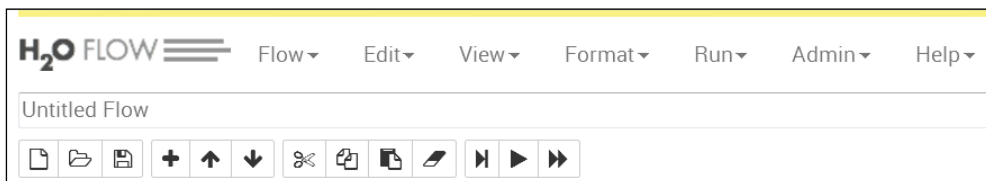
The previous example shows that I can access the H2O interface using the port number 54323 on the host IP address 192.168.1.108. I can simply check my host's file to confirm that the host name is hc2r1m2:

```
[hadoop@hc2nn ~]$ cat /etc/hosts | grep hc2
192.168.1.103 hc2nn.semtech-solutions.co.nz hc2nn
192.168.1.105 hc2r1m1.semtech-solutions.co.nz hc2r1m1
192.168.1.108 hc2r1m2.semtech-solutions.co.nz hc2r1m2
192.168.1.109 hc2r1m3.semtech-solutions.co.nz hc2r1m3
192.168.1.110 hc2r1m4.semtech-solutions.co.nz hc2r1m4
```

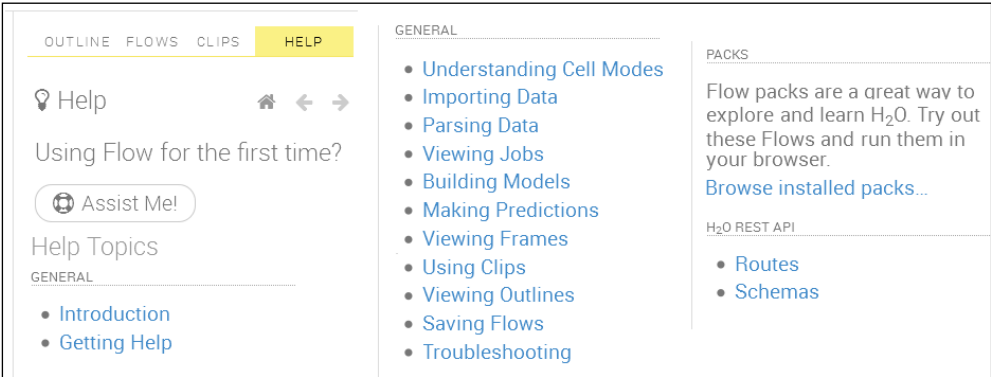
So, I can access the interface using the hc2r1m2:54323 URL. The following screenshot shows the Flow interface with no data loaded. There are data processing and administration menu options and buttons at the top of the page. To the right, there are help options to enable you to learn more about H2O:



The following screenshot shows the menu options and buttons in greater detail. In the following sections, I will use a practical example to explain some of these options, but there will not be enough space in this chapter to cover all the functionality. Check the <http://h2o.ai/> website to learn about the Flow application in detail, available at <http://h2o.ai/product/flow/>:



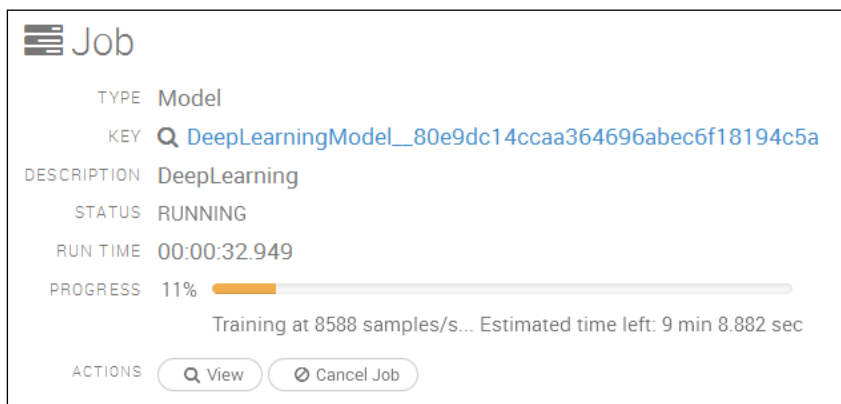
In greater definition, you can see that the previous menu options and buttons allow you to both administer your H2O Spark cluster, and also manipulate the data that you wish to process. The following screenshot shows a reformatted list of the help options available, so that, if you get stuck, you can investigate solving your problem from the same interface:



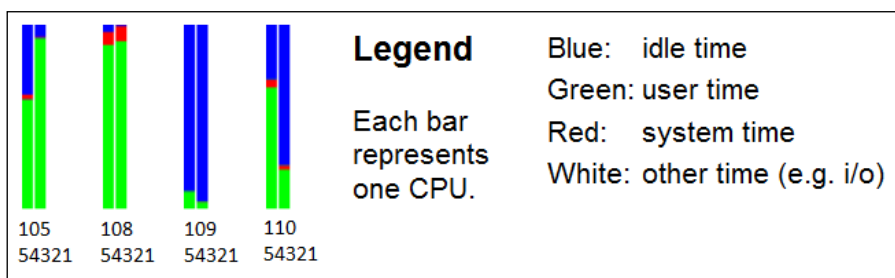
If I use the menu option, **Admin | Cluster Status**, I will obtain the following screenshot, which shows me the status of each cluster server in terms of memory, disk, load, and cores. It's a useful snapshot that provides me with a color-coded indication of the status:

sparkling-water-hadoop									
CLOUD STATUS									
HEALTHY		CONSENSUS		LOCKED		VERSION	STARTED	NODES (USED / ALL)	Show advanced
N O D E S						0.3.0.1109	3 minutes ago	4 / 4	
NAME	PING	CORES	LOAD	DATA	DATA (%)	GC (FREE / TOTAL / MAX)		DISK (FREE / MAX)	DISK
192.168.1.105:54321	few seconds	2	0.120	- / -	NaN%	25.49 MB / 83.00 MB / 83.00 MB		39.02 GB / 49.21 GB	79%
192.168.1.108:54321	few seconds	2	0.200	- / -	NaN%	31.76 MB / 79.00 MB / 79.00 MB		39.08 GB / 49.21 GB	79%
192.168.1.109:54321	few seconds	2	0.070	- / -	NaN%	28.03 MB / 79.00 MB / 79.00 MB		39.41 GB / 49.21 GB	80%
192.168.1.110:54321	few seconds	2	0.080	- / -	NaN%	29.57 MB / 79.50 MB / 79.50 MB		39.30 GB / 49.21 GB	79%
TOTAL	-	8	0.470	- / -	NaN%	114.84 MB / 320.50 MB / 320.50 MB		156.81 GB / 196.86 GB	79%

The menu option, **Admin | Jobs**, provides details of the current cluster jobs in terms of the start, end, and run times, as well as status. Clicking on the job name provides further details, as shown next, including data processing details, and an estimated run time, which is useful. Also, if you select the **Refresh** button, the display will continuously refresh until it is deselected:



The **Admin | Water Meter** option provides a visual display of the CPU usage on each node in the cluster. As you can see in the following screenshot, my meter shows that my cluster was idle:



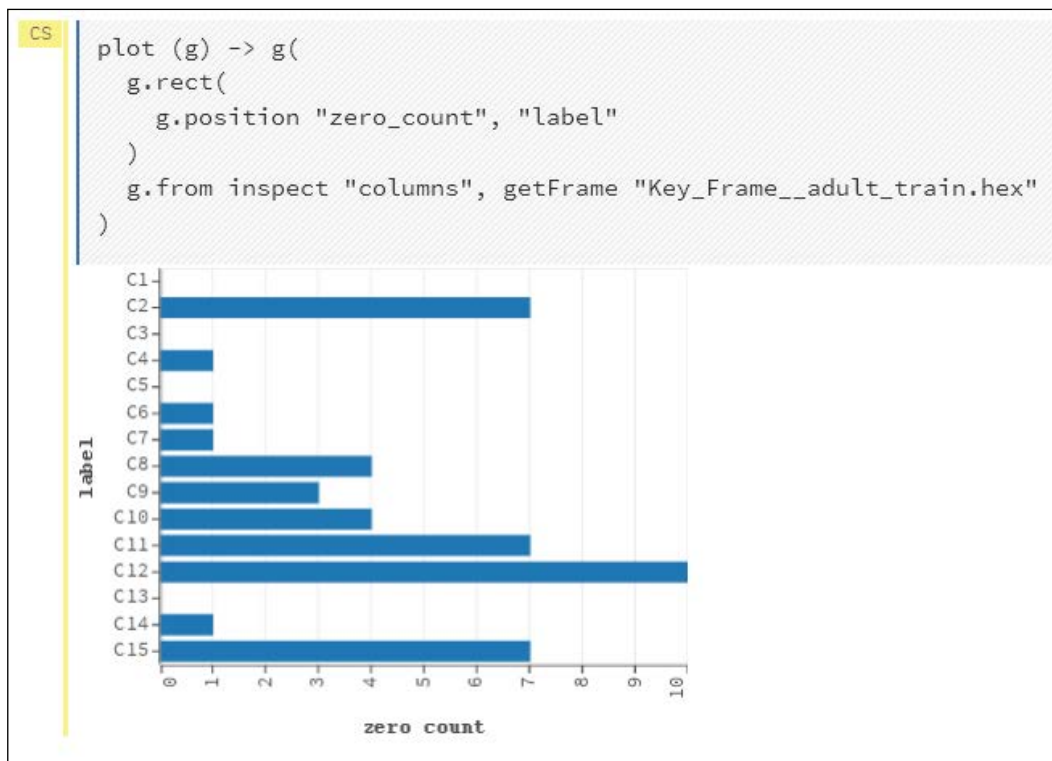
Using the menu option, **Flow | Upload File**, I have uploaded some of the training data used in the previous deep learning Scala-based example. The data has been loaded into a data preview pane; I can see a sample of the data that has been organized into cells. Also, an accurate guess has been made of the data types so that I can see which columns can be enumerated. This is useful if I want to consider classification:

DATA PREVIEW						
Numeric ▾	Enum ▾	Numeric ▾	Enum ▾	Numeric ▾	Enum ▾	Enum ▾
39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical
50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial
38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners
53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners

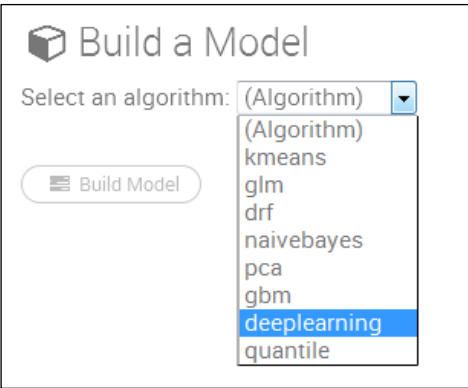
Having loaded the data, I am now presented with a **Frame** display, which offers me the ability to view, inspect, build a model, create a prediction, or download the data. The data display shows information like min, max, and mean. It shows data types, labels, and a zero data count, as shown in the following screenshot:

Key_Frame__adult_train.hex								
ACTIONS: View Data Inspect Build Model... Predict Download								
ROWS		COLUMNS		COMPRESSED SIZE				
10		15		1KB				
LABEL	MISSING_COUNT	ZERO_COUNT	POSITIVE_INFINITY_COUNT	NEGATIVE_INFINITY_COUNT	MIN	MAX	MEAN	
C1	0	0	0	0	28	53	41.9	
C2	0	7	0	0	0	2	0.4	
C3	0	0	0	0	45781	338409	180924.4	
C4	0	1	0	0	0	4	2.3	
C5	0	0	0	0	5	14	11	
C6	0	1	0	0	0	3	1.4	

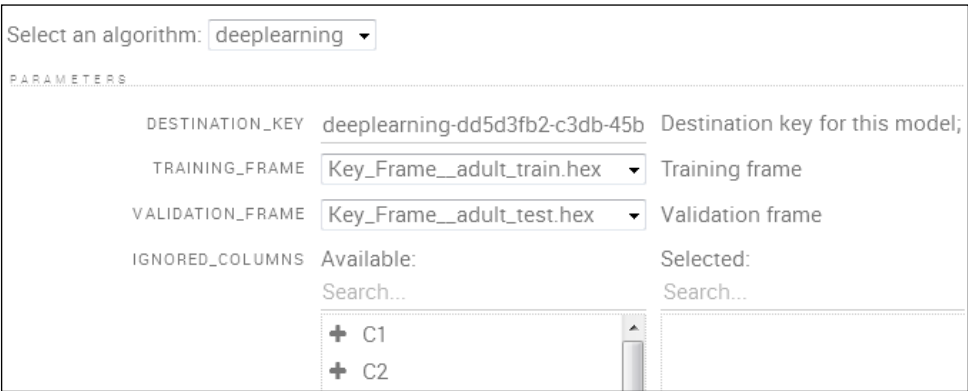
I thought that it would be useful to create a deep learning classification model, based on this data, to compare the Scala-based approach to this H2O user interface. Using the view and inspect options, it is possible to visually, and interactively check the data, as well as create plots relating to the data. For instance, using the previous inspect option followed by the plot columns option, I was able to create a plot of data labels versus zero counts in the column data. The following screenshot shows the result:



By selecting the build model option, a menu option is offered that lets me choose a model type. I will select deep learning, as I already know that this data is suited to this classification approach. The previous Scala-based model resulted in an accuracy level of 83 percent:



I have selected the deep learning option. Having chosen this option, I am then able to set model parameters, such as training and validation data sets, as well as choosing the data columns that my model should use (obviously, the two data sets should contain the same columns). The following screenshot displays the data sets, and the model columns being selected:



There are a large range of basic and advanced model options available. A selection of them are shown in the following screenshot. I have set the response column to 15 as the income column. I have also set the **VARIABLE_IMPORTANCES** option. Note that I don't need to enumerate the response column, as it has been done automatically:

DROPN_A20_COLS ☐		CHECKPOINT	
RESPONSE_COLUMN	C15	USE_ALL_FACTOR_LEVELS	☒
N_FOLDS	0	TRAIN_SAMPLES_PER_ITERATION	-2
ACTIVATION	Rectifier	ADAPTIVE_RATE	☒
HIDDEN	200, 200	RHO	0.99
EPOCHS	100	EPSILON	1e-8
VARIABLE_IMPORTANCES	☒	INPUT_DROPOUT_RATIO	0
REPLICATE_TRAINING_DATA	☒	L1	0
		L2	0

Note also that the epochs or iterations option is set to **100** as before. Also, the figure 200, 200 for the hidden layers indicates that the network has two hidden layers, each with 200 neurons. Selecting the build model option causes the model to be created from these parameters. The following screenshot shows the model being trained, including an estimation of training time and an indication of the data processed so far.

CS

```
buildModel 'deeplearning', {"destination_key":"deeplearning-ded16717-75cb-4014-a1fb-b99d257a3056","training_frame":"Key_Frame__adult_train2.hex","validation_frame":"Key_Frame__adult_val2.hex","response_column":"C15","activation":"Rectifier","hidden":[200,200],"epochs":"100","variable_importances":true,"replicate_training_data":true}
```

Job

TYPE Model

KEY deeplearning-ded16717-75cb-4014-a1fb-b99d257a3056

DESCRIPTION DeepLearning

STATUS RUNNING

RUN TIME 00:00:57.911

PROGRESS 13%

Scoring on 10018 training samples, 16281 validation samples)

ACTIONS [View](#) [Cancel Job](#)

Viewing the model, once trained, shows training and validation metrics, as well as a list of the important training parameters:

 Model

KEY: deeplearning-ded16717-75cb-4014-a1fb-b99d257a3056

ALGORITHM: deeplearning

ACTIONS: ⚡ Predict... 📄 Clone this model... 🔍 Inspect 🗑 Delete

▶ MODEL PARAMETERS

▼ TRAINING METRICS

MODEL_CATEGORY	AUC	GINI	MSE	URATION_IN_MS	SCORING_TIME
Binomial	0.917392	0.834784	0.097503	0	0

▼ VALIDATION METRICS

MODEL_CATEGORY	AUC	GINI	MSE	DURATION_IN_MS	SCORING_TIME
Binomial	0.908921	0.817843	0.101004	0	0

▼ VARIABLE IMPORTANCES

C9.White-

C14.United-States-

C6.Married-civ-spouse-

C11-

Selecting the **Predict** option allows an alternative validation data set to be specified. Choosing the **Predict** option using the new data set causes the already trained model to be validated against a new test dataset:

 Predict

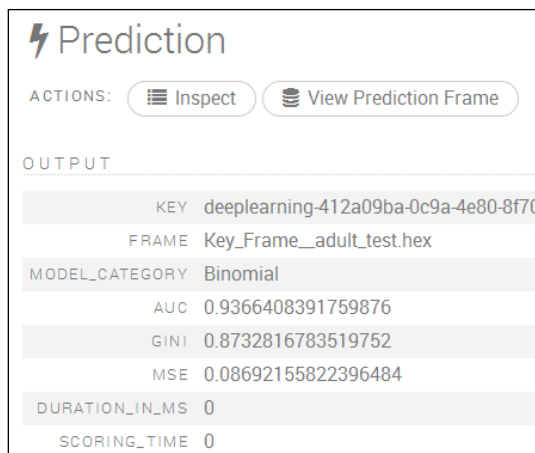
KEY:

MODEL: deeplearning-ded16717-75cb-4014-a1fb-b99d257a3056

FRAME:

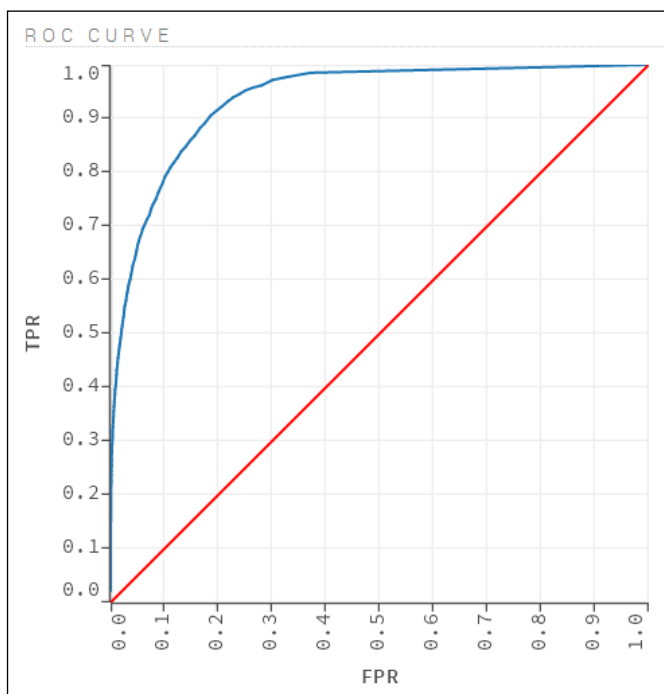
ACTIONS: ⚡ Predict

Selecting the **Predict** option causes the prediction details for the deep learning model, and dataset to be displayed as shown in the following screenshot:



The preceding screenshot shows the test data frame and the model category, as well as the validation statistics in terms of AUC, GINI, and MSE.

The AUC value, or area under the curve, relates to the ROC, or the receiver operator characteristics curve, which is also shown in the following screenshot. TPR means **True Positive Rate**, and FPR means **False Positive Rate**. AUC is a measure of accuracy with a value of one being perfect. So, the blue line shows greater accuracy than that of the red line:



There is a great deal of functionality available within this interface that I have not explained, but I hope that I have given you a feel for its power and potential. You can use this interface to inspect your data, and create reports before attempting to develop code, or as an application in its own right to delve into your data.

Summary

My continuing theme, when examining both Apache Hadoop and Spark, is that none of these systems stand alone. They need to be integrated to form ETL-based processing systems. Data needs to be sourced and processed in Spark, and then passed to the next link in the ETL chain, or stored. I hope that this chapter has shown you that Spark functionality can be extended with extra libraries, and systems such as H2O.

Although Apache Spark MLlib (machine learning library) has a lot of functionality, the combination of H2O Sparkling Water and the Flow web interface provides an extra wealth of data analysis modeling options. Using Flow, you can also visually, and interactively process your data. I hope that this chapter shows you, even though it cannot cover all that H2O offers, that the combination of Spark and H2O widens your data processing possibilities.

I hope that you have found this chapter useful. As a next step, you might consider checking the <http://h2o.ai/> website or the H2O Google group, which is available at <https://groups.google.com/forum/#!forum/h2ostream>.

The next chapter will examine the Spark-based service <https://databricks.com/>, which will use Amazon AWS storage for Spark cluster creation in the cloud.

8

Spark Databricks

Creating a big data analytics cluster, importing data, and creating ETL streams to cleanse and process the data are hard to do, and also expensive. The aim of Databricks is to decrease the complexity and make the process of cluster creation, and data processing easier. They have created a cloud-based platform, based on Apache Spark that automates cluster creation, and simplifies data import, processing, and visualization. Currently, the storage is based upon AWS but, in the future, they plan to expand to other cloud providers.

The same people who designed Apache Spark are involved in the Databricks system. At the time of writing this book, the service was only accessible via registration. I have been offered a 30-day trial period. Over the next two chapters, I will examine the service, and its components, and offer some sample code to show how it works. This chapter will cover the following topics:

- Installing Databricks
- AWS configuration
- Account management
- The menu system
- Notebooks and folders
- Importing jobs via libraries
- Development environments
- Databricks tables
- The Databricks DbUtils package

Given that this book is provided in a static format, it will be difficult to fully examine functionality such as streaming.

Overview

The Databricks service, available at the <https://databricks.com/> website, is based upon the idea of a cluster. This is similar to a Spark cluster, which has already been examined and used in previous chapters. It contains a master, workers, and executors. However, the configuration and the size of the cluster are automated, depending upon the amount of memory that you specify. Features such as security, isolation, process monitoring, and resource management are all automatically managed for you. If you have an immediate requirement for a Spark-based cluster using 200 GB of memory, for a short period of time, this service can be used to dynamically create it, and process your data. You can terminate the cluster to reduce your costs when the processing is finished.

Within a cluster, the idea of a Notebook is introduced, along with a location for you to create scripts and run programs. Folders can be created within Notebooks, which can be based upon Scala, Python, or SQL. Jobs can be created to execute the functionality, and can be called from the Notebook code or the imported libraries. Notebooks can call Notebook functionality. Also, the functionality is provided to schedule jobs, based on time or event.

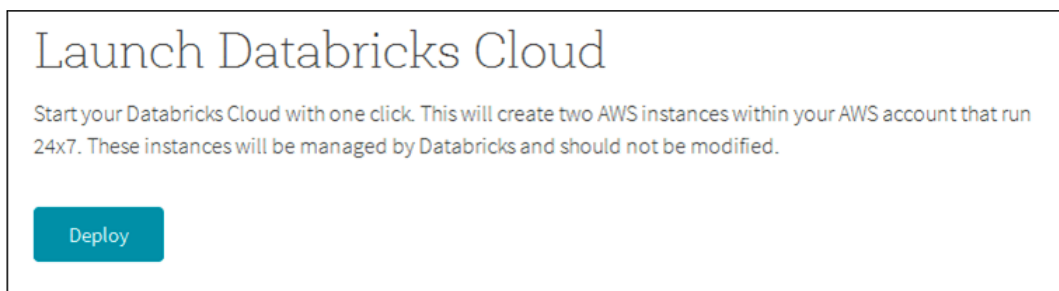
This provides you with a feel of what the Databricks service provides. The following sections will explain each major item that has been introduced. Please keep in mind that what is presented here is new and evolving. Also, I used the AWS US East (North Virginia) region for this demonstration, as the Asia Sydney region currently has limitations that caused the Databricks install to fail.

Installing Databricks

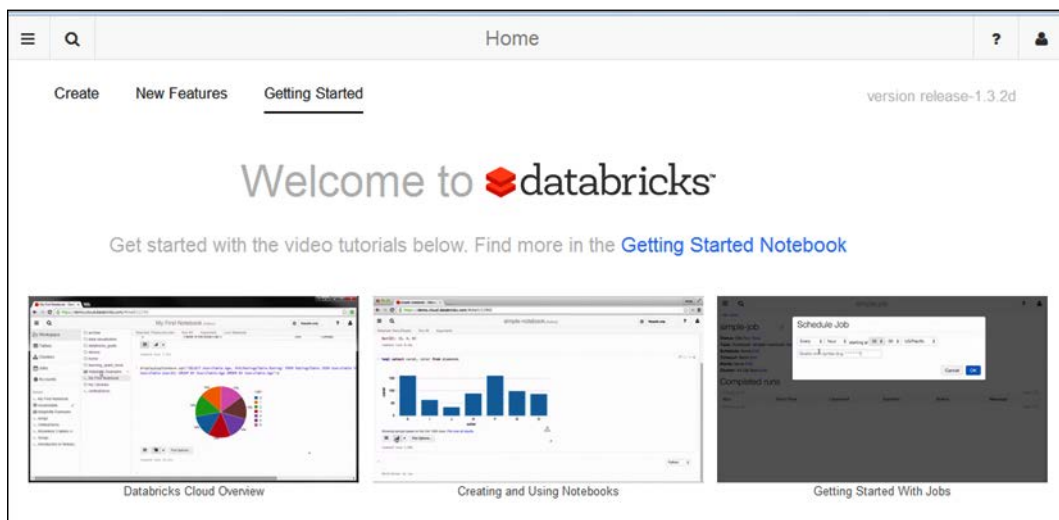
In order to create this demonstration, I used the AWS offer of a year's free access, which was available at <http://aws.amazon.com/free/>. This has limitations such as 5 GB of S3 storage, and 750 hours of **Amazon Elastic Compute Cloud (EC2)**, but it allowed me low-cost access and reduced my overall EC2 costs. The AWS account provides the following:

- An account ID
- An access Key ID
- A secret access Key

These items of information are used by Databricks to access your AWS storage, install the Databricks systems, and create the cluster components that you specify. From the moment of the install, you begin to incur AWS EC2 costs, as the Databricks system uses at least two running instances without any clusters. Once you have successfully entered your AWS and billing information, you will be prompted to launch the Databricks cloud.



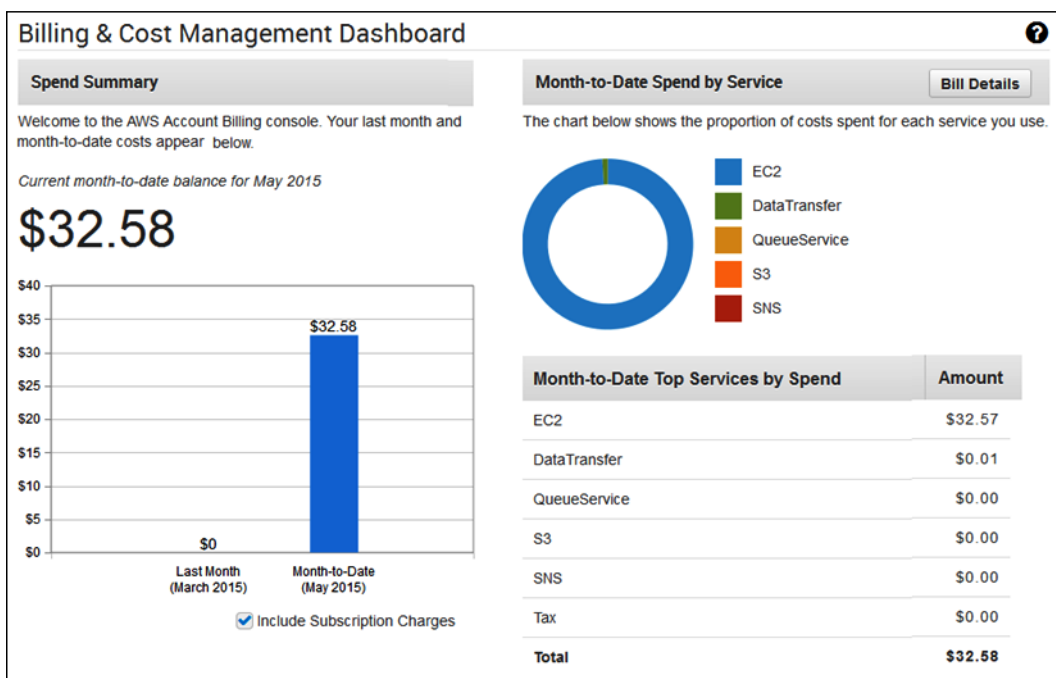
Having done this, you will be provided with a URL to access your cloud, an admin account, and password. This will allow you to access the Databricks web-based user interface, as shown in the following screenshot:



This is the welcome screen. It shows the menu bar at the top of the image, which, from left to right, contains the menu, search, help, and account icons. While using the system, there may also be a clock-faced icon that shows the recent activity. From this single interface, you may search through help screens, and usage examples before creating your own clusters and code.

AWS billing

Please note that, once you have the Databricks system installed, you will start incurring the AWS EC2 storage costs. Databricks attempts to minimize your costs by keeping EC2 resources active for a full charging period. For instance if you terminate a Databricks cluster the cluster-based EC2 instances will still exist for the hour in which AWS bills for them. In this way, Databricks can reuse them if you create a new cluster. The following screenshot shows that, although I am using a free AWS account, and though I have carefully reduced my resource usage, I have incurred AWS EC2 costs in a short period of time:

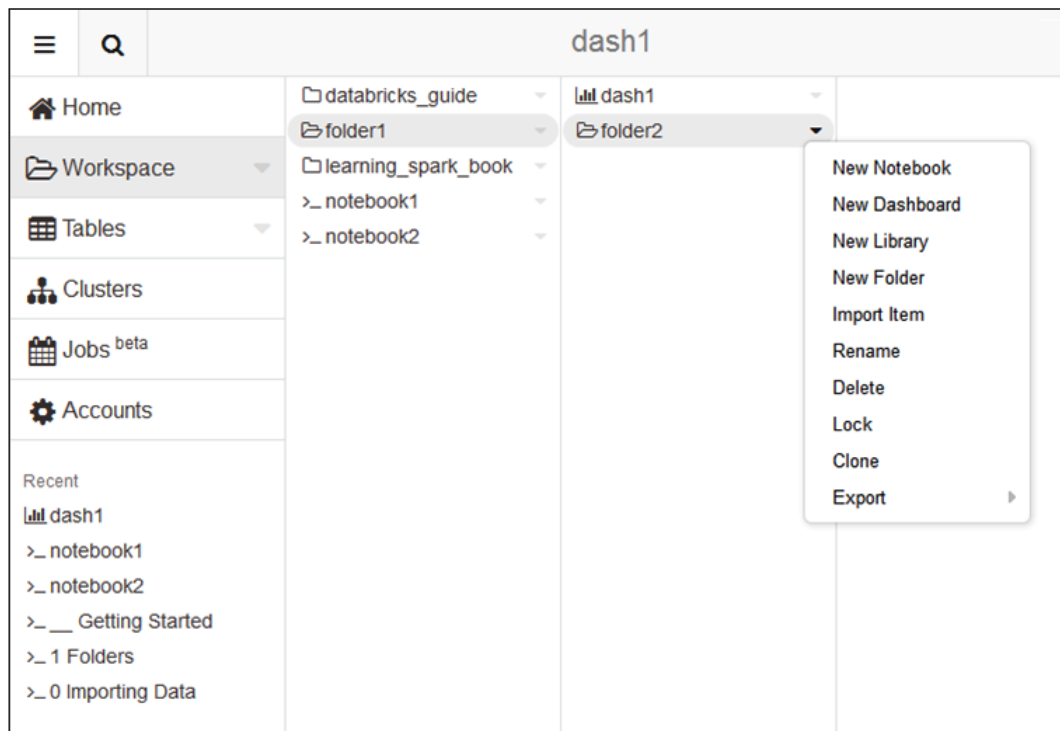


You need to be aware of the Databricks clusters that you create, and understand that, while they exist and are used, AWS costs are being incurred. Only keep the clusters that you really require, and terminate any others.

In order to examine the Databricks data import functionality, I also created an AWS S3 bucket, and uploaded data files to it. This will be explained later in this chapter.

Databricks menus

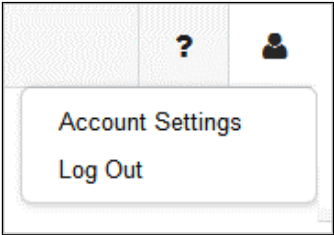
By selecting the top-left menu icon on the Databricks web interface, it is possible to expand the menu system. The following screenshot shows the top-level menu options, as well as the **Workspace** option, expanded to a folder hierarchy of /folder1/folder2/. Finally, it shows the actions that can be carried out on folder2, that is, creating a notebook, creating a dashboard, and more.



All of these actions will be expanded in future sections. The next section will examine account management, before moving on to clusters.

Account management

Account management is quite simplified within Databricks. There is a default Administrator account and subsequent accounts can be created, but you need to know the Administrator password to do so. Passwords need to be more than eight characters long; they should contain at least one digit, one upper case character, and one non-alphanumeric character. **Account** options can be accessed from the top-right menu option, shown in the following screenshot:

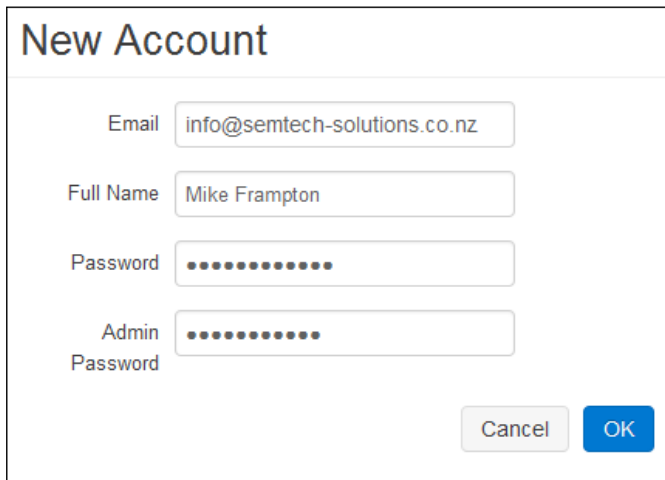


This also allows the user to logout. By selecting the account setting, you can change your password. By selecting the **Accounts** menu option, an **Accounts** list is generated. There, you will find an option to **Add Account**, and each account can be deleted via an **X** option on each account line, as shown in the following screenshot:

Accounts			
+ Add Account			
Username	Name	Password	
admin	Administrator	Reset password	x
info@semtech-solutions.co.nz	Mike Frampton	Reset password	x

It is also possible to reset the account passwords from the accounts list. Selecting the **Add Account** option creates a new account window that requires an email address, a full name, the administrator password, and the user's password. So, if you want to create a new user, you need to know your Databricks instance Administrator password. You must also follow the rules for new passwords, which are as follows:

- Minimum of eight characters
- Must contain at least one digit in the range: 0-9
- Must contain at least one upper case character in the range: A-Z
- Must contain at least one non-alphanumeric character: !@#\$\$%



The screenshot shows a 'New Account' dialog box. It has a title bar with the text 'New Account'. Below the title bar, there are four input fields. The first field is labeled 'Email' and contains the text 'info@semtech-solutions.co.nz'. The second field is labeled 'Full Name' and contains the text 'Mike Frampton'. The third field is labeled 'Password' and is filled with dots. The fourth field is labeled 'Admin Password' and is also filled with dots. At the bottom right of the dialog, there are two buttons: 'Cancel' and 'OK'.

The next section will examine the **Clusters** menu option, and will enable you to manage your own Databricks Spark clusters.

Cluster management

Selecting the **Clusters** menu option provides a list of your current Databricks clusters and their status. Of course, currently you have none. Selecting the **Add Cluster** option allows you to create one. Note that the amount of memory you specify determines the size of your cluster. There is a minimum of 54 GB required to create a cluster with a single master and worker. For each additional 54 GB specified, another worker is added.

New Cluster

Name

semclust1

Memory (GB)

53

Use spot instances

☐

Note:

A cluster needs at least 54 GB memory to run Spark commands or import tables.

Cancel

OK

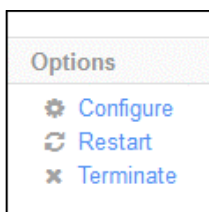
The following screenshot is a concatenated image, showing a new cluster called semclust1 being created and in a **Pending** state. While **Pending**, the cluster has no dashboard, and the cluster nodes are not accessible.

+ Add Cluster

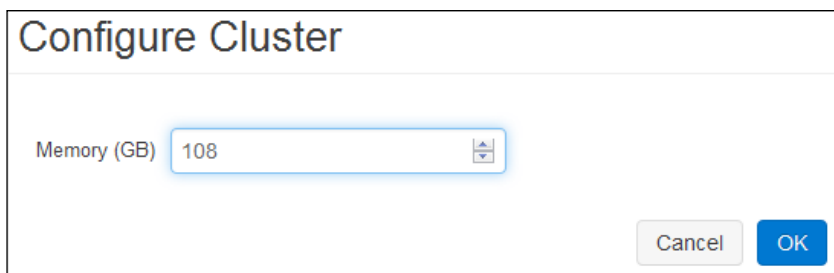
Name	Memory	Type	State	Nodes	Libraries	Notebooks	Dashboards
semclust1		On-demand	Pending	View Spark UI 1 Nodes	--	0 Notebooks	Make Dashboard Cluster

Name	Memory	Type	State	Nodes	Libraries	Notebooks	Dashboards
semclust1	54 GB	On-demand	Running	View Spark UI 2 Nodes Master Worker 0	--	0 Notebooks	Attached

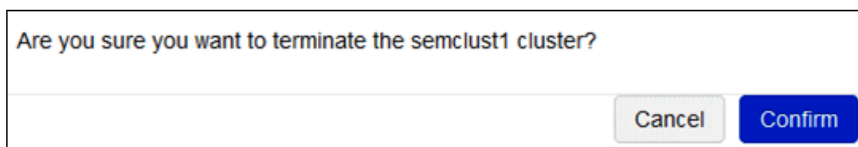
Once created the cluster memory is listed and its status changes from **Pending** to **Running**. A default dashboard has automatically been attached, and the Spark master and worker user interfaces can be accessed. It is important to note here that Databricks automatically starts and manages the cluster processes. There is also an **Option** column to the right of this display that offers the ability to **Configure**, **Restart**, or **Terminate** a cluster as shown in the following screenshot:



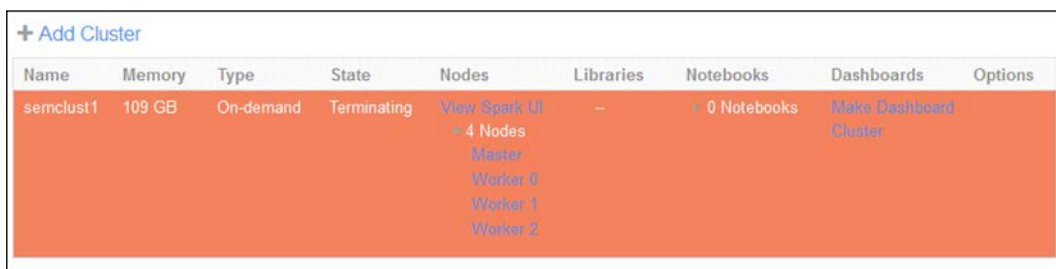
By reconfiguring a cluster, you can change its size. By adding more memory, you can add more workers. The following screenshot shows a cluster, created at the default size of 54 GB, having its memory extended to 108 GB.



Terminating a cluster removes it, and it cannot be recovered. So, you need to be sure that deletion is the correct course of action. Databricks prompts you to confirm your action before the termination actually takes place.



It takes time for a cluster to be both, created and terminated. During termination, the cluster is marked with an orange banner, and a state of **Terminating**, as shown in the following screenshot:



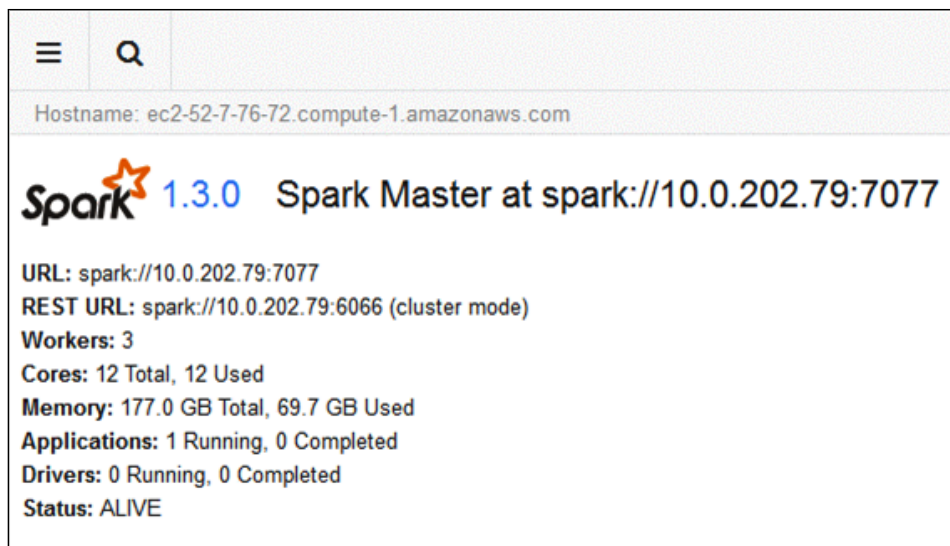
The screenshot shows a table of Databricks clusters. The cluster 'semclust1' is highlighted with an orange background, indicating it is in a 'Terminating' state. The table has columns for Name, Memory, Type, State, Nodes, Libraries, Notebooks, Dashboards, and Options. The 'Nodes' column for the highlighted cluster shows '4 Nodes' with a sub-list: 'Master', 'Worker 0', 'Worker 1', and 'Worker 2'. The 'Dashboards' column has a link 'Make Dashboard Cluster'.

Name	Memory	Type	State	Nodes	Libraries	Notebooks	Dashboards	Options
semclust1	109 GB	On-demand	Terminating	View Spark UI 4 Nodes Master Worker 0 Worker 1 Worker 2	--	0 Notebooks	Make Dashboard Cluster	

Note that the cluster type in the previous screenshot is shown to be **On-demand**. When creating a cluster, it is possible to select a check box called **Use spot instances to create a spot cluster**. These clusters are cheaper than the on-demand clusters, as they bid for a cheaper AWS spot price. However, they can be slower to start than the on-demand clusters.

The Spark user interfaces are the same as those you would expect on a non-Databricks Spark cluster. You can examine workers, executors, configuration, and log files. As you create clusters, they will be added to your cluster list. One of the clusters will be used as the cluster where the dashboards are run. This can be changed by using the **Make Dashboard Cluster** option. As you add libraries and Notebooks to your cluster, the cluster details entry will be updated with a count of the numbers added.

The only thing that I would say about the Databricks Spark user interface option at this time, because it is familiar, is that it displays the Spark version that is used. The following screenshot, extracted from the master user interface, shows that the Spark version being used (1.3.0) is very up-to-date. At the time of writing, the latest Apache Spark release was 1.3.1, dated 17 April, 2015.



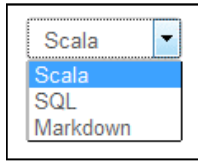
The next section will examine Databricks Notebooks and folders—how to create them, and how they can be used.

Notebooks and folders

A Notebook is a special type of Databricks folder that can be used to create Spark scripts. Notebooks can call the Notebook scripts to create a hierarchy of functionality. When created, the type of Notebook must be specified (Python, Scala, or SQL), and a cluster can then specify that the Notebook functionality can be run against it. The following screenshot shows the Notebook creation.

A screenshot of the "Create New Notebook" dialog box. The title "Create New Notebook" is at the top. Below the title, there are three input fields: "Name" with the value "notebook1", "Language" with a dropdown menu showing "Scala", and "Cluster" with a dropdown menu showing "semclust1 (54 GB, Running)". At the bottom right, there are two buttons: "Cancel" and "Create".

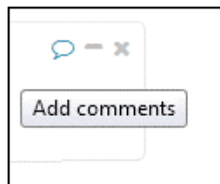
Note that a menu option, to the right of a Notebook session, allows the type of Notebook that is to be changed. The following example shows that a Python notebook can be changed to **Scala**, **SQL**, or **Markdown**:



Note that a Scala Notebook cannot be changed to Python, and a Python Notebook cannot be changed to Scala. The terms Python, Scala, and SQL are well understood as the development languages, however, **Markdown** is new. Markdown allows formatted documentation to be created from formatted commands in text. A simple reference can be found at <https://forums.databricks.com/static/markdown/help.html>.

This means that formatted comments can be added to the Notebook session as scripts are created. Notebooks are further subdivided into cells, which contain the commands to be executed. Cells can be moved within a Notebook by hovering over the top-left corner, and dragging them into position. New cells can be inserted into a cell list within a Notebook.

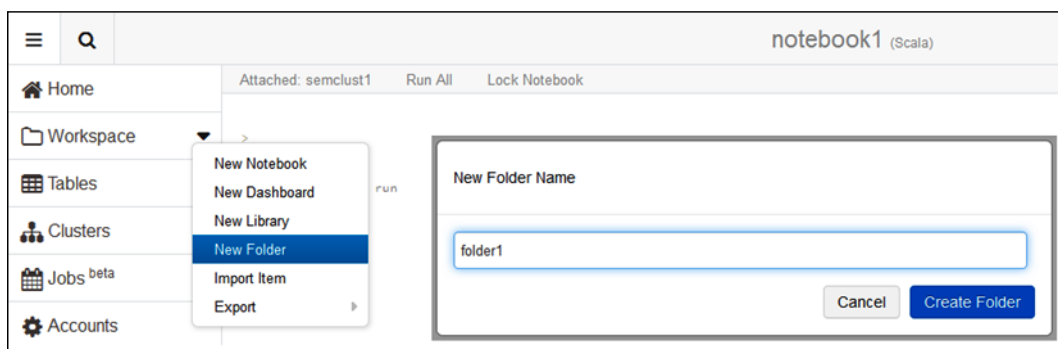
Also, using the `%sql` command, within a Scala or Python Notebook cell, allows SQL syntax to be used. Typically, the key combination of `Shift + Enter` causes text blocks in a Notebook or folder to be executed. Using the `%md` command allows Markdown comments to be added within a cell. Also, comments can be added to a Notebook cell. The menu options available at the top-right section of a Notebook cell, shown in the following screenshot, shows comment, as well as the minimize and maximize options:



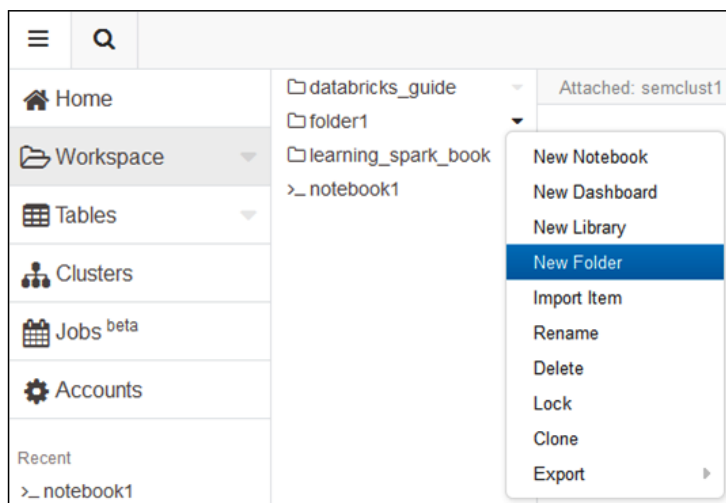
Multiple web-based sessions may share a Notebook. The actions that occur within the Notebook will be populated to each web interface viewing it. Also, the Markdown and comment options can be used to enable communication between users to aid the interactive data investigation between a distributed group.



The previous screenshot shows the header of a Notebook session for **notebook1**. It shows the Notebook name and type (**Scala**). It also shows the option to lock the Notebook to make it read only, as well as the option to detach it from its cluster. The following screenshot shows the creation of a folder within a Notebook workspace:

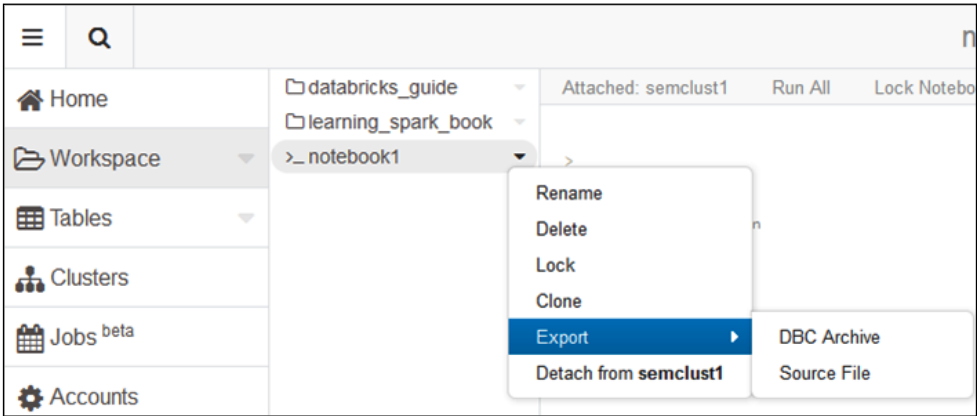


A drop-down menu, from the **Workspace** main menu option, allows for the creation of a folder—in this case, named `folder1`. The later sections will describe other options in this menu. Once created and selected, a drop-down menu from the new folder called `folder1` shows the actions associated with it in the following screenshot:

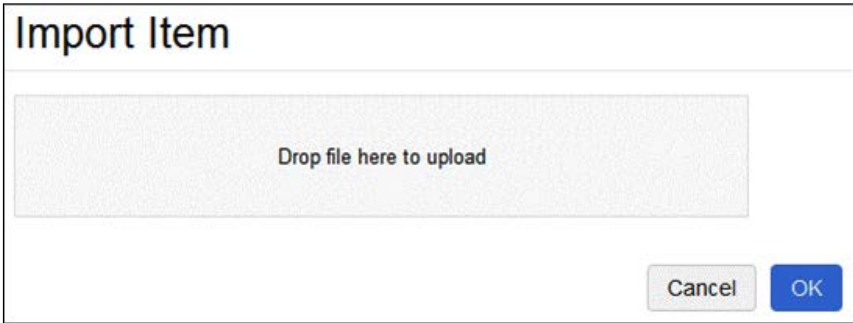


So, a folder can be exported to a DBC archive. It can be locked, or cloned to create a copy. It can also be renamed, or deleted. Items can be imported into it; for instance, files, which will be explained by example later. Also, new notebooks, dashboards, libraries, and folders can be created within it.

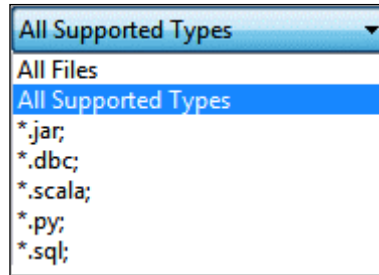
In the same way as actions can be carried out against a folder, a Notebook has a set of possible actions. The following screenshot shows the actions available via a drop-down menu for the Notebook called `notebook1`, which is currently attached to the running cluster called `semclust1`. It is possible to rename, delete, lock, or clone a Notebook. It is also possible to detach it from its current cluster, or attach it if it is detached. It is also possible to export the Notebook to a file, or a DBC archive.



From the folder **Import** option, files can be imported to a folder. The following screenshot shows the file drop-option window that is invoked if this option is selected. It is possible to either drop a file onto the upload pane from the local server, or click on this pane to open a navigation browser to search the local server for files to upload.



Note that the files that are uploaded need to be of a specific type. The following screenshot shows the supported file types. This is a screenshot taken from the file browser when browsing for a file to upload. It also makes sense. The supported file types are Scala, SQL, and Python; as well as DBC archives and JAR file libraries.



Before leaving this section, it should also be noted that Notebooks and folders can be dragged and dropped to change their position. The next section will examine Databricks jobs and libraries via simple worked examples.

Jobs and libraries

Within Databricks, it is possible to import JAR libraries and run the classes in them on your clusters. I will create a very simple piece of Scala code to print out the first 100 elements of the Fibonacci series as `BigInt` values, locally on my Centos Linux server. I will compile my class into a JAR file using SBT, run it locally to check the result, and then run it on my Databricks cluster to compare the results. The code looks as following:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object db_ex1 extends App
{
  val appName = "Databricks example 1"
  val conf = new SparkConf()

  conf.setAppName(appName)

  val sparkCxt = new SparkContext(conf)

  var seed1:BigInt = 1
```

```
var seed2:BigInt = 1
val limit = 100
var resultStr = seed1 + " " + seed2 + " "

for( i <- 1 to limit ){

    val fib:BigInt = seed1 + seed2
    resultStr += fib.toString + " "

    seed1 = seed2
    seed2 = fib
}

println()
println( "Result : " + resultStr )
println()

sparkCxt.stop()

} // end application
```

Not that the most elegant piece of code, or the best way to create Fibonacci, but I just want a sample JAR and class to use with Databricks. When run locally, I get the first 100 terms, which look as follows (I've clipped this data to save space):

```
Result : 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040
1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169
63245986 102334155 165580141 267914296 433494437 701408733 1134903170
1836311903 2971215073 4807526976 7778742049 12586269025 20365011074
32951280099 53316291173
```

```
4660046610375530309 7540113804746346429 12200160415121876738
19740274219868223167 31940434634990099905 51680708854858323072
83621143489848422977 135301852344706746049 218922995834555169026
354224848179261915075 573147844013817084101 927372692193078999176
```

The library that has been created is called `data-bricks_2.10-1.0.jar`. From my folder menu, I can create a new Library using the menu drop-down option. This allows me to specify the library source as a JAR file, name the new library, and load the library JAR file from my local server. The following screenshot shows an example of this process:

New Library

Source Upload Java/Scala JAR

Library Name db example 1

JAR File



data-bricks_2

5.7 KB

[Remove file](#)

Create Library

When the library has been created, it can be attached to the cluster called `semclust1`, my Databricks cluster, using the **Attach** option. The following screenshot shows the new library in the process of attaching:

≡
Q
db example 1

db example 1

Files

[1.0.jar](#)

Clusters

☐ Attach automatically to all clusters.

Attach	Name	Status
☒	semclust1	Attaching

In the following example, a job called **job2** has been created by selecting the **jar** option on the **Task** item. For the job, the same JAR file has been loaded and the class `db_ex1` has been assigned to run in the library. The cluster has been specified as on-demand, meaning that a cluster will be created automatically to run the job. The **Active runs** section shows the job running in the following screenshot:

Q

job2 beta

[< All Jobs](#)

job2

Status: Running [Cancel](#)

Task: db_ex1 in data-bricks_2.10-1.0.jar uploaded at 2015-05-26 16:55:45 [Edit](#) [Remove](#)

◦ Arguments:

Schedule: None [Edit](#)

Timeout: None [Edit](#)

Alerts: None [Edit](#)

Cluster: 54 GB On-demand [Edit](#)

Permalink: [Latest successful results](#)

Active runs

Run	Start Time	Launched	Duration	Status	Message
Run 1	2015-05-26 16:56:43	Manually	18s	Running	In run

Completed runs

[< Previous 20](#)

Run	Start Time	Launched	Duration	Status	Message
-----	------------	----------	----------	--------	---------

[< Previous 20](#)

Once run, the job is moved to the **Completed runs** section of the display. The following screenshot, for the same job, shows that it took 47 seconds to run, that it was launched manually, and that it succeeded.

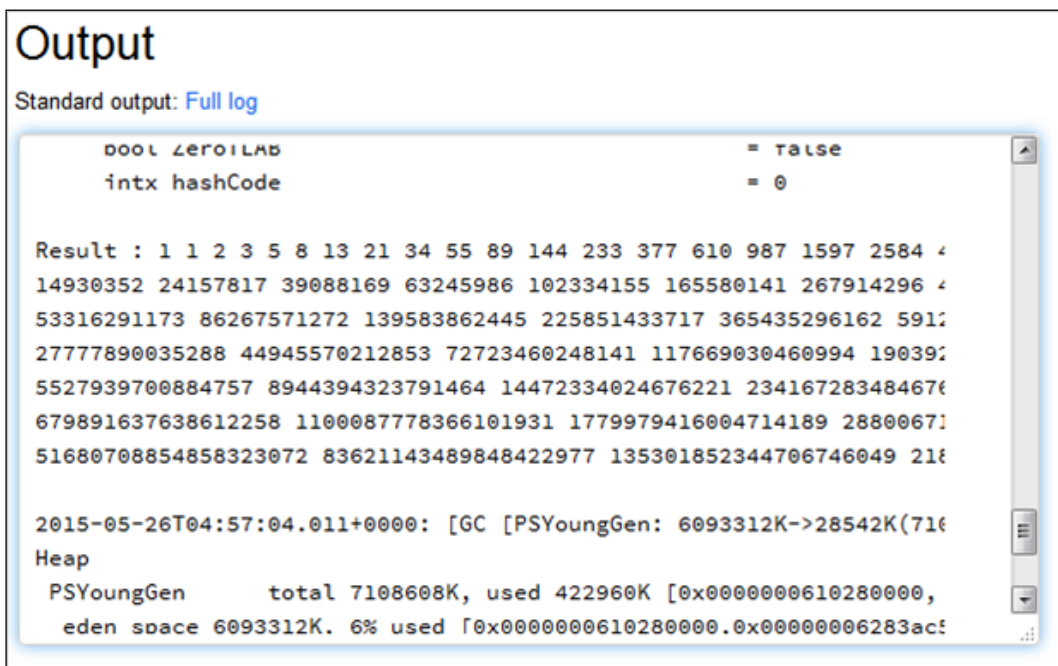
Completed runs

[< Previous 20](#)

Run	Start Time	Launched	Duration	Status	Message
Run 1	2015-05-26 16:56:43	Manually	47s	Succeeded	

[< Previous 20](#)

By selecting the run named **Run 1** in the previous screenshot, it is possible to see the run output. The following screenshot shows the same result as the local run, displayed from my local server execution. I have clipped the output text to make it presentable and readable on this page, but you can see that the output is the same.



```

Output
Standard output: Full log

    pool zeroILAB          = false
    intx hashCode          = 0

Result : 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4149
30352 24157817 39088169 63245986 102334155 165580141 267914296 421609757
53316291173 86267571272 139583862445 225851433717 365435296162 5912581520617
27777890035288 44945570212853 72723460248141 117669030460994 190392721
5527939700884757 8944394323791464 14472334024676221 234167283484676
679891637638612258 1100087778366101931 1779979416004714189 288006711
51680708854858323072 83621143489848422977 135301852344706746049 216
2015-05-26T04:57:04.011+0000: [GC [PSYoungGen: 6093312K->28542K(716
Heap
PSYoungGen      total 7108608K, used 422960K [0x0000000610280000,
eden space 6093312K. 6% used [0x0000000610280000.0x00000006283ac5

```

So, even from this very simple example, it is obvious that it is possible to develop applications remotely, and load them onto a Databricks cluster as JAR files in order to execute. However, each time a Databricks cluster is created on AWS EC2 storage, the Spark URL changes, so the application must not hard-code details such as the Spark master URL. Databricks will automatically set the Spark URL.

When running the JAR file classes in this way, it is also possible to define class parameters. The jobs may be scheduled to run at a given time, or periodically. The job timeouts, and alert email addresses may also be specified.

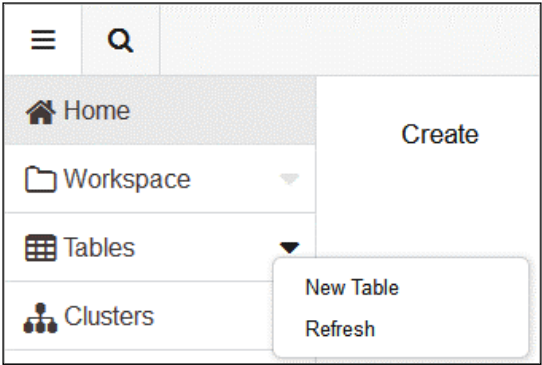
Development environments

It has been shown that scripts can be created in Notebooks in Scala, Python, or SQL, but it is also possible to use an IDE such as IntelliJ or Eclipse to develop code. By installing an SBT plugin into this development environment, it is possible to develop code for your Databricks environment. The current release of Databricks, as I write this book, is 1.3.2d. The **Release Notes** link, under **New Features** on the start page, contains a link to the IDE integration, which is `https://dbc-xxxxxxx-xxxx.cloud.databricks.com/#shell/1547`.

The URL will be of this form, with the section starting with `dbc` changed to match the URL for the Databricks cloud that you will create. I won't expand on this here, but leave it to you to investigate. In the next section, I will investigate the Databricks table data processing functionality.

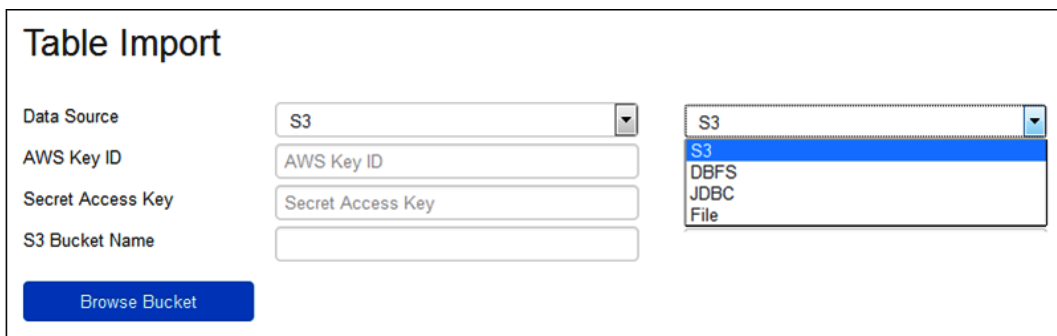
Databricks tables

The Databricks **Tables** menu option allows you store your data in a tabular form with an associated schema. The **Tables** menu option allows you to both create a table, and refresh your tables list, as the following screenshot shows:



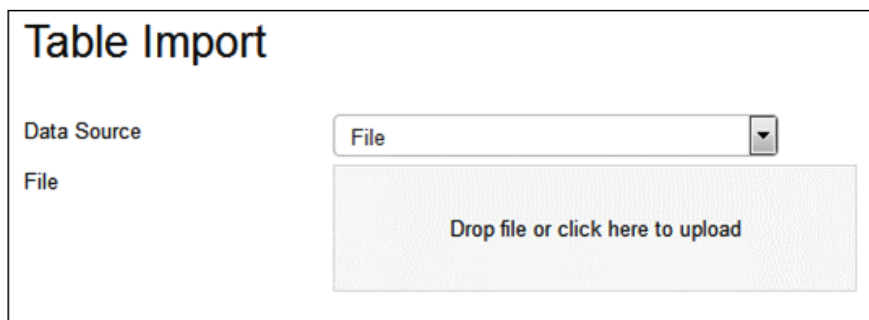
Data import

You can create tables via data import, and specify the table structure, at the same time, in terms of column names and types. If the data that is being imported has a header, then the column names can be taken from that, although all the column types are assumed to be strings. The following screenshot shows a concatenated view of the data import options and form, available when creating a table. The import file location options are **S3**, **DBFS**, **JDBC**, and **File**.



The screenshot shows a 'Table Import' form. On the left, there are four input fields: 'Data Source' (a dropdown menu), 'AWS Key ID' (a text box), 'Secret Access Key' (a text box), and 'S3 Bucket Name' (a text box). Below these fields is a blue button labeled 'Browse Bucket'. On the right, there is a dropdown menu showing a list of options: 'S3' (highlighted in blue), 'DBFS', 'JDBC', and 'File'.

The previous screenshot shows **S3** selected. In order to browse my **S3** bucket for a file to import to a table, I will need to enter the **AWS Key ID**, the **Secret Access Key**, and the **AWS S3 Bucket Name**. Then, I could browse, select the file, and create a table via preview. In the following screenshot, I have selected the **File** option:



The screenshot shows the 'Table Import' form with 'File' selected in the 'Data Source' dropdown. Below the dropdown, there is a large, light gray rectangular area with the text 'Drop file or click here to upload' centered inside it.

I can either drop my file to import into the upload frame in the following screenshot, or click on the frame to browse the local server to select a file to upload. Once a file is selected, it is then possible to define the data column delimiter, and whether the data contains a header row. It is possible to preview the data, and change the column names and data types. It is also possible to specify the new table name, and the file type. The following screenshot shows a sample file data load to create the table called `shuttle`:

Table Import

Data Source
File

bankfull.txt

4.6 MB

Remove file

Table Details

Table name
shuttle

File type
CSV

Column Delimiter
:

☒ First row is header

Create Table

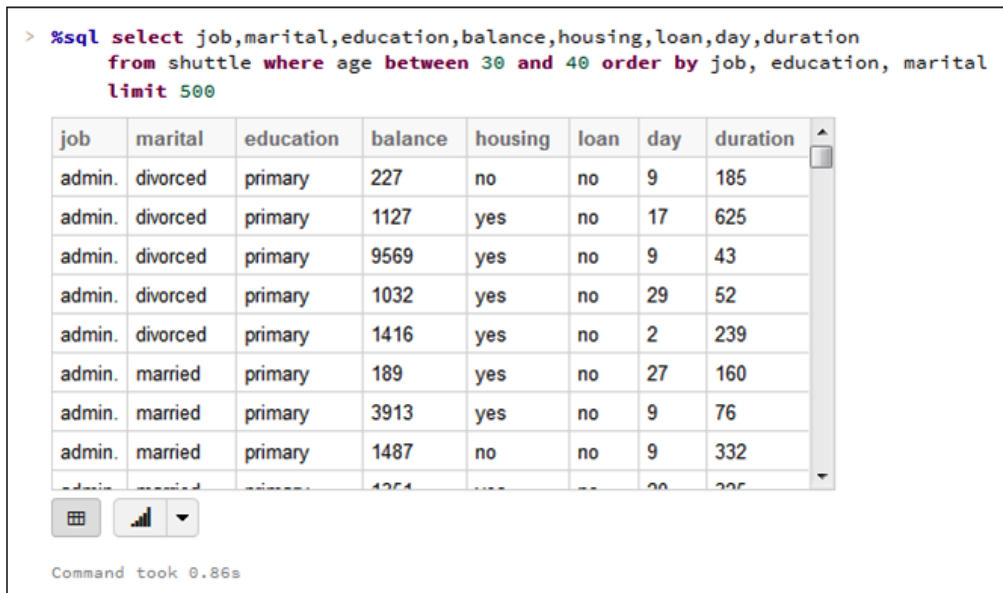
Previewing table

age	job	marital
INT	STRING	STRING
58	management	married
44	technician	single
33	entrepreneur	married
47	blue-collar	married
33	unknown	single
35	management	married
28	management	single
42	entrepreneur	divorced

Once created, the menu table list can be refreshed and the table schema viewed to confirm the column names and types. In this way, a sample of the table data can also be previewed. The table can now be viewed and accessed from an SQL session. The following screenshot shows that the **shuttle** table is visible using the `show tables` command:



Once imported, the data in this table can also be accessed via an SQL session. The following screenshot shows a simple SQL session statement to show the data extracted from the new **shuttle** table:

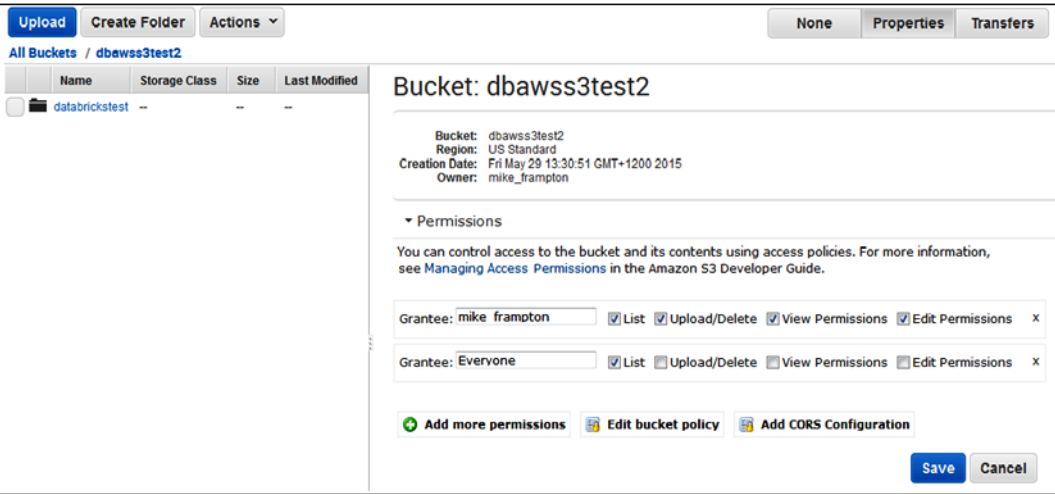


So, this provides the means to import multiple tables from a variety of data sources, and create a complex schema in order to filter and join the data by columns and rows, just as you would in a traditional, relational database. It provides a familiar approach to big data processing.

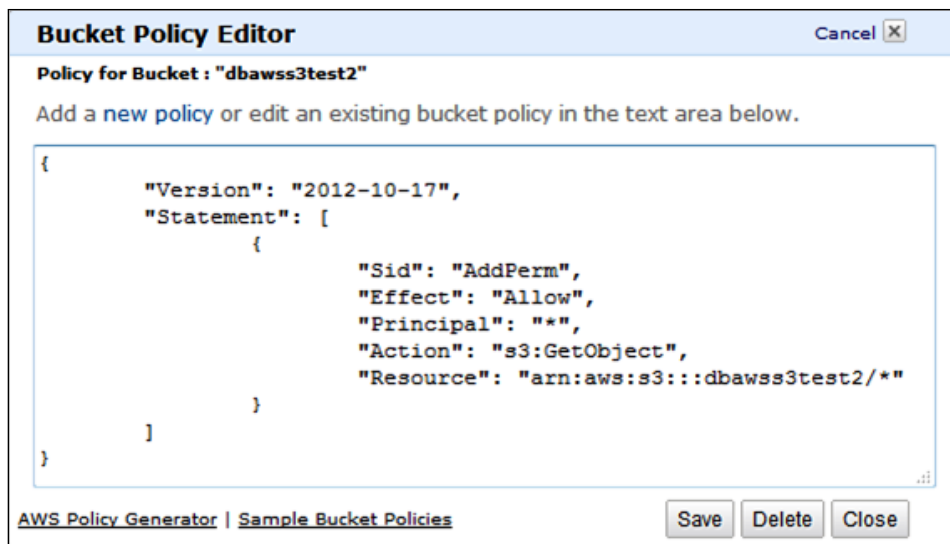
This section has described the process by which tables can be created via data import, but what about creating tables programmatically, or creating tables as external objects? The following sections will provide examples of this approach to table management.

External tables

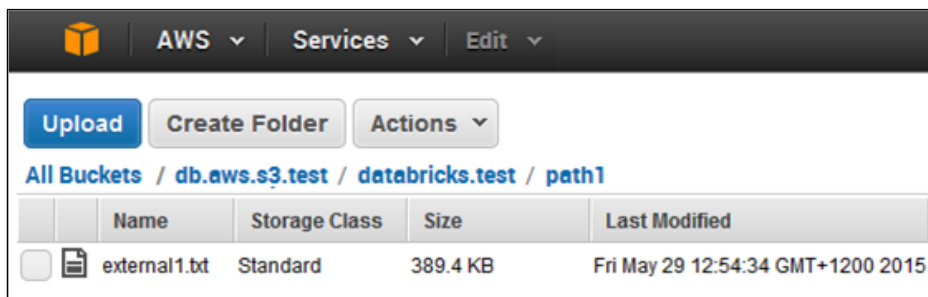
Databricks allows you to create tables against external resources, such as AWS S3 files, or local file system files. In this section, I will create an external table against an S3-based bucket, path, and a set of files. I will also examine both the permissions required in AWS and the access policy used. The following screenshot shows an AWS S3 bucket called **dbawss3test2** being created. Permissions have been granted to everyone to access the list. I am not suggesting that you do this, but ensure that your group can access your bucket.



Also, a policy has been added to aid access. In this case, anonymous users have been granted read-only access to the bucket and sub contents. You can create a more complex policy to limit the access to your group and assorted files. The following screenshot shows the new policy:



With an access policy, and a bucket created with the correct access policy, I can now create folders and upload files for use with a Databricks external table. As the following screenshot shows, I have done just that. The uploaded file has ten columns in CSV file format:



Now that the AWS S3 resources have been set up, they need to be mounted to Databricks, as the Scala-based example shows next. I have removed my AWS and secret keys from the script for security purposes. Your mounted directory will need to start with `/mnt` and any of the `/` characters, and your secret key value will need to be replaced with `%2F`. The `dbutils.fs` class is being used to create the mount and the code executes within a second, as the following result shows:

```
> import dbutils.fs

val mountDir = "/mnt/s3data1"

// If you have '/' characters in your secret key they must be escaped with '%2F'

val awsKey      = " "
val awsSecretKey = " "
val bucketName  = "dbawss3test2"
val awsPath     = "/databrickstest/path1/"

val s3Path      = "s3n://" + awsKey + ":" + awsSecretKey + "@" + bucketName + awsPath

fs.mount( s3Path, mountDir )

Successfully mounted to /mnt/s3data1!
import dbutils.fs
mountDir: String = /mnt/s3data1
awsKey: String =
awsSecretKey: String =
bucketName: String = dbawss3test2
awsPath: String = /databrickstest/path1/
s3Path: String = s3n://.
res7: Boolean = true

Command took 0.93s
```

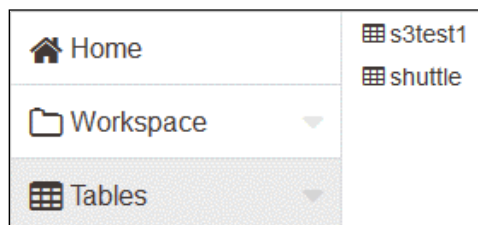
Now, an external table can be created against this mounted path and the files that it contains using a Notebook-based SQL session, as the following screenshot shows. The table called `s3test1` has been created against the files that the mounted directory contains, and a delimiter is specified as a comma, in order to parse the CSV-based content.

```
> CREATE TABLE s3test1
(
  col0 String, col1 String, col2 String,
  col3 String, col4 String, col5 String,
  col6 String, col7 String, col8 String,
  col9 String
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION "/mnt/s3data1/*";

OK

Command took 0.61s
```

The **Tables** menu option now shows that the **s3test1** table exists, as shown in the following screenshot. So, it should be possible to run some SQL against this table:

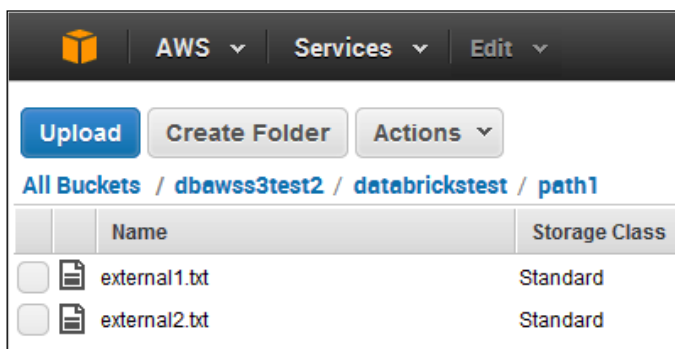


I have run a `SELECT` statement in an SQL-based Notebook session to get a row count from the external table, using the `COUNT(*)` function, as shown in the following screenshot. It can be seen that the table contains **14500** rows.

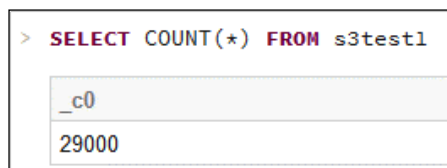
```
> SELECT COUNT(*) FROM s3test1
```

_c0
14500

I will now add another file to the S3-based folder. In this case, it is just a copy of the first file in CSV format, so the row count in the external table should double. The following screenshot shows the file that is added:



Running the same `SELECT` statement against the external table does indeed provide a doubled row count of **29000** rows. The following screenshot shows the SQL statement, and the output:



So, it is easily possible to create external tables within Databricks, and run SQL against content that is dynamically changed. The file structure will need to be uniform, and the S3 bucket access must be defined if using AWS. The next section will examine the DbUtils package provided with Databricks.

The DbUtils package

The previous Scala-based script, which uses the DbUtils package, and creates the mount in the last section, only uses a small portion of the functionality of this package. In this section, I would like to introduce some more features of the DbUtils package, and the **Databricks File System (DBFS)**. The help option within the DbUtils package can be called within a Notebook connected to a Databricks cluster, to learn more about its structure and functionality. As the following screenshot shows, executing `dbutils.fs.help()` in a Scala Notebook provides help on `fsutils`, `cache`, and the `mount`-based functionality:

```
> dbutils.fs.help()
```

```
dbutils.fs provides utilities for working with FileSystems. Most methods in
this package can take either a DBFS path (e.g., "/foo"), an S3 URI ("s3n://bucket/"), or
another Hadoop FileSystem URI.
```

```
For more info about a method, use dbutils.fs.help("methodName").
```

```
If you find yourself using these functions a lot, consider doing an "import dbutils.fs",
allowing you to access them via "fs.ls()", etc.
```

It is also possible to obtain help on individual functions, as the text in the previous screenshot shows. The example in the following screenshot explains the **cacheTable** function, providing descriptive text and a sample function call with the parameter and return types:

```
> dbutils.fs.help("cacheTable")
```

```
/**
 * Caches the contents of the given table on the local SSDs of this cluster. The provided
 * table must be backed by files stored in DBFS or S3; this cannot cache arbitrary tables
 * constructed via transformations.
 *
 * Note that this simply calls cacheFiles() after looking up the location of the source data
 * for this table, so the cache is actually based on the files, not the metadata of the table.
 * For instance, if another table uses the same underlying data as this one, it will also
 * start using the cache.
 *
 * See the documentation for cacheFiles() for the semantics of this cache, and how it differs
 * from normal sqlContext.cacheTable().
 *
 * Example: cacheTable("sales")
 *
 * @param tableName name of the Hive or registered temp table.
 * @return True if the table was successfully cached or re-cached.
 */
cacheTable(tableName: java.lang.String): boolean
```

The next section will briefly examine the DBFS before moving on to examining more of the `dbutils` functionality.

Databricks file system

The DBFS can be accessed using URL's of the `dbfs:/*` form, and using the functions available within `dbutils.fs`.

```
> dbutils.fs.ls("dbfs:/mnt/")

res6: com.databricks.SchemaSeq[com.databricks.backend.daemon.dbutils.FileInfo] =
SchemaSeq(FileInfo(dbfs:/mnt/s3data/, s3data/, 0),
FileInfo(dbfs:/mnt/s3data1/, s3data1/, 0))
```

The previous screenshot shows the `/mnt` file system being examined using the `ls` function, and then showing mount directories—`s3data` and `s3data1`. These were the directories created during the previous Scala S3 mount example.

Dbutils fsutils

The `fsutils` group of functions, within the `dbutils` package, covers functions such as `cp`, `head`, `makedirs`, `mv`, `put`, and `rm`. The help calls, shown previously, can provide more information about them. You can create a directory on DBFS using the `makedirs` call, as shown next. Note that I have created a number of directories under `dbfs:/`, named as `data*` in this session. The following example has created the directory called `data2`:

```
> dbutils.fs.makedirs("dbfs:/data2/")
dbutils.fs.ls("dbfs:/")

res10: com.databricks.SchemaSeq[com.databricks.backend.daemon.dbutils.FileInfo] = SchemaSeq(
FileInfo(dbfs:/FileStore/, FileStore/, 0), FileInfo(dbfs:/data/, data/, 0), FileInfo(dbfs:/data1/, data1/, 0),
FileInfo(dbfs:/data2/, data2/, 0),          FileInfo(dbfs:/databricks-datasets/, databricks-datasets/, 0),
FileInfo(dbfs:/mnt/, mnt/, 0),             FileInfo(dbfs:/mount/, mount/, 0),
FileInfo(dbfs:/tmp/, tmp/, 0),             FileInfo(dbfs:/user/, user/, 0))
```

The previous screenshot shows by executing an `ls` that there are many default directories that already exist on DBFS. For instance, see the following:

- `/tmp` is a temporary area
- `/mnt` is a mount point for remote directories—that is, S3
- `/user` is a user storage area that currently contains Hive

- /mount is an empty directory
- /FileStore is a storage area for tables, JARs, and job JARs
- /databricks-datasets is datasets provided by Databricks

The `dbutils.copy` command, shown next, allows a file to be copied to a DBFS location. In this instance, the `external1.txt` file had been copied to the `/data2` directory, as shown in the following screenshot:

```
> dbutils.fs.cp("dbfs:/mnt/s3data1/external1.txt","dbfs:/data2/")
dbutils.fs.ls("dbfs:/data2/")

res21: com.databricks.SchemaSeq[com.databricks.backend.daemon.dbutils.FileInfo] =
SchemaSeq(FileInfo(dbfs:/data2/external1.txt, external1.txt, 398755))
```

The `head` function can be used to return the first `maxBytes` characters from the head of a file on DBFS. The following example shows the format of the `external1.txt` file. This is useful, as it tells me that this is a CSV file, and so shows me how to process it.

```
> dbutils.fs.head("dbfs:/data2/external1.txt")

[Truncated to first 65536 bytes]
res24: String =
"55,0,81,0,-6,11,25,88,64,4
56,0,96,0,52,-4,40,44,4,4
50,-1,89,-7,50,0,39,40,2,1
53,9,79,0,42,-2,25,37,12,4
55,2,82,0,54,-6,26,28,2,1"
```

It is also possible to move files within DBFS. The following screenshot shows the `mv` command being used to move the `external1.txt` file from the directory `data2` to the directory called `data1`. The `ls` command is then used to confirm the move.

```
> dbutils.fs.mv("dbfs:/data2/external1.txt","dbfs:/data1/")
dbutils.fs.ls("dbfs:/data1/")

res26: com.databricks.SchemaSeq[com.databricks.backend.daemon.dbutils.FileInfo] =
SchemaSeq(FileInfo(dbfs:/data1/external1.txt, external1.txt, 398755))
```

Finally, the remove function (`rm`) is used to remove the file called `external1.txt`, which was just moved. The following `ls` function call shows that the file no longer exists within the `data1` directory, because there is no `FileInfo` record in the function output:

```
> dbutils.fs.rm("dbfs:/data1/external1.txt")
dbutils.fs.ls("dbfs:/data1/")

res27: com.databricks.SchemaSeq[com.databricks.backend.daemon.dbutils.FileInfo]
= SchemaSeq()
```

The DbUtils cache

The cache functionality, within `DbUtils`, provides the means to cache (and uncache) both tables and files to DBFS. Actually, the tables are saved as files also to the DBFS directory called `/FileStore`. The following screenshot shows that the cache functions are available:

cache

```
cacheFiles(files: Seq): boolean -> Caches a set of files on the local SSDs of this cluster
cacheTable(tableName: String): boolean -> Caches the contents of the given table on the local SSDs of this cluster
uncacheFiles(files: Seq): boolean -> Removes the cached version of the files
uncacheTable(tableName: String): boolean -> Removes the cached version of the given table from SSDs
```

The DbUtils mount

The mount functionality allows you to mount remote file systems, refresh mounts, display mount details, and unmount specific mounted directories. An example of an S3 mount was already given in the previous sections, so I won't repeat it here. The following screenshot shows the output from the `mounts` function. The `s3data` and `s3data1` mounts have been created by me. The other two mounts for root and datasets already existed. The mounts are listed in a sequence of the `MountInfo` objects. I have rearranged the text to be more meaningful, and to be better presented on the page.

```
> dbutils.fs.mounts()

res23: com.databricks.SchemaSeq[com.databricks.backend.daemon.dbutils.MountInfo] =
SchemaSeq(MountInfo(/mnt/s3data1, s3n://dbawss3test2/databrickstest/path1/),
MountInfo(/mnt/s3data, s3n://db.aws.s3.test/databricks.test/path1/),
MountInfo(/, DatabricksRoot),
MountInfo(/databricks-datasets, databricks-datasets))
```

Summary

This chapter has introduced Databricks. It shows how the service can be accessed, and also shows how it uses AWS resources. Remember that, in the future, the people who invented Databricks plan to support other cloud-based platforms, such as Microsoft Azure. I thought that it was important to introduce Databricks, because the same people who were involved in the development of Apache Spark are involved in this system. The natural progression seems to be Hadoop, Spark, then Databricks.

I will continue the Databricks investigation in the next chapter, because important features, such as visualization, have not yet been examined. Also, the major Spark functionality modules called GraphX, streaming, MLlib, and SQL have not been introduced in Databricks terms. How easy is it to use these modules within Databricks to process real data? Read on to find out.

9

Databricks Visualization

This chapter builds on the work done in *Chapter 8, Spark Databricks*, and continues to investigate the functionality of the Apache Spark-based service at <https://databricks.com/>. Although I will use Scala-based code examples in this chapter, I wish to concentrate on the Databricks functionality rather than the traditional Spark processing modules: MLlib, GraphX, Streaming, and SQL. This chapter will explain the following Databricks areas:

- Data visualization using Dashboards
- An RDD-based report
- A Data stream-based report
- The Databricks Rest interface
- Moving data with Databricks

So, this chapter will examine the functionality in Databricks to analytically visualize data via reports, and dashboards. It will also examine the REST interface, as I believe it to be a useful tool for both, remote access, and integration purposes. Finally, it will examine the options for moving data, and libraries, into a Databricks cloud instance.

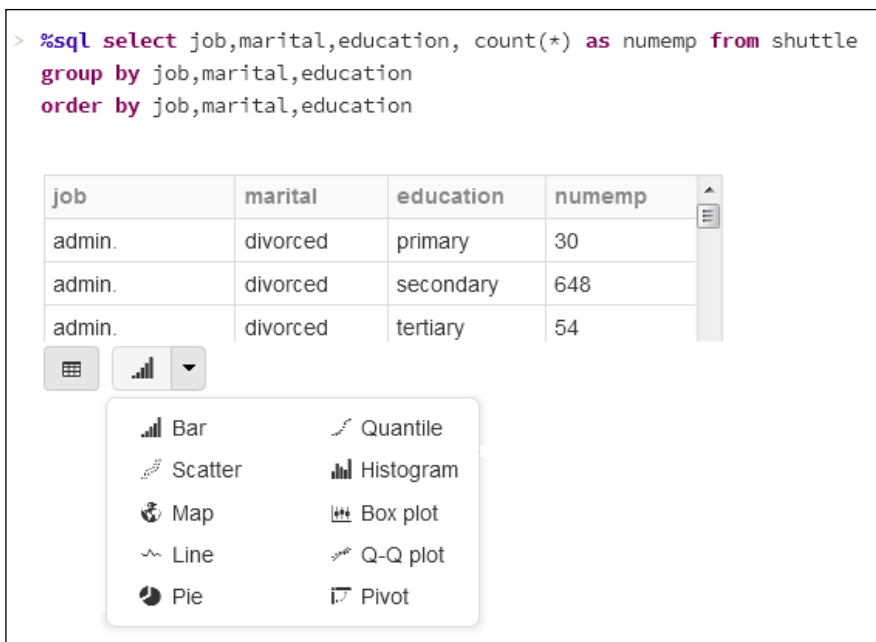
Data visualization

Databricks provides tools to access S3, and the local file system-based files. It offers the ability to import data into tables, as already shown. In the last chapter, raw data was imported into the shuttle table to provide the table-based data that SQL could be run against, to filter against rows and columns, allow data to be sorted, and then aggregated. This is very useful, but we are still left looking at raw data output when images, and reports, present information that can be more readily, and visually, interpreted.

Databricks provides a visualization interface, based on the tabular result data that your SQL session produces. The following screenshot shows some SQL that has been run. The resulting data, and the visualization drop-down menu under the data, show the possible options.

```
> %sql select job,marital,education, count(*) as numemp from shuttle
group by job,marital,education
order by job,marital,education
```

job	marital	education	numemp
admin.	divorced	primary	30
admin.	divorced	secondary	648
admin.	divorced	tertiary	54

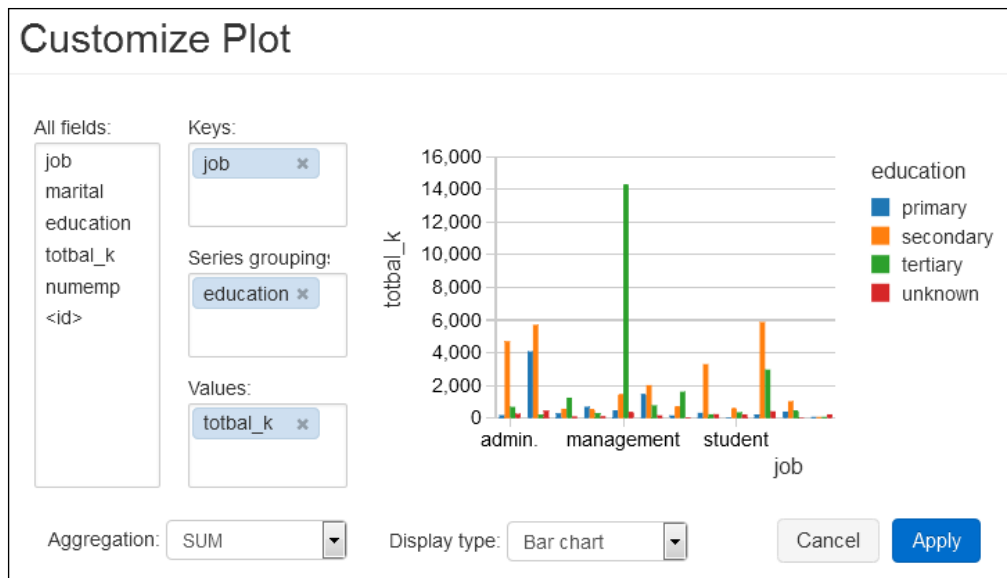


The screenshot shows a Databricks SQL interface. At the top, a SQL query is executed: `%sql select job,marital,education, count(*) as numemp from shuttle group by job,marital,education order by job,marital,education`. Below the query, a table displays the results with columns: job, marital, education, and numemp. The table contains three rows of data. Below the table, there is a visualization toolbar with icons for table, bar chart, and a dropdown menu. The dropdown menu is open, showing a list of visualization options: Bar, Scatter, Map, Line, Pie, Quantile, Histogram, Box plot, Q-Q plot, and Pivot.

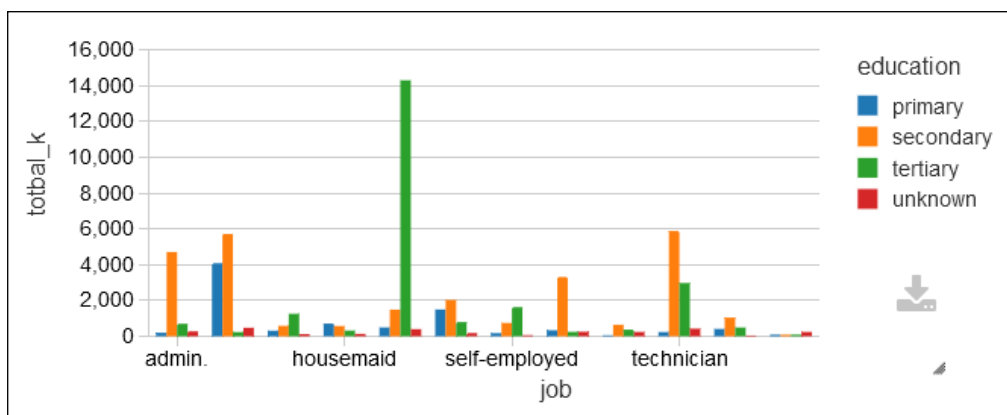
There is a range of visualization options here, starting with the more familiar **Bar** graphs, and **Pie** charts through to **Quantiles**, and **Box** plots. I'm going to change my SQL so that I get more options to plot a graph, which is as follows:

```
> %sql select job,marital,education,sum(balance)/1000 as totbal_k, count(*) as numemp from shuttle
group by job,marital,education
order by job,marital,education
```

Then, having selected the visualization option; **Bar** graph, I will select the **Plot** options which will allow me to choose the data for the graph vertices. It will also allow me to select a data column to pivot on. The following screenshot shows the values that I have chosen.

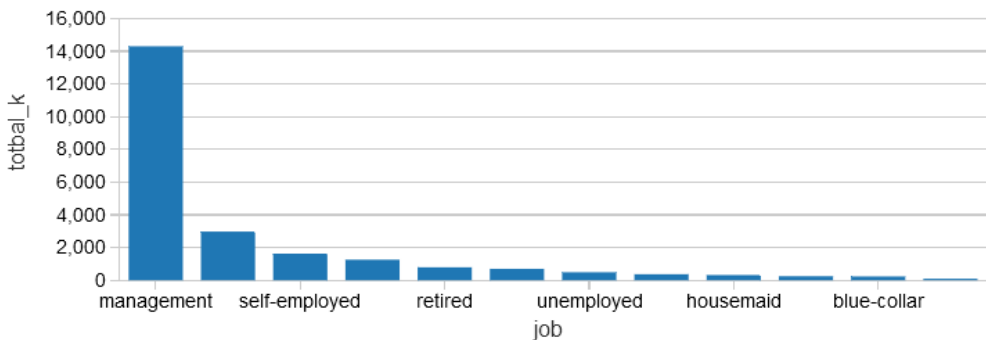


The **All fields** section, from the **Plot** options display, shows all of the fields available for the graph display from the SQL statement result data. The **Keys** and **Values** sections define the data fields that will form the graph axes. The **Series grouping** field allows me to define a value, education, to pivot on. By selecting **Apply**, I can now create a graph of total balance against a job type, grouped by the education type, as the following screenshot shows:

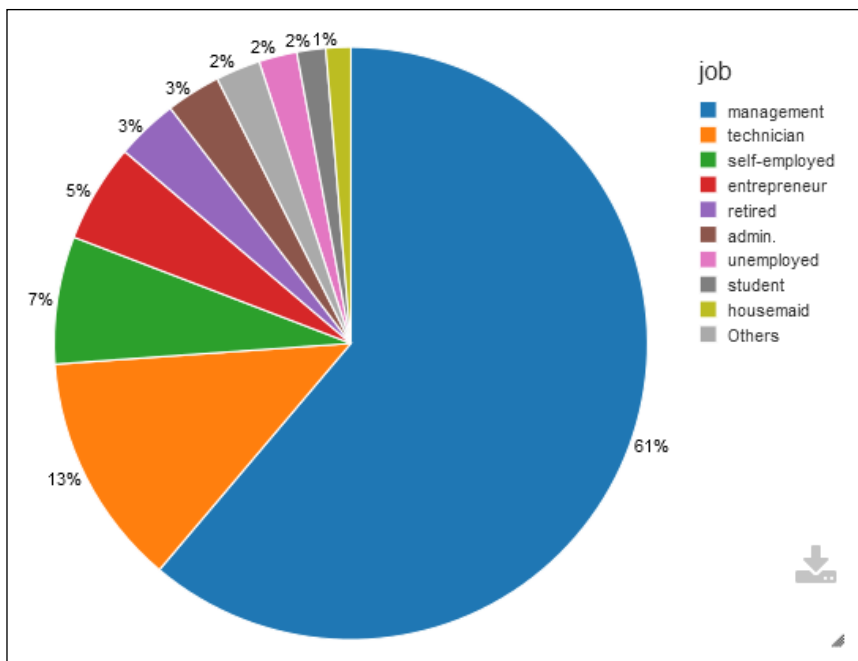


If I were an accountant trying to determine the factors affecting wage costs, and groups of employees within the company that cost the most, I would then see the green spike in the previous graph. It seems to indicate that the **management** employees with a tertiary education are the most costly group within the data. This can be confirmed by changing the SQL to filter on a **tertiary education**, ordering the result by balance descending, and creating a new bar graph.

```
> %sql select job,education,sum(balance)/1000 as totbal_k from shuttle
where education = "tertiary"
group by job,education
order by totbal_k desc
```



Clearly, the **management** grouping is approximately **14 million**. Changing the display option to **Pie** represents the data as a pie graph, with clearly sized segments and colors, which visually and clearly present the data, and the most important items.

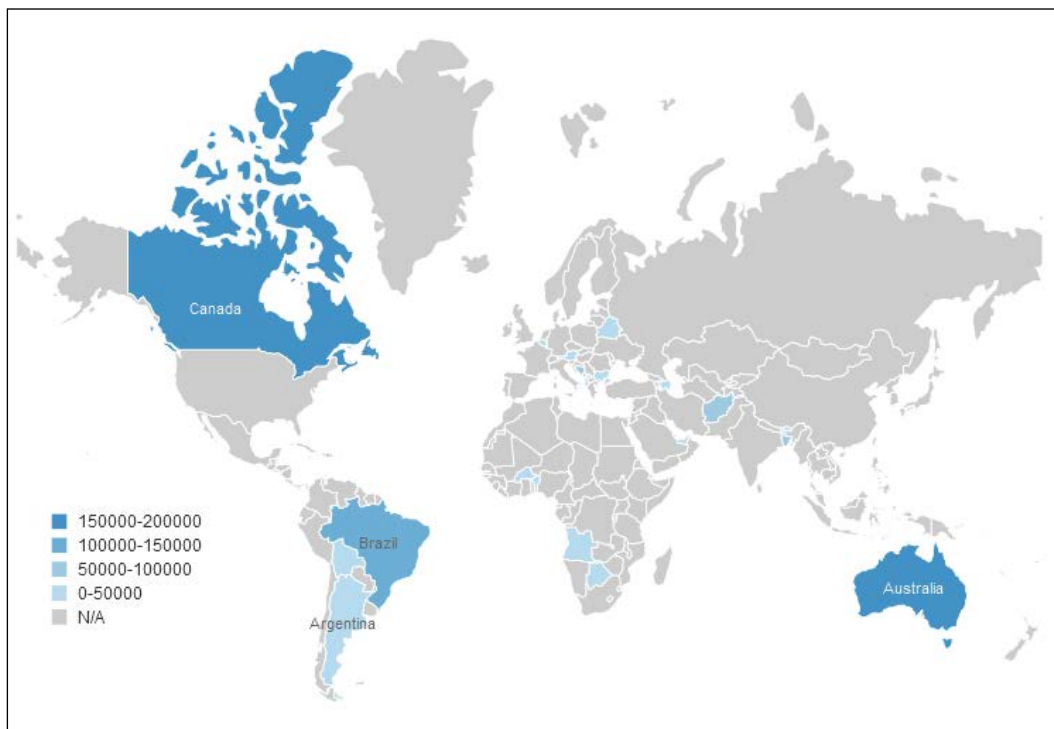


I cannot examine all of the display options in this small chapter, but what I did want to show is the world map graph that can be created using geographic information. I have downloaded the `Countries.zip` file from <http://download.geonames.org/export/dump/>.

This will offer a sizeable data set of around 281 MB compressed, which can be used to create a new table. It is displayed as a world map graph. I have also sourced an ISO2 to ISO3 set of mapping data, and stored it in a Databricks table called `cmap`. This allows me to convert ISO2 country codes in the data above i.e. "AU" to ISO3 country codes i.e. "AUS" (needed by the map graph I am about to use). The first column in the data that we will use for the map graph, must contain the geo location data. In this instance, the country codes in the ISO 3 format. So from the countries data, I will create a count of records for each country by ISO3 code. It is also important to ensure that the plot options are set up correctly in terms of keys, and values. I have stored the downloaded country-based data in a table called `geo1`. The SQL used is shown in the following screenshot:

```
> %sql select iso3 as country, 1 as value from
      geo1 left outer join cmap
      on countrycode = iso2
```

As shown previously, this gives two columns of data an ISO3-based value called `country`, and a numeric count called `value`. Setting the display option to `Map` creates a color-coded world map, shown in the following screenshot:



These graphs show how data can be visually represented in various forms, but what can be done if a report is needed for external clients or a dashboard is required? All this will be covered in the next section.

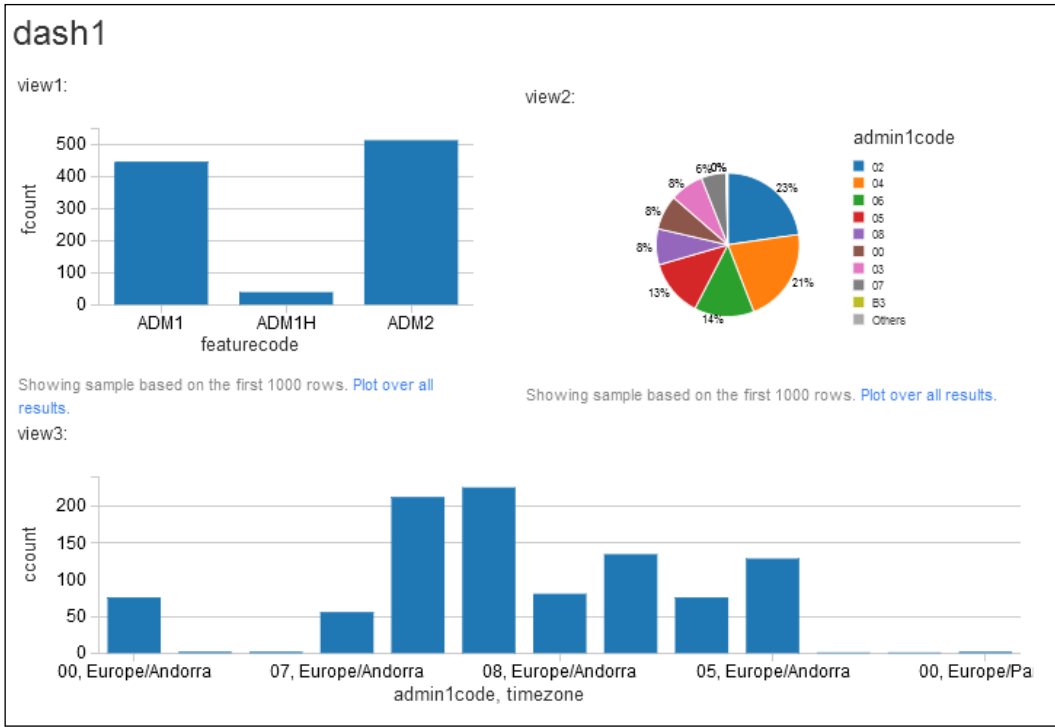
Dashboards

In this section, I will use the data in the table called `geo1`, which was created in the last section for a map display. It was made to create a simple dashboard, and publish the dashboard to an external client. From the **Workspace** menu, I have created a new dashboard called `dash1`. If I edit the controls tab of this dashboard, I can start to enter SQL, and create graphs, as shown in the following screenshot. Each graph is represented as a view and can be defined via SQL. It can be resized, and configured using the plot options as per the individual graphs. Use the **Add** drop-down menu to add a view. The following screenshot shows that `view1` is already created, and added to `dash1`. `view2` is being defined.

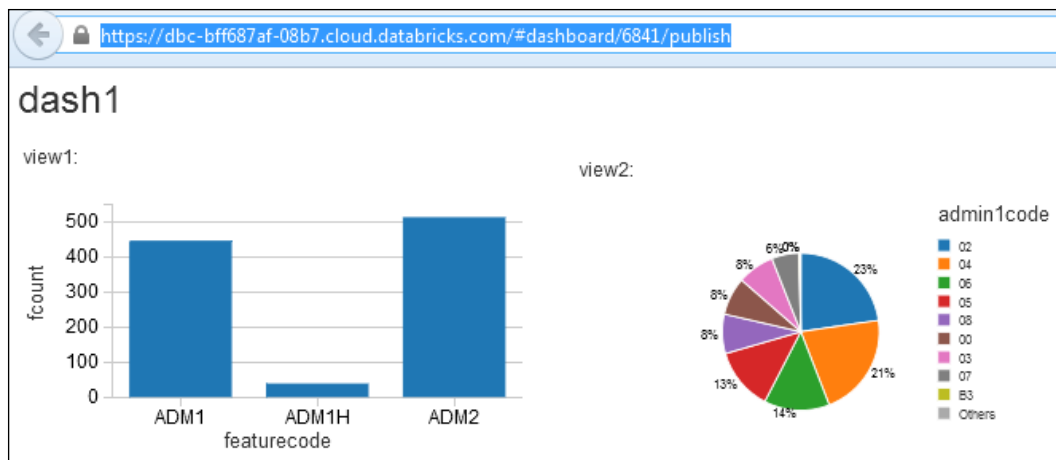
The screenshot shows a dashboard editor interface. At the top right, there are buttons for 'Edit', 'Publish', and 'Clone'. Below these, a 'Size' dropdown is set to '800x600' and an 'Add' button is visible. The dashboard itself is titled 'dash1' and has two tabs: 'Setup queries' and 'Controls'. The 'Controls' tab is active, showing a bar chart labeled 'view1'. The chart has a y-axis labeled 'fcount' ranging from 0 to 500 and an x-axis labeled 'featurecode' with categories 'ADM1', 'ADM1H', and 'ADM2'. The bars show counts of approximately 450 for ADM1, 50 for ADM1H, and 500 for ADM2. Below the chart, a note says 'Showing sample based on the first 1000 rows. Plot over all'. A 'Plot Options...' button is at the bottom left of the chart area. Overlaid on the right is a 'Create Data View' dialog box. It has a 'Label' field with 'view2', a 'Language' dropdown set to 'SQL', and a 'Query' text area containing the SQL: `select adminlcode,1 as account from geo1`. At the bottom of the dialog are 'Cancel' and 'Save' buttons.

featurecode	fcount
ADM1	450
ADM1H	50
ADM2	500

Once all the views have been added, positioned, and resized, the edit tab can be selected to present the finalized dashboard. The following screenshot now shows the finalized dashboard called `dash1` with three different graphs in different forms, and segments of the data:



This is very useful for giving a view of the data, but this dashboard is within the Databricks cloud environment. What if I want a customer to see this? There is a **publish** menu option in the top-right part of the dashboard screen, which allows you to publish the dashboard. This displays the dashboard under a new publicly published URL, as shown in the following screenshot. Note the new URL at the top of the following screenshot. You can now share this URL with your customers to present results. There are also options to periodically update the display to represent updates in the underlying data.



This gives you an idea of the available display options. All of the reports, and dashboards created so far have been based upon SQL, and the data returned. In the next section, I will show that reports can be created programmatically using a Scala-based Spark RDD, and streamed data.

An RDD-based report

The following Scala-based example uses a user-defined class type called `birdType`, based on the bird name, and the volume encountered. An RDD is created of the bird type records, and then converted into a data frame. The data frame is then displayed. Databricks allows the displayed data to be presented as a table or using plot options as a graph. The following image shows the Scala that is used:

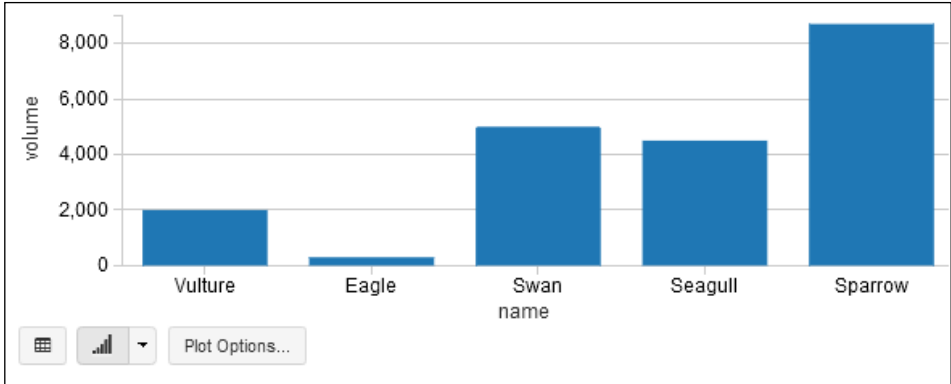
```
> case class birdType( name:String, volume: Int)

val birdRDD = sc.parallelize(

  birdType("Vulture",2000) ::
  birdType("Eagle" ,300) ::
  birdType("Swan" ,5000) ::
  birdType("Seagull",4500) ::
  birdType("Sparrow",8700) ::
  Nil).toDF()

display(birdRDD)
```

The bar graph, which this Scala example allows to be created, is shown in the following screenshot. The previous Scala code and the following screenshot are less important than the fact that this graph has been created programmatically using a data frame:

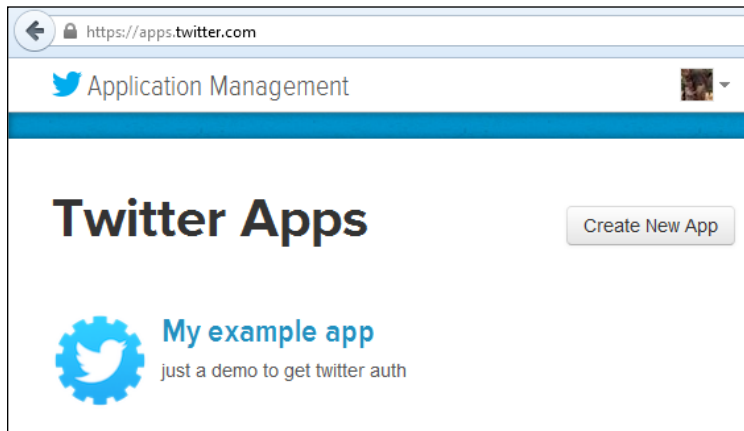


This opens up the possibility of programmatically creating data frames, and temporary tables from calculation-based data sources. It also allows for streamed data to be processed, and the refresh functionality of dashboards to be used, to constantly present a window of streamed data. The next section will examine a stream-based example of report generation.

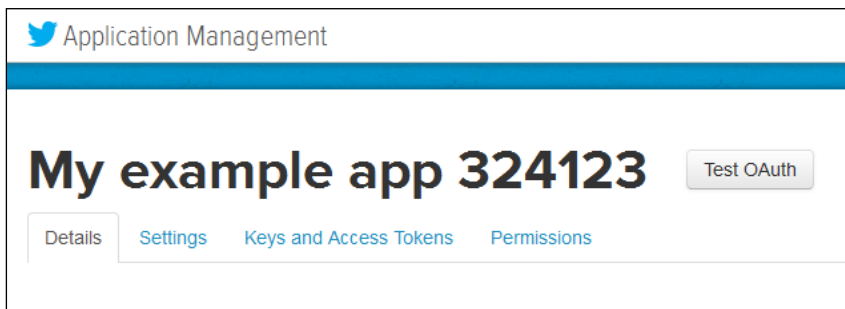
A stream-based report

In this section, I will use Databricks capability to upload a JAR-based library, so that we can run a Twitter-based streaming Apache Spark example. In order to do this, I must first create a Twitter account, and a sample application at: <https://apps.twitter.com/>.

The following screenshot shows that I have created an application called `My example app`. This is necessary, because I need to create the necessary access keys, and tokens to create a Scala-based twitter feed.



If I now select the application name, I can see the application details. This provides a menu option, which provides access to the application details, settings, access tokens, and permissions. There is also a button which says **Test OAuth**, this enables the access and token keys that will be created to be tested. The following screenshot shows the application menu options:



By selecting the **Keys and Access Tokens** menu option, the access keys, and the access tokens can be generated for the application. Each of the application settings and tokens, in this section, have an API key, and a secret key. The top of the form, in the following screenshot, shows the consumer key, and consumer secret (of course, the key and account details have been removed from these images for security reasons).

Details

Settings

Keys and Access Tokens

Permissions

Application Settings

Keep the "Consumer Secret" a secret. This key should never be human-readable in your application.

Consumer Key (API Key)

QQp

Consumer Secret (API Secret)

0HFz

Access Level

Read, write, and direct messages (modify app permissions)

Owner

frampton_mike

Owner ID

3239

Application Actions

Regenerate Consumer Key and Secret

Change App Permissions

There are also options in the previous screenshot to regenerate the keys, and set permissions. The next screenshot shows the application access token details. There is an access token, and an access token secret. It also has the options to regenerate the values, and revoke access:

Your Access Token

This access token can be used to make API requests on your own account's behalf. Do not share

Access Token	3239
Access Token Secret	llQv:
Access Level	Read, write, and direct messages
Owner	frampton_mike
Owner ID	323

Token Actions

Regenerate My Access Token and Token Secret
Revoke Token Access

Using these four alpha numeric value strings, it is possible to write a Scala example to access a Twitter stream. The values that will be needed are as follows:

- Consumer Key
- Consumer Secret
- Access Token
- Access Token Secret

In the following code sample, I will remove my own key values for security reasons. You just need to add your own values to get the code to work. I have developed my library, and run the code locally to check whether it will work. I did this before loading it to Databricks, in order to reduce time, and costs due to debugging. My Scala code sample looks like the following code. First, I define a package, import Spark streaming, and twitter resources. Then, I define an object class called `twitter1`, and create a main function:

```
package nz.co.semtechsolutions

import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.streaming._
import org.apache.spark.streaming.twitter._
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.sql._
```

```
import org.apache.spark.sql.types.{StructType, StructField, StringType}

object twitter1 {

  def main(args: Array[String]) {
```

Next, I create a Spark configuration object using an application name. I have not used a Spark master URL, as I will let both, `spark-submit`, and Databricks assign the default URL. From this, I will create a Spark context, and define the Twitter consumer, and access values:

```
val appName = "Twitter example 1"
val conf     = new SparkConf()

conf.setAppName(appName)
val sc = new SparkContext(conf)

val consumerKey      = "QQpl8xx"
val consumerSecret   = "0HFzxx"
val accessToken      = "323xx"
val accessTokenSecret = "Ilxx"
```

I set the Twitter access properties using the `System.setProperty` call, and use it to set the four `twitter4j` oauth access properties using the access keys, which were generated previously:

```
System.setProperty("twitter4j.oauth.consumerKey", consumerKey)
System.setProperty("twitter4j.oauth.consumerSecret",
  consumerSecret)
System.setProperty("twitter4j.oauth.accessToken", accessToken)
System.setProperty("twitter4j.oauth.accessTokenSecret",
  accessTokenSecret)
```

A streaming context is created from the Spark context, which is used to create a Twitter-based Spark DStream. The stream is split by spaces to create words, and it gets filtered by the words starting with `#`, to select hash tags:

```
val ssc = new StreamingContext(sc, Seconds(5) )
val stream = TwitterUtils.createStream(ssc, None)
  .window( Seconds(60) )

// split out the hash tags from the stream

val hashTags = stream.flatMap( status => status.getText.split("
").filter(_.startsWith("#")))
```

The function used below to get a singleton SQL Context is defined at the end of this example. So, for each RDD in the stream of hash tags, a single SQL context is created. This is used to import implicits which allows an RDD to be implicitly converted to a data frame using `toDF`. A data frame is created from each `rdd` called `dfHashTags`, and this is then used to register a temporary table. I have then run some SQL against the table to get a count of rows. The count of rows is then printed. The horizontal banners in the code are just used to enable easier viewing of the output of results when using `spark-submit`:

```
hashTags.foreachRDD{ rdd =>

  val sqlContext = SQLContextSingleton.getInstance(rdd.sparkContext)
  import sqlContext.implicits._

  val dfHashTags = rdd.map(hashT => hashRow(hashT) ).toDF()

  dfHashTags.registerTempTable("tweets")

  val tweetcount = sqlContext.sql("select count(*) from tweets")

  println("\n=====")
  println( "=====\\n")

  println("Count of hash tags in stream table : "
    + tweetcount.toString )

  tweetcount.map(c => "Count of hash tags in stream table : "
    + c(0).toString ).collect().foreach(println)

  println("\n=====")
  println( "=====\\n")

} // for each hash tags rdd
```

I have also output a list of the top five tweets by volume in my current tweet stream data window. You might recognize the following code sample. It is from the Spark examples on GitHub. Again, I have used the banner to help with the results that will be seen in the output:

```
val topCounts60 = hashTags.map((_, 1))
  .reduceByKeyAndWindow(_ + _, Seconds(60))
  .map{case (topic, count) => (count, topic)}
```

```
.transform(_.sortByKey(false))

topCounts60.foreachRDD(rdd => {

    val topList = rdd.take(5)

    println("\n=====")
    println( "=====\\n")
    println("\\nPopular topics in last 60 seconds (%s total):"
        .format(rdd.count()))
    topList.foreach{case (count, tag) => println("%s (%s tweets)"
        .format(tag, count))}
    println("\\n=====")
    println( "=====\\n")
})
```

Then, I have used `start` and `awaitTermination`, via the Spark stream context `ssc`, to start the application, and keep it running until stopped:

```
ssc.start()
ssc.awaitTermination()

} // end main
} // end twitter1
```

Finally, I have defined the singleton SQL context function, and the dataframe case class for each row in the hash tag data stream `rdd`:

```
object SQLContextSingleton {
    @transient private var instance: SQLContext = null

    def getInstance(sparkContext: SparkContext):
        SQLContext = synchronized {
            if (instance == null) {
                instance = new SQLContext(sparkContext)
            }
            instance
        }
}

case class hashRow( hashTag: String)
```

I compiled this Scala application code using SBT into a JAR file called `data-bricks_2.10-1.0.jar`. My SBT file looks as follows:

```
[hadoop@hc2nn twitter1]$ cat twitter.sbt
```

```
name := "Databricks"
```

```

version := "1.0"
scalaVersion := "2.10.4"
libraryDependencies += "org.apache.spark" % "streaming" % "1.3.1" from
"file:///usr/local/spark/lib/spark-assembly-1.3.1-hadoop2.3.0.jar"
libraryDependencies += "org.apache.spark" % "sql" % "1.3.1" from
"file:///usr/local/spark/lib/spark-assembly-1.3.1-hadoop2.3.0.jar"
libraryDependencies += "org.apache.spark.streaming" % "twitter" % "1.3.1"
from file:///usr/local/spark/lib/spark-examples-1.3.1-hadoop2.3.0.jar

```

I downloaded the correct version of Apache Spark onto my cluster to match the current version used by Databricks at this time (1.3.1). I then installed it under `/usr/local/` on each node in my cluster, and ran it in local mode with spark as the cluster manager. My spark-submit script looks as follows:

```

[hadoop@hc2nn twitter1]$ more run_twitter.bash
#!/bin/bash

SPARK_HOME=/usr/local/spark
SPARK_BIN=$SPARK_HOME/bin
SPARK_SBIN=$SPARK_HOME/sbin

JAR_PATH=/home/hadoop/spark/twitter1/target/scala-2.10/data-bricks_2.10-1.0.jar
CLASS_VAL=nz.co.semtechsolutions.twitter1

TWITTER_JAR=/usr/local/spark/lib/spark-examples-1.3.1-hadoop2.3.0.jar

cd $SPARK_BIN

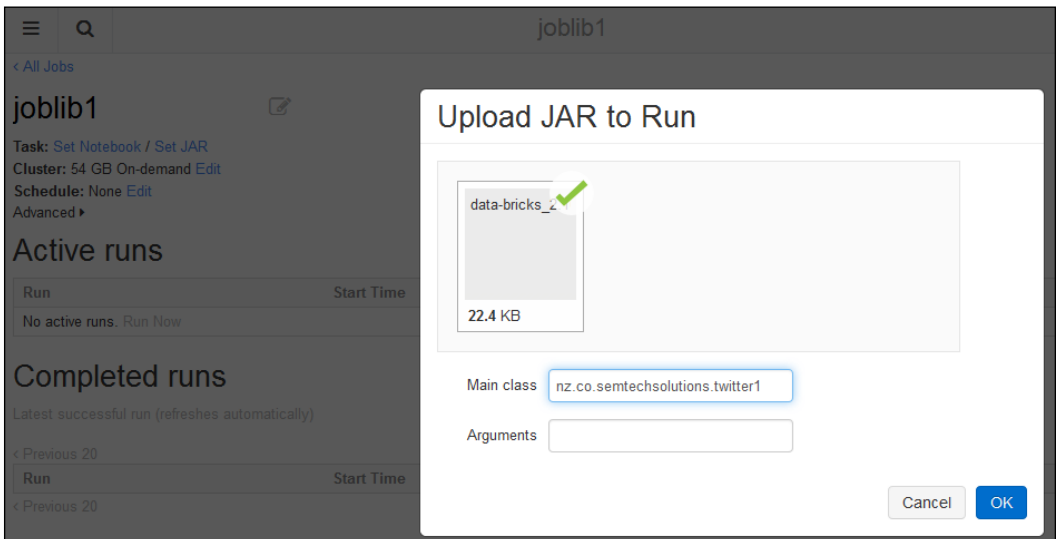
./spark-submit \
  --class $CLASS_VAL \
  --master spark://hc2nn.semtech-solutions.co.nz:7077 \
  --executor-memory 100M \
  --total-executor-cores 50 \
  --jars $TWITTER_JAR \
  $JAR_PATH

```

I won't go through the details, as it has been covered quite a few times, except to note that the class value is now `nz.co.semtechsolutions.twitter1`. This is the package class name, plus the application object class name. So, when I run it locally, I get an output as follows:

```
=====
Count of hash tags in stream table : 707
=====
Popular topics in last 60 seconds (704 total):
#KCAMÉXICO (139 tweets)
#BE3 (115 tweets)
#Fallout4 (98 tweets)
#OrianaSabatini (69 tweets)
#MartinaStoessel (61 tweets)
=====
```

This tells me that the application library works. It connects to Twitter, creates a data stream, is able to filter the data into hash tags, and creates a temporary table using the data. So, having created a JAR library for Twitter data streaming, and proving that it works, I'm now able to load it onto the Databricks cloud. The following screenshot shows that a job has been created from the Databricks cloud jobs menu called `joblib1`. The **Set Jar** option has been used to upload the JAR library that was just created. The full package-based name to the `twitter1` application object class has been specified.



The following screenshot shows the `joblib1` job, which is ready to run. A Spark-based cluster will be created on demand, as soon as the job is executed using the **Run Now** option, under the **Active runs** section. No scheduling options have been specified, although the job can be defined to run at a given date and time.

[< All Jobs](#)

joblib1



Task: nz.co.semtechsolutions.twitter1 in [data-bricks_2.10-1.0.jar](#) uploaded at 2015-06-15 18:56:55 - [Edit](#) / [Remove](#)

◦ Arguments:

Cluster: 54 GB On-demand [Edit](#)

Schedule: None [Edit](#)

Advanced ▶

Active runs

Run	Start Time	Launched	Duration	Status
No active runs. Run Now				

I selected the **Run Now** option to start the job run, as shown in the following screenshot. This shows that there is now an active run called `Run 1` for this job. It has been running for six seconds. It was launched manually, and is pending while a on-demand cluster is created. By selecting the run name `Run 1`, I can see details about the job, especially the logged output.

joblib1



Task: nz.co.semtechsolutions.twitter1 in [data-bricks_2.10-1.0.jar](#) uploaded at 2015-06-15 18:56:55 - [Edit](#) / [Remove](#)

◦ Arguments:

Cluster: 54 GB On-demand [Edit](#)

Schedule: None [Edit](#)

Advanced ▶

Active runs

Run	Start Time	Launched	Duration	Status
Run 1	2015-06-15 18:57:26	Manually	6s	Pending - Cancel

The following screenshot shows an example of the output for Run 1 of joblib1. It shows the time started and duration, it also shows the running status and job details in terms of class and JAR file. It would have shown class parameters, but there were none in this case. It also shows the details of the 54 GB on-demand cluster. More importantly, it shows the list of the top five tweet hash tag values.

[< All Jobs / joblib1](#)

Run 1 of joblib1

Started: 2015-06-15 18:57:26
Duration: 35s
Status: Running - [Cancel](#)
Task: nz.co.semtechsolutions.twitter1 in [data-bricks_2.10-1.0.jar](#)
 ◦ Arguments:
Cluster: 54 GB On-demand
Message: In run

Output

Standard output: [Full log](#)

```
=====
Popular topics in last 60 seconds (91 total):
#iChooseNicki (8 tweets)
#Masturbation (3 tweets)
#SexToys (3 tweets)
#Matures (3 tweets)
#MILFs (3 tweets)
=====
```

The following screenshot shows the same job run output window in the Databricks cloud instance. But this shows the output from the SQL `count (*)`, showing the number of tweet hash tags in the current data stream tweet window from the temporary table.

Output

Standard output: [Full log](#)

```
=====
Count of hash tags in stream table : 523
=====
```

So, this proves that I can create an application library locally, using Twitter-based Apache Spark streaming, and convert the data stream into data frames, and a temporary table. It shows that I can reduce costs by developing locally, and then port my library to my Databricks cloud. I am aware that I have neither visualized the temporary table, nor the DataFrame in this example, into a Databricks graph, but time scales did not allow me to do this. Also, another thing that I would have done, if I had time, would be to checkpoint, or periodically save the stream to file, in case of application failure. However, this topic is covered in *Chapter 3, Apache Spark Streaming* with an example, so you can take a look there if you are interested. In the next section, I will examine the Databricks REST API, which will allow better integration between your external applications, and your Databricks cloud instance.

REST interface

Databricks provides a REST interface for Spark cluster-based manipulation. It allows for cluster management, library management, command execution, and the execution of contexts. To be able to access the REST API, the port 34563 must be accessible for your instance in the AWS EC2-based Databricks cloud. The following Telnet command shows an attempt to access the port 34563 of my Databricks cloud instance. Note that the Telnet attempt has been successful:

```
[hadoop@hc2nn ~]$ telnet dbc-bff687af-08b7.cloud.databricks.com 34563
Trying 52.6.229.109...
Connected to dbc-bff687af-08b7.cloud.databricks.com.
Escape character is '^['.
```

If you do not receive a Telnet session, then contact Databricks via help@databricks.com. The next sections provide examples of REST interface access to your instance on the Databricks cloud.

Configuration

In order to use the interface, I needed to whitelist the IP address that I use to access my Databricks cluster instance. This is the IP address of the machine from which I will be running the REST API commands. By whitelisting the IP addresses, Databricks can ensure that a secure list of users access each Databricks cloud instance.

I contacted Databricks support via the previous help email address, but there is also a Whitelist IP Guide, found in the **Workspace** menu in your cloud instance:

Workspace | databricks_guide | DevOps Utilities | Whitelist IP.

REST API calls can now be submitted to my Databricks cloud instance, from the Linux command line, using the Linux `curl` command. The example general form of the `curl` command is shown next using my Databricks cloud instance username, password, cloud instance URL, REST API path, and parameters.

The Databricks forum, and the previous help email address can be used to gain further information. The following sections will provide some REST API worked examples:

```
curl -u '<user>:<paswd>' <dbc url> -d "<parameters>"
```

Cluster management

You will still need to create Databricks Spark clusters from your cloud instance user interface. The list REST API command is as follows:

```
/api/1.0/clusters/list
```

It needs no parameters. This command will provide a list of your clusters, their status, IP addresses, names, and the port numbers that they run on. The following output shows that the cluster `semclust1` is in a pending state in the process of being created:

```
curl -u 'xxxx:yyyyy' 'https://dbc-bff687af-08b7.cloud.databricks.com:34563/api/1.0/clusters/list'
```

```
[{"id":"0611-014057-waist9","name":"semclust1","status":"Pending","driverIp":"","jdbcPort":10000,"numWorkers":0}]
```

The same REST API command run when the cluster is available, shows that the cluster called `semclust1` is running, and has one worker:

```
[{"id":"0611-014057-waist9","name":"semclust1","status":"Running","driverIp":"10.0.196.161","jdbcPort":10000,"numWorkers":1}]
```

Terminating this cluster, and creating a new one called `semclust` changes the results of the REST API call as shown:

```
curl -u 'xxxx:yyyy' 'https://dbc-bff687af-08b7.cloud.databricks.com:34563/api/1.0/clusters/list'
```

```
[{"id":"0611-023105-moms10","name":"semclust", "status":"Pending","driverIp":"","jdbcPort":10000,"numWorkers":0},
 {"id":"0611-014057-waist9","name":"semclust1","status":"Terminated","driverIp":"10.0.196.161","jdbcPort":10000,"numWorkers":1}]
```

The execution context

With these API calls, you can create, show the status of, or delete an execution context. The REST API calls are as follows:

- `/api/1.0/contexts/create`
- `/api/1.0/contexts/status`
- `/api/1.0/contexts/destroy`

In the following REST API call example, submitted via `curl`, a Scala context has been created for the cluster `semclust` identified by its cluster ID.

```
curl -u 'xxxx:yyyy' https://dbc-bff687af-08b7.cloud.databricks.com:34563/api/1.0/contexts/create -d "language=scala&clusterId=0611-023105-moms10"
```

The result returned is either an error, or a context ID. The following three example return values show an error caused by an invalid URL, and two successful calls returning context IDs:

```
{"error":"ClusterNotFoundException: Cluster not found: semclust1"}
{"id":"8689178710930730361"}
{"id":"2876384417314129043"}
```

Command execution

These commands allow you to run a command, list a command status, cancel a command, or show the results of a command. The REST API calls are as follows:

- `/api/1.0/commands/execute`
- `/api/1.0/commands/cancel`
- `/api/1.0/commands/status`

The following example shows an SQL statement being run against an existing table called `cmap`. The context must exist, and must be of the SQL type. The parameters have been passed on to the HTTP GET call via a `-d` option. The parameters are language, the cluster ID, the context ID, and the SQL command. The command ID is returned as follows:

```
curl -u 'admin:FirmWare1$34' https://dbc-bff687af-08b7.cloud.databricks.com:34563/api/1.0/commands/execute -d
"language=sql&clusterId=0611-023105-moms10&contextId=7690632266172649068&command=select count(*) from cmap"
```

```
{"id":"d8ec4989557d4a4ea271d991a603a3af"}
```

Libraries

The REST API also allows for libraries to be uploaded to a cluster and their statuses checked. The REST API call paths are as follows:

- `/api/1.0/libraries/upload`
- `/api/1.0/libraries/list`

An example is given next of a library upload to the cluster instance called `semclust`. The parameters passed on to the HTTP GET API call via a `-d` option are the language, cluster ID, the library name and URI. A successful call results in the name and URI of the library, which is as follows:

```
curl -u 'xxxx:yyyy' https://dbc-bff687af-08b7.cloud.databricks.com:34563/api/1.0/libraries/upload
-d "language=scala&clusterId=0611-023105-moms10&name=lib1&uri=file:///home/hadoop/spark/ann/target/scala-2.10/a-n-n_2.10-1.0.jar"
```

```
{"name":"lib1","uri":"file:///home/hadoop/spark/ann/target/scala-2.10/a-n-n_2.10-1.0.jar"}
```

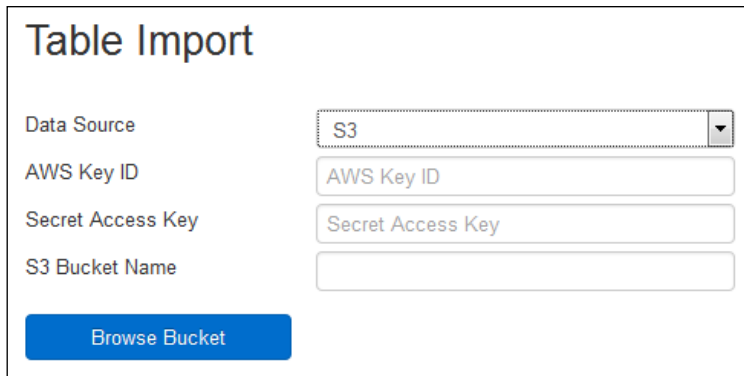
Note that this REST API can change by content and version overtime, so check in the Databricks forum, and use the previous help email address to check the API details with Databricks support. I do think though that, with these simple example calls, it is clear that this REST API can be used to integrate Databricks with the external systems, and ETL chains. In the next section, I will provide an overview of data movement within the Databricks cloud.

Moving data

Some of the methods of moving data in and out of Databricks have already been explained in *Chapter 8, Spark Databricks* and *Chapter 9, Databricks Visualization*. What I would like to do in this section is provide an overview of all of the methods available for moving data. I will examine the options for tables, workspaces, jobs, and Spark code.

The table data

The table import functionality for Databricks cloud allows data to be imported from an AWS **S3** bucket, from the **Databricks file system (DBFS)**, via JDBC and finally from a local file. This section gives an overview of each type of import, starting with **S3**. Importing the table data from AWS **S3** requires the AWS Key, the AWS secret key, and the **S3** bucket name. The following screenshot shows an example. I have already provided an example of **S3** bucket creation, including adding an access policy, so I will not cover it again.



The screenshot displays the 'Table Import' interface. It features a title 'Table Import' at the top left. Below the title, there are four input fields arranged vertically. The first field is labeled 'Data Source' and has a dropdown menu with 'S3' selected. The second field is labeled 'AWS Key ID' and contains the placeholder text 'AWS Key ID'. The third field is labeled 'Secret Access Key' and contains the placeholder text 'Secret Access Key'. The fourth field is labeled 'S3 Bucket Name' and is currently empty. At the bottom left of the form, there is a blue button labeled 'Browse Bucket'.

Once the form details are added, you will be able to browse your **S3** bucket for a data source. Selecting **DBFS** as a table data source enables your **DBFS** folders, and files to be browsed. Once a data source is selected, it can display a preview as the following screenshot shows:

Table Import

Data Source: DBFS

Select table source

- FileStore
- data
- data1
- data2
- mnt
- mount
- tmp
- user

Preview

Selecting **JDBC** as a table data source allows you to specify a remote SQL database as a data source. Just add an access **URL**, **Username**, and **Password**. Also, add some SQL to define the table, and columns to source. There is also an option of adding extra properties to the call via the **Add Property** button, as the following screenshot shows:

Table Import

Data Source: JDBC

URL: (Required)

Username: (Leave blank if not needed)

Password: (Leave blank if not needed)

Table / Query: (Required)

Key (e.g. useSSL): Value (e.g. true) ✕

Connect Add Property

Selecting the **File** option to populate a Databricks cloud instance table, from a file, creates a drop down or browse. This upload method was used previously to upload CSV-based data into a table. Once the data source is specified, it is possible to specify a data separator string or header row, define column names or column types and preview the data before creating the table.

Table Import

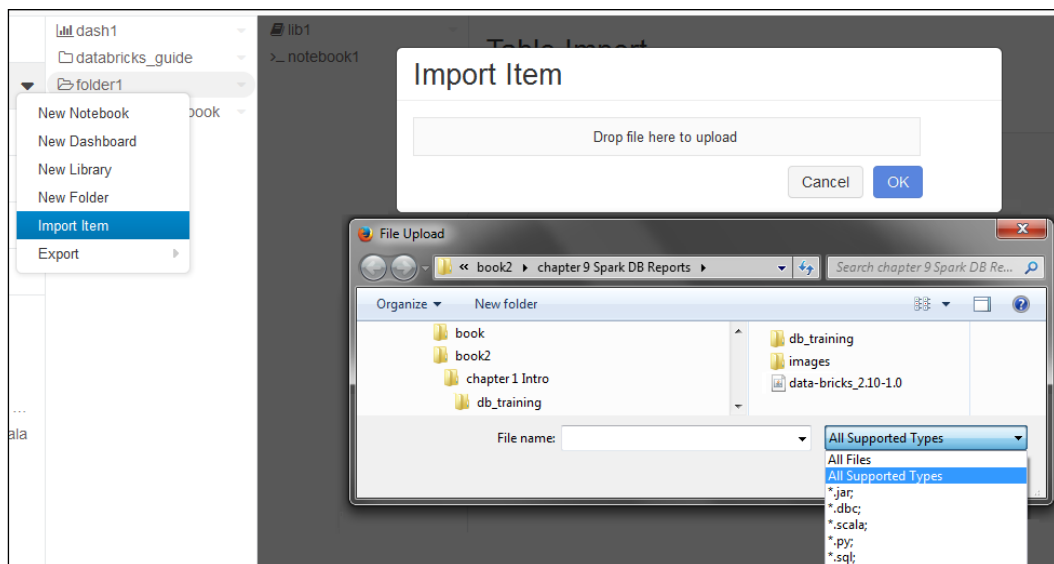
Data Source: File

File

Drop file or click here to upload

Folder import

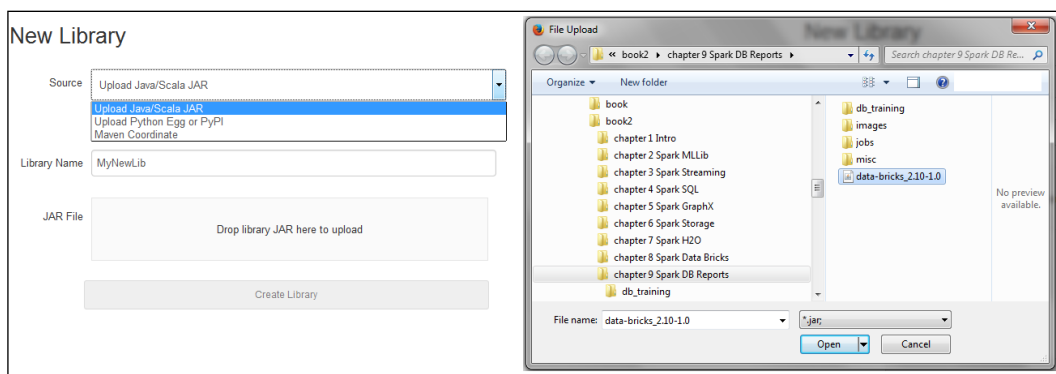
From either a workspace, or a folder drop-down menu, it is possible to import an item. The following screenshot shows a compound image from the **Import Item** menu option:



This creates a file drop or browse window, which when clicked, allows you to browse the local server for the items to import. Selecting the **All Supported Types** option shows that the items to import can be JAR files, dbc archives, Scala, Python, or SQL files.

Library import

The following screenshot shows the **New Library** functionality, from the Workspace and folder menu options. This allows an externally created and tested library to be loaded to your Databricks cloud instance. The library can be in the form of a Java or Scala JAR file, a Python Egg or a Maven coordinate for repository access. In the following screenshot, a JAR file is being selected from the local server via a browse window. This functionality has been used in this chapter to test stream-based Scala programming:



Further reading

Before summing up this chapter, and the last for cloud-based Apache Spark usage in Databricks, I wanted to mention some resources for gaining extra information on both, Apache Spark, and Databricks. First, there is the Databricks forum available at: forums.databricks.com/ for questions, and answers related to the use of <https://databricks.com/>. Also, within your Databricks instance, under the Workspace menu option, there will be a Databricks guide that contains a lot of useful information. The Apache Spark website at <http://spark.apache.org/> also contains a lot of useful information, as well as module-based API documentation. Finally, there is the Spark mailing list, user@spark.apache.org, which provides a great deal of Spark usage information, and problem solving.

Summary

Chapter 8, Spark Databricks and *Chapter 9, Databricks Visualization*, have provided an introduction to Databricks in terms of cloud installation, and the use of Notebooks and folders. Account and cluster management have been examined. Also, job creation, the idea of remote library creation, and importing have been examined. The functionality of the Databricks `dbutils` package, and the Databricks file system was explained in *Chapter 8, Spark Databricks*. Tables, and an example of data import was also shown so that SQL can be run against a dataset.

The idea of data visualization has been examined, and a variety of graphs have also been created. Dashboards have been made to show how easy it is to both, create, and share this kind of data presentation. The Databricks REST interface has been shown via worked examples, as an aid to using a Databricks cloud instance remotely, and integrating it with external systems. Finally, the data and library movement options have been examined in terms of workspace, folders, and tables.

You might ask why I have committed two chapters to a cloud-based service such as Databricks. The reason is that Databricks seems to be a logical, cloud-based progression, from Apache Spark. It is supported by the people who originally developed Apache Spark and although in its infancy as a service and subject to change still capable of providing a Spark cloud based production service. This means that a company wishing to use a Spark could use Databricks and grow their cloud as demand grows and have access to dynamic Spark-based machine learning, graph processing, SQL, streaming and visualization functionality.

As ever, these Databricks chapters have just scratched the surface of the functionality available. The next step will be to create an AWS and Databricks account yourself, and use the information provided here to gain practical experience.

As this is the last chapter, I will provide my contact details again. I would be interested in the ways that people are using Apache Spark. I would be interested in the size of clusters you are creating, and the data that you are processing. Are you using Spark as a processing engine? Or are you building systems on top of it? You can connect with me at LinkedIn at: [linkedin.com/profile/view?id=73219349](https://www.linkedin.com/profile/view?id=73219349).

You can contact me via my website at semtech-solutions.co.nz or finally, by email at: info@semtech-solutions.co.nz.

Finally, I maintain a list of open-source-software-related presentations when I have the time. Anyone is free to use, and download them. They are available on SlideShare at: <http://www.slideshare.net/mikejf12/presentations>.

If you have any challenging opportunities or problems, please feel free to contact me using the previous details.

Index

A

account management, Databricks 226, 227

Amazon AWS

pricing, URL 12

URL 11

Amazon EC2

about 10-13

URL 10

Amazon Elastic Compute Cloud (EC2) 222

Apache Giraph 133

Apache Kafka

about 2, 82-93

JAR library file, URL 82

URL 82

Apache Mesos 9

Apache Spark

about 1, 2

cluster design 5-7

cluster management 8

Databricks 221

extended eco system 4

future 5

graph processing 4

GraphX 131

MLlib 17

overview 2

performance, examining 13

Spark Machine Learning 3

SQL context 96

SQL module 4

stream processing 3

Titan, accessing with 177

URL 3, 63, 282

used, for accessing Cassandra 174-177

used, for accessing HBase 165-169

Apache Spark SQL 95

Apache Spark streaming

data sources 66

errors 63

HDFS-based checkpoint, setting up 64-66

overview 62, 63

recovery 63

URL 62

Apache YARN 9

architecture, H2O 195-198

Artificial Neural Networks (ANN)

about 41, 198

Spark server, sparkling 44-47

theory 41-44

using 48-58

AWS billing 224

B

BaseConfiguration method 183

Bruce Penn

URL 14

C

Cassandra

about 2

accessing, with Apache Spark 174-177

installing 169-171

Titan, accessing with 169

classifications, with Naïve Bayes

about 25, 26

theory 25, 26

closeness centrality algorithm 150, 151

Cloudera

URL 120

cluster design, Apache Spark 5-7

clustering, with K-Means

about 36

theory 36

cluster management

about 8

Amazon EC2 10-12

Apache Mesos 9

Apache YARN 9

local mode 8

standalone mode 8

connected components algorithm 152

D

dashboards 62

data

folder, importing 281

importing 97

JSON files, processing 97-99

library, importing 282

moving 279

Parquet files, processing 100

quality 199

saving 97

sourcing 198, 199

table data, importing 279-281

text files, processing 97

databases 62

Databricks

about 221, 255

account management 226

AWS billing 224

cluster management 228-231

development environments 240

folder 231

installing 222, 223

jobs 235

libraries 235

menu 225

Notebooks 231

overview 222

references 282

URL 10, 15, 282

Databricks filesystem (DBFS)

about 279

accessing 250

Databricks tables

about 240

creating, via data import 241-243

external tables 244-248

DataFrames 101, 102

data sources, Apache Spark streaming

about 66

Apache Kafka 82-93

file streams 69

Flume 62, 70-82

HDFS 62

Kafka 62

TCP stream 67-69

DataStax Spark Cassandra connector

URL 173

data visualization

about 255-260

dashboards 261, 263

RDD-based report 263, 264

stream-based report 264-275

dbutils.fs class 246

dbutils package

about 248, 249

cache functionality 252

DBFS 248

fsutils group 250-252

mount functionality 252

deep learning

about 200, 201

MNIST 207, 208

Scala-based H2O Sparkling

Water example 202-207

URL 201

discrete stream (DStream) 63

Docker

installing 144-147

URL 144

E

end of file markers (EOF) 184

environment configuration, MLlib

architecture 18, 19

development environment 19-21

- Spark, installing 21-24
- environment, H2O**
 - processing 190, 191
- Extract, Transform, Load (ETL)** 19

F

- False Positive Rate (FPR)** 219

Flume

- about 2, 70-82
- URL 70

- folder** 231-235

G

graph, creating

- connected components 141-143
- counting example 138, 139
- filtering example 139
- PageRank algorithm 140
- triangle counting 141

- GraphInputFormat class** 185

GraphX

- about 131
- coding 134
- overview 131-133

GraphX coding

- about 134
- environment 134-136
- graph, creating 137, 138

- Gremlin language** 157

H

H2O

- about 189
- architecture 195-198
- build environment 192-195
- environment, processing 190, 191
- installing 191, 192
- overview 190
- performance tuning 200
- Sparkling Water download option,
 - URL 191
- system versions, URL 191
- URL 189, 195

H2O flow

- about 208-220
- URL 211

- Hadoop file system** 14

Hadoop Gremlin

- URL 181

HBase

- about 2
- accessing, with Apache Spark 165-169
- Titan, accessing with 159

- head function** 251

Hive

- Hive-based Metastore server 121-128
- local Metastore server 115-121
- using 115

- Hive-based Metastore server**

- using 121-128

J

- JavaScript Object Notation (JSON) files**

- processing 97, 98

K

K-Means

- clustering 36
- using 36-41

L

LabeledPoint

- URL 27

- local Hive Metastore server**

- using 115-121

M

- Machine Learning library (MLlib)**

- about 2, 17
- environment configuration 17

markdown

- URL 232

- Mazerunner algorithms**

- about 149
- closeness centrality algorithm 150, 151
- connected components algorithm 152
- PageRank algorithm 150

- strongly connected components
 - algorithm 152
- triangle count algorithm 151

Mazrunner, for Neo4j

- about 143, 144
- algorithms 149
- Docker, installing 144-147
- Neo4j browser 147-149

MNIST

- about 207, 208
- URL 198

N

Naïve Bayes

- classification 25
- URL 26
- using 26-35

Neo4j browser

- about 147, 148
- URL 147

Notebook 231-235

O

Oryx system

- URL 15

Out of Memory(OOM) messages 14

P

PageRank algorithm 150

Parquet files

- about 97
- processing 100

performance

- cluster structure 13
- code, tuning 15
- data locality 14
- examining 13
- Hadoop file system 14
- Out of Memory(OOM) messages,
 - avoiding 14

PostgreSQL connector library

- URL, for download 122

PredictionIO

- URL 15

R

remove function (rm) 252

REST interface

- about 275
- cluster management 276, 277
- command execution 277
- configuration 276
- execution context 277
- libraries 278

S

SeldonIO

- URL 15

Sister property 133

Sparkling Water component, H2O

- about 189
- URL 191

Spark Machine Learning 3

SparkOnHBase module

- URL 164

Spark SQL 4

SQL context 96

stream processing 3

- strongly connected components
 - algorithm 152

T

tertiary education 258

textFile method 97

text files

- processing 97

TinkerPop

- about 157
- URL 157

Titan

- about 156, 157
- accessing, with Apache Spark 177-186
- accessing, with Cassandra 169-173
- accessing, with HBase 159-165
- installing 158
- URL 156-158

Titan, accessing with Apache Spark

- about 178
- alternative Groovy configuration 183
- Cassandra, using 184, 185

- filesystem, using 185, 186
- Gremlin shell 179, 180
- Groovy commands, executing 179-181
- HBase, using 185
- TinkerPop Hadoop Gremlin
 - package 181, 182
- Titan, accessing with Cassandra**
 - about 169
 - Cassandra, installing 169-171
 - Gremlin Cassandra script 172, 173
 - Spark Cassandra connector 173
- Titan, accessing with HBase**
 - about 159
 - Gremlin HBase script 160-164
 - HBase cluster, using 159, 160
 - SparkOnHBase module, using 164, 165
- TitanFactory.open method 184**
- triangle count algorithm 151**
- True Positive Rate (TPR) 219**
- Twitter**
 - URL 264

U

- user-defined functions (UDFs) 110-114**

V

- velox system**
 - URL 15
- Vendor API 157**



Thank you for buying Mastering Apache Spark

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

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

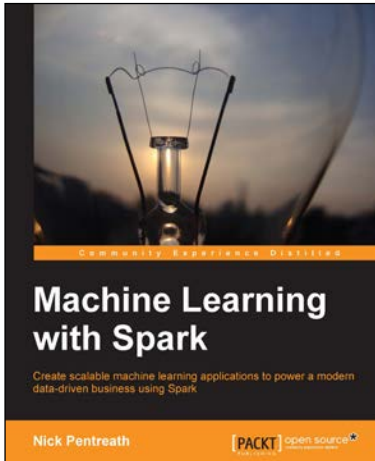
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



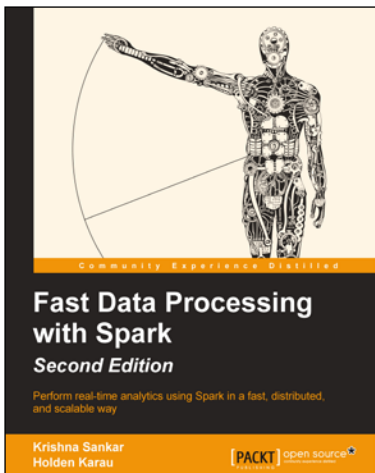
Machine Learning with Spark

ISBN: 978-1-78328-851-9

Paperback: 338 pages

Create scalable machine learning applications to power a modern data-driven business using Spark

1. A practical tutorial with real-world use cases allowing you to develop your own machine learning systems with Spark.
2. Combine various techniques and models into an intelligent machine learning system.
3. Use Spark's powerful tools to load, analyze, clean, and transform your data.



Fast Data Processing with Spark

Second Edition

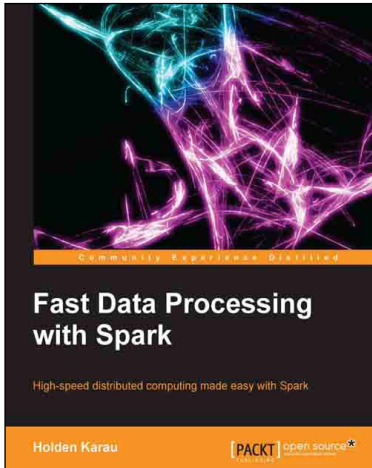
ISBN: 978-1-78439-257-4

Paperback: 184 pages

Perform real-time analytics using Spark in a fast, distributed, and scalable way

1. Develop a machine learning system with Spark's MLlib and scalable algorithms.
2. Deploy Spark jobs to various clusters such as Mesos, EC2, Chef, YARN, EMR, and so on.
3. This is a step-by-step tutorial that unleashes the power of Spark and its latest features.

Please check www.PacktPub.com for information on our titles



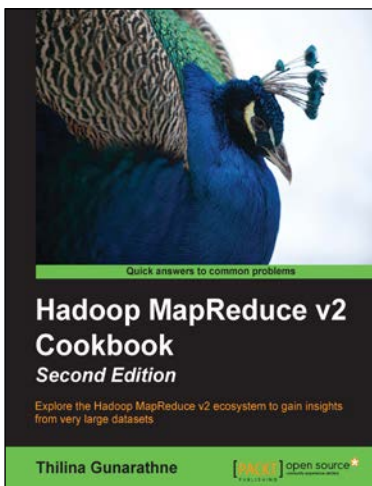
Fast Data Processing with Spark

ISBN: 978-1-78216-706-8

Paperback: 120 pages

High-speed distributed computing made easy with Spark

1. Implement Spark's interactive shell to prototype distributed applications.
2. Deploy Spark jobs to various clusters such as Mesos, EC2, Chef, YARN, EMR, and so on.
3. Use Shark's SQL query-like syntax with Spark.



Hadoop MapReduce v2 Cookbook *Second Edition*

ISBN: 978-1-78328-547-1

Paperback: 322 pages

Explore the Hadoop MapReduce v2 ecosystem to gain insights from very large datasets

1. Process large and complex datasets using next generation Hadoop.
2. Install, configure, and administer MapReduce programs and learn what's new in MapReduce v2.
3. More than 90 Hadoop MapReduce recipes presented in a simple and straightforward manner, with step-by-step instructions and real-world examples.

Please check www.PacktPub.com for information on our titles