Distributed Computing with Apache Spark

A Big Data Approach

Bhim Upadhyaya

Contents

Contents i

I Framework Exploration 2

1 Installation and Getting Started 3

1.1 Downloading, Installing, and Configuring . . . 4

1.1.1 A Note on Data Configuration . . . 5

1.2 Scala Shell . . . 6

1.3 Python Shell . . . 8

1.4 R Shell . . . 8

1.5 Spark Standalone Application . . . 9

1.6 Summary . . . 12

2 Spark Components 13

2.1 Core . . . 13

2.2 SQL . . . 13

2.3 Streaming . . . 13

2.4 MLlib . . . 13

2.5 GraphX . . . 14

2.6 Summary . . . 14

3 Data Input and Output 15

3.1 File Formats . . . 15

3.1.1 Text . . . 15
3.1.2 JSON . . . 15
3.1.3 CSV and TSV . . . 15
3.1.4 Sequence Files . . . 15
3.1.5 Hadoop I/O Formats . . . 15
3.2 File Compression . . . 15
3.3 File Systems . . . 16
3.3.1 Local . . . 16
3.3.2 HDFS . . . 16
3.3.3 Amazon S3 . . . 16
3.4 Summary . . . 16

4 Data Sources **17**
4.1 JDBC . . . 17
4.2 Cassandra . . . 17
4.3 HBase . . . 17
4.4 Elasticsearch . . . 17
4.5 Summary . . . 18

5 Spark Core **19**
5.1 High Level Architecture . . . 19
5.1.1 Driver Programs . . . 19
5.1.2 Cluster Managers . . . 19
5.1.3 Workers . . . 19
5.1.4 Executors . . . 19
5.1.5 Tasks . . . 19
5.2 RDD . . . 20
5.2.1 Creating RDDs . . . 20
5.2.2 RDD Operations . . . 20
5.3 Caching . . . 20

5.3.1 Caching Methods 20
5.3.2 Cache Memory Management 20
5.4 Shared Variables 20
5.5 Datasets 20
5.6 DataFrames 21
5.7 Summary 21

6 Spark SQL **22**
6.1 Purpose 22
6.2 Linking with Other Spark Libraries 22
6.3 Using Spark SQL 22
6.3.1 Initialization 22
6.3.2 Sample Spark SQL 22
6.3.3 DataFrames 22
6.3.4 Caching 22
6.4 Loading and Saving Data 23
6.4.1 RDDs 23
6.4.2 JSON 23
6.4.3 Parquet 23
6.4.4 Apache Hive 23
6.5 JDBC / ODBC Connection 23
6.6 User Defined Functions 23
6.7 Performance Tuning 23
6.8 Summary 23

7 Spark Streaming **24**
7.1 Architecture 24
7.2 Sample Application 24
7.3 Transformations 24
7.4 Sources 24

7.5 Highly Available Systems . . . 25
7.6 Performance Tuning . . . 25
7.7 ??? . . . 25
7.8 Summary . . . 25

8 MLlib **26**
8.1 Introduction . . . 26
8.2 Sample Application . . . 26
8.3 Selected ML Algorithms . . . 26
8.3.1 Liner Regression . . . 26
8.3.2 Clustering . . . 26
8.3.3 Collaborative Filtering and Recommendation . . . 27
8.4 Performance Tuning . . . 27
8.5 Summary . . . 27

9 GraphX **28**
9.1 Introduction . . . 28
9.2 Sample Graph Application . . . 28
9.3 Graph Operators . . . 28
9.4 Graph Builders . . . 28
9.5 RDDs . . . 29
9.6 Optimizations . . . 29
9.7 Graph Algorithms . . . 29
9.7.1 Page Rank . . . 29
9.7.2 Connected Components . . . 29
9.7.3 Triangle Counting . . . 29
9.8 Summary . . . 29

II Distributed Computing 30

10 Cluster Managers 31

10.1 Standalone Cluster Manager . . . 31
10.2 Apache Mesos . . . 31
10.3 Yarn . . . 31
10.4 Summary . . . 31

11 Cluster Setup 32

11.1 Installing Operating System . . . 32
11.2 Installing Spark . . . 32
11.3 Cluster Configuration . . . 32
11.4 Summary . . . 32

12 Deploying and Running an Application on a Cluster 33

12.1 Spark Runtime Components . . . 33
12.1.1 The Driver . . . 33
12.1.2 Cluster Manager . . . 33
12.1.3 Executors . . . 33
12.1.4 Launching the Application . . . 33
12.2 Packaging and Dependencies Management . . . 33
12.3 Application Deployment . . . 34
12.4 Scheduling . . . 34
12.5 Summary . . . 34

13 Monitoring, Tuning, and Debugging 35

13.1 Monitoring . . . 35
13.1.1 Standalone Cluster . . . 35
13.1.2 Jobs Monitoring . . . 35
13.1.3 Tasks Monitoring . . . 35
13.1.4 Stages Monitoring . . . 35

13.1.5 RDD Storage Monitoring . . . 35
13.1.6 Environment Monitoring . . . 35
13.2 Finding Information . . . 36
13.2.1 Web UI . . . 36
13.2.2 Driver and Executor Logs . . . 36
13.3 Performance . . . 36
13.3.1 Level of Parallelism . . . 36
13.3.2 Serialization Format . . . 36
13.3.3 Memory Management . . . 36
13.3.4 Hardware Provisioning . . . 36
13.4 Summary . . . 36

III Applications **37**

14 Clustering Application **38**
14.1 Problem Definition . . . 38
14.2 Solution Design . . . 38
14.3 Implementation . . . 38
14.4 Testing and Verification . . . 38
14.5 Summary . . . 39

15 Dashboard **40**
15.1 Problem Definition . . . 40
15.2 Solution Design . . . 40
15.3 Implementation . . . 40
15.4 Testing and Verification . . . 40
15.5 Summary . . . 41

16 Recommendation Engine **42**
16.1 Problem Definition . . . 42

16.2 Solution Design 42
16.3 Implementation 42
16.4 Testing and Verification 42
16.5 Summary 43

17 NLP Application **44**
17.1 Problem Definition 44
17.2 Solution Design 44
17.3 Implementation 44
17.4 Testing and Verification 44
17.5 Summary 45

18 GrahhX Application **46**
18.1 Problem Definition 46
18.2 Solution Design 46
18.3 Implementation 46
18.4 Testing and Verification 46
18.5 Summary 47

List of Figures

1.1 Sample Big Data from *kaggle* 6
1.2 Scala Shell 8
1.3 Line Count Standalone Application 10
1.4 SBT File 11
1.5 Spark Submit for Line Count Application 11

List of Tables

Part I

Framework Exploration

Chapter 1

Installation and Getting Started

Spark is one of the most popular cluster computing frameworks available today, specially designed for big data processing. Also it is one of the fastest evolving open source frameworks with a large community of developers. In this context, it maintains many versions for compatibility and maintainability. While spark was initially developed for distributed processing of big data, there has been many requests for making it equally useful for single machine and traditional web applications. One of the discussions is available online at https://www.youtube.com/watch?v=hEJtwezHjk8.

In this chapter, we demonstrate a typical process of installing and running spark on a single node. After successful installation, we will show how to start and work with both spark shell and python shell. Also we will create a standalone Spark application. A convenient single node can be our local machine for framework exploration purpose because we have full control of the machine. Specially, configuration changes and tuning is most sought after skills in the industry as it is difficult to achieve in the distributed environments. Our local installation will allow us to make any changes that we would like to experiment with.

Our observation is that some of the richest companies in the world spent millions of dollars to setup Spark clusters but could not take advantage of that because of tuning. In this kind of contexts, we will take a slightly different approach than other contemporary books. We will first play with the framework in a single machine and get a good grounding. This will prepare us for faster troubleshooting in a clustered environment.

1.1 Downloading, Installing, and Configuring

Spark installation can be done in two ways—using source code and using pre-compiled version. If we plan to create a custom version then source code is the option. If we choose this option, we can modify Spark source code to meet our specific needs. There are some responsibilities associated with this flexibility. The first one is that we are maintaining our own version, which probably is not a big deal. The second one, however, is not very convenient. We are required to synchronize our code if we want to upgrade or downgrade our Spark version. Having mentioned that, there are many organizations which maintain their own version of Spark source code.

At the time of writing, the source code can be built using Maven. There is a pom.xml, almost 3000 lines, in the root directory of the source code. We should be able to get deployable by following a typical maven build steps. Here, we will use pre-compiled version. The latest available version as of April, 2018 is 2.3. The binary can be downloaded from https://spark.apache.org/downloads.html. The default selection picks the latest version. For us, it is Spark 2.3.0 built for Apache Hadoop 2.7 and later. We can pick the right combination by selecting appropriate items from the drop down options provided.

Once downloaded, we can move it to a preferred location. We will be taking a minimum system change approach in the sense that we modify minimum number of parameters in our operating system configuration files. In a Unix type of operating systems, *.tgz* file can be extracted using the following command:

```
tar -xf spark-2.3.0-bin-hadoop2.7.tgz
```

A *.tgz* file is a tar archieve file that was compressed using GNU zip (gzip). If we are using a Mac OS then it can simply be extracted by double clicking. But it is a good practice to try unix commands since most of the servers do not provide GUI. Once decompressed, the first level of folder structure looks like the tree shown below. We will be re-visiting these directories later in other chapters as we explore more on Spark. To get started, the *bin* folder houses executable files, including the ones to start Spark shell, Python shell, and R shell; R shell is relatively a new feature. The *conf* contains configuration files, including logging configuration. Similarly, Scala, Java, Python, and R examples are available in the *examples* folder.

```
|--LICENSE
|--NOTICE
|--R
|--README.md
```

```
|--RELEASE
|--bin
|--conf
|--data
|--examples
|--jars
|--kubernetes
|--licenses
|--python
|--sbin
|--yarn
```

In order to run shells (Spark, Python, or R), no additional configuration is required. However, if we want to change log levels, which seems to be a common need as some stage, we can do that by renaming *log4j.properties.template* to *log4j.properties* and changing the log levels appropriately. For example, we see logs printed on the console when we start shells. The default log level set for *log4j.rootCategory* in the template is *INFO*, which prints detailed information. If we don't want to see all those details, we can change it to lower log level like *WARN*. Next time we start a shell, we see only *WARN* level messages. The *log4j* template template and other configuration templates are available in the *conf* folder.

1.1.1 A Note on Data Configuration

For the purpose of exploring the framework, we can use the sample data that are available with the Spark framework. We will use those data whenever we can take advantage. In a local environment, as long as the correct path is provided, no additional configuration is required. However, data loading becomes a significant exercise when we deal with clustered environment. We will deal with distributed data loading later in the respective chapters. For now, we will take two types of data in terms of size. The first type is the small data that comes with the framework itself. These files are handy to explore framework capabilities in terms of varieties of data. For the second type, we will take real word big data from kaggle, available online at https://www.kaggle.com/competitions. At the time of writing, *kaggle* is the world's largest community for big data challenges.

Figure 1.1 shows top 4 competitions in *kaggle* based on prize value. This shows the importance of big data analytics. In the following sections, we will use some of sample data from *kaggle*.

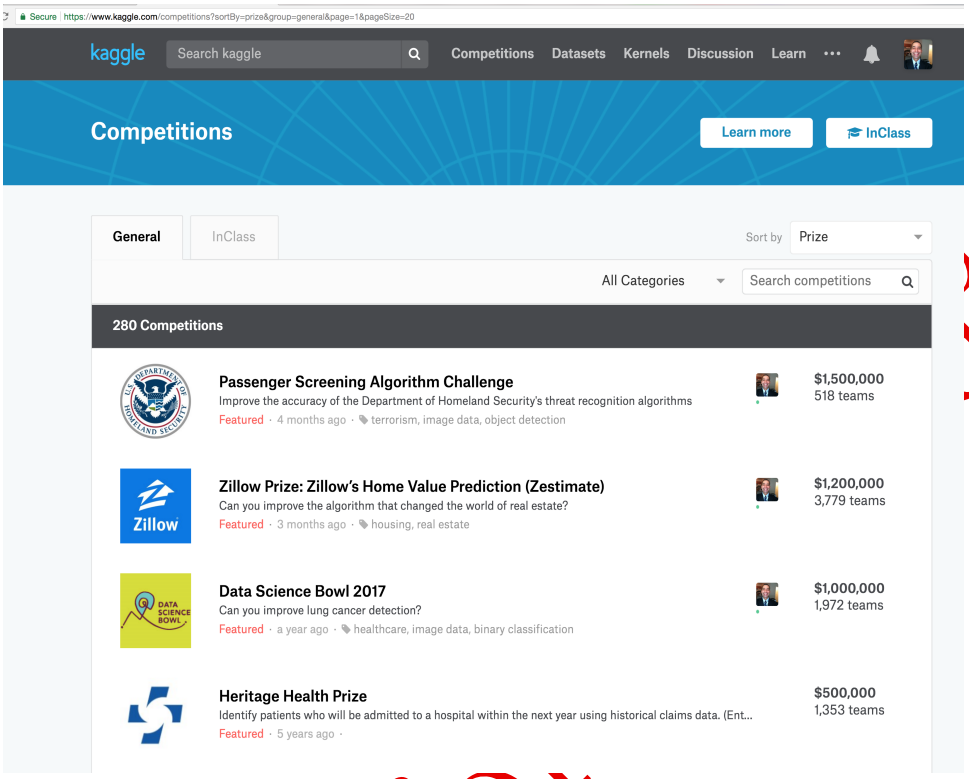

Figure 1.1: Sample Big Data from *kaggle*

1.2 Scala Shell

A Scala shell is an interactive REPL-like (Read-Evaluate-Print-Loop) shell that allows users to perform ad hoc data analysis using Scala syntax. The major difference between Scala REPL and Spark's Scala shell is that Scala shell can be used to perform cluster computing. The same set of commands that are used in a single machine shell can also be used in the cluster mode. This allows users to test program with more predictable computing environment and migrate the same code to large scale computing without modifying the code. This is certainly a big advantage from programming point of view.

Scala shell can be started by running the command *bin/spark-shell*. Figure 1.2 shows a typical screen shot for Scala shell. Let us note the following items from the screen shot.

- *spark*: Spark session is available as *spark*.
- *sc*: Spark context is available as *sc*.

- *Web UI*: Web UI is available at http://10.0.0.10:4040.
- *mode*: The mode is local.
- *Spark version*: The spark version used is 2.3.0.
- *Java version*: The Java version used is 1.8.0_25.
- *Scala version*: The Scala version used is 2.11.8.

The following code snippet demonstrates a typical use of Scala shell to count the number of lines in a file. We make use of data available in *kaggle*. This particular example uses *yelp* user data; the file is 1.36 GB in size. The line count is 1,326,101. In the code snipped, *sc* denotes Spark context. The line of code (LOC) creates an RDD from the text file, a CSV file in this case. The second LOC performs the count and returns value to the shell. Please observe the computation time for first iteration and subsequent iterations for *linesRDD.count()*.

```
scala> val linesRDD = sc.textFile("/Users/.../yelp-dataset/
 yelp_user.csv")
linesRDD: org.apache.spark.rdd.RDD[String] = /Users/.../
 yelp-dataset/yelp_review.csv MapPartitionsRDD[1]
 at textFile at <console>:24

scala> linesRDD.count()
res0: Long = 1326101

scala>
```

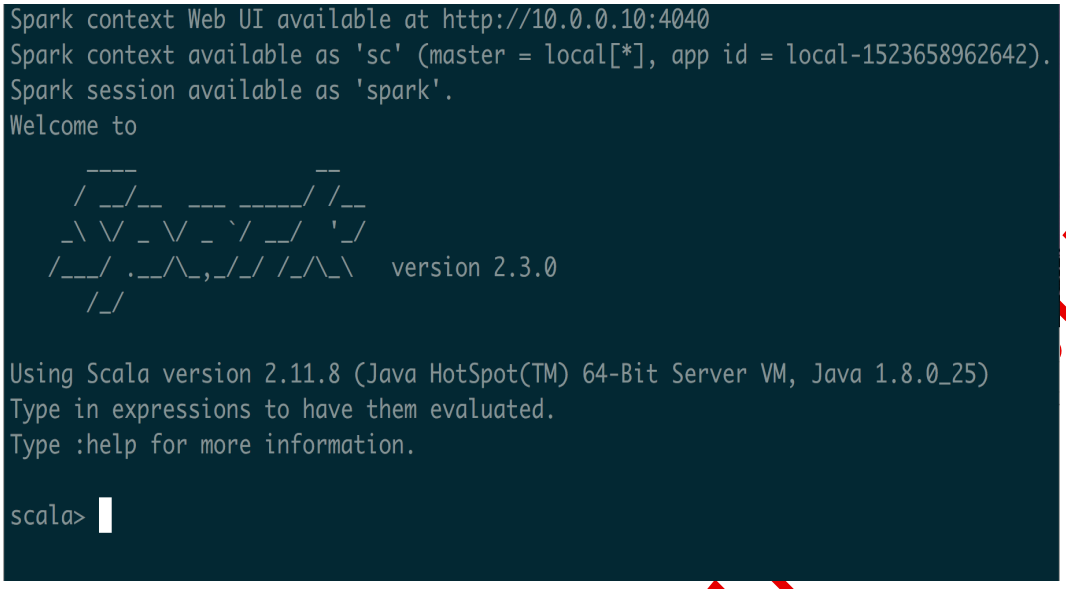

Figure 1.2: Scala Shell

1.3 Python Shell

A Python shell is similar to Scala shell except it accepts Python syntax and it can be started by running the command *bin/pyspark*. In this way, Python developers can use Python syntax and use Spark's scalability features without requiring them to learn new syntax. The Spark session is available as *spark*; similarly, Spark context is available as *sc*. The Spark version as well as Python version used are displayed. The following code snipped reads a file name *yelp_user.csv* from the specified location and counts number of lines. This is exactly the same file that we used earlier for Scala shell. Please note the count, it is exactly same as expected.

```
>>> linesRDD = sc.textFile("/Users/.../yelp-dataset/yelp_user.csv")
>>> linesRDD.count()
1326101
>>>
```

1.4 R Shell

R shell allows R programmers to leverage Spark's scalability without additional syntactical burden. It can be started by running the command *bin/sparkR*. The code snippet below demonstrates a typical use of R Shell. We use *yelp_user.csv*; it

was the same file used for Scala shell and Python shell. With *header* = *true*, the count is 1326100; this means it is 1326101 if header is counted.

```
> userDF = read.df("/Users/.../yelp-dataset/yelp_user.csv",
 source="csv", header="true", inferSchema="true")

> count(userDF)

[1] 1326100
```

Let's try to count by setting *header* to *false*. Now, the count is 136101, as shown in the code snippet below. This count is exactly same to that of the Scala shell and the Python shell in earlier sections.

```
> userDF = read.df("/Users/.../yelp-dataset/yelp_user.csv",
 source="csv", header="false", inferSchema="true")

> count(userDF)

[1] 1326101
```

1.5 Spark Standalone Application

In earlier sections, we made use of Spark's shells. Shells are convenient way of programming and getting results and are better for interactive programs or queries. Some computations run for long period of time, some computations have large outputs. If we try to return large output to Spark shell, it takes long time and the computational experience becomes less pleasant. Also it is more efficient to let long running computations continue and collect results later.

In this section, we will create a standalone program for the same computational problem that we dealt earlier. We will use the same file, same computation method and get the count. For that, we make use of an IDE, IntelliJ in this case; we can also use Eclipse. Figure 1.3 shows complete code for standalone application for the exercise that we did earlier. It takes the file name as an argument. Since it is a local mode printing the value on the console should be fine as we get to see the count.

Figure 1.4 lists corresponding SBT build file entries. *name* is the name of the application, *version* is appended in the jar file to recognize the version of the build for this application. Similarly, *scalaVersion* tells us which version of Scala

```
package com.equalinformation.spark.scala
import org.apache.spark._

object LineCount {
   def main(args: Array[String]): Unit = {
      val inputFileName = args(0)

      val conf = new SparkConf().setMaster("local").
       setAppName("LineCount")
      val sc = new SparkContext(conf)

      val userRDD = sc.textFile(inputFileName)
      val count = userRDD.count()

      println(count) // Printing is ok in local mode
   }
}
```

Figure 1.3: Line Count Standalone Application

should be available in the deployment environment. The last LOC provides dependencies information. Please note *provided*, which means the jars are provided in the deployment environment. This helps us to reduce the jar size of our application.

Now, the final part is to submit the jar using *spark-submit*. Figure 1.5 shows the syntax for spark submit for our line count standalone application. Please note the count, we got the same count that we got from Spark shells.

```
name := "standalone"

version := "1.0"

scalaVersion := "2.11.7"

libraryDependencies ++= Seq(
"org.apache.spark" %% "spark-core" % "2.3.0" % "provided"
)
```

Figure 1.4: SBT File

```
Bhims-iMac:spark-2.3.0-bin-hadoop2.7 bhim$ bin/spark-submit
 --class com.equalinformation.spark.scala.LineCount
 standalone_2.11-1.0.jar /Users/.../yelp-dataset/yelp_user.csv
1326101
Bhims-iMac:spark-2.3.0-bin-hadoop2.7 bhim$
```

Figure 1.5: Spark Submit for Line Count Application

1.6 Summary

In this Chapter, we started by discussing where to download Apache Spark from and what combination to use. Then we covered installation and configuration. Also we discussed a sample big data source, *kaggle*, which has well funded competitions. Next, we presented Scala shell and showed how to work with it. Similarly, we showed Python and R shells operations using the same data sample. We also demonstrated operation accuracy in all three shells. Finally, we demonstrated how to write a stand alone Spark application. Again, we took the same sample data and demonstrated a consistent result.

Chapter 2

Spark Components

(content here)

2.1 Core

(content here)

2.2 SQL

(content here)

2.3 Streaming

(content here)

2.4 MLlib

(content here)

2.5 GraphX

(content here)

2.6 Summary

(content here)

Chapter 3

Data Input and Output

(content here)

3.1 File Formats

3.1.1 Text

3.1.2 JSON

3.1.3 CSV and TSV

3.1.4 Sequence Files

3.1.5 Hadoop I/O Formats

(content here)

3.2 File Compression

(content here)

3.3 File Systems

3.3.1 Local

3.3.2 HDFS

3.3.3 Amazon S3

(content here)

3.4 Summary

(content here)

Chapter 4

Data Sources

(content here)

4.1 JDBC

(content here)

4.2 Cassandra

(content here)

4.3 HBase

(content here)

4.4 Elasticsearch

(content here)

4.5 Summary

(content here)

Chapter 5

Spark Core

(content here)

5.1 High Level Architecture

5.1.1 Driver Programs

5.1.2 Cluster Managers

5.1.3 Workers

5.1.4 Executors

5.1.5 Tasks

(content here)

5.2 RDD

5.2.1 Creating RDDs

5.2.2 RDD Operations

Transformation

Actions

Lazy Evaluation

(content here)

5.3 Caching

5.3.1 Caching Methods

5.3.2 Cache Memory Management

(content here)

5.4 Shared Variables

Broadcast Variables

Accumulators

(content here)

5.5 Datasets

(content here)

5.6 DataFrames

(content here)

5.7 Summary

(content here)

Chapter 6

Spark SQL

(content here)

6.1 Purpose

(content here)

6.2 Linking with Other Spark Libraries

(content here)

6.3 Using Spark SQL

6.3.1 Initialization

6.3.2 Sample Spark SQL

6.3.3 DataFrames

6.3.4 Caching

(content here)

6.4 Loading and Saving Data

6.4.1 RDDs

6.4.2 JSON

6.4.3 Parquet

6.4.4 Apache Hive

(content here)

6.5 JDBC / ODBC Connection

(content here)

6.6 User Defined Functions

(content here)

6.7 Performance Tuning

(content here)

6.8 Summary

(content here)

Chapter 7

Spark Streaming

(content here)

7.1 Architecture

(content here)

7.2 Sample Application

(content here)

7.3 Transformations

(content here)

7.4 Sources

(content here)

7.5 Highly Available Systems

(content here)

7.6 Performance Tuning

(content here)

7.7 ???

(content here)

7.8 Summary

(content here)

Chapter 8

MLlib

(content here)

8.1 Introduction

(content here)

8.2 Sample Application

(content here)

8.3 Selected ML Algorithms

8.3.1 Liner Regression

(content here)

8.3.2 Clustering

(content here)

8.3.3 Collaborative Filtering and Recommendation

(content here)

(content here)

8.4 Performance Tuning

(content here)

8.5 Summary

(content here)

Chapter 9

GraphX

(content here)

9.1 Introduction

(content here)

9.2 Sample Graph Application

(content here)

9.3 Graph Operators

(content here)

9.4 Graph Builders

(content here)

9.5 RDDs

(content here)

9.6 Optimizations

(content here)

9.7 Graph Algorithms

9.7.1 Page Rank

(content here)

9.7.2 Connected Components

(content here)

9.7.3 Triangle Counting

(content here)

(content here)

9.8 Summary

(content here)

Part II

Distributed Computing

Chapter 10

Cluster Managers

(content here)

10.1 Standalone Cluster Manager

(content here)

10.2 Apache Mesos

(content here)

10.3 Yarn

(content here)

10.4 Summary

(content here)

Chapter 11

Cluster Setup

(content here)

11.1 Installing Operating System

(content here)

11.2 Installing Spark

(content here)

11.3 Cluster Configuration

(content here)

11.4 Summary

(content here)

Chapter 12

Deploying and Running an Application on a Cluster

(content here)

12.1 Spark Runtime Components

12.1.1 The Driver

12.1.2 Cluster Manager

12.1.3 Executors

12.1.4 Launching the Application

(content here)

12.2 Packaging and Dependencies Management

(content here)

12.3 Application Deployment

(content here)

12.4 Scheduling

(content here)

12.5 Summary

(content here)

Chapter 13

Monitoring, Tuning, and Debugging

(content here)

13.1 Monitoring

13.1.1 Standalone Cluster

Monitoring a Spark Master

Monitoring a Spark Worker

13.1.2 Jobs Monitoring

13.1.3 Tasks Monitoring

13.1.4 Stages Monitoring

13.1.5 RDD Storage Monitoring

13.1.6 Environment Monitoring

(content here)

13.2 Finding Information

13.2.1 Web UI

13.2.2 Driver and Executor Logs

(content here)

13.3 Performance

13.3.1 Level of Parallelism

13.3.2 Serialization Format

13.3.3 Memory Management

13.3.4 Hardware Provisioning

(content here)

13.4 Summary

(content here)

Part III

Applications

Chapter 14

Clustering Application

(content here)

14.1 Problem Definition

(content here)

14.2 Solution Design

(content here)

14.3 Implementation

(content here)

14.4 Testing and Verification

(content here)

14.5 Summary

(content here)

Chapter 15

Dashboard

(content here)

15.1 Problem Definition

(content here)

15.2 Solution Design

(content here)

15.3 Implementation

(content here)

15.4 Testing and Verification

(content here)

15.5 Summary

(content here)

Chapter 16

Recommendation Engine

(content here)

16.1 Problem Definition

(content here)

16.2 Solution Design

(content here)

16.3 Implementation

(content here)

16.4 Testing and Verification

(content here)

16.5 Summary

(content here)

Chapter 17

NLP Application

(content here)

17.1 Problem Definition

(content here)

17.2 Solution Design

(content here)

17.3 Implementation

(content here)

17.4 Testing and Verification

(content here)

17.5 Summary

(content here)

Chapter 18

GrahhX Application

(content here)

18.1 Problem Definition

(content here)

18.2 Solution Design

(content here)

18.3 Implementation

(content here)

18.4 Testing and Verification

(content here)

18.5 Summary

(content here)

Index

cluster computing, 3

preface, 1