Looking for a cluster computing system that provides high-level APIs? Apache Spark is your answer—an open source, fast, and general purpose cluster computing system. Are you a Python developer inclined to work with the Spark engine? If so, this book will be your companion as you create a data-intensive app using Spark as a processing engine, Python visualization libraries, and web frameworks such as Flask. To begin with, you will learn the most effective way to install the Python development environment powered by Spark, Blaze, and Bokeh. You will then find out how to connect with data stores such as MySQL, MongoDB, Cassandra, and Hadoop. You'll expand your skills throughout, becoming familiar with the various data sources (Github, Twitter, Meetup, and blogs), their data structures, and solutions to effectively tackle complexities. By the end of the book, you will have created a real-time and insightful trend tracker data-intensive app with Spark.

Who this book is written for
This book is for data scientists and software developers with a focus on Python who want to work with the Spark engine, and it will also benefit Enterprise architects. All you need to have is a good background in Python and an inclination to work with Spark.

What you will learn from this book
- Create a Python development environment powered by Spark (PySpark), Blaze, and Bokeh
- Build a real-time trend tracker data-intensive app
- Visualize the trends and insights gained from data using Bokeh
- Generate insights from data using machine learning through Spark MLLIB
- Juggle data using Blaze
- Create training data sets and train Machine Learning models
- Test machine learning models on test datasets
- Deploy machine learning algorithms and models and scale them for real-time events
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Setting Up a Spark Virtual Environment'
- A synopsis of the book’s content
- More information on Spark for Python Developers
Amit Nandi studied physics at the Free University of Brussels in Belgium, where he did his research on computer generated holograms. Computer generated holograms are the key components of an optical computer, which is powered by photons running at the speed of light. He then worked with the university Cray supercomputer, sending batch jobs of programs written in Fortran. This gave him a taste for computing, which kept growing. He has worked extensively on large business reengineering initiatives, using SAP as the main enabler. He focused for the last 15 years on start-ups in the data space, pioneering new areas of the information technology landscape. He is currently focusing on large-scale data-intensive applications as an enterprise architect, data engineer, and software developer. He understands and speaks seven human languages. Although Python is his computer language of choice, he aims to be able to write fluently in seven computer languages too.
Spark for Python Developers aims to combine the elegance and flexibility of Python with the power and versatility of Apache Spark. Spark is written in Scala and runs on the Java virtual machine. It is nevertheless polyglot and offers bindings and APIs for Java, Scala, Python, and R. Python is a well-designed language with an extensive set of specialized libraries. This book looks at PySpark within the PyData ecosystem. Some of the prominent PyData libraries include Pandas, Blaze, Scikit-Learn, Matplotlib, Seaborn, and Bokeh. These libraries are open source. They are developed, used, and maintained by the data scientist and Python developers community. PySpark integrates well with the PyData ecosystem, as endorsed by the Anaconda Python distribution. The book puts forward a journey to build data-intensive apps along with an architectural blueprint that covers the following steps: first, set up the base infrastructure with Spark. Second, acquire, collect, process, and store the data. Third, gain insights from the collected data. Fourth, stream live data and process it in real time. Finally, visualize the information.

The objective of the book is to learn about PySpark and PyData libraries by building apps that analyze the Spark community’s interactions on social networks. The focus is on Twitter data.

What this book covers

Chapter 1, Setting Up a Spark Virtual Environment, covers how to create a segregated virtual machine as our sandbox or development environment to experiment with Spark and PyData libraries. It covers how to install Spark and the Python Anaconda distribution, which includes PyData libraries. Along the way, we explain the key Spark concepts, the Python Anaconda ecosystem, and build a Spark word count app.
Chapter 2, Building Batch and Streaming Apps with Spark, lays the foundation of the Data Intensive Apps Architecture. It describes the five layers of the apps architecture blueprint: infrastructure, persistence, integration, analytics, and engagement. We establish API connections with three social networks: Twitter, GitHub, and Meetup. This chapter provides the tools to connect to these three nontrivial APIs so that you can create your own data mashups at a later stage.

Chapter 3, Juggling Data with Spark, covers how to harvest data from Twitter and process it using Pandas, Blaze, and SparkSQL with their respective implementations of the dataframe data structure. We proceed with further investigations and techniques using Spark SQL, leveraging on the Spark dataframe data structure.

Chapter 4, Learning from Data Using Spark, gives an overview of the ever expanding library of algorithms of Spark MLlib. It covers supervised and unsupervised learning, recommender systems, optimization, and feature extraction algorithms. We put the Twitter harvested dataset through a Python Scikit-Learn and Spark MLlib K-means clustering in order to segregate the Apache Spark relevant tweets.

Chapter 5, Streaming Live Data with Spark, lays down the foundation of streaming architecture apps and describes their challenges, constraints, and benefits. We illustrate the streaming concepts with TCP sockets, followed by live tweet ingestion and processing directly from the Twitter firehose. We also describe Flume, a reliable, flexible, and scalable data ingestion and transport pipeline system. The combination of Flume, Kafka, and Spark delivers unparalleled robustness, speed, and agility in an ever-changing landscape. We end the chapter with some remarks and observations on two streaming architectural paradigms, the Lambda and Kappa architectures.

Chapter 6, Visualizing Insights and Trends, focuses on a few key visualization techniques. It covers how to build word clouds and expose their intuitive power to reveal a lot of the key words, moods, and memes carried through thousands of tweets. We then focus on interactive mapping visualizations using Bokeh. We build a world map from the ground up and create a scatter plot of critical tweets. Our final visualization is to overlay an actual Google map of London, highlighting upcoming meetups and their respective topics.
Setting Up a Spark Virtual Environment

In this chapter, we will build an isolated virtual environment for development purposes. The environment will be powered by Spark and the PyData libraries provided by the Python Anaconda distribution. These libraries include Pandas, Scikit-Learn, Blaze, Matplotlib, Seaborn, and Bokeh. We will perform the following activities:

- Setting up the development environment using the Anaconda Python distribution. This will include enabling the IPython Notebook environment powered by PySpark for our data exploration tasks.
- Installing and enabling Spark, and the PyData libraries such as Pandas, Scikit-Learn, Blaze, Matplotlib, and Bokeh.
- Building a word count example app to ensure that everything is working fine.

The last decade has seen the rise and dominance of data-driven behemoths such as Amazon, Google, Twitter, LinkedIn, and Facebook. These corporations, by seeding, sharing, or disclosing their infrastructure concepts, software practices, and data processing frameworks, have fostered a vibrant open source software community. This has transformed the enterprise technology, systems, and software architecture. This includes new infrastructure and DevOps (short for development and operations), concepts leveraging virtualization, cloud technology, and software-defined networks.
To process petabytes of data, Hadoop was developed and open sourced, taking its inspiration from the Google File System (GFS) and the adjoining distributed computing framework, MapReduce. Overcoming the complexities of scaling while keeping costs under control has also led to a proliferation of new data stores. Examples of recent database technology include Cassandra, a columnar database; MongoDB, a document database; and Neo4J, a graph database.

Hadoop, thanks to its ability to process huge datasets, has fostered a vast ecosystem to query data more iteratively and interactively with Pig, Hive, Impala, and Tez. Hadoop is cumbersome as it operates only in batch mode using MapReduce. Spark is creating a revolution in the analytics and data processing realm by targeting the shortcomings of disk input-output and bandwidth-intensive MapReduce jobs.

Spark is written in Scala, and therefore integrates natively with the Java Virtual Machine (JVM) powered ecosystem. Spark had early on provided Python API and bindings by enabling PySpark. The Spark architecture and ecosystem is inherently polyglot, with an obvious strong presence of Java-led systems.

This book will focus on PySpark and the PyData ecosystem. Python is one of the preferred languages in the academic and scientific community for data-intensive processing. Python has developed a rich ecosystem of libraries and tools in data manipulation with Pandas and Blaze, in Machine Learning with Scikit-Learn, and in data visualization with Matplotlib, Seaborn, and Bokeh. Hence, the aim of this book is to build an end-to-end architecture for data-intensive applications powered by Spark and Python. In order to put these concepts in to practice, we will analyze social networks such as Twitter, GitHub, and Meetup. We will focus on the activities and social interactions of Spark and the Open Source Software community by tapping into GitHub, Twitter, and Meetup.

Building data-intensive applications requires highly scalable infrastructure, polyglot storage, seamless data integration, multiparadigm analytics processing, and efficient visualization. The following paragraph describes the data-intensive app architecture blueprint that we will adopt throughout the book. It is the backbone of the book. We will discover Spark in the context of the broader PyData ecosystem.

---

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Understanding the architecture of data-intensive applications

In order to understand the architecture of data-intensive applications, the following conceptual framework is used. The architecture is designed on the following five layers:

- Infrastructure layer
- Persistence layer
- Integration layer
- Analytics layer
- Engagement layer

The following screenshot depicts the five layers of the **Data Intensive App Framework**:

<table>
<thead>
<tr>
<th>Data Intensive App Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engagement Layer</td>
</tr>
<tr>
<td>Visualization (Charts, Time Series, Maps...)</td>
</tr>
<tr>
<td>Analytics Layer</td>
</tr>
<tr>
<td>Exploration</td>
</tr>
<tr>
<td>Integration Layer</td>
</tr>
<tr>
<td>Connect – Collect – Correct – Compose – Consume – Control</td>
</tr>
<tr>
<td>Persistence Layer</td>
</tr>
<tr>
<td>RDBMS</td>
</tr>
<tr>
<td>Infrastructure Layer</td>
</tr>
<tr>
<td>Virtualization</td>
</tr>
</tbody>
</table>

From the bottom up, let's go through the layers and their main purpose.
Setting Up a Spark Virtual Environment

Infrastructure layer
The infrastructure layer is primarily concerned with virtualization, scalability, and continuous integration. In practical terms, and in terms of virtualization, we will go through building our own development environment in a VirtualBox and virtual machine powered by Spark and the Anaconda distribution of Python. If we wish to scale from there, we can create a similar environment in the cloud. The practice of creating a segregated development environment and moving into test and production deployment can be automated and can be part of a continuous integration cycle powered by DevOps tools such as Vagrant, Chef, Puppet, and Docker. Docker is a very popular open source project that eases the installation and deployment of new environments. The book will be limited to building the virtual machine using VirtualBox. From a data-intensive app architecture point of view, we are describing the essential steps of the infrastructure layer by mentioning scalability and continuous integration beyond just virtualization.

Persistence layer
The persistence layer manages the various repositories in accordance with data needs and shapes. It ensures the set up and management of the polyglot data stores. It includes relational database management systems such as MySQL and PostgreSQL; key-value data stores such as Hadoop, Riak, and Redis; columnar databases such as HBase and Cassandra; document databases such as MongoDB and Couchbase; and graph databases such as Neo4j. The persistence layer manages various file systems such as Hadoop's HDFS. It interacts with various storage systems from native hard drives to Amazon S3. It manages various file storage formats such as csv, json, and parquet, which is a column-oriented format.

Integration layer
The integration layer focuses on data acquisition, transformation, quality, persistence, consumption, and governance. It is essentially driven by the following five Cs: connect, collect, correct, compose, and consume.

The five steps describe the lifecycle of data. They are focused on how to acquire the dataset of interest, explore it, iteratively refine and enrich the collected information, and get it ready for consumption. So, the steps perform the following operations:

- **Connect**: Targets the best way to acquire data from the various data sources, APIs offered by these sources, the input format, input schemas if they exist, the rate of data collection, and limitations from providers
Chapter 1

- **Correct**: Focuses on transforming data for further processing and also ensures that the quality and consistency of the data received are maintained
- **Collect**: Looks at which data to store where and in what format, to ease data composition and consumption at later stages
- **Compose**: Concentrates its attention on how to mash up the various data sets collected, and enrich the information in order to build a compelling data-driven product
- **Consume**: Takes care of data provisioning and rendering and how the right data reaches the right individual at the right time
- **Control**: This sixth *additional* step will sooner or later be required as the data, the organization, and the participants grow and it is about ensuring data governance

The following diagram depicts the iterative process of data acquisition and refinement for consumption:

![Data Integration Diagram](image)

**Analytics layer**

The analytics layer is where Spark processes data with the various models, algorithms, and machine learning pipelines in order to derive insights. For our purpose, in this book, the analytics layer is powered by Spark. We will delve deeper in subsequent chapters into the merits of Spark. In a nutshell, what makes it so powerful is that it allows multiple paradigms of analytics processing in a single unified platform. It allows batch, streaming, and interactive analytics. Batch processing on large datasets with longer latency periods allows us to extract patterns and insights that can feed into real-time events in streaming mode. Interactive and iterative analytics are more suited for data exploration. Spark offers bindings and APIs in Python and R. With its **SparkSQL** module and the Spark Dataframe, it offers a very familiar analytics interface.
Engagement layer
The engagement layer interacts with the end user and provides dashboards, interactive visualizations, and alerts. We will focus here on the tools provided by the PyData ecosystem such as Matplotlib, Seaborn, and Bokeh.

Understanding Spark
Hadoop scales horizontally as the data grows. Hadoop runs on commodity hardware, so it is cost-effective. Intensive data applications are enabled by scalable, distributed processing frameworks that allow organizations to analyze petabytes of data on large commodity clusters. Hadoop is the first open source implementation of map-reduce. Hadoop relies on a distributed framework for storage called HDFS (Hadoop Distributed File System). Hadoop runs map-reduce tasks in batch jobs. Hadoop requires persisting the data to disk at each map, shuffle, and reduce process step. The overhead and the latency of such batch jobs adversely impact the performance.

Spark is a fast, distributed general analytics computing engine for large-scale data processing. The major breakthrough from Hadoop is that Spark allows data sharing between processing steps through in-memory processing of data pipelines.

Spark is unique in that it allows four different styles of data analysis and processing. Spark can be used in:

- **Batch**: This mode is used for manipulating large datasets, typically performing large map-reduce jobs
- **Streaming**: This mode is used to process incoming information in near real time
- **Iterative**: This mode is for machine learning algorithms such as a gradient descent where the data is accessed repetitively in order to reach convergence
- **Interactive**: This mode is used for data exploration as large chunks of data are in memory and due to the very quick response time of Spark

The following figure highlights the preceding four processing styles:
Spark operates in three modes: one single mode, standalone on a single machine and two distributed modes on a cluster of machines—on Yarn, the Hadoop distributed resource manager, or on Mesos, the open source cluster manager developed at Berkeley concurrently with Spark:

Spark offers a polyglot interface in Scala, Java, Python, and R.

**Spark libraries**

Spark comes with batteries included, with some powerful libraries:

- **Spark SQL**: This provides the SQL-like ability to interrogate structured data and interactively explore large datasets
- **SparkMLLIB**: This provides major algorithms and a pipeline framework for machine learning
- **Spark Streaming**: This is for near real-time analysis of data using micro batches and sliding widows on incoming streams of data
- **Spark GraphX**: This is for graph processing and computation on complex connected entities and relationships

**PySpark in action**

Spark is written in Scala. The whole Spark ecosystem naturally leverages the JVM environment and capitalizes on HDFS natively. Hadoop HDFS is one of the many data stores supported by Spark. Spark is agnostic and from the beginning interacted with multiple data sources, types, and formats.

PySpark is not a transcribed version of Spark on a Java-enabled dialect of Python such as Jython. PySpark provides integrated API bindings around Spark and enables full usage of the Python ecosystem within all the nodes of the cluster with the pickle Python serialization and, more importantly, supplies access to the rich ecosystem of Python’s machine learning libraries such as Scikit-Learn or data processing such as Pandas.
Setting Up a Spark Virtual Environment

When we initialize a Spark program, the first thing a Spark program must do is to create a SparkContext object. It tells Spark how to access the cluster. The Python program creates a PySparkContext. Py4J is the gateway that binds the Python program to the Spark JVM SparkContext. The JVM SparkContext serializes the application codes and the closures and sends them to the cluster for execution. The cluster manager allocates resources and schedules, and ships the closures to the Spark workers in the cluster who activate Python virtual machines as required. In each machine, the Spark Worker is managed by an executor that controls computation, storage, and cache.

Here’s an example of how the Spark driver manages both the PySpark context and the Spark context with its local filesystems and its interactions with the Spark worker through the cluster manager:

The Resilient Distributed Dataset

Spark applications consist of a driver program that runs the user’s main function, creates distributed datasets on the cluster, and executes various parallel operations (transformations and actions) on those datasets.

Spark applications are run as an independent set of processes, coordinated by a SparkContext in a driver program.

The SparkContext will be allocated system resources (machines, memory, CPU) from the Cluster manager.
The SparkContext manages executors who manage workers in the cluster. The driver program has Spark jobs that need to run. The jobs are split into tasks submitted to the executor for completion. The executor takes care of computation, storage, and caching in each machine.

The key building block in Spark is the RDD (Resilient Distributed Dataset). A dataset is a collection of elements. Distributed means the dataset can be on any node in the cluster. Resilient means that the dataset could get lost or partially lost without major harm to the computation in progress as Spark will re-compute from the data lineage in memory, also known as the DAG (short for Directed Acyclic Graph) of operations. Basically, Spark will snapshot in memory a state of the RDD in the cache. If one of the computing machines crashes during operation, Spark rebuilds the RDDs from the cached RDD and the DAG of operations. RDDs recover from node failure.

There are two types of operation on RDDs:

- **Transformations**: A transformation takes an existing RDD and leads to a pointer of a new transformed RDD. An RDD is immutable. Once created, it cannot be changed. Each transformation creates a new RDD. Transformations are lazily evaluated. Transformations are executed only when an action occurs. In the case of failure, the data lineage of transformations rebuilds the RDD.

- **Actions**: An action on an RDD triggers a Spark job and yields a value. An action operation causes Spark to execute the (lazy) transformation operations that are required to compute the RDD returned by the action. The action results in a DAG of operations. The DAG is compiled into stages where each stage is executed as a series of tasks. A task is a fundamental unit of work.

Here's some useful information on RDDs:

- RDDs are created from a data source such as an HDFS file or a DB query. There are three ways to create an RDD:
  - Reading from a datastore
  - Transforming an existing RDD
  - Using an in-memory collection

- RDDs are transformed with functions such as `map` or `filter`, which yield new RDDs.

- An action such as `first`, `take`, `collect`, or `count` on an RDD will deliver the results into the Spark driver. The Spark driver is the client through which the user interacts with the Spark cluster.
Understanding Anaconda

Anaconda is a widely used free Python distribution maintained by Continuum (https://www.continuum.io/). We will use the prevailing software stack provided by Anaconda to generate our apps. In this book, we will use PySpark and the PyData ecosystem. The PyData ecosystem is promoted, supported, and maintained by Continuum and powered by the Anaconda Python distribution. The Anaconda Python distribution essentially saves time and aggravation in the installation of the Python environment; we will use it in conjunction with Spark. Anaconda has its own package management that supplements the traditional pip install and easy-install. Anaconda comes with batteries included, namely some of the most important packages such as Pandas, Scikit-Learn, Blaze, Matplotlib, and Bokeh. An upgrade to any of the installed library is a simple command at the console:

$ conda update
A list of installed libraries in our environment can be obtained with command:

```
$ conda list
```

The key components of the stack are as follows:

- **Anaconda:** This is a free Python distribution with almost 200 Python packages for science, math, engineering, and data analysis.
- **Conda:** This is a package manager that takes care of all the dependencies of installing a complex software stack. This is not restricted to Python and manages the install process for R and other languages.
- **Numba:** This provides the power to speed up code in Python with high-performance functions and just-in-time compilation.
- **Blaze:** This enables large scale data analytics by offering a uniform and adaptable interface to access a variety of data providers, which include streaming Python, Pandas, SQLAlchemy, and Spark.
- **Bokeh:** This provides interactive data visualizations for large and streaming datasets.
- **Wakari:** This allows us to share and deploy IPython Notebooks and other apps on a hosted environment.

The following figure shows the components of the Anaconda stack:
Setting up the Spark powered environment

In this section, we will learn to set up Spark:

- Create a segregated development environment in a virtual machine running on Ubuntu 14.04, so it does not interfere with any existing system.
- Install Spark 1.3.0 with its dependencies, namely.
- Install the Anaconda Python 2.7 environment with all the required libraries such as Pandas, Scikit-Learn, Blaze, and Bokeh, and enable PySpark, so it can be accessed through IPython Notebooks.
- Set up the backend or data stores of our environment. We will use MySQL as the relational database, MongoDB as the document store, and Cassandra as the columnar database.

Each storage backend serves a specific purpose depending on the nature of the data to be handled. The MySQL RDBMs is used for standard tabular processed information that can be easily queried using SQL. As we will be processing a lot of JSON-type data from various APIs, the easiest way to store them is in a document. For real-time and time-series-related information, Cassandra is best suited as a columnar database.

The following diagram gives a view of the environment we will build and use throughout the book:
Chapter 1

Setting up an Oracle VirtualBox with Ubuntu

Setting up a clean new VirtualBox environment on Ubuntu 14.04 is the safest way to create a development environment that does not conflict with existing libraries and can be later replicated in the cloud using a similar list of commands.

In order to set up an environment with Anaconda and Spark, we will create a VirtualBox virtual machine running Ubuntu 14.04.

Let's go through the steps of using VirtualBox with Ubuntu:

1. Oracle VirtualBox VM is free and can be downloaded from https://www.virtualbox.org/wiki/Downloads. The installation is pretty straightforward.

2. After installing VirtualBox, let's open the Oracle VM VirtualBox Manager and click the New button.

3. We'll give the new VM a name, and select Type Linux and Version Ubuntu (64 bit).

4. You need to download the ISO from the Ubuntu website and allocate sufficient RAM (4 GB recommended) and disk space (20 GB recommended). We will use the Ubuntu 14.04.1 LTS release, which is found here: http://www.ubuntu.com/download/desktop.

5. Once the installation completed, it is advisable to install the VirtualBox Guest Additions by going to (from the VirtualBox menu, with the new VM running) Devices | Insert Guest Additions CD image. Failing to provide the guest additions in a Windows host gives a very limited user interface with reduced window sizes.

6. Once the additional installation completes, reboot the VM, and it will be ready to use. It is helpful to enable the shared clipboard by selecting the VM and clicking Settings, then go to General | Advanced | Shared Clipboard and click on Bidirectional.

Installing Anaconda with Python 2.7

PySpark currently runs only on Python 2.7. (There are requests from the community to upgrade to Python 3.3.) To install Anaconda, follow these steps:

1. Download the Anaconda Installer for Linux 64-bit Python 2.7 from http://continuum.io/downloads#all.
Setting Up a Spark Virtual Environment

2. After downloading the Anaconda installer, open a terminal and navigate to the directory or folder where the installer has been saved. From here, run the following command, replacing the `2.x.x` in the command with the version number of the downloaded installer file:

```bash
# install anaconda 2.x.x
bash Anaconda-2.x.x-Linux-x86[_64].sh
```

3. After accepting the license terms, you will be asked to specify the install location (which defaults to `~/anaconda`).

4. After the self-extraction is finished, you should add the anaconda binary directory to your PATH environment variable:

```bash
# add anaconda to PATH
bash Anaconda-2.x.x-Linux-x86[_64].sh
```

Installing Java 8

Spark runs on the JVM and requires the Java SDK (short for Software Development Kit) and not the JRE (short for Java Runtime Environment), as we will build apps with Spark. The recommended version is Java Version 7 or higher. Java 8 is the most suitable, as it includes many of the functional programming techniques available with Scala and Python.

To install Java 8, follow these steps:

1. Install Oracle Java 8 using the following commands:

```bash
# install oracle java 8
$ sudo apt-get install software-properties-common
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

2. Set the `JAVA_HOME` environment variable and ensure that the Java program is on your PATH.

3. Check that `JAVA_HOME` is properly installed:

```bash
# $ echo JAVA_HOME
```
Installing Spark


The Spark download page offers the possibility to download earlier versions of Spark and different package and download types. We will select the latest release, pre-built for Hadoop 2.6 and later. The easiest way to install Spark is to use a Spark package prebuilt for Hadoop 2.6 and later, rather than build it from source. Move the file to the directory ~/spark under the root directory.

Download the latest release of Spark—Spark 1.5.2, released on November 9, 2015:

1. Select Spark release 1.5.2 (Nov 09 2015),
2. Chose the package type Prebuilt for Hadoop 2.6 and later,
3. Chose the download type Direct Download,
4. Download Spark: spark-1.5.2-bin-hadoop2.6.tgz,
5. Verify this release using the 1.3.0 signatures and checksums,

This can also be accomplished by running:

# download spark
$ wget http://d3kbcqa49mib13.cloudfront.net/spark-1.5.2-bin-hadoop2.6.tgz

Next, we'll extract the files and clean up:

# extract, clean up, move the unzipped files under the spark directory
$ tar -xf spark-1.5.2-bin-hadoop2.6.tgz
$ rm spark-1.5.2-bin-hadoop2.6.tgz
$ sudo mv spark-* spark

Now, we can run the Spark Python interpreter with:

# run spark
$ cd ~/spark
./bin/pyspark
You should see something like this:

Welcome to

   ___              ___
  /  __/_  __ _  __/ /__
 / _/ _ \/ _ \ \__ \/ _ `
/__/ \__/\_\/_/__/ \_

Using Python version 2.7.6 (default, Mar 22 2014 22:59:56)
SparkContext available as sc.

The interpreter will have already provided us with a Spark context object, sc, which we can see by running:

>>> print(sc)
<pyspark.context.SparkContext object at 0x7f34b61c4e50>

Enabling IPython Notebook

We will work with IPython Notebook for a friendlier user experience than the console.

You can launch IPython Notebook by using the following command:

$ IPYTHON_OPTS="notebook --pylab inline" ./bin/pyspark

Launch PySpark with IPYNB in the directory examples/AN_Spark where Jupyter or IPython Notebooks are stored:

# cd to /home/an/spark/spark-1.5.0-bin-hadoop2.6/examples/AN_Spark
# launch command using python 2.7 and the spark-csv package:
$ IPYTHON_OPTS='notebook' /home/an/spark/spark-1.5.0-bin-hadoop2.6/bin/pyspark --packages com.databricks:spark-csv_2.11:1.2.0

# launch command using python 3.4 and the spark-csv package:
$ IPYTHON_OPTS='notebook' PYSPARK_PYTHON=python3
   /home/an/spark/spark-1.5.0-bin-hadoop2.6/bin/pyspark --packages com.databricks:spark-csv_2.11:1.2.0
Building our first app with PySpark

We are ready to check now that everything is working fine. The obligatory word count will be put to the test in processing a word count on the first chapter of this book.

The code we will be running is listed here:

```python
# Word count on 1st Chapter of the Book using PySpark

# import regex module
import re
# import add from operator module
from operator import add

# read input file
file_in = sc.textFile('/home/an/Documents/A00_Documents/Spark4Py 20150315')

# count lines
print('number of lines in file: %s' % file_in.count())

# add up lengths of each line
chars = file_in.map(lambda s: len(s)).reduce(add)
print('number of characters in file: %s' % chars)

# Get words from the input file
words = file_in.flatMap(lambda line: re.split('\W+', line.lower().strip()))
# words of more than 3 characters
words = words.filter(lambda x: len(x) > 3)
# set count 1 per word
words = words.map(lambda w: (w,1))
# reduce phase - sum count all the words
words = words.reduceByKey(add)
```

In this program, we are first reading the file from the directory `/home/an/Documents/A00_Documents/Spark4Py 20150315` into `file_in`. We are then introspecting the file by counting the number of lines and the number of characters per line.
Setting Up a Spark Virtual Environment

We are splitting the input file into words and getting them in lower case. For our word count purpose, we are choosing words longer than three characters in order to avoid shorter and much more frequent words such as the, and, for to skew the count in their favor. Generally, they are considered stop words and should be filtered out in any language processing task.

At this stage, we are getting ready for the MapReduce steps. To each word, we map a value of 1 and reduce it by summing all the unique words.

Here are illustrations of the code in the IPython Notebook. The first 10 cells are preprocessing the word count on the dataset, which is retrieved from the local file directory.

```
In [1]: # pyspark context sc is up and running
sc
In [2]: # sc master - running locally
sc.master
Out[2]: u'local[*]'

## Word count on manuscript using PySpark

In [3]: # Import regex module
import re
# Import add from operator module
from operator import add

In [4]: # read input file
file_in = sc.textFile('/home/an/Documents/A00_Documents/SparkPy 20150315')

In [5]: # count lines
print('number of lines in file: %s' % file_in.count())
#
# add up lengths of each line
chars = file_in.map(lambda s: len(s)).reduce(add)
print('number of characters in file: %s' % chars)

number of lines in file: 176
number of characters in file: 11819

In [6]: # Get words from the input file
words = file_in.flatMap(lambda line: re.split('\\s+', line.lower().strip()))

In [7]: # words of more than 3 characters
words = words.filter(lambda x: len(x) > 3)

In [8]: # set count 1 per word
words = words.map(lambda w: (w, 1))

In [9]: # reduce phase - sum count all the words
words = words.reduceByKey(add)

In [10]: # create tuple (count, word) and sort in descending
words = words.map(lambda x: (x[1], x[0])).sortByKey(False)
```
Swap the word count tuples in the format \((\text{count}, \text{word})\) in order to sort by count, which is now the primary key of the tuple:

```python
# create tuple (count, word) and sort in descending
words = words.map(lambda x: (x[1], x[0])).sortByKey(False)

# take top 20 words by frequency
words.take(20)
```

In order to display our result, we are creating the tuple \((\text{count}, \text{word})\) and displaying the top 20 most frequently used words in descending order:

```
In [9]: # reduce phase - sum count all the words
    words = words.reduceByKey(add)

In [10]: # create tuple (count, word) and sort in descending
    words = words.map(lambda x: (x[1], x[0])).sortByKey(False)

In [11]: # take top 20 words by frequency
    words.take(20)
```

```
Out[11]: [(45, u'spark'),
           (26, u'data'),
           (20, u'with'),
           (18, u'anaconda'),
           (16, u'layer'),
           (15, u'python'),
           (13, u'hadoop'),
           (12, u'such'),
           (11, u'from'),
           (11, u'install'),
           (10, u'distributed'),
           (10, u'will'),
           (9, u'processing'),
           (8, u'download'),
           (8, u'pyspark'),
           (8, u'cluster'),
           (7, u'environmont'),
           (7, u'that'),
           (7, u'which'),
           (7, u'analytics')]

In [12]: # create function for histogram of most frequent words
    #
    % matplotlib inline
    import matplotlib.pyplot as plt
    #
    def histogram(words):
        count = map(lambda x: x[1], words)
        word = map(lambda x: x[0], words)
        plt.barh(range(len(count)), count, color = 'grey')
        plt.yticks(range(len(count)), word)

In [13]: # Change order of tuple (word, count) from (count, word)
    words = words.map(lambda x:(x[1], x[0]))
    words.take(20)
```
Let's create a histogram function:

```python
# create function for histogram of most frequent words

%matplotlib inline
import matplotlib.pyplot as plt
#

def histogram(words):
    count = map(lambda x: x[1], words)
    word = map(lambda x: x[0], words)
    plt.barh(range(len(count)), count, color = 'grey')
    plt.yticks(range(len(count)), word)
    # Change order of tuple (word, count) from (count, word)
    words = words.map(lambda x: (x[1], x[0]))
    words.take(25)
    # display histogram
    histogram(words.take(25))
```

Here, we visualize the most frequent words by plotting them in a bar chart. We have to first swap the tuple from the original (count, word) to (word, count):
So here you have it: the most frequent words used in the first chapter are **Spark**, followed by **Data** and **Anaconda**.
Virtualizing the environment with Vagrant

In order to create a portable Python and Spark environment that can be easily shared and cloned, the development environment can be built with a `vagrantfile`.

We will point to the Massive Open Online Courses (MOOCs) delivered by Berkeley University and Databricks:

- **Introduction to Big Data with Apache Spark, Professor Anthony D. Joseph** can be found at [https://www.edx.org/course/introduction-big-data-apache-spark-uc-berkeleyx-cs100-1x](https://www.edx.org/course/introduction-big-data-apache-spark-uc-berkeleyx-cs100-1x)
- **Scalable Machine Learning, Professor Ameet Talwalkar** can be found at [https://www.edx.org/course/scalable-machine-learning-uc-berkeleyx-cs190-1x](https://www.edx.org/course/scalable-machine-learning-uc-berkeleyx-cs190-1x)

The course labs were executed on IPython Notebooks powered by PySpark. They can be found in the following GitHub repository: [https://github.com/spark-mooc/mooc-setup/](https://github.com/spark-mooc/mooc-setup/).

Once you have set up Vagrant on your machine, follow these instructions to get started: [https://docs.vagrantup.com/v2/getting-started/index.html](https://docs.vagrantup.com/v2/getting-started/index.html).

Clone the `spark-mooc/mooc-setup/` GitHub repository in your work directory and launch the command `vagrant up`, within the cloned directory:

Be aware that the version of Spark may be outdated as the `vagrantfile` may not be up-to-date.

You will see an output similar to this:

C:\Programs\spark\edx1001\mooc-setup-master>vagrant up
Bringing machine 'sparkvm' up with 'virtualbox' provider...
==> sparkvm: Checking if box 'sparkmooc/base' is up to date...
==> sparkvm: Clearing any previously set forwarded ports...
==> sparkvm: Clearing any previously set network interfaces...
==> sparkvm: Preparing network interfaces based on configuration...
 火花: Adapter 1: nat
 火花: Forwarding ports...
    火花: 8001 => 8001 (adapter 1)
    火花: 4040 => 4040 (adapter 1)
    火花: 22 => 2222 (adapter 1)
==> sparkvm: Booting VM...
==> sparkvm: Waiting for machine to boot. This may take a few minutes...
 火花: SSH address: 127.0.0.1:2222
 火花: SSH username: vagrant
 火花: SSH auth method: private key
sparkvm: Warning: Connection timeout. Retrying...
sparkvm: Warning: Remote connection disconnect. Retrying...

==> sparkvm: Machine booted and ready!
==> sparkvm: Checking for guest additions in VM...
==> sparkvm: Setting hostname...
==> sparkvm: Mounting shared folders...
  sparkvm: /vagrant => C:/Programs/spark/edx1001/mooc-setup-master
==> sparkvm: Machine already provisioned. Run `vagrant provision` or use the `--provision`
==> sparkvm: to force provisioning. Provisioners marked to run always will still run.

C:\Programs\spark\edx1001\mooc-setup-master>

This will launch the IPython Notebooks powered by PySpark on localhost:8001:
Setting Up a Spark Virtual Environment

Moving to the cloud
As we are dealing with distributed systems, an environment on a virtual machine running on a single laptop is limited for exploration and learning. We can move to the cloud in order to experience the power and scalability of the Spark distributed framework.

Deploying apps in Amazon Web Services
Once we are ready to scale our apps, we can migrate our development environment to Amazon Web Services (AWS).

How to run Spark on EC2 is clearly described in the following page: https://spark.apache.org/docs/latest/ec2-scripts.html.

We emphasize five key steps in setting up the AWS Spark environment:

2. Export your key pair to your environment:
   ```
   export AWS_ACCESS_KEY_ID=accesskeyid
   export AWS_SECRET_ACCESS_KEY=secretaccesskey
   ```
3. Launch your cluster:
   ```
   ~$ cd $SPARK_HOME/ec2
   ec2$ ./spark-ec2 -k <keypair> -i <key-file> -s <num-slaves> launch <cluster-name>
   ```
4. SSH into a cluster to run Spark jobs:
   ```
   ec2$ ./spark-ec2 -k <keypair> -i <key-file> login <cluster-name>
   ```
5. Destroy your cluster after usage:
   ```
   ec2$ ./spark-ec2 destroy <cluster-name>
   ```

Virtualizing the environment with Docker
In order to create a portable Python and Spark environment that can be easily shared and cloned, the development environment can be built in Docker containers.

We wish capitalize on Docker's two main functions:

- Creating isolated containers that can be easily deployed on different operating systems or in the cloud.
• Allowing easy sharing of the development environment image with all its dependencies using The DockerHub. The DockerHub is similar to GitHub. It allows easy cloning and version control. The snapshot image of the configured environment can be the baseline for further enhancements.

The following diagram illustrates a Docker-enabled environment with Spark, Anaconda, and the database server and their respective data volumes.

Docker offers the ability to clone and deploy an environment from the Dockerfile.

You can find an example Dockerfile with a PySpark and Anaconda setup at the following address: https://hub.docker.com/r/thisgokeboysef/pyspark-docker/-/dockerfile/.
Setting Up a Spark Virtual Environment

Install Docker as per the instructions provided at the following links:

- http://docs.docker.com/mac/started/ if you are on Mac OS X
- http://docs.docker.com/linux/started/ if you are on Linux
- http://docs.docker.com/windows/started/ if you are on Windows

Install the docker container with the Dockerfile provided earlier with the following command:

```
$ docker pull thisgokeboysef/pyspark-docker
```

Other great sources of information on how to dockerize your environment can be seen at Lab41. The GitHub repository contains the necessary code:

https://github.com/Lab41/ipython-spark-docker

The supporting blog post is rich in information on thought processes involved in building the docker environment: http://lab41.github.io/blog/2015/04/13/ipython-on-spark-on-docker/.

Summary

We set the context of building data-intensive apps by describing the overall architecture structured around the infrastructure, persistence, integration, analytics, and engagement layers. We also discussed Spark and Anaconda with their respective building blocks. We set up an environment in a VirtualBox with Anaconda and Spark and demonstrated a word count app using the text content of the first chapter as input.

In the next chapter, we will delve more deeply into the architecture blueprint for data-intensive apps and tap into the Twitter, GitHub, and Meetup APIs to get a feel of the data we will be mining with Spark.
Where to buy this book

You can buy Spark for Python Developers from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.