Spark for Big Data Analytics

As the use of Hadoop and related technologies in the respective ecosystem gained prominence, a few obvious and salient deficiencies of the Hadoop operational model became apparent. In particular, the ingrained reliance on the MapReduce paradigm, and other facets related to MapReduce, made a truly functional use of the Hadoop ecosystem possible only for major firms that were invested deeply in the respective technologies.

At the **UC Berkeley Electrical Engineering and Computer Sciences** (**EECS**) Annual Research Symposium of 2011, a vision for a new research group at the university was announced during a presentation by Prof. Ian Stoica (`https://amplab.cs.berkeley.edu/about/`). It laid out the foundation of what was to become a pivotal unit that would profoundly change the landscape of Big Data. The **AMPLab**, launched in February 2011, aimed to deliver a scalable and unified solution by integrating Algorithms, Machines, and People that could cater to future needs without requiring any major re-engineering efforts.

The most well-known and most widely used project to evolve from the AMPLab initiative was Spark, arguably a superior alternative - or more precisely, *extension* - of the Hadoop ecosystem.

In this chapter, we will visit some of the salient characteristics of Spark and end with a real-world tutorial on how to use Spark. The topics we will cover are:

- The advent of Spark
- Theoretical concepts in Spark
- Core components of Spark
- The Spark architecture
- Spark solutions
- Spark tutorial

The advent of Spark

When the first release of Spark became available in 2014, Hadoop had already enjoyed several years of growth since 2009 onwards in the commercial space. Although Hadoop solved a major hurdle in analyzing large terabyte-scale datasets efficiently, using distributed computing methods that were broadly accessible, it still had shortfalls that hindered its wider acceptance.

Limitations of Hadoop

A few of the common limitations with Hadoop were as follows:

- **I/O Bound operations**: Due to the reliance on local disk storage for saving and retrieving data, any operation performed in Hadoop incurred an I/O overhead. The problem became more acute in cases of larger datasets that involved thousands of blocks of data across hundreds of servers. To be fair, the ability to co-ordinate concurrent I/O operations (via HDFS) formed the foundation of distributed computing in Hadoop world. However, leveraging the capability and *tuning* the Hadoop cluster in an efficient manner across different use cases and datasets required an immense and perhaps disproportionate level of expertise. Consequently, the I/O bound nature of workloads became a deterrent factor for using Hadoop against extremely large datasets. As an example, machine learning use cases that required hundreds of iterative operations meant that the system would incur an I/O overhead for each pass of the iteration.
- **MapReduce programming (MR) Model**: As discussed in the earlier parts of this book, all operations in Hadoop require expressing problems in terms of the MapReduce Programming Model - namely, the user would have to express the problem in terms of key-value pairs where each pair can be independently computed. In Hadoop, coding efficient MapReduce programs, mainly in Java, was non-trivial, especially for those new to Java or to Hadoop (or both).
- **Non-MR Use Cases**: Due to the reliance on MapReduce, other more common and simpler concepts such as filters, joins, and so on would have to also be expressed in terms of a MapReduce program. Thus, a join across two files across a primary key would have to adopt a key-value pair approach. This meant that operations, both simple and complex, were hard to achieve without significant programming efforts.

- **Programming APIs**: The use of Java as the central programming language across Hadoop meant that to be able to properly administer and use Hadoop, developers had to have a strong knowledge of Java and related topics such as JVM tuning, Garbage Collection, and others. This also meant that developers in other popular languages such as R, Python, and Scala had very little recourse for re-using or at least implementing their solution in the language they knew best.
- On the whole, even though the Hadoop world had championed the Big Data revolution, it fell short of being able to democratize the use of the technology for Big Data on a broad scale.

The team at AMPLab recognized these shortcomings early on, and set about creating Spark to address these and, in the process, hopefully develop a new, superior alternative.

Overcoming the limitations of Hadoop

We'll now look at some of the limitations discussed in the earlier section and understand how Spark addresses these areas, by virtue of which it provides a superior alternative to the Hadoop ecosystem.

A key difference to bear in mind at the onset is that Spark does NOT need Hadoop in order to operate. In fact, the underlying backend from which Spark accesses data can be technologies such as HBase, Hive and Cassandra in addition to HDFS.

This means that organizations that wish to leverage a standalone Spark system can do so without building a separate Hadoop infrastructure if one does not already exist.

The Spark solutions are as follows:

- **I/O Bound operations**: Unlike Hadoop, Spark can store and access data stored in *memory*, namely RAM - which, as discussed earlier, is 1,000+ times faster than reading data from a disk. With the emergence of SSD drives, the standard in today's enterprise systems, the difference has gone down significantly. Recent NVMe drives can deliver up to 3-5 GB (Giga Bytes) of bandwidth per second. Nevertheless, RAM, which averages about 25-30 GB per second in read speed, is still 5-10x faster compared to reading from the newer storage technologies. As a result, being able to store data in RAM provides a 5x or more improvement to the time it takes to read data for Spark operations. This is a significant improvement over the Hadoop operating model which relies on disk read for all operations. In particular, tasks that involve iterative operations as in machine learning benefit immensely from the Spark's facility to store and read data from memory.

- **MapReduce programming (MR) Model**: While MapReduce is the primary programming model through which users can benefit from a Hadoop platform, Spark does not have the same requirement. This is particularly helpful for more complex use cases such as quantitative analysis involving calculations that cannot be easily *parallelized*, such as machine learning algorithms. By decoupling the programming model from the platform, Spark allows users to write and execute code written in various languages without forcing any specific programming model as a pre-requisite.
- **Non-MR use cases**: Spark SQL, Spark Streaming and other components of the Spark ecosystem provide a rich set of functionalities that allow users to perform common tasks such as SQL joins, aggregations, and related database-like operations without having to leverage other, external solutions. Spark SQL queries are generally executed against data stored in Hive (JSON is another option), and the functionality is also available in other Spark APIs such as R and Python.
- **Programming APIs**: The most commonly used APIs in Spark are Python, Scala and Java. For R programmers, there is a separate package called `SparkR` that permits direct access to Spark data from R. This is a major differentiating factor between Hadoop and Spark, and by exposing APIs in these languages, Spark becomes immediately accessible to a much larger community of developers. In Data Science and Analytics, Python and R are the most prominent languages of choice, and hence, any Python or R programmer can leverage Spark with a much simpler learning curve relative to Hadoop. In addition, Spark also includes an interactive shell for ad-hoc analysis.

Theoretical concepts in Spark

The following are the core concepts in Spark:

- Resilient distributed datasets
- Directed acyclic graphs
- SparkContext
- Spark DataFrames
- Actions and transformations
- Spark deployment options

Resilient distributed datasets

Resilient distributed datasets, more commonly known as **RDD**s, are the primary data structure used in Spark. RDDs are essentially a collection of records that are stored across a Spark cluster in a distributed manner. RDDs are *immutable*, which is to say, they cannot be altered once created. RDDs that are stored across nodes can be accessed in parallel, and hence support parallel operations natively.

The user does not need to write separate code to get the benefits of parallelization but can get the benefits of *actions and transformations* of data simply by running specific commands that are native to the Spark platform. Because RDDs can be also stored in memory, as an additional benefit, the parallel operations can act on the data directly in memory without incurring expensive I/O access penalties.

Directed acyclic graphs

In computer science and mathematics parlance, a directed acyclic graph represents pairs of nodes (also known as **vertices**) connected with edges (or **lines**) that are unidirectional. Namely, given Node A and Node B, the edge can connect A à B or B à A but not both. In other words, there isn't a circular relationship between any pair of nodes.

Spark leverages the concept of DAG to build an internal workflow that delineates the different stages of processing in a Spark job. Conceptually, this is akin to creating a virtual flowchart of the series of steps needed to obtain a certain output. For instance, if the required output involves producing a count of words in a document, the intermediary steps map-shuffle-reduce can be represented as a series of actions that lead to the final result. By maintaining such a **map**, Spark is able to keep track of the dependencies involved in the operation. More specifically, RDDs are the **nodes**, and transformations, which are discussed later in this section, are the **edges** of the DAG.

SparkContext

A SparkContext is the entry point for all Spark operations and means by which the application connects to the resources of the Spark cluster. It initializes an instance of Spark and can thereafter be used to create RDDs, perform actions and transformations on the RDDs, and extract data and other Spark functionalities. A SparkContext also initializes various properties of the process, such as the application name, number of cores, memory usage parameters, and other characteristics. Collectively, these properties are contained in the object SparkConf, which is passed to SparkContext as a parameter.

`SparkSession` is the new abstraction through which users initiate their connection to Spark. It is a superset of the functionality provided in `SparkContext` prior to Spark 2.0.0. However, practitioners still use `SparkSession` and `SparkContext` interchangeably to mean one and the same entity; namely, the primary mode of interacting with `Spark.SparkSession` has essentially combined the functionalities of both SparkContext and `HiveContext`.

Spark DataFrames

A DataFrame in Spark is the raw data organized into rows and columns. This is conceptually similar to CSV files or SQL tables. Using R, Python and other Spark APIs, the user can interact with a DataFrame using common Spark commands used for filtering, aggregating, and more generally manipulating the data. The data contained in DataFrames are physically located across the multiple nodes of the Spark cluster. However, by representing them in a **DataFrame** they appear to be a cohesive unit of data without exposing the complexity of the underlying operations.

Note that DataFrames are not the same as Datasets, another common term used in Spark. Datasets refer to the actual data that is held across the Spark cluster. A DataFrame is the tabular representation of the Dataset.

Starting with Spark 2.0, the DataFrame and Dataset APIs were merged and a DataFrame in essence now represents a Dataset of Row. That said, DataFrame still remains the primary abstraction for users who want to leverage Python and R for interacting with Spark data.

Actions and transformations

There are 2 types of Spark operations:

- Transformations
- Actions

Transformations specify general data manipulation operations such as filtering data, joining data, performing aggregations, sampling data, and so on. Transformations do not return any result when the line containing the transformation operation in the code is executed. Instead, the command, upon execution, supplements Spark's internal DAG with the corresponding operation request. Examples of common transformations include: `map`, `filter`, `groupBy`, `union`, `coalesce`, and many others.

Actions, on the other hand, return results. Namely, they execute the series of transformations (if any) that the user may have specified on the corresponding RDD and produce an output. In other words, actions trigger the execution of the steps in the DAG. Common Actions include: `reduce`, `collect`, `take`, `aggregate`, `foreach`, and many others.

Note that RDDs are immutable. They cannot be changed; transformations and actions will always produce new RDDs, but never modify existing ones.

Spark deployment options

Spark can be deployed in various modes. The most important ones are:

- **Standalone mode**: As an independent cluster not dependent upon any external cluster manager
- **Amazon EC2**: On EC2 instances of Amazon Web Services where it can access data from S3
- **Apache YARN**: The Hadoop ResourceManager

Other options include **Apache Mesos** and **Kubernetes.**

Further details can be found at the Spark documentation website, `https://spark.apache.org/docs/latest/index.html`.

Spark APIs

The Spark platform is easily accessible through Spark APIs available in Python, Scala, R, and Java. Together they make working with data in Spark simple and broadly accessible. During the inception of the Spark project, it only supported Scala/Java as the primary API. However, since one of the overarching objectives of Spark was to provide an easy interface to a diverse set of developers, the Scala API was followed by a Python and R API.

In Python, the PySpark package has become a widely used standard for writing Spark applications by the Python developer community. In R, users interact with Spark via the SparkR package. This is useful for R developers who may also be interested in working with data stored in a Spark ecosystem. Both of these languages are very prevalent in the Data Science community, and hence, the introduction of the Python and R APIs set the groundwork for democratizing **Big Data** Analytics on Spark for analytical use cases.

Core components in Spark

The following components are quite important in Spark:

- Spark Core
- Spark SQL
- Spark Streaming
- GraphX
- MLlib

Spark Core

Spark Core provides fundamental functionalities in Spark, such as working with RDDs, performing actions, and transformations, in addition to more administrative tasks such as storage, high availability, and other topics.

Spark SQL

Spark SQL provides the user with the ability to query data stored in Apache Hive using standard SQL commands. This adds an additional level of accessibility by providing developers with a means to interact with datasets via the Spark SQL interface using common SQL terminologies. The platform hosting the underlying data is not limited to Apache Hive, but can also include JSON, Parquet, and others.

Spark Streaming

The streaming component of Spark allows users to interact with streaming data such as web-related content and others. It also includes enterprise characteristics such as high availability. Spark can read data from various middleware and data streaming services such as Apache Kafka, Apache Flume, and Cloud based solutions from vendors such as Amazon Web Services.

GraphX

The GraphX component of Spark supports graph-based operations, similar to technologies such as graph databases that support specialized data structures. These make it easy to use, access, and represent inter-connected points of data, such as social networks. Besides analytics, the Spark GraphX platform supports graph algorithms that are useful for business use cases that require relationships to be represented at scale. As an example, credit card companies use Graph based databases similar to the GraphX component of Spark to build recommendation engines that detect users with similar characteristics. These characteristics may include buying habits, location, demographics, and other qualitative and quantitative factors. Using Graph systems in these cases allows companies to build networks with nodes representing individuals and edges representing relationship metrics to find common features amongst them.

MLlib

MLlib is one of the flagship components of the Spark ecosystem. It provides a scalable, high-performance interface to perform resource intensive machine learning tasks in Spark. Additionally, MLlib can natively connect to HDFS, HBase, and other underlying storage systems supported in Spark. Due to this versatility, users do not need to rely on a pre-existing Hadoop environment to start using the algorithms built into MLlib. Some of the supported algorithms in MLlib include:

- **Classification**: logistic regression
- **Regression**: generalized linear regression, survival regression and others
- Decision trees, random forests, and gradient-boosted trees
- **Recommendation**: Alternating least squares
- **Clustering**: K-means, Gaussian mixtures and others
- **Topic modeling**: Latent Dirichlet allocation
- **Apriori**: Frequent Itemsets, Association Rules

ML workflow utilities include:

- **Feature transformations**: Standardization, normalization and others
- ML Pipeline construction
- Model evaluation and hyper-parameter tuning
- **ML persistence**: Saving and loading models and Pipelines

The architecture of Spark

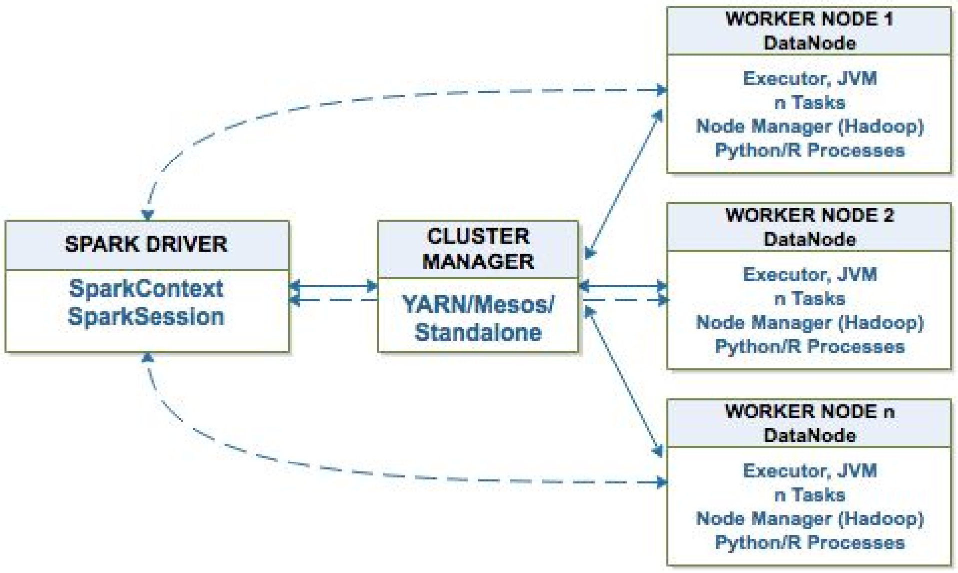
Spark consists of 3 primary architectural components:

- The SparkSession / SparkContext
- The Cluster Manager
- The Worker Nodes (that hosts executor processes)

The **SparkSession/SparkContext**, or more generally the Spark Driver, is the entry point for all Spark applications as discussed earlier. The SparkContext will be used to create RDDs and perform operations against RDDs. The SparkDriver sends instructions to the worker nodes to schedule tasks.

The **Cluster manager** is conceptually similar to Resource Managers in Hadoop and indeed, one of the supported solutions is YARN. Other Cluster Managers include Mesos. Spark can also operate in a Standalone mode in which case YARN/Mesos are not required. Cluster Managers co-ordinate communications between the Worker Nodes, manage the nodes (such as starting, stopping, and so on), and perform other administration tasks.

Worker nodes are servers where Spark applications are hosted. Each application gets its own unique **executor process**, namely, processes that perform the actual action and transformation tasks. By assigning dedicated executor processes, Spark ensures that an issue in any particular application does not impact other applications. Worker Nodes consist of the Executor, the JVM, and the Python/R/other application process required by the Spark application. Note that in the case of Hadoop, the Worker Node and Data Nodes are one and the same:


WORKER NODE 1
DataNode
Executor, JVM
n Tasks
Node Manager (Hadoop)
Python/R Processes
SPARK DRIVER
SparkContext
SparkSession
CLUSTER MANAGER
YARN/Mesos/ Standalone
WORKER NODE 2
DataNode
Executor, JVM
n Tasks
Node Manager (Hadoop)
Python/R Processes
WORKER NODE n
DataNode
Executor, JVM
n Tasks
Node Manager (Hadoop)
Python/R Processes

Spark solutions

Spark is directly available from `spark.apache.org` as an open-source solution. **Databricks** is the leading provider of the commercial solution of Spark. For those who are familiar with programming in Python, R, Java, or Scala, the time required to start using Spark is minimal due to efficient interfaces, such as the PySpark API that allows users to work in Spark using just Python.

Cloud-based Spark platforms, such as the Databricks Community Edition, provide an easy and simple means to work on Spark without the additional work of installing and configuring Spark. Hence, users who wish to use Spark for programming and related tasks can get started much more rapidly without spending time on administrative tasks.

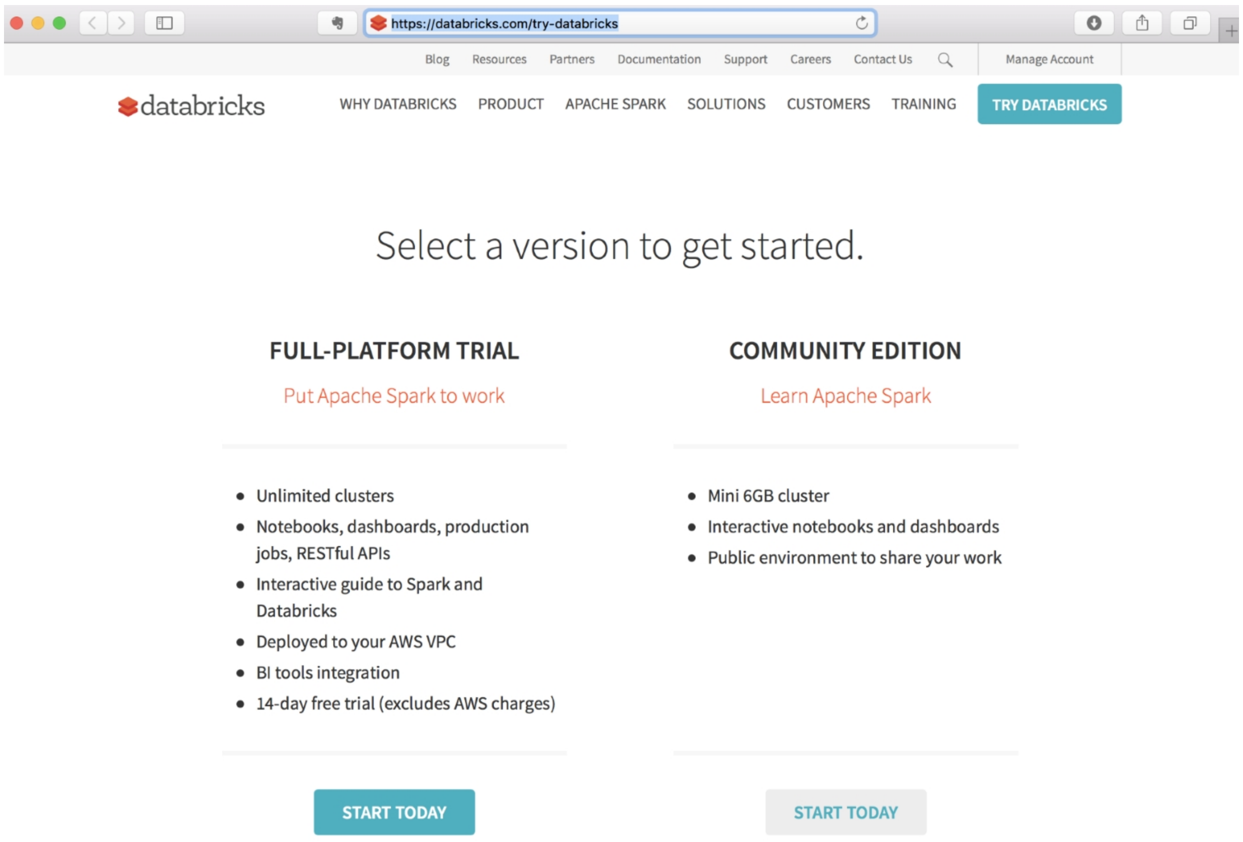
Spark practicals

In this section, we will create an account on Databricks' Community Edition and complete a hands-on exercise that will walk the reader through the basics of actions, transformations, and RDD concepts in general.

Signing up for Databricks Community Edition

The following steps outline the process of signing up for the **Databricks Community Edition**:

1. Go to `https://databricks.com/try-databricks`:



Select a version to get started.

FULL-PLATFORM TRIAL

Put Apache Spark to work

- Unlimited clusters
- Notebooks, dashboards, production jobs, RESTful APIs
- Interactive guide to Spark and Databricks
- Deployed to your AWS VPC
- BI tools integration
- 14-day free trial (excludes AWS charges)

START TODAY

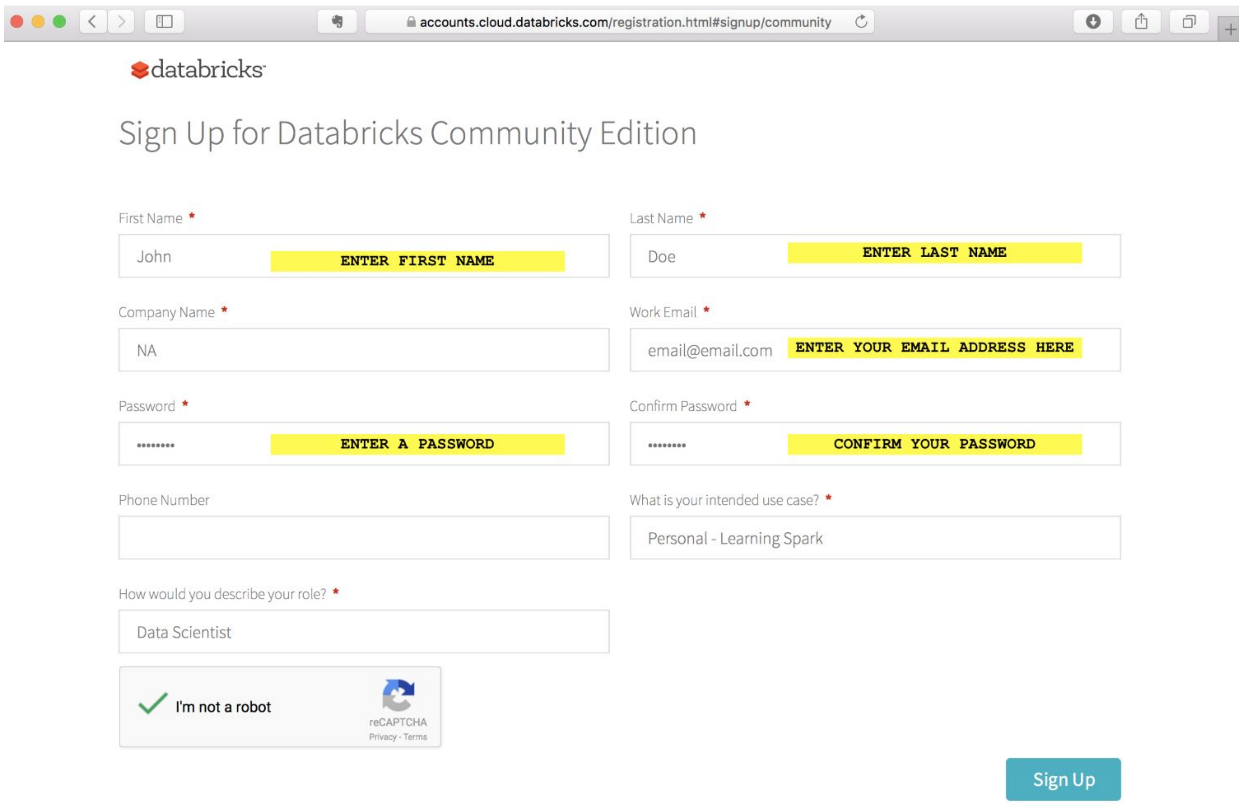
COMMUNITY EDITION

Learn Apache Spark

- Mini 6GB cluster
- Interactive notebooks and dashboards
- Public environment to share your work

START TODAY

2. Click on the START TODAY button and enter your information:



accounts.cloud.databricks.com/registration.html#signup/community

databricks

Sign Up for Databricks Community Edition

First Name * John ENTER FIRST NAME

Last Name * Doe ENTER LAST NAME

Company Name * NA

Work Email * email@email.com ENTER YOUR EMAIL ADDRESS HERE

Password * •••••••• ENTER A PASSWORD

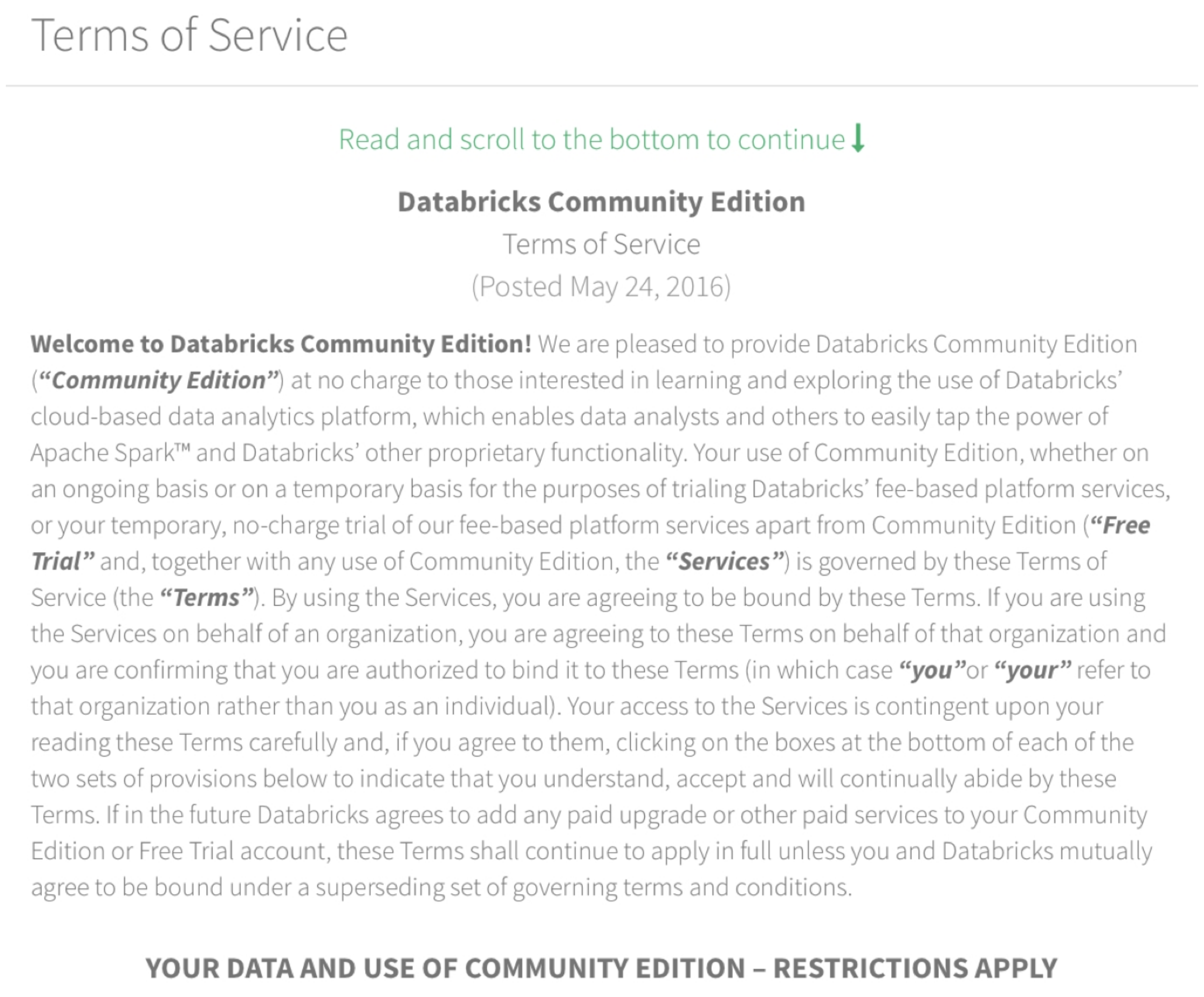
Confirm Password * •••••••• CONFIRM YOUR PASSWORD

Phone Number

What is your intended use case? * Personal - Learning Spark

How would you describe your role? * Data Scientist

3. Confirm that you have read and agree to the terms in the popup menu (scroll down to the bottom for the **Agree** button):



Terms of Service

Read and scroll to the bottom to continue

Databricks Community Edition

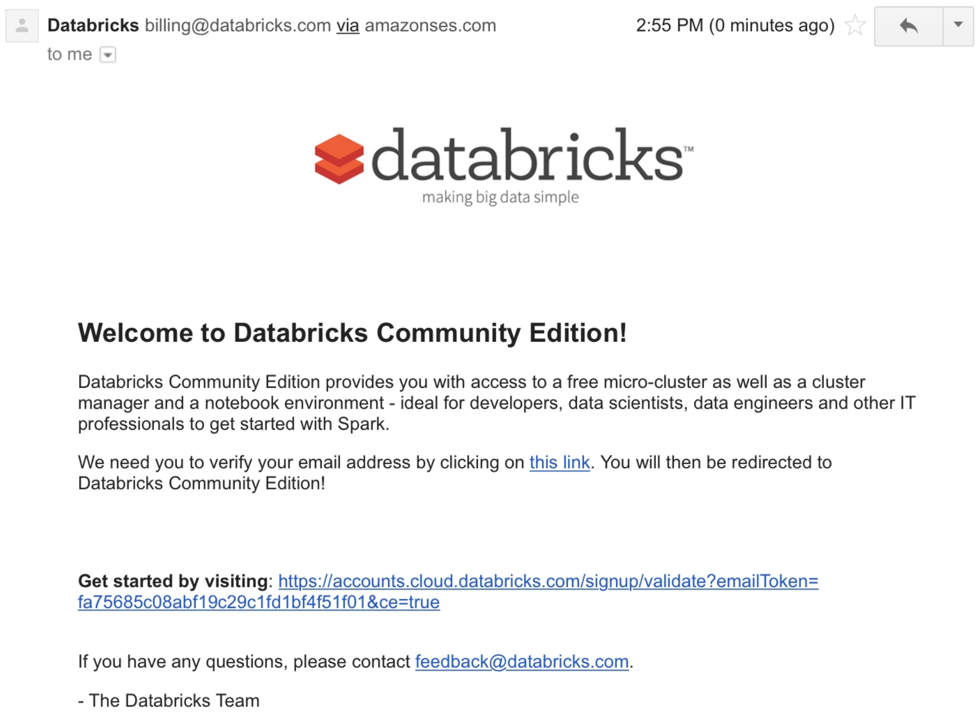
Terms of Service

(Posted May 24, 2016)

Welcome to Databricks Community Edition! We are pleased to provide Databricks Community Edition (***"Community Edition"***) at no charge to those interested in learning and exploring the use of Databricks' cloud-based data analytics platform, which enables data analysts and others to easily tap the power of Apache Spark™ and Databricks' other proprietary functionality. Your use of Community Edition, whether on an ongoing basis or on a temporary basis for the purposes of trialing Databricks' fee-based platform services, or your temporary, no-charge trial of our fee-based platform services apart from Community Edition (***"Free Trial"*** and, together with any use of Community Edition, the ***"Services"***) is governed by these Terms of Service (the ***"Terms"***). By using the Services, you are agreeing to be bound by these Terms. If you are using the Services on behalf of an organization, you are agreeing to these Terms on behalf of that organization and you are confirming that you are authorized to bind it to these Terms (in which case ***"you"***or ***"your"*** refer to that organization rather than you as an individual). Your access to the Services is contingent upon your reading these Terms carefully and, if you agree to them, clicking on the boxes at the bottom of each of the two sets of provisions below to indicate that you understand, accept and will continually abide by these Terms. If in the future Databricks agrees to add any paid upgrade or other paid services to your Community Edition or Free Trial account, these Terms shall continue to apply in full unless you and Databricks mutually agree to be bound under a superseding set of governing terms and conditions.

YOUR DATA AND USE OF COMMUNITY EDITION – RESTRICTIONS APPLY

4. Check your email for a confirmation email from Databricks and click on the link to confirm your account:



Databricks billing@databricks.com via amazonses.com 2:55 PM (0 minutes ago)

to me

databricks
making big data simple

Welcome to Databricks Community Edition!

Databricks Community Edition provides you with access to a free micro-cluster as well as a cluster manager and a notebook environment - ideal for developers, data scientists, data engineers and other IT professionals to get started with Spark.

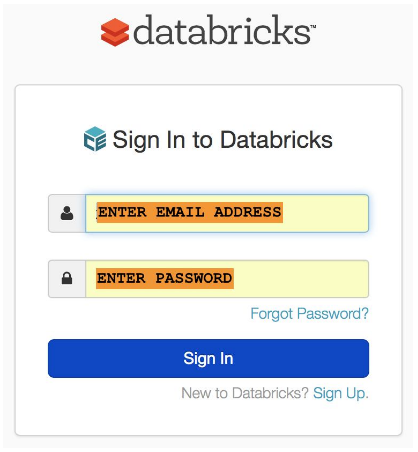
We need you to verify your email address by clicking on this link. You will then be redirected to Databricks Community Edition!

Get started by visiting: https://accounts.cloud.databricks.com/signup/validate?emailToken=fa75685c08abf19c29c1fd1bf4f51f01&ce=true

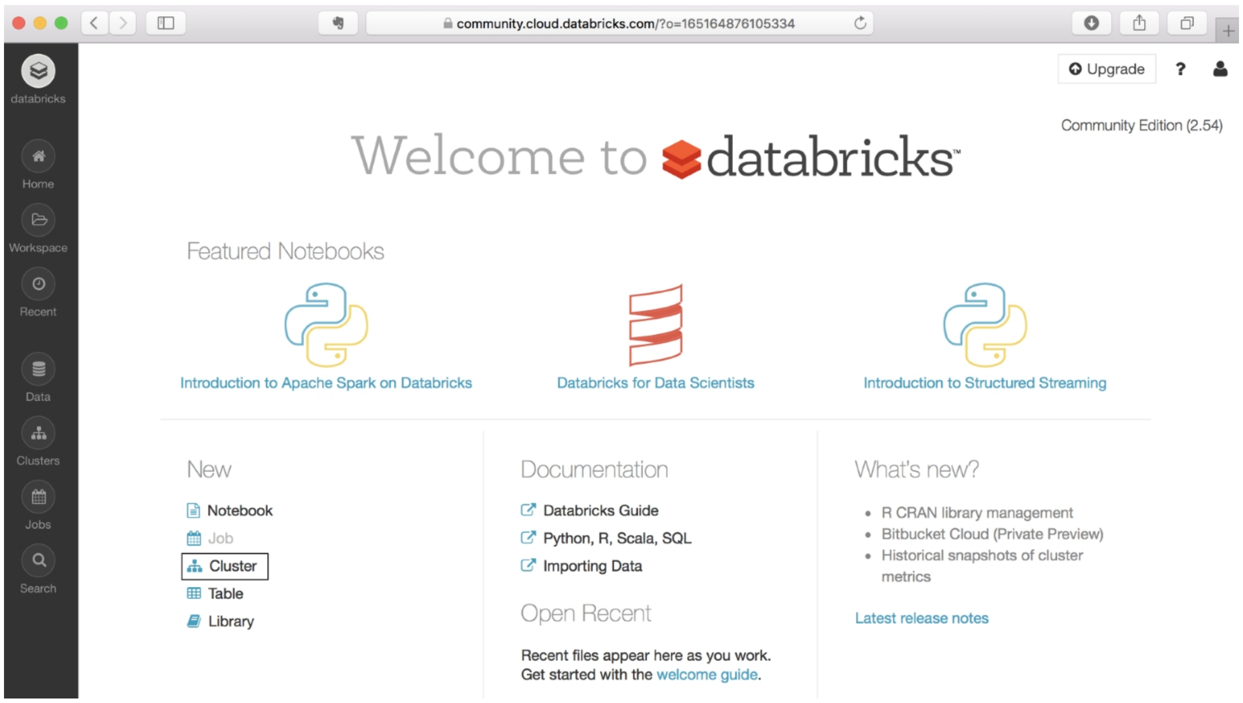
If you have any questions, please contact feedback@databricks.com.

- The Databricks Team

5. Once you click on the link to confirm your account, you'll be taken to a login screen where you can log on using the email address and password you used to sign up for the account:



6. After logging in, click on Cluster to set up a Spark cluster, as shown in the following figure:



Community Edition (2.54)

Welcome to databricks

Featured Notebooks

Introduction to Apache Spark on Databricks

Databricks for Data Scientists

Introduction to Structured Streaming

New

- Notebook
- Job
- Cluster
- Table
- Library

Documentation

- Databricks Guide
- Python, R, Scala, SQL
- Importing Data

Open Recent

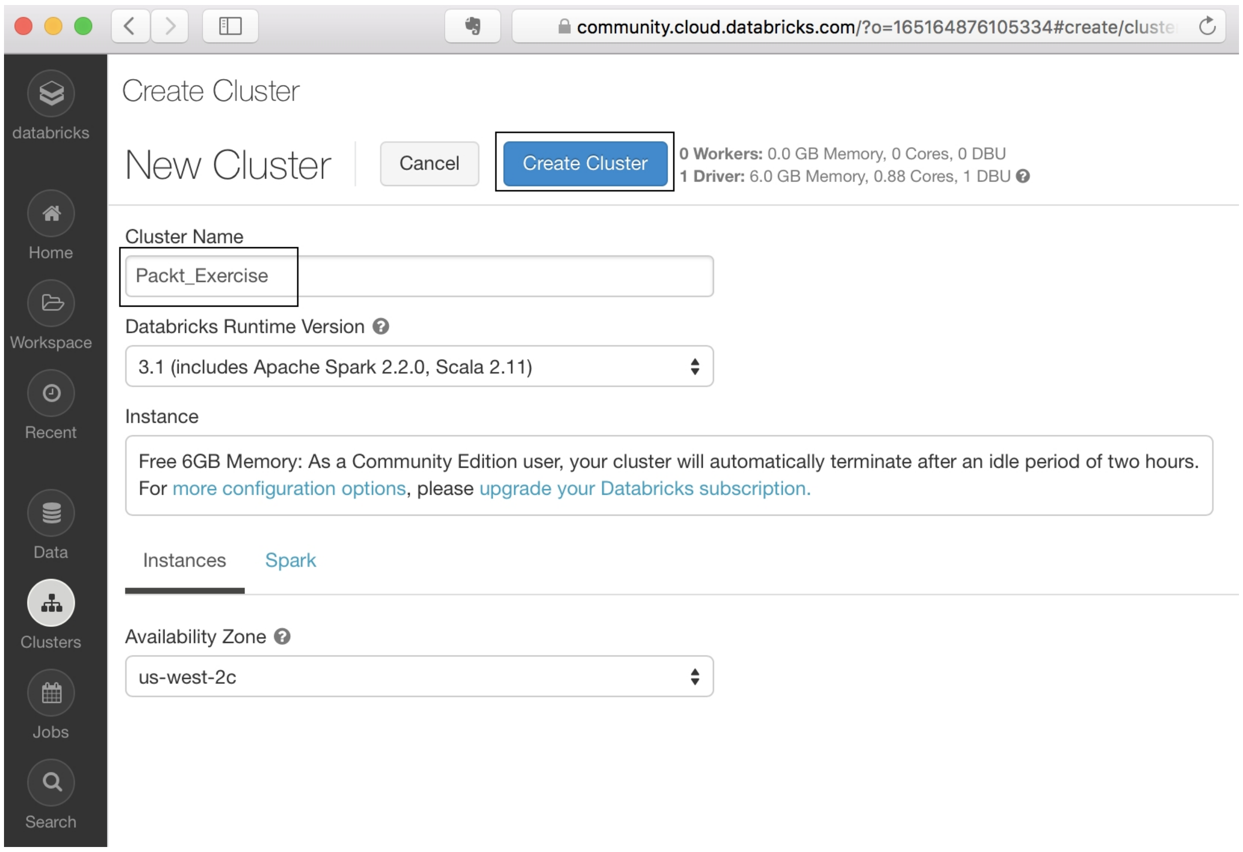
Recent files appear here as you work.
Get started with the welcome guide.

What's new?

- R CRAN library management
- Bitbucket Cloud (Private Preview)
- Historical snapshots of cluster metrics

Latest release notes

7. Enter `Packt_Exercise` as the Cluster Name and click on the Create Cluster button at the top of the page:



Create Cluster

New Cluster

0 Workers: 0.0 GB Memory, 0 Cores, 0 DBU
1 Driver: 6.0 GB Memory, 0.88 Cores, 1 DBU

Cluster Name: Packt_Exercise

Databricks Runtime Version: 3.1 (includes Apache Spark 2.2.0, Scala 2.11)

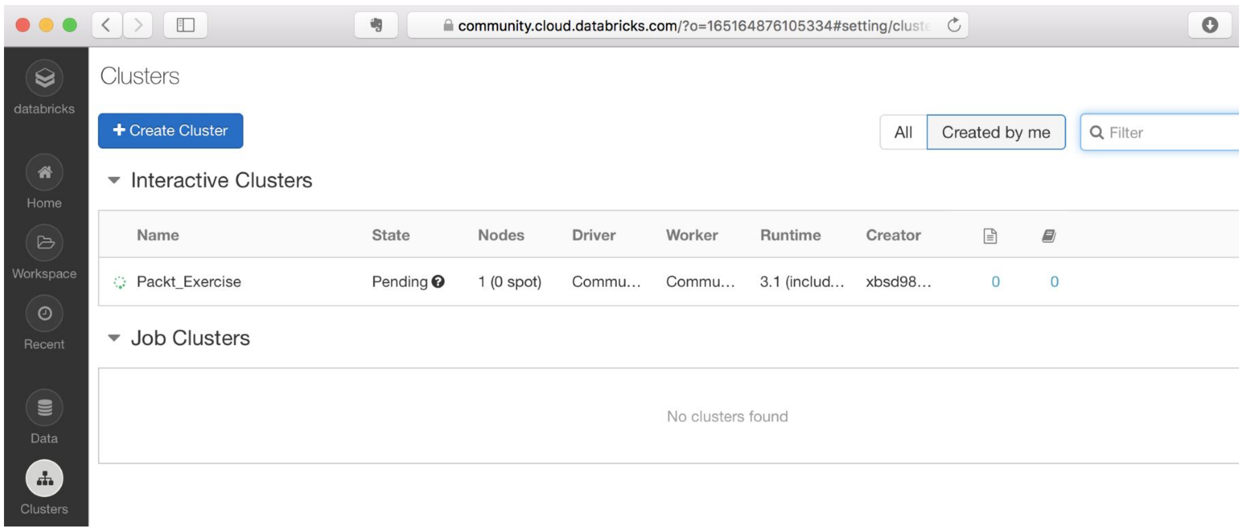
Instance

Free 6GB Memory: As a Community Edition user, your cluster will automatically terminate after an idle period of two hours. For more configuration options, please upgrade your Databricks subscription.

Availability Zone: us-west-2c

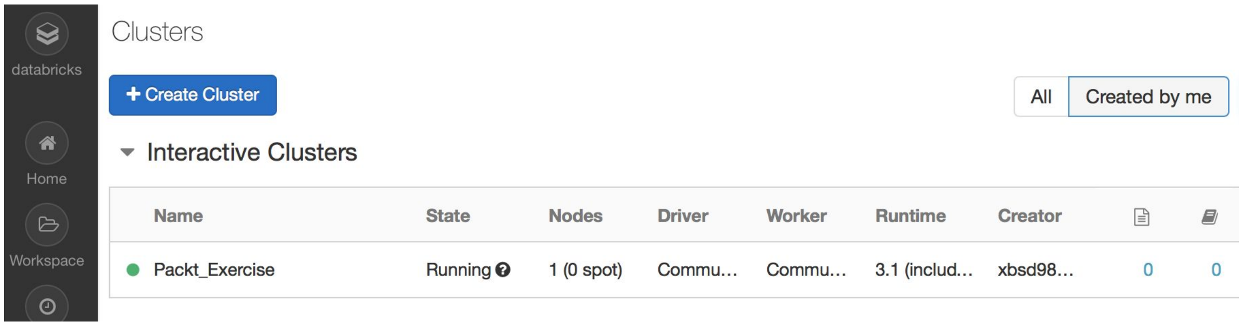
8. This will initiate the process of starting up a Spark Cluster on which we will execute our Spark commands using an iPython notebook. An iPython Notebook is the name given to a commonly used IDE - a web-based development application used for writing and testing Python code. The notebook can also support other languages through the use of kernels, but for the purpose of this exercise, we will focus on the Python kernel.

   After a while, the Status will change from Pending to Running:



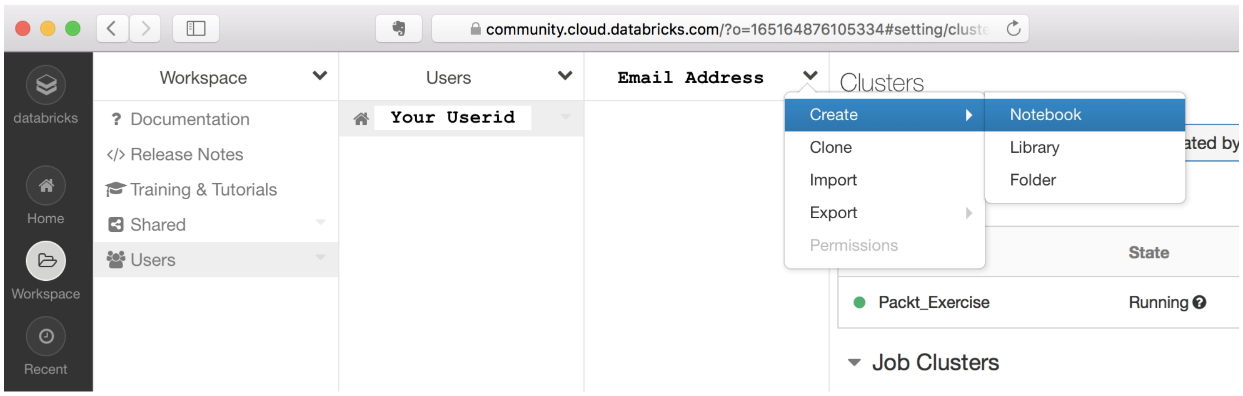
Name	State	Nodes	Driver	Worker	Runtime	Creator		
Packt_Exercise	Pending	1 (0 spot)	Commu…	Commu…	3.1 (includ…	xbsd98…	0	0

Status changes to Running after a few minutes:

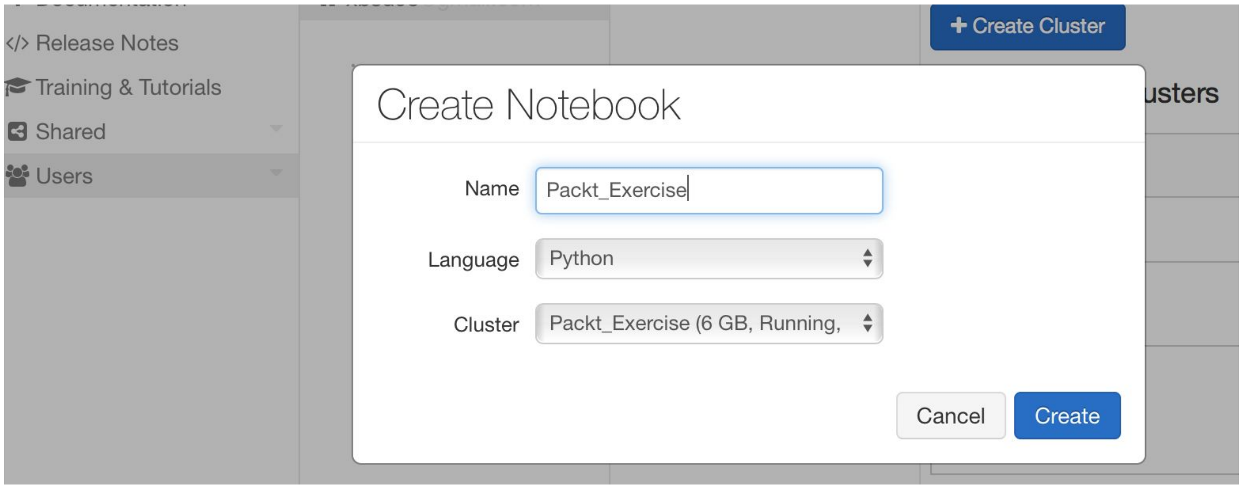


Name	State	Nodes	Driver	Worker	Runtime	Creator		
Packt_Exercise	Running	1 (0 spot)	Commu…	Commu…	3.1 (includ…	xbsd98…	0	0

9. Click on **Workspace** (on the left hand bar) and select **options**, **Users** | (`Your userid`) and click on the drop-down arrow next to your email address. Select Create | Notebook:

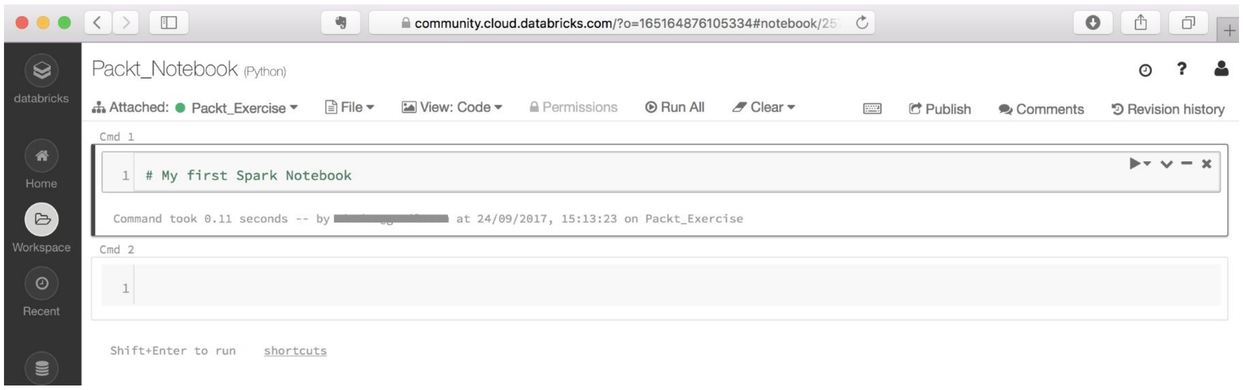


10. In the popup screen, enter `Packt_Exercise` as the name of the notebook and click on the Create button:



Name: Packt_Exercise
Language: Python
Cluster: Packt_Exercise (6 GB, Running,

11. Once you click on the **Create** button, you'll be taken directly to the Notebook as shown in the following screenshot. This is the Spark Notebook, where you'll be able to execute the rest of the code given in the next few sections. The code should be typed in the cells of the notebook as shown. After entering your code, press *Shift* + *Enter* to execute the corresponding cell:

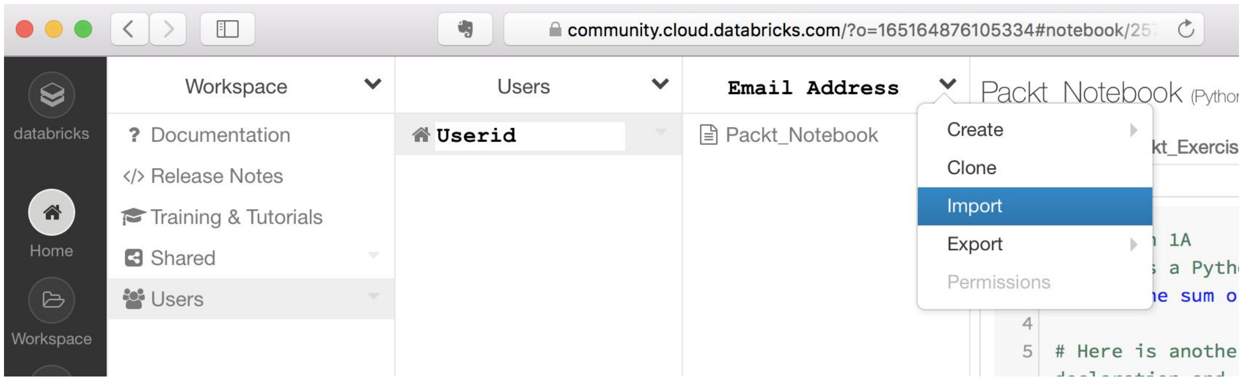


```
# My first Spark Notebook
```

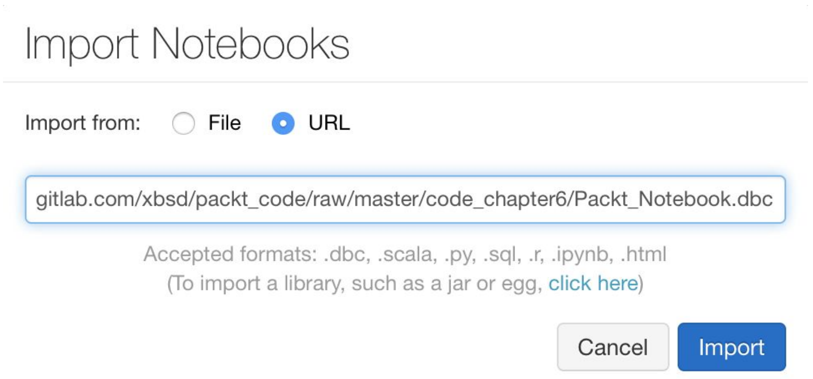
```
Command took 0.11 seconds -- by [redacted] at 24/09/2017, 15:13:23 on Packt_Exercise
```

12. For the next few exercises, you can copy-paste the text into the cells of the Notebook. Alternatively, you can also import the notebook and load it directly in your workspace. If you do so, you'll not need to type in the commands (although typing in the commands will provide more hands-on familiarity).

13. An alternative approach to copy-pasting commands: You can import the notebook by clicking on Import as shown in the following screenshot:



14. Enter the following **URL** in the popup menu (select **URL** as the **Import from** option):

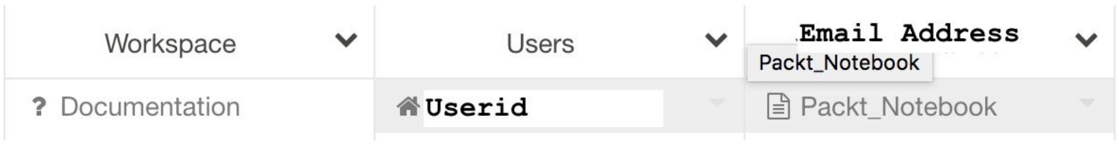

Import Notebooks

Import from: File [✓ URL]

gitlab.com/xbsd/packt_code/raw/master/code_chapter6/Packt_Notebook.dbc

Accepted formats: .dbc, .scala, .py, .sql, .r, .ipynb, .html
(To import a library, such as a jar or egg, click here)

15. The notebook will then show up under your email ID. Click on the name of the notebook to load it:



Spark exercise - hands-on with Spark (Databricks)

This notebook is based on tutorials conducted by Databricks (`https://databricks.com/`). The tutorial will be conducted using the Databricks' Community Edition of Spark, available to sign up to at `https://databricks.com/try-databricks`. Databricks is a leading provider of the commercial and enterprise supported version of Spark.

In this tutorial, we will introduce a few basic commands used in Spark. Users are encouraged to try out more extensive Spark tutorials and notebooks that are available on the web for more detailed examples.

Documentation for Spark's Python API can be found at `https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.sql`.

The data for this book was imported into the Databricks' Spark Platform. For more information on importing data, go to **Importing Data - Databricks** (`https://docs.databricks.com/user-guide/importing-data.html`).

```
# COMMAND ----------

# The SparkContext/SparkSession is the entry point for all Spark operations
# sc = the SparkContext = the execution environment of Spark, only 1 per JVM
# Note that SparkSession is now the entry point (from Spark v2.0)
# This tutorial uses SparkContext (was used prior to Spark 2.0)

from pyspark import SparkContext
# sc = SparkContext(appName = "some_application_name") # You'd normally run this,
but in this case, it has already been created in the Databricks' environment

# COMMAND ----------

quote = "To be, or not to be, that is the question: Whether 'tis nobler in the
mind to suffer The slings and arrows of outrageous fortune, Or to take Arms
against a Sea of troubles, And by opposing end them: to die, to sleep No more;
and by a sleep, to say we end the heart-ache, and the thousand natural shocks
that Flesh is heir to? 'Tis a consummation devoutly to be wished. To die, to
sleep, To sleep, perchance to Dream; aye, there's the rub, for in that sleep of
death, what dreams may come, when we have shuffled off this mortal coil, must
give us pause."
```

```
# COMMAND ----------
sparkdata = sc.parallelize(quote.split(' '))

# COMMAND ----------
print "sparkdata = ", sparkdata
print "sparkdata.collect = ", sparkdata.collect
print "sparkdata.collect() = ", sparkdata.collect()[1:10]

# COMMAND ----------
# A simple transformation - map
def mapword(word):
 return (word,1)

print sparkdata.map(mapword) # Nothing has happened here
print sparkdata.map(mapword).collect()[1:10] # collect causes the DAG to execute

# COMMAND ----------
# Another Transformation

def charsmorethan2(tuple1):
 if len(tuple1[0])>2:
 return tuple1
 pass

rdd3 = sparkdata.map(mapword).filter(lambda x: charsmorethan2(x))
# Multiple Transformations in 1 statement, nothing is happening yet
rdd3.collect()[1:10]
# The DAG gets executed. Note that since we didn't remove punctuation marks ...
'be,', etc are also included

# COMMAND ----------
# With Tables, a general example
cms = sc.parallelize([[1,"Dr. A",12.50,"Yale"],[2,"Dr. B",5.10,"Duke"],[3,"Dr.
C",200.34,"Mt. Sinai"],[4,"Dr. D",5.67,"Duke"],[1,"Dr. E",52.50,"Yale"]])

# COMMAND ----------
def findPayment(data):
 return data[2]

print "Payments = ", cms.map(findPayment).collect()
print "Mean = ", cms.map(findPayment).mean() # Mean is an action

# COMMAND ----------
# Creating a DataFrame (familiar to Python programmers)

cms_df = sqlContext.createDataFrame(cms, ["ID","Name","Payment","Hosp"])
print cms_df.show()
print cms_df.groupby('Hosp').agg(func.avg('Payment'),
func.max('Payment'),func.min('Payment'))
print cms_df.groupby('Hosp').agg(func.avg('Payment'),
func.max('Payment'),func.min('Payment')).collect()
print
print "Converting to a Pandas DataFrame"
print "--------------------------------"
pd_df = cms_df.groupby('Hosp').agg(func.avg('Payment'),
func.max('Payment'),func.min('Payment')).toPandas()
print type(pd_df)
print
print pd_df
```

```
# COMMAND ----------
wordsList = ['to','be','or','not','to','be']
wordsRDD = sc.parallelize(wordsList, 3) # Splits into 2 groups
# Print out the type of wordsRDD
print type(wordsRDD)

# COMMAND ----------
# Glom coallesces all elements within each partition into a list
print wordsRDD.glom().take(2) # Take is an action, here we are 'take'-ing the
first 2 elements of the wordsRDD
print wordsRDD.glom().collect() # Collect

# COMMAND ----------
# An example with changing the case of words
# One way of completing the function
def makeUpperCase(word):
 return word.upper()

print makeUpperCase('cat')

# COMMAND ----------
upperRDD = wordsRDD.map(makeUpperCase)
print upperRDD.collect()

# COMMAND ----------
upperLambdaRDD = wordsRDD.map(lambda word: word.upper())
print upperLambdaRDD.collect()

# COMMAND ----------

# Pair RDDs
wordPairs = wordsRDD.map(lambda word: (word, 1))
print wordPairs.collect()

# COMMAND ----------

# #### Part 2: Counting with pair RDDs
# There are multiple ways of performing group-by operations in Spark
# One such method is groupByKey()
#
# ** Using groupByKey() **
#
# This method creates a key-value pair whereby each key (in this case word) is
assigned a value of 1 for our wordcount operation. It then combines all keys into
a single list. This can be quite memory intensive, especially if the dataset is
large.

# COMMAND ----------
# Using groupByKey
wordsGrouped = wordPairs.groupByKey()
for key, value in wordsGrouped.collect():
 print '{0}: {1}'.format(key, list(value))

# COMMAND ----------
# Summation of the key values (to get the word count)
wordCountsGrouped = wordsGrouped.map(lambda (k,v): (k, sum(v)))
print wordCountsGrouped.collect()

# COMMAND ----------
# ** (2c) Counting using reduceByKey **
#
# reduceByKey creates a new pair RDD. It then iteratively applies a function
first to each key (i.e., within the key values) and then across all the keys,
i.e., in other words it applies the given function iteratively.

# COMMAND ----------

wordCounts = wordPairs.reduceByKey(lambda a,b: a+b)
print wordCounts.collect()

# COMMAND ----------
# %md
# ** Combining all of the above into a single statement **

# COMMAND ----------

wordCountsCollected = (wordsRDD
 .map(lambda word: (word, 1))
 .reduceByKey(lambda a,b: a+b)
 .collect())
print wordCountsCollected

# COMMAND ----------

# %md
#
# This tutorial has provided a basic overview of Spark and introduced the
Databricks community edition where users can upload and execute their own Spark
notebooks. There are various in-depth tutorials on the web and also at Databricks
on Spark and users are encouraged to peruse them if interested in learning
further about Spark.
```

Summary

In this chapter, we read about some of the core features of Spark, one of the most prominent technologies in the Big Data landscape today. Spark has matured rapidly since its inception in 2014, when it was released as a Big Data solution that alleviated many of the shortcomings of Hadoop, such as I/O contention and others.

Today, Spark has several components, including dedicated ones for streaming analytics and machine learning, and is being actively developed. Databricks is the leading provider of the commercially supported version of Spark and also hosts a very convenient cloud-based Spark environment with limited resources that any user can access at no charge. This has dramatically lowered the barrier to entry as users do not need to install a complete Spark environment to learn and use the platform.

In the next chapter, we will begin our discussion on machine learning. Most of the text, until this section, has focused on the management of large scale data. Making use of the data effectively and gaining *insights* from the data is always the final aim. In order to do so, we need to employ the advanced algorithmic techniques that have become commonplace today. The next chapter will discuss the basic tenets of machine learning, and thereafter we will delve deeper into the subject area in the subsequent chapter.