PySpark

# Introduction to Spark

## Resilient Distributed Data Set (RDD)

The core data structure in Spark is a **resilient distributed data set** (RDD). As the name suggests, an RDD is Spark's representation of a data set that's distributed across the RAM, or memory, of a cluster of many machines. An RDD object is essentially a collection of elements we can use to hold lists of tuples, dictionaries, lists, etc. Similar to a pandas DataFrame, we can load a data set into an RDD, and then run any of the methods accessible to that object.

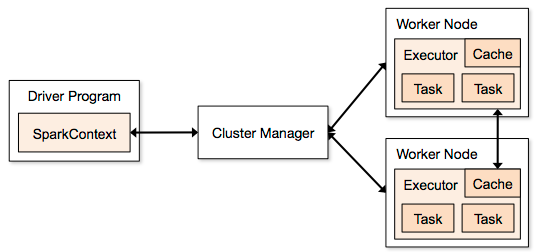
While the Spark toolkit is in Scala, a language that compiles down to bytecode for the JVM, the open source community has developed a wonderful toolkit called [PySpark](https://spark.apache.org/docs/0.9.0/python-programming-guide.html) (<https://spark.apache.org/docs/0.9.0/python-programming-guide.html>) that allows us to interface with RDDs in Python. Thanks to a library called [Py4J](https://github.com/bartdag/py4j) (<https://github.com/bartdag/py4j>), Python can interface with Java objects (in our case RDDs). Py4J is also one of the tools that makes PySpark work.

In this mission, we'll work with a data set containing the names of all of the guests who have appeared on [The Daily Show](https://en.wikipedia.org/wiki/The_Daily_Show).

To start off, we'll load the data set into an RDD. We're using the TSV version of [FiveThirtyEight's data set](https://github.com/fivethirtyeight/data/tree/master/daily-show-guests). TSV files use a tab character ("\t") as the delimiter, instead of the comma (",") that CSV files use.

## SparkContext

In Spark, the SparkContext object manages the connection to the clusters, and coordinates the running of processes on those clusters. More specifically, it connects to the cluster managers. The cluster managers control the executors that run the computations. Here's a diagram from the Spark documentation that will help you visualize the architecture:



We automatically have access to the SparkContext object sc. We then run the following code to read the TSV data set into an RDD object raw\_data:

raw\_data = sc.textFile("daily\_show.tsv")

The RDD object raw\_data closely resembles a list of string objects, with one object for each line in the data set. We then use the take() method to print the first five elements of the RDD:

raw\_data.take(5)

To explore the other methods an RDD object has access to, check out the [PySpark documentation](https://spark.apache.org/docs/1.1.1/api/python/pyspark.rdd.RDD-class.html#take) ([https://spark.apache.org/docs/1.1.1/api/python/pyspark.rdd.RDD-class.html#take](https://spark.apache.org/docs/1.1.1/api/python/pyspark.rdd.RDD-class.html%23take)). take(n) will return the first n elements of the RDD.

## Lazy Evaluation

You may be wondering why, if an RDD resembles a Python list, we don't just use bracket notation to access elements in the RDD.

The answer is that Spark distributes RDD objects across many partitions, and the RDD object is specifically designed to handle distributed data. We can't rely on the standard implementation of a list for these reasons.

Spark offers many advantages over regular Python, though. For example, thanks to RDD [abstraction](https://en.wikipedia.org/wiki/Abstraction_%28software_engineering%29) (<https://en.wikipedia.org/wiki/Abstraction_%28software_engineering%29> - a technique for hiding complexity of computer systems), you can run Spark locally on your own computer. Spark will simulate distributing your calculations over many machines by automatically slicing your computer's memory into partitions.

Spark's RDD implementation also lets us evaluate code "lazily," meaning we can postpone running a calculation until absolutely necessary. On the previous screen, Spark waited to load the TSV file into an RDD until raw\_data.take(5) executed. When our code called raw\_data = sc.textFile("dail\_show.tsv"), Spark created a pointer to the file, but didn't actually read it into raw\_data until raw\_data.take(5) needed that variable to run its logic.

The advantage of "lazy" evaluation is that we can build up a queue of tasks and let Spark optimize the overall workflow in the background. In regular Python, the interpreter can't do much workflow optimization. We'll see more examples of lazy evaluation later on.

## Pipelines

While Spark borrowed heavily from Hadoop's MapReduce pattern, it's still quite different in many ways. If you have experience with Hadoop and traditional MapReduce, you may want to read this great [post by Cloudera](http://blog.cloudera.com/blog/2014/09/how-to-translate-from-mapreduce-to-apache-spark/) (<http://blog.cloudera.com/blog/2014/09/how-to-translate-from-mapreduce-to-apache-spark/>) about the difference between them. Don't worry if you've never worked with MapReduce or Hadoop before; we'll cover the concepts you need to know in this course.

The key idea to understand when working with Spark is data **pipelining**. Every operation or calculation in Spark is essentially a series of steps that we can chain together and run in succession to form a **pipeline**. Each step in the **pipeline** returns either a Python value (such as an integer), a Python data structure (such as a dictionary), or an RDD object. We'll start with the map() function.

## Map()

The map(f) function applies the function f to every element in the RDD. Because RDDs are iterable objects (like most Python objects), Spark runs function f on each iteration and returns a new RDD.

We'll walk through an example of a map function so you can get a better sense of how it works. If you look carefully, you'll see that raw\_data is in a format that's hard to work with. While the elements are currently all strings, we'd like to convert each of them into a list to make the data more manageable. To do this the traditional way, we would:

1. Use a 'for' loop to iterate over the collection
2. Split each `string` on the delimiter
3. Store the result in a `list`

Let's see how we can use map to do this with Spark instead.

In the code cell:

1. Call the RDD function `map()` to specify we want to apply the logic in the parentheses to every line in our data set.
2. Write a lambda function that splits each line using the tab delimiter (\t), and assign the resulting RDD to `daily\_show`.
3. Call the RDD function `take()` on `daily\_show` to display the first five elements (or rows) of the resulting RDD.

We call the map(f) function a transformation step. It requires either a named or lambda function f.

## Python & Scala, Friends Forever

One of the wonderful features of PySpark is the ability to separate our logic - which we prefer to write in Python - from the actual data transformation. In the previous code cell, we wrote this lambda function in Python code:

raw\_data.map(lambda line: line.split('\t'))

Even though the function was in Python, we also took advantage of Scala when Spark actually ran the code over our RDD. **This** is the power of PySpark. Without learning any Scala, we get to harness the data processing performance gains from Spark's Scala architecture. Even better, when we ran the following code, it returned the results to us in Python-friendly notation:

daily\_show.take(5)

**Transformations and Actions**

There are two types of methods in Spark:

1. Transformations - map(), reduceByKey()
2. Actions - take(), reduce(), saveAsTextFile(), collect()

Transformations are lazy operations that always return a reference to an RDD object. Spark doesn't actually run the transformations, though, until an action needs to use the RDD resulting from a transformation. Any function that returns an RDD is a transformation, and any function that returns a value is an action. These concepts will become more clear as we work through this lesson and practice writing PySpark code.

**Immutability**

You may be wondering why we couldn't just split each string in place, instead of creating a new object daily\_show. In Python, we could have modified the collection element-by-element in place, without returning and assigning the results to a new object.

RDD objects are [immutable](https://www.quora.com/Why-is-a-spark-RDD-immutable), meaning that we can't change their values once we've created them. In Python, list and dictionary objects are mutable (we can change their values), while tuple objects are immutable. The only way to modify a tuple object in Python is to create a new tuple object with the necessary updates. Spark uses the immutability of RDDs to enhance calculation speeds. The mechanics of how it does this are outside the scope of this lesson.

## ReduceByKey()

We'd like to tally up the number of guests who have appeared on The Daily Show during each year. If daily\_show were a list of lists, we could write the following Python code to achieve this result:

tally = dict()

for line in daily\_show:

year = line[0]

if year in tally.keys():

tally[year] = tally[year] + 1

else:

tally[year] = 1

The keys in tally will be the years, and the values will be the totals for the number of lines associated with each year.

To achieve the same result with Spark, we'll have to use a Map step, then a ReduceByKey step.

## Explanation

You may have noticed that printing tally didn't return the histogram we were hoping for. Because of lazy evaluation, PySpark delayed executing the map and reduceByKey steps until we actually need them. Before we use take() to preview the first few elements in tally, we'll walk through the code we just wrote.

daily\_show.map(lambda x: (x[0], 1)).reduceByKey(lambda x, y: x+y)

During the map step, we used a lambda function to create a tuple consisting of:

* key: x[0] (the first value in the list)
* value: 1 (the integer)

Our high-level strategy was to create a tuple with the key representing the year, and the value representing 1. After running the map step, Spark will maintain in memory a list of tuples resembling the following:

('YEAR', 1)

('1991', 1)

('1991', 1)

('1991', 1)

('1991', 1)

...

We'd like to reduce that down to:

('YEAR', 1)

('1991', 4)

...

reduceByKey(f) combines tuples with the same key using the function we specify, f.

To see the results of these two steps, we'll use the take command, which forces lazy code to run immediately. Because tally is an RDD, we can't use Python's len function to find out how many elements are in the collection. Instead, we'll need to use the RDD count() function.

## Filter

Unlike pandas, Spark knows nothing about column headers, and didn't set them aside. We need a way to remove the element ('YEAR', 1) from our collection. We'll need a workaround, though, because RDD objects are immutable once we create them. The only way to remove that tuple is to create a new RDD object that doesn't have it.

Spark comes with a filter(f) function that creates a new RDD by filtering an existing one for specific criteria. If we specify a function f that returns a binary value, True or False, the resulting RDD will consist of elements where the function evaluated to True. You can read more about the filter function in the [Spark documentation](https://spark.apache.org/docs/1.1.1/api/python/pyspark.rdd.RDD-class.html#filter) ([https://spark.apache.org/docs/1.1.1/api/python/pyspark.rdd.RDD-class.html#filter](https://spark.apache.org/docs/1.1.1/api/python/pyspark.rdd.RDD-class.html%23filter)).

## Practice with Pipelines

To flex Spark's muscles, we'll demonstrate how to chain together a series of data transformations into a pipeline, and observe Spark managing everything in the background. The developers who wrote Spark had this functionality in mind, and optimized it for running tasks in succession.

Before Spark came along, running lots of tasks in succession in Hadoop was incredibly time consuming. Hadoop had to write intermediate results to disk, and wasn't aware of the full pipeline. Thanks to its aggressive approach to memory use and well-architected core, Spark improves on Hadoop's turnaround time significantly. If you're curious, you can read more about this topic in a [Quora thread](http://qr.ae/RHWrT2) (<https://www.quora.com/What-are-the-advantages-of-DAG-directed-acyclic-graph-execution-of-big-data-algorithms-over-MapReduce-I-know-that-Apache-Spark-Storm-and-Tez-use-the-DAG-execution-model-over-MapReduce-Why-Are-there-any-disadvantages/answer/Tathagata-Das?share=1&srid=umKP>).

In the following code cell, we'll filter out actors for whom the profession is blank, lowercase each profession, generate a histogram of professions, and output the first five tuples in the histogram.

# Spark Installation & Jupyter Notebook Integration

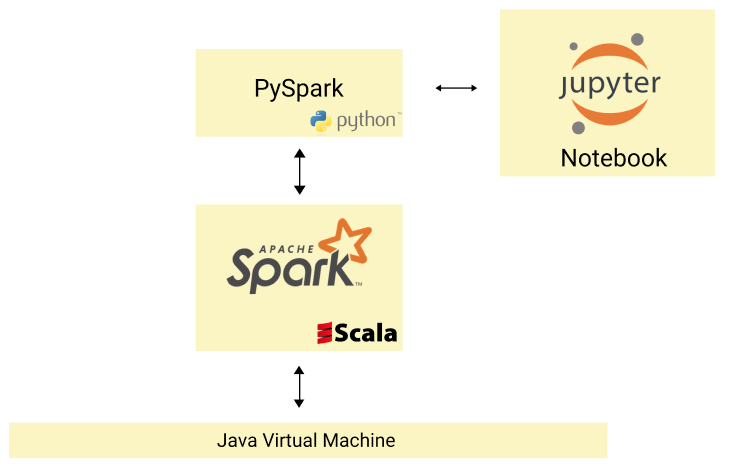
## Introduction

In the last mission, we introduced the Spark cluster computing framework and explored some basic PySpark methods, all within the Dataquest interface. In this project, we'll walk through how to set up Spark on your own computer and integrate PySpark with Jupyter Notebook. We can use Spark in two modes:

* **Local mode** - The entire Spark application runs on a single machine. Local mode is what you'll use to prototype Spark code on your own computer. It's also easier to set up.
* **Cluster mode** - The Spark application runs across multiple machines. Cluster mode is what you'll use when you want to run your Spark application across multiple machines in a cloud environment like Amazon Web Services, Microsoft Azure, or Digital Ocean.

For now, we'll walk through the instructions for installing Spark in local mode on Windows, Mac, and Linux. We'll cover how to install Spark in cluster mode as part of the data engineering track.

Here's a diagram describing the high-level components you'll be setting up today:



## Java

Spark runs on the Java Virtual Machine, or JVM for short, which comes in the Java SE Development Kit (JDK for short). We recommend installing Java SE Development Kit version 7 or higher, which you can download from Oracle’s website:

* <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

As of this writing, Java SE Development Kit 8u111 and 8u112 are the two latest releases of the JDK. Any version after JDK 7 works, so you can download any of the versions on this page. Select the appropriate installation file for your operating system.

If you're on Windows or Linux, be sure to choose the correct instruction set architecture (x86 or x64) for your computer. Each computer chip has a specific instruction set architecture that determines the maximum amount of memory it can work with. The two main types are x86 (32 bit) and x64 (64-bit). If you're not sure which one your computer has, you can find out by following [this guide if you're on Windows](http://support.wdc.com/KnowledgeBase/answer.aspx?ID=9405) (<https://support.wdc.com/knowledgebase/answer.aspx?ID=9405>) or [this one if you're on Linux](http://www.howtogeek.com/198615/how-to-check-if-your-linux-system-is-32-bit-or-64-bit/) (<https://www.howtogeek.com/198615/how-to-check-if-your-linux-system-is-32-bit-or-64-bit/>).

To verify that the installation worked, launch your command line application (**Command Prompt** for Windows and **Terminal** for Mac and Linux) and run:

java –version

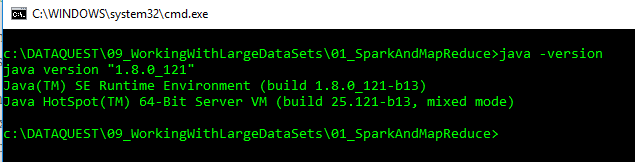
The output should be similar to:

java version "1.7.0\_79"

Java(TM) SE Runtime Environment (build 1.7.0\_79-b15)

Java HotSpot(TM) 64-Bit Server VM (build 24.79-b02, mixed mode)

My output:

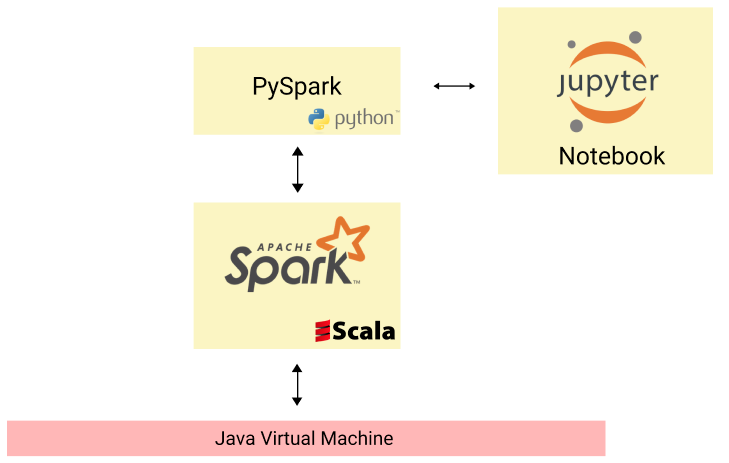


While the exact numbers probably won't match, the key thing to verify is that the version is larger than 1.7. This number actually represents Version 7. If you're interested, you can read why at [Oracle's website](http://www.oracle.com/technetwork/java/javase/jdk7-naming-418744.html) (<http://www.oracle.com/technetwork/java/javase/jdk7-naming-418744.html>).

If running java -version returned an error or a different version than the one you just installed, your Java JDK installation most likely wasn't added to your PATH properly. Read this [post](http://cloudlink.soasta.com/t5/CloudTest-Knowledge-Base/Adding-JDK-Path-in-Mac-OS-X-Linux-or-Windows/ta-p/43867) (<https://community.akamai.com/community/web-performance/blog/2017/07/19/welcome-to-the-akamai-community>) to learn more about how to properly add the Java executable to your PATH.

Now that we have the JVM set up, let's move on to Spark.

*Note, I already did this when learning Java, if you’re doing this again and struggling, see my early notes from Java, I’m not positive, but I’ll bet there is some information there on Java installation in a windows system.*



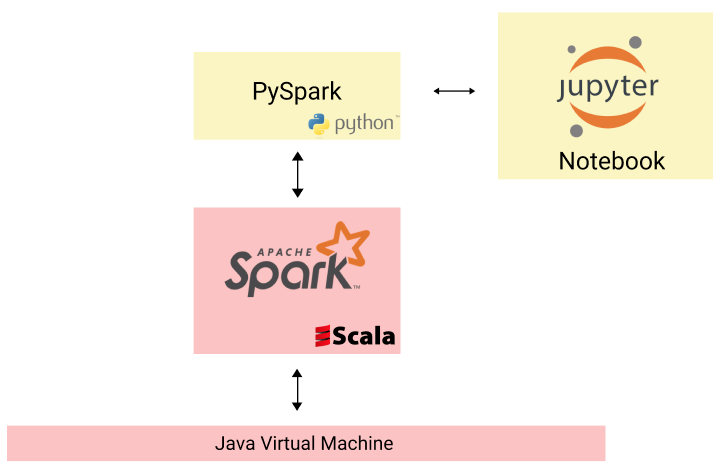
## Spark

Because you've installed JDK, you could technically download the original source code and build Spark on your computer. Building from the source code is the process of generating an executable program for your machine. It involves [many steps](http://stackoverflow.com/a/1622520) ([https://stackoverflow.com/questions/1622506/programming-definitions-what-exactly-is-building/1622520#1622520](https://stackoverflow.com/questions/1622506/programming-definitions-what-exactly-is-building/1622520%231622520)). While there are some performance benefits to building Spark from source, it takes a while to do, and it's hard to debug if the build fails.

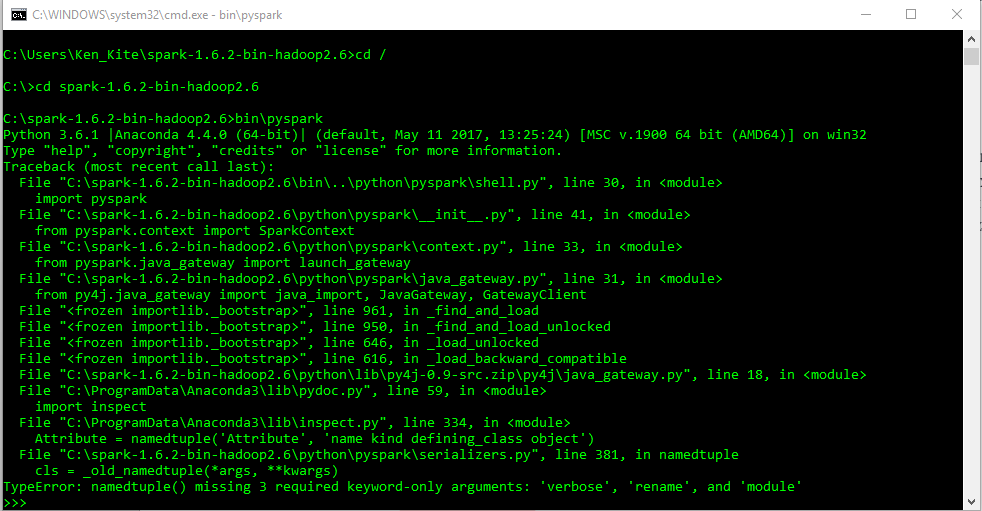
We'll download and work with a pre-built version of Spark instead. Navigate to the [Spark downloads page](http://spark.apache.org/downloads.html) (<http://spark.apache.org/downloads.html>) and select the following options:

1. **1.6.2**
2. **Pre-built for Hadoop 2.6**
3. **Direct Download**

Next, click the link that appears in Step 4 to download Spark as a .TGZ file to your computer. Open your command line application and navigate to the folder you downloaded it to. Unzip the file and move the resulting folder into your home directory. Windows does not have a built in utility that can unzip tgz files - we recommend downloading and using [7-Zip](http://www.7-zip.org/). Once you have unzipped the file, move the resulting folder into your home directory (c:\Users\xxxx\).



Note, it appears that spark-1.6.2-bin-hadoop2.6 is not compatible with Python 3.6.1 |Anaconda 4.4.0 (64-bit) in windows. I am seeing the following error:



Tried using spark-2.2.0-bin-hadoop2.7 and when I run:

bin\pyspark

I get the following…

