**Using Spark in Python**

The first step in using Spark is connecting to a cluster.

In practice, the cluster will be hosted on a remote machine that's connected to all other nodes. There will be one computer, called the *master* that manages splitting up the data and the computations. The master is connected to the rest of the computers in the cluster, which are called *worker*. The master sends the workers data and calculations to run, and they send their results back to the master.

When you're just getting started with Spark it's simpler to just run a cluster locally. Thus, for this course, instead of connecting to another computer, all computations will be run on DataCamp's servers in a simulated cluster.

Creating the connection is as simple as creating an instance of the SparkContext class. The class constructor takes a few optional arguments that allow you to specify the attributes of the cluster you're connecting to.

An object holding all these attributes can be created with the SparkConf() constructor. Take a look at the [**documentation**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.html) for all the details!

For the rest of this course you'll have a SparkContext called sc already available in your workspace.

Note: You can connect to a cluster by creating instance of SparkContent()

# Examining The SparkContext

In this exercise you'll get familiar with the SparkContext.

You'll probably notice that code takes longer to run than you might expect. This is because Spark is some serious software. It takes more time to start up than you might be used to. You may also find that running simpler computations might take longer than expected. That's because all the optimizations that Spark has under its hood are designed for complicated operations with big data sets. That means that for simple or small problems Spark may actually perform worse than some other solutions!

Get to know the SparkContext.

* Call print() on sc to verify there's a SparkContext in your environment.
* print() sc.version to see what version of Spark is running on your cluster.

# Using DataFrames

Spark's core data structure is the Resilient Distributed Dataset (RDD). This is a low level object that lets Spark work its magic by splitting data across multiple nodes in the cluster. However, RDDs are hard to work with directly, so in this course you'll be using the **Spark DataFrame abstraction** built on top of RDDs.

The Spark DataFrame was designed to behave a lot like a SQL table (a table with variables in the columns and observations in the rows). Not only are they easier to understand, DataFrames are also more optimized for complicated operations than RDDs.

When you start modifying and combining columns and rows of data, there are many ways to arrive at the same result, but some often take much longer than others. When using RDDs, it's up to the data scientist to figure out the right way to optimize the query, but the DataFrame implementation has much of this optimization built in!

To start working with Spark DataFrames, you first have to create a SparkSession object from your SparkContext. You can think of the SparkContext as your connection to the cluster and the SparkSession as your interface with that connection.

Remember, for the rest of this course you'll have a SparkSession called spark available in your workspace!

Note: SparkContext you connect with the cluster, and you use SparkSession to interface with that connection. To work with DataFrames you have to create a SparkSession from your SparkContext.

# Creating a SparkSession

We've already created a SparkSession for you called spark, but what if you're not sure there already is one? Creating multiple SparkSessions and SparkContexts can cause issues, so it's best practice to use the SparkSession.builder.getOrCreate()method. This returns an existing SparkSession if there's already one in the environment, or creates a new one if necessary!

1. Import SparkSession from pyspark.sql.
2. Make a new SparkSession called my\_spark using SparkSession.builder.getOrCreate().
3. Print my\_spark to the console to verify it's a SparkSession

# Viewing tables

Once you've created a SparkSession, you can start poking around to see what data is in your cluster!

Your SparkSession has an attribute called catalog which lists all the data inside the cluster. This attribute has a few methods for extracting different pieces of information.

One of the most useful is the .listTables() method, which returns the names of all the tables in your cluster as a list.

Note: .catalog tells all the data inside a cluster and it has an attribute called .listTables() which returns names of tables in your cluster

# Are you query-ious?

One of the advantages of the DataFrame interface is that you can run SQL queries on the tables in your Spark cluster. If you don't have any experience with SQL, don't worry, we'll provide you with queries! (To learn more SQL, start with our [**Introduction to SQL**](https://www.datacamp.com/courses/intro-to-sql-for-data-science) course.)

As you saw in the last exercise, one of the tables in your cluster is the flights table. This table contains a row for every flight that left Portland International Airport (PDX) or Seattle-Tacoma International Airport (SEA) in 2014 and 2015.

Running a query on this table is as easy as using the .sql()method on your SparkSession. This method takes a string containing the query and returns a DataFrame with the results!

If you look closely, you'll notice that the table flights is only mentioned in the query, not as an argument to any of the methods. This is because there isn't a local object in your environment that holds that data, so it wouldn't make sense to pass the table as an argument.

Remember, we've already created a SparkSession called sparkin your workspace. (It's no longer called my\_spark because we created it for you!)

##### Instructions

**100 XP**

* Use the .sql() method to get the first 10 rows of the flights table and save the result to flights10. The variable query contains the appropriate SQL query.
* Use the DataFrame method .show() to print flights10.

# Don't change this query

query = "FROM flights SELECT \* LIMIT 10"

# Get the first 10 rows of flights

flights10 = spark.sql(query)

# Show the results

flights10.show()

# Pandafy a Spark DataFrame

Suppose you've run a query on your huge dataset and aggregated it down to something a little more manageable.

Sometimes it makes sense to then take that table and work with it locally using a tool like pandas. Spark DataFrames make that easy with the .toPandas() method. Calling this method on a Spark DataFrame returns the corresponding pandas DataFrame. It's as simple as that!

This time the query counts the number of flights to each airport from SEA and PDX.

Remember, there's already a SparkSession called spark in your workspace!

##### Instructions

**100 XP**

* Run the query using the .sql() method. Save the result in flight\_counts.
* Use the .toPandas() method on flight\_counts to create a pandas DataFrame called pd\_counts.
* Print the .head() of pd\_counts to the console.

# Don't change this query

query = "SELECT origin, dest, COUNT(\*) as N FROM flights GROUP BY origin, dest"

# Run the query

flight\_counts = spark.sql(query)

# Convert the results to a pandas DataFrame

pd\_counts = flight\_counts.toPandas()

# Print the head of pd\_counts

print(pd\_counts.head())

# Put some Spark in your data

In the last exercise, you saw how to move data from Spark to pandas. However, maybe you want to go the other direction, and put a pandas DataFrame into a Spark cluster! The SparkSession class has a method for this as well.

The .createDataFrame() method takes a pandas DataFrame and returns a Spark DataFrame.

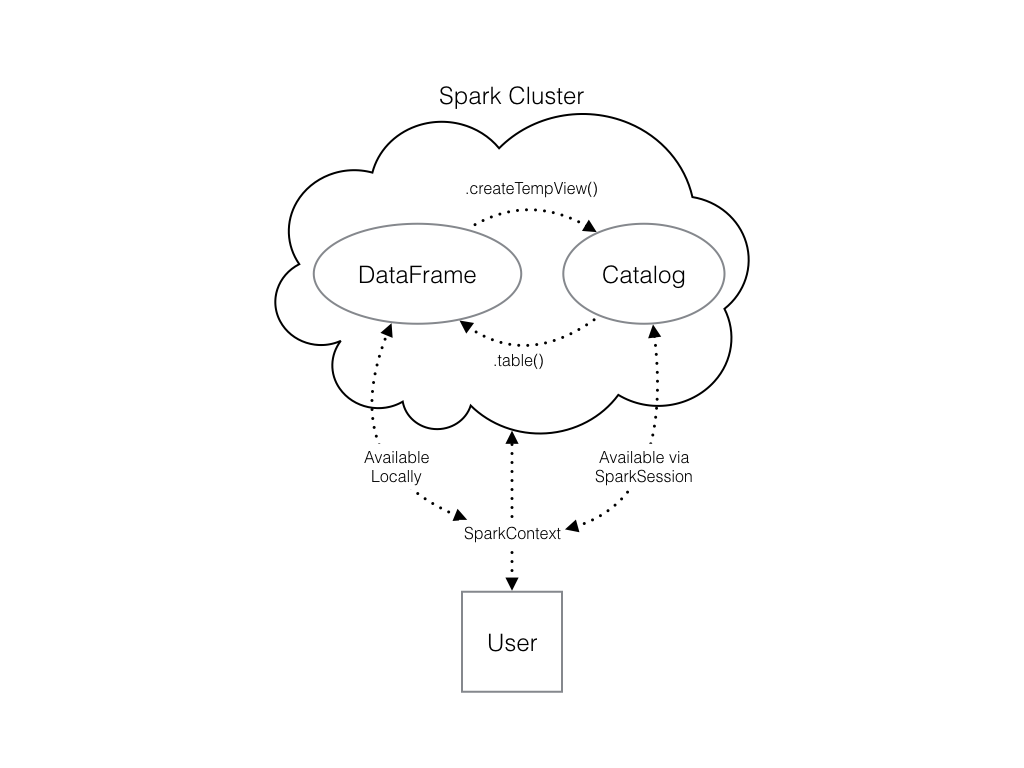
The output of this method is stored locally, not in the SparkSession catalog. This means that you can use all the Spark DataFrame methods on it, but you can't access the data in other contexts.

For example, a SQL query (using the .sql() method) that references your DataFrame will throw an error. To access the data in this way, you have to save it as a temporary table.

You can do this using the .createTempView() Spark DataFrame method, which takes as its only argument the name of the temporary table you'd like to register. This method registers the DataFrame as a table in the catalog, but as this table is temporary, it can only be accessed from the specific SparkSession used to create the Spark DataFrame.

There is also the method .createOrReplaceTempView(). This safely creates a new temporary table if nothing was there before, or updates an existing table if one was already defined. You'll use this method to avoid running into problems with duplicate tables.

Check out the diagram to see all the different ways your Spark data structures interact with each other.



There's already a SparkSession called spark in your workspace, numpy has been imported as np, and pandas as pd.

##### Instructions

**100 XP**

##### Instructions

**100 XP**

* The code to create a pandas DataFrame of random numbers has already been provided and saved under pd\_temp.
* Create a Spark DataFrame called spark\_temp by calling the .createDataFrame() method with pd\_temp as the argument.
* Examine the list of tables in your Spark cluster and verify that the new DataFrame is not present. Remember you can use spark.catalog.listTables() to do so.
* Register spark\_temp as a temporary table named "temp"using the .createOrReplaceTempView() method. Remember that the table name is set including it as the only argument!
* Examine the list of tables again!

# Create pd\_temp

pd\_temp = pd.DataFrame(np.random.random(10))

# Create spark\_temp from pd\_temp

spark\_temp = spark.createDataFrame(pd\_temp)

# Examine the tables in the catalog

print(spark.catalog.listTables())

# Add spark\_temp to the catalog

spark\_temp.createOrReplaceTempView("temp")

# Examine the tables in the catalog again

print(spark.catalog.listTables())

# Dropping the middle man

Now you know how to put data into Spark via pandas, but you're probably wondering why deal with pandas at all? Wouldn't it be easier to just read a text file straight into Spark? Of course it would!

Luckily, your SparkSession has a .read attribute which has several methods for reading different data sources into Spark DataFrames. Using these you can create a DataFrame from a .csv file just like with regular pandas DataFrames!

The variable file\_path is a string with the path to the file airports.csv. This file contains information about different airports all over the world.

A SparkSession named spark is available in your workspace.

##### Instructions

**100 XP**

##### Instructions

**100 XP**

* Use the .read.csv() method to create a Spark DataFrame called airports
  + The first argument is file\_path
  + Pass the argument header=True so that Spark knows to take the column names from the first line of the file.
* Print out this DataFrame by calling .show().

# Don't change this file path

file\_path = "/usr/local/share/datasets/airports.csv"

# Read in the airports data

airports = spark.read.csv(file\_path,header=True)

# Show the data

airports.show()