# Spark DataFrame Basics

Spark DataFrames are the workhouse and main way of working with Spark and Python post Spark 2.0. DataFrames act as powerful versions of tables, with rows and columns, easily handling large datasets. The shift to DataFrames provides many advantages:

* A much simpler syntax
* Ability to use SQL directly in the dataframe
* Operations are automatically distributed across RDDs

If you've used R or even the pandas library with Python you are probably already familiar with the concept of DataFrames. Spark DataFrame expand on a lot of these concepts, allowing you to transfer that knowledge easily by understanding the simple syntax of Spark DataFrames. Remember that the main advantage to using Spark DataFrames vs those other programs is that Spark can handle data across many RDDs, huge data sets that would never fit on a single computer. That comes at a slight cost of some "peculiar" syntax choices, but after this course you will feel very comfortable with all those topics!

## Creating a DataFrame

First we need to start a SparkSession:

**from** **pyspark.sql** **import** SparkSession

Then start the SparkSession

*# May take a little while on a local computer*

spark = SparkSession.builder.appName("Basics").getOrCreate()

You will first need to get the data from a file (or connect to a large distributed file like HDFS, we'll talk about this later once we move to larger datasets on AWS EC2).

*# We'll discuss how to read other options later.* *# This dataset is from Spark's examples* *# Might be a little slow locally* df = spark.read.json('people.json')

*# showing data; Note how data is missing!*

df.show()

df.printSchema()

root

|-- age: long (nullable = true)

|-- name: string (nullable = true)

df.columns

df.describe()

df.describe().show()

Some data types make it easier to infer schema (like tabular formats such as csv which we will show later).

However you often have to set the schema yourself if you aren't dealing with a. read method that doesn't have inferSchema() built-in.

Spark has all the tools you need for this; it just requires a very specific structure:

**from** **pyspark.sql.types** **import** StructField,StringType,IntegerType,StructType

Next we need to create the list of Structure fields

\*:param name: string, name of the field.

\*:param dataType: :class:`DataType` of the field.

\*:param nullable: boolean, whether the field can be null (None) or not

data\_schema = [StructField("age", IntegerType(), **True**),StructField("name", StringType(), **True**)]

final\_struc = StructType(fields=data\_schema)

df = spark.read.json('people.json', schema=final\_struc)

df.printSchema()

root

|-- age: integer (nullable = true)

|-- name: string (nullable = true)

**Grabbing the data**

df['age']

type(df['age'])

df.select('age')

type(df.select('age'))

df.select('age').show()

df.head(2)

**Creating new columns**

*# Adding a new column with a simple copy*

df.withColumn('newage',df['age']).show()

df.show()

*# Simple Rename*

df.withColumnRenamed('age','supernewage').show()

df.withColumn('doubleage',df['age']\*2).show()

df.withColumn('add\_one\_age',df['age']+1).show()

df.withColumn('half\_age',df['age']/2).show()

### Using SQL

To use SQL queries directly with the dataframe, you will need to register it to a temporary view:

*# Register the DataFrame as a SQL temporary view*

df.createOrReplaceTempView("people")

sql\_results = spark.sql("SELECT \* FROM people")

sql\_results

sql\_results.show()

spark.sql("SELECT \* FROM people WHERE age=30").show()

# Basic Operations

This lecture will cover some basic operations with Spark DataFrames.

We will play around with some stock data from Apple.

**from** **pyspark.sql** **import** SparkSession

*# May take awhile locally*

spark = SparkSession.builder.appName("Operations").getOrCreate()

*# Let Spark know about the header and infer the Schema types!*

df = spark.read.csv('appl\_stock.csv',inferSchema=**True**,header=**True**)

df.printSchema()

root

|-- Date: timestamp (nullable = true)

|-- Open: double (nullable = true)

|-- High: double (nullable = true)

|-- Low: double (nullable = true)

|-- Close: double (nullable = true)

|-- Volume: integer (nullable = true)

|-- Adj Close: double (nullable = true)

## Filtering Data

A large part of working with DataFrames is the ability to quickly filter out data based on conditions. Spark DataFrames are built on top of the Spark SQL platform, which means that is you already know SQL, you can quickly and easily grab that data using SQL commands, or using the DataFram methods (which is what we focus on in this course).

*# Using SQL*

df.filter("Close<500").show()

+--------------------+------------------+------------------+------------------+------------------+---------+------------------+ | Date| Open| High| Low| Close| Volume| Adj Close| +--------------------+------------------+------------------+------------------+------------------+---------+------------------+ |2010-01-04 00:00:...| 213.429998| 214.499996|212.38000099999996| 214.009998|123432400| 27.727039| |2010-01-05 00:00:...| 214.599998| 215.589994| 213.249994| 214.379993|150476200|27.774976000000002| |2010-01-06 00:00:...| 214.379993| 215.23| 210.750004| 210.969995|138040000|27.333178000000004| |2010-01-07 00:00:...| 211.75| 212.000006| 209.050005| 210.58|119282800| 27.28265|

*# Using SQL with .select()*

df.filter("Close<500").select('Open').show()

*# Using SQL with .select()*

df.filter("Close<500").select(['Open','Close']).show()

df.filter(df["Close"] < 200).show()

*# Make sure to add in the parenthesis separating the statements!* df.filter( (df["Close"] < 200) & (df['Open'] > 200) ).show()

*# Make sure to add in the parenthesis separating the statements!*

df.filter( (df["Close"] < 200) | (df['Open'] > 200) ).show()

*# Make sure to add in the parenthesis separating the statements!* df.filter( (df["Close"] < 200) & ~(df['Open'] < 200) ).show()

df.filter(df["Low"] == 197.16).show()

*# Collecting results as Python objects*

df.filter(df["Low"] == 197.16).collect()

result = df.filter(df["Low"] == 197.16).collect()

*# Note the nested structure returns a nested row object*

type(result[0])

pyspark.sql.types.Row

row = result[0]

row.asDict()

{'Adj Close': 25.620401,

'Close': 197.75,

'Date': datetime.datetime(2010, 1, 22, 0, 0),

'High': 207.499996,

'Low': 197.16,

'Open': 206.78000600000001,

'Volume': 220441900}

**for** item **in** result[0]:

print(item)

2010-01-22 00:00:00

206.78000600000001

207.499996

197.16

197.75

220441900

25.620401

# GroupBy and Aggregate Functions

Let's learn how to use GroupBy and Aggregate methods on a DataFrame. GroupBy allows you to group rows together based off some column value, for example, you could group together sales data by the day the sale occured, or group repeast customer data based off the name of the customer. Once you've performed the GroupBy operation you can use an aggregate function off that data. An aggregate function aggregates multiple rows of data into a single output, such as taking the sum of inputs, or counting the number of inputs.

**from** **pyspark.sql** **import** SparkSession

*# May take a while locally*

spark = SparkSession.builder.appName("Operations").getOrCreate()

*# Let Spark know about the header and infer the Schema types!*

df = spark.read.csv('appl\_stock.csv',inferSchema=**True**,header=**True**)

df.groupBy("Company")

*# Mean* df.groupBy("Company").mean().show()

*# Count* df.groupBy("Company").count().show()

*# Max sales across everything*

df.agg({'Sales':'max'}).show()

grouped = df.groupBy("Company")

grouped.agg({"Sales":'max'}).show()

*# OrderBy*

*# Ascending*

df.orderBy("Sales").show()

*# Descending call off the column itself.* df.orderBy(df["Sales"].desc()).show()

**Drop the missing data**

*# Drop any row that contains missing data*

df.na.drop().show()

*# Has to have at least 2 NON-null values*

df.na.drop(thresh=2).show()

df.na.drop(subset=["Sales"]).show()

df.na.drop(how='any').show()

f.na.drop(how='all').show()

## Fill the missing values

We can also fill the missing values with new values. If you have multiple nulls across multiple data types, Spark is actually smart enough to match up the data types. For example:

df.na.fill('NEW VALUE').show()

df.na.fill(0).show()

df.na.fill('No Name',subset=['Name']).show()

**from** **pyspark.sql.functions** **import** mean mean\_val = df.select(mean(df['Sales'])).collect() *# Weird nested formatting of Row object!* mean\_val[0][0]

mean\_sales = mean\_val[0][0]

df.na.fill(mean\_sales,["Sales"]).show()

*# One (very ugly) one-liner*

df.na.fill(df.select(mean(df['Sales'])).collect()[0][0],['Sales']).show()

**Bonus Codes!**

*# Uh oh Strings!*

df.describe().printSchema()

**from** **pyspark.sql.functions** **import** format\_number

result = df.describe()

result.select(result['summary'],

format\_number(result['Open'].cast('float'),2).alias('Open'),

format\_number(result['High'].cast('float'),2).alias('High'),

format\_number(result['Low'].cast('float'),2).alias('Low'),

format\_number(result['Close'].cast('float'),2).alias('Close'),

result['Volume'].cast('int').alias('Volume')

).show()

**Create a new dataframe with a column called HV Ratio that is the ratio of the High Price versus volume of stock traded for a day.**

df2 = df.withColumn("HV Ratio",df["High"]/df["Volume"])*#.show()* *# df2.show()* df2.select('HV Ratio').show()

**What day had the Peak High in Price?**

*# Didn't need to really do this much indexing* *# Could have just shown the entire row*

df.orderBy(df["High"].desc()).head(1)[0][0]

*# Also could have gotten this from describe()*

**from** **pyspark.sql.functions** **import** mean

df.select(mean("Close")).show()

#### max and min of the Volume column:

*# Could have also used describe*

**from** **pyspark.sql.functions** **import** max,min

df.select(max("Volume"),min("Volume")).show()

**How many days was the Close lower than 60 dollars?**

df.filter("Close < 60").count()

df.filter(df['Close'] < 60).count()

**from** **pyspark.sql.functions** **import** corr

df.select(corr("High","Volume")).show()

**from** **pyspark.sql.functions** **import** year

yeardf = df.withColumn("Year",year(df["Date"]))

max\_df = yeardf.groupBy('Year').max()

*# 2015*

max\_df.select('Year','max(High)').show()

#### What is the average Close for each Calendar Month?

#### In other words, across all the years, what is the average Close price for Jan,Feb, Mar, etc... Your result will have a value for each of these months.

**from** **pyspark.sql.functions** **import** month

monthdf = df.withColumn("Month",month("Date"))

monthavgs = monthdf.select("Month","Close").groupBy("Month").mean()

monthavgs.select("Month","avg(Close)").orderBy('Month').show()