



# Big Data

PySpark

Instructor: Trong-Hop Do

April 24<sup>th</sup> 2021

**S<sup>3</sup>Lab**

*Smart Software System Laboratory*



“Big data is at the foundation of all the megatrends that are happening today, from social to mobile to cloud to gaming.”


– Chris Lynch, Vertica Systems

# Install Spark on Windows



## Install Java 8 or Later

- To install Apache Spark on windows, you would need Java 8 or later version hence download the Java version from Oracle and install it on your system.
- <https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html>

Windows x64	166.79 MB	 <a href="#">jdk-8u271-windows-x64.exe</a>
-------------	-----------	--

# Apache Spark Installation on Windows

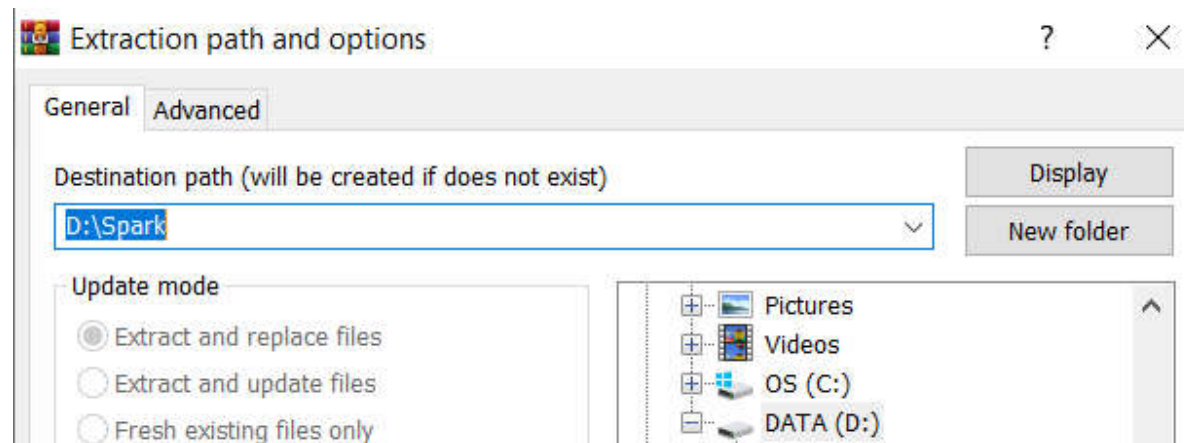
- Download Apache spark
- <https://spark.apache.org/downloads.html>

## Download Apache Spark™

1. Choose a Spark release:
2. Choose a package type:
3. Download Spark: [spark-3.0.1-bin-hadoop2.7.tgz](#)
4. Verify this release using the 3.0.1 [signatures](#), [checksums](#) and [project release KEYS](#).

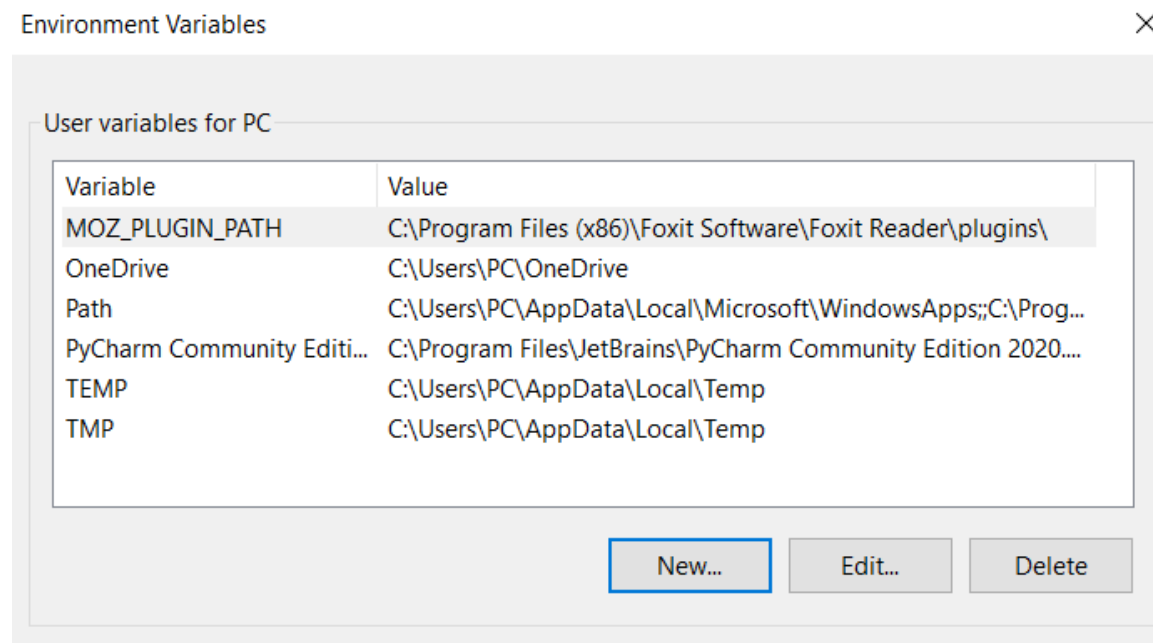
# Apache Spark Installation on Windows

- Extract the zip file to any folder

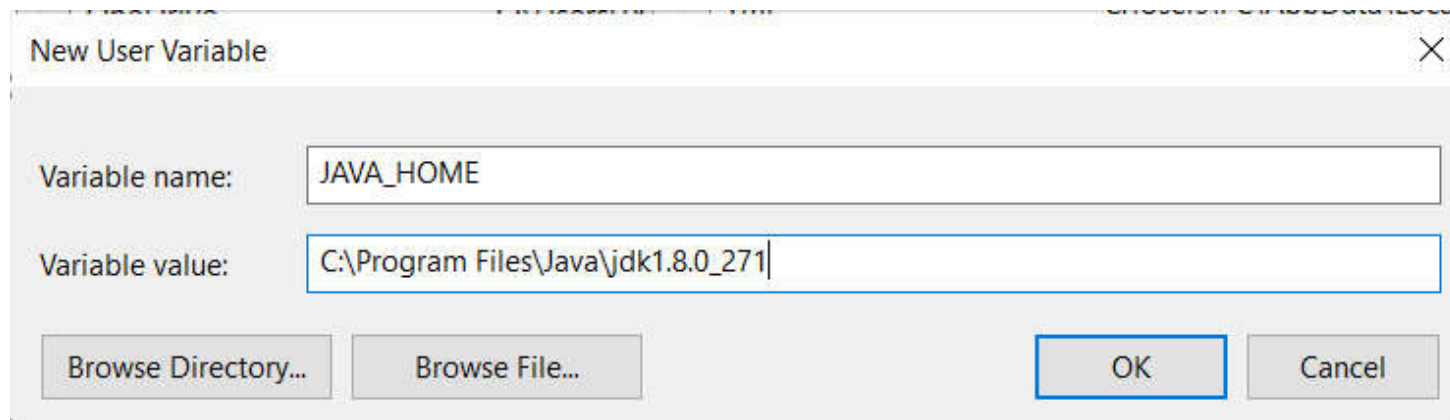


# Environment Variables Setting

- Open System Environment Variables window and select Environment Variables.

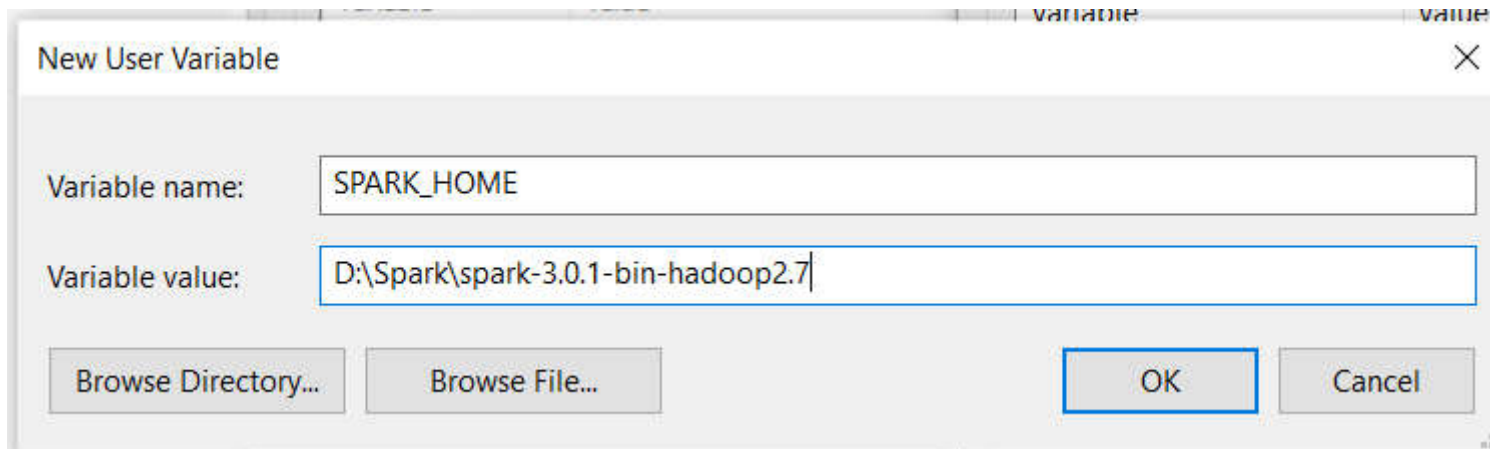


# Environment Variables Setting





# Apache Spark Installation on Windows

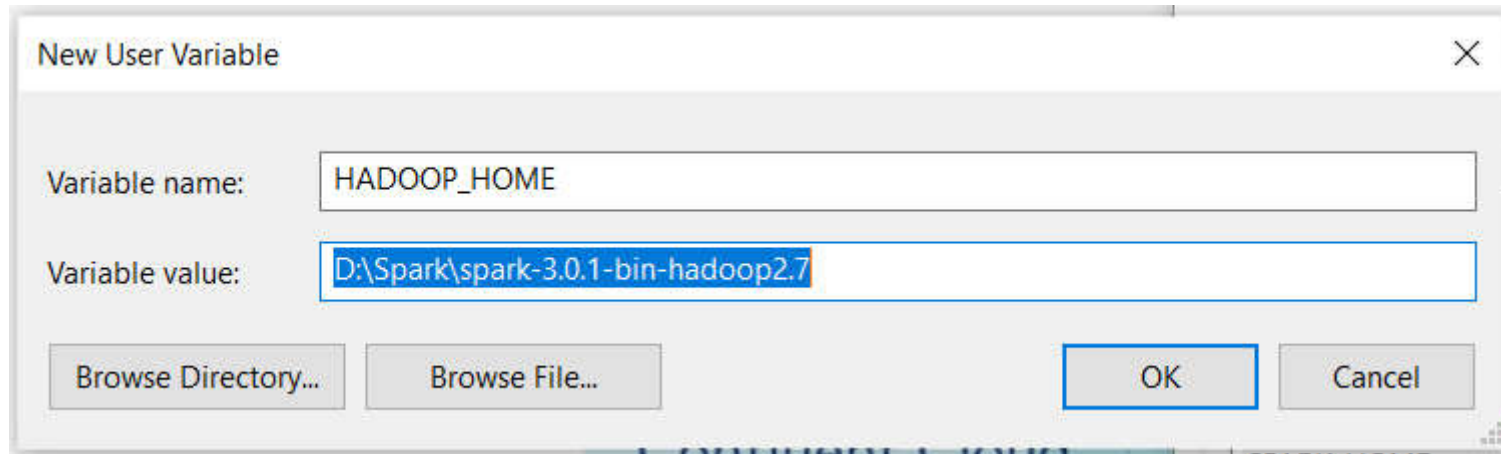


The screenshot shows a 'New User Variable' dialog box. It has two input fields: 'Variable name' with the text 'SPARK\_HOME' and 'Variable value' with the text 'D:\Spark\spark-3.0.1-bin-hadoop2.7'. Below these fields are four buttons: 'Browse Directory...', 'Browse File...', 'OK', and 'Cancel'. The 'OK' button is highlighted with a blue border. The dialog box has a title bar with 'New User Variable' and a close button (X) in the top right corner.

	variable	value
Variable name:	SPARK_HOME	
Variable value:	D:\Spark\spark-3.0.1-bin-hadoop2.7	

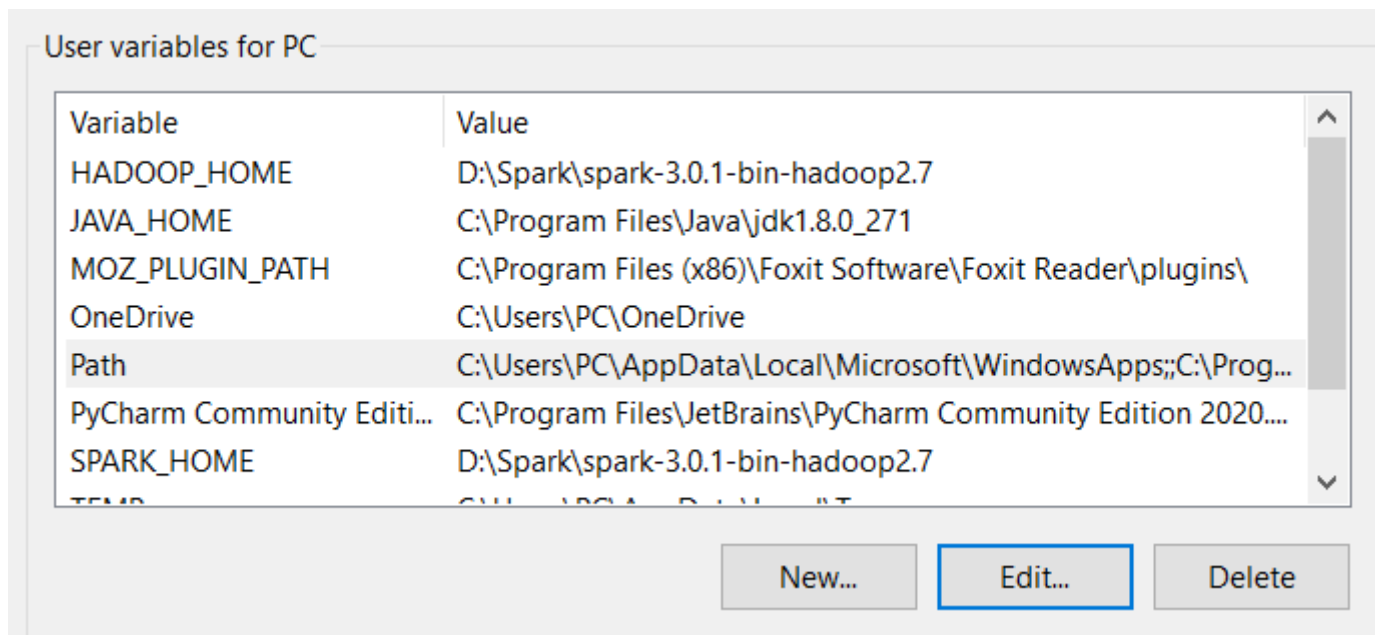
Browse Directory... Browse File... OK Cancel

# Apache Spark Installation on Windows



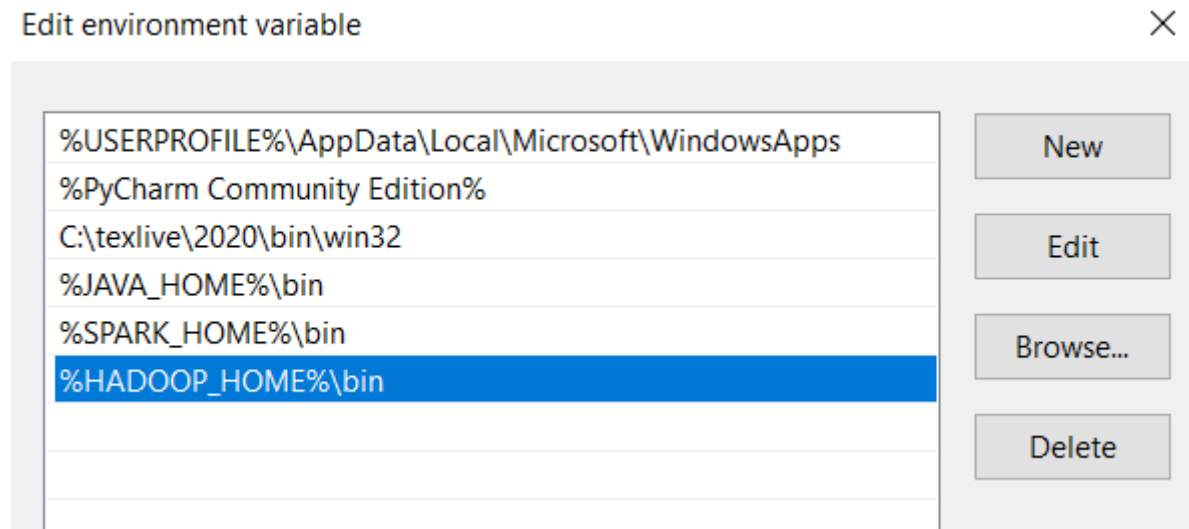
# Apache Spark Installation on Windows

- Now Edit the PATH variable



# Apache Spark Installation on Windows

- Add Spark, Java, and Hadoop bin location by selecting New option.



```
C:\Users\PC>spark-shell
00/12/07 16:50:25 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://192.168.56.1:4040
Spark context available as 'sc' (master = local[*], app id = local-1607334630777).
Spark session available as 'spark'.
Welcome to

      / _ \   / _ \   / _ \   / _ \
     / _ \ V _/ V _/ V _/ V _/
    / _ \|_/_/_/_/_/_/_/_/_/_/_/_ version 3.0.1
     / _ \
      / _ \

Using Scala version 2.12.10 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_271)
Type in expressions to have them evaluated.
Type :help for more information.

scala> 00/12/07 16:50:45 WARN ProcsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of ProcessTree metrics is stopped

scala>
```

Type **pyspark** on command prompt

```
C:\Users\PC>pyspark
Python 3.9.5 (tags/v3.9.5:0a7dcdb, May 3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
21/05/09 15:58:47 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java cl
asses where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to

  ____
 /  _ \
/_/_/  \_/_/  version 3.0.1

Using Python version 3.9.5 (tags/v3.9.5:0a7dcdb, May 3 2021 17:27:52)
SparkSession available as 'spark'.
>>> 21/05/09 15:59:03 WARN ProcfsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of Proc
essTree metrics is stopped

>>>
```

## Run PySpark on Jupyter lab

- Open Anaconda prompt and type **“python -m pip install findspark”**. This package is necessary to run spark from Jupyter notebook.

```
(base) D:\Spark\spark-3.0.1-bin-hadoop2.7>python -m pip install findspark
Collecting findspark
  Downloading findspark-1.4.2-py2.py3-none-any.whl (4.2 kB)
Installing collected packages: findspark
Successfully installed findspark-1.4.2
```

# Run PySpark on Jupyter lab

- Open jupyter notebook
- New -> Python 3

```
In [1]: import findspark  
findspark.init()
```

```
In [2]: import pyspark  
  
from pyspark.sql import SparkSession  
spark = SparkSession.builder.getOrCreate()  
df = spark.sql("select 'spark' as hello ")  
df.show()
```

```
+-----+  
|hello|  
+-----+  
|spark|  
+-----+
```



# Big Data Analytics with PySpark SQL



# What is PySpark

PySpark is a Spark library written in Python to run Python application using Apache Spark capabilities, using PySpark we can run applications parallelly on the distributed cluster (multiple nodes).

In other words, PySpark is a Python API for Apache Spark. Apache Spark is an analytical processing engine for large scale powerful distributed data processing and machine learning applications.



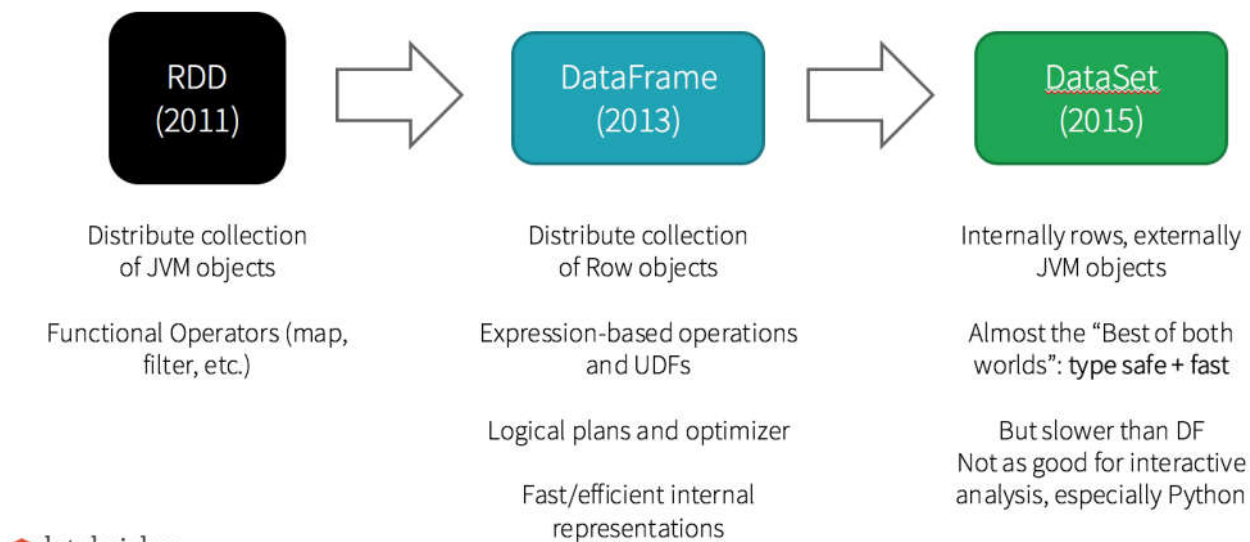
source: <https://databricks.com/>

# PySpark Modules and Packages

- PySpark RDD ([pyspark.RDD](#))
- PySpark DataFrame and SQL ([pyspark.sql](#))
- PySpark Streaming ([pyspark.streaming](#))
- PySpark MLib ([pyspark.ml](#), [pyspark.mllib](#))
- PySpark GraphFrames ([GraphFrames](#))
- PySpark Resource ([pyspark.resource](#)) It's new in PySpark 3.0

# RDD vs DataFrame vs DataSet

## History of Spark APIs



In version 2.0, DataSet and DataFrame APIs are unified to provide a single API for developers. A DataFrame is a specific Dataset[T], where T=Row type, so DataFrame shares the same methods as Dataset.

## RDD vs DataFrame vs DataSet

Feature	RDD	DataFrame	DataSet
Immutable	Yes	Yes	Yes
Fault tolerant	Yes	Yes	Yes
Type-safe	Yes	No	Yes
Schema	No	Yes	Yes
Execution optimization	No	Yes	Yes
Level	Low	High	High

## What is SparkSession?

- Since Spark 2.0 SparkSession has become an entry point to PySpark to work with RDD, DataFrame. Prior to 2.0, SparkContext used to be an entry point.
- Spark Session also includes all the APIs available in different contexts –
  - Spark Context,
  - SQL Context,
  - Streaming Context,
  - Hive Context.

## SparkSession in PySpark shell

- By default PySpark shell provides “**spark**” object; which is an instance of SparkSession class. We can directly use this object where required in spark-shell.

```
>>> spark.version
'3.0.1'
>>> spark.createDataFrame([("Java","20000"),("Python","10000"),("Scala","5000")]).show()
+-----+-----+
|   _1   |   _2   |
+-----+-----+
|  Java  | 20000  |
| Python | 10000  |
|  Scala |  5000  |
+-----+-----+
```

## Create SparkSession in Jupyter lab

```
[1]: import findspark
      findspark.init()
      import pyspark
      from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder.appName("VeryFirstSparkExample").getOrCreate()
```

```
[2]: spark.version
```

```
[2]: '3.0.1'
```

```
[3]: spark.createDataFrame([("Java", "20000"), ("Python", "10000"), ("Scala", "5000")]).show()
```

```
+-----+-----+
|   _1|   _2|
+-----+-----+
|  Java|20000|
|Python|10000|
|  Scala| 5000|
+-----+-----+
```



# SparkSession Commonly Used Methods

**version** – Returns Spark version where your application is running, probably the Spark version your cluster is configured with.

**createDataFrame()** – This creates a DataFrame from a collection and an RDD

**getActiveSession()** – returns an active Spark session.

**read()** – Returns an instance of DataFrameReader class, this is used to read records from csv, parquet, avro and more file formats into DataFrame.

**readStream()** – Returns an instance of DataStreamReader class, this is used to read streaming data. that can be used to read streaming data into DataFrame.

**sparkContext()** – Returns a SparkContext.

**sql** – Returns a DataFrame after executing the SQL mentioned.

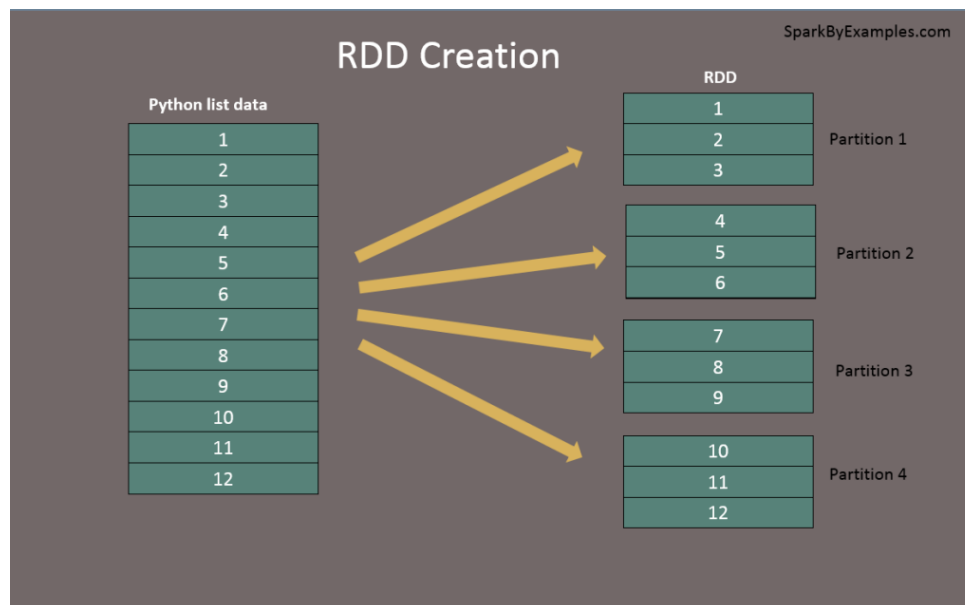
**sqlContext()** – Returns SQLContext.

**stop()** – Stop the current SparkContext.

**table()** – Returns a DataFrame of a table or view.

**udf()** – Creates a PySpark UDF to use it on DataFrame, Dataset, and SQL.

## Create RDD using `sparkContext.parallelize()`



By using `parallelize()` function of `SparkContext` (`sparkContext.parallelize()`) you can create an RDD. This function loads the existing collection from your driver program into parallelizing RDD. This is a basic method to create RDD and used when you already have data in memory that either loaded from a file or from a database. and it required all data to be present on the driver program prior to creating RDD.

## Create RDD using `sparkContext.parallelize()`

```
[4]: #Create RDD from parallelize  
data = [1,2,3,4,5,6,7,8,9,10,11,12]  
rdd=spark.sparkContext.parallelize(data)
```

```
[6]: rdd.collect()
```

```
[6]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

## Create RDD using sparkContext.textFile()

```
[7]: #Create RDD from external Data source  
rdd2 = spark.sparkContext.textFile("/path/textFile.txt")  
  
[8]: #Reads entire file into a RDD as single record.  
rdd3 = spark.sparkContext.wholeTextFiles("/path/textFile.txt")
```

# PySpark RDD Operations

- **RDD transformations** – Transformations are lazy operations, instead of updating an RDD, these operations return another RDD.
- **RDD actions** – operations that trigger computation and return non-RDD values.
- Transformations on PySpark RDD returns another RDD and transformations are lazy meaning they don't execute until you call an action on RDD. Some transformations on RDD's are flatMap(), map(), reduceByKey(), filter(), sortByKey() and return new RDD instead of updating the current.

## RDD transformation: flatMap

**flatMap** – flatMap() transformation flattens the RDD after applying the function and returns a new RDD. On the below example, first, it splits each record by space in an RDD and finally flattens it. Resulting RDD consists of a single word on each record.

```
[9]: #Create RDD from external Data source
transRDD = spark.sparkContext.textFile("trans.txt")

[10]: transRDD.collect()

[10]: ['00000000,06-26-2011,4000001,040.33,Exercise & Fitness,Cardio Machine Accessories,Clarksville,Tennessee,credit',
      '00000001,05-26-2011,4000002,198.44,Exercise & Fitness.Weightlifting Gloves,Long Beach,California,credit'.

[11]: transRDD.flatMap(lambda x: x.split(",")).collect()

[11]: ['00000000',
      '06-26-2011',
      '4000001',
      '040.33',
      'Exercise & Fitness',
      'Cardio Machine Accessories',
      'Clarksville',
```

## RDD transformation: map

**map** – map() transformation is used to apply any complex operations like adding a column, updating a column e.t.c, the output of map transformations would always **have the same number of records** as input.

```
[11]: transRDD.flatMap(lambda x: x.split(",")).collect()
```

```
[11]: ['00000000',  
      '06-26-2011',  
      '4000001',  
      '040.33',  
      'Exercise & Fitness',  
      'Cardio Machine Accessories',  
      'Clarksville',  
      'Tennessee',  
      'credit',  
      '00000001',  
      '05-26-2011',
```

```
[16]: transRDD.flatMap(lambda x: x.split(",")).count()
```

```
[16]: 540
```

```
[14]: transRDD.map(lambda x: x.split(",")).collect()
```

```
[14]: [['00000000',  
      '06-26-2011',  
      '4000001',  
      '040.33',  
      'Exercise & Fitness',  
      'Cardio Machine Accessories',  
      'Clarksville',  
      'Tennessee',  
      'credit'],  
      ['00000001',  
      '05-26-2011',
```

```
[15]: transRDD.map(lambda x: x.split(",")).count()
```

```
[15]: 60
```

## RDD transformation: map

```
[22]: #Show customer ID and amount of each transaction  
transRDD.map(lambda x: x.split(",")).map(lambda x: (x[2],x[3])).collect()
```

```
[22]: [('4000001', '040.33'),  
      ('4000002', '198.44'),  
      ('4000002', '005.58'),  
      ('4000003', '198.19'),  
      ('4000002', '098.81'),  
      ('4000004', '193.63'),  
      ('4000005', '027.89'),  
      ('4000006', '096.01'),  
      ('4000006', '010.44'),  
      ('4000006', '152.46'),  
      ('4000007', '180.28'),  
      ('4000009', '121.39'),
```



## RDD transformation: reduceByKey

**reduceByKey** – `reduceByKey()` merges the values for each key with the function specified. In our example, it reduces the word string by applying the sum function on value. The result of our RDD contains unique words and their count.

```
[27]: #Show ID and total cost of all transctions of each customer
transRDD.map(lambda x: x.split(",")).map(lambda x: (x[2],x[3])).reduceByKey(lambda amount1,amount2: float(amount1)+float(amount2) ).collect()
```

```
[27]: [('4000004', 337.06),
      ('4000007', 699.55),
      ('4000008', 859.42),
      ('4000001', 651.0500000000001),
      ('4000002', 706.97),
      ('4000003', 527.5899999999999),
      ('4000005', 325.15),
      ('4000006', 539.3800000000001),
      ('4000009', 457.83),
      ('4000010', 447.09000000000003)]
```

## RDD transformation: sortByKey

```
[32]: #Show ID and total cost of all transctions of each customer, sorted by customer ID
transRDD.map(lambda x: x.split(",")).map(lambda x: (x[2],x[3])).reduceByKey(lambda amount1,amount2: float(amount1)+float(amount2) ).sortByKey().collect()
```

```
[32]: [('4000001', 651.0500000000001),
      ('4000002', 706.97),
      ('4000003', 527.5899999999999),
      ('4000004', 337.06),
      ('4000005', 325.15),
      ('4000006', 539.3800000000001),
      ('4000007', 699.55),
      ('4000008', 859.42),
      ('4000009', 457.83),
      ('4000010', 447.09000000000003)]
```

## RDD transformation: filter

```
[40]: #Show customer IDs and games of all transaction where games include 'Sport'  
transRDD.map(lambda x: x.split(",")).map(lambda x: (x[2],x[4])).filter(lambda x: 'Sport' in x[1]).collect()
```

```
[40]: [('4000002', 'Team Sports'),  
      ('4000006', 'Winter Sports'),  
      ('4000010', 'Team Sports'),  
      ('4000001', 'Combat Sports'),  
      ('4000008', 'Water Sports'),  
      ('4000008', 'Team Sports'),  
      ('4000008', 'Water Sports'),  
      ('4000005', 'Air Sports'),  
      ('4000009', 'Water Sports'),  
      ('4000003', 'Water Sports'),  
      ('4000009', 'Combat Sports'),  
      ('4000008', 'Team Sports'),  
      ('4000001', 'Water Sports'),  
      ('4000008', 'Team Sports'),  
      ('4000008', 'Team Sports'),  
      ('4000007', 'Team Sports'),  
      ('4000005', 'Team Sports'),  
      ('4000004', 'Water Sports'),
```

## RDD functions

<https://spark.apache.org/docs/latest/api/python/reference/pyspark.html>

# Exercises

- Show the ID and all game types played by customers who play “Water Sports”.

Hint: use `reduceByKey()` to concatenate the game types of each customer IDs and then apply `filter()`. To remove duplicate game types for each ID, use `distinct()` function

```
[('4000008', 'Water Sports;Team Sports;Games;Outdoor Play Equipment;Outdoor Recreation'),  
 ('4000004', 'Indoor Games;Water Sports;Outdoor Recreation'),  
 ('4000003', 'Gymnastics;Outdoor Recreation;Water Sports'),  
 ('4000006', 'Jumping;Outdoor Play Equipment;Winter Sports;Water Sports'),  
 ('4000001', 'Combat Sports;Outdoor Recreation;Gymnastics;Exercise & Fitness;Water Sports;Winter Sports'),  
 ('4000009', 'Gymnastics;Combat Sports;Outdoor Play Equipment;Indoor Games;Water Sports'),  
 ('4000002', 'Outdoor Recreation;Exercise & Fitness;Team Sports;Water Sports')]
```

- Other exercises
  1. Show IDs and number of transactions of each customer
  2. Show IDs and number of transactions of each customer, sorted by customer ID
  3. Show IDs and total cost of transactions of each customer, sorted by total cost
  4. Show ID, number of transactions, and total cost for each customer, sorted by customer ID
  5. Show name, number of transactions, and total cost for each customer, sorted by total cost
  6. Show ID, name, game types played by each customer
  7. Show ID, name, game types of all players who play 5 or more game types
  8. Show name of all distinct players of each game types
  9. Show all game types which don't have player under 40
  10. Show min, max, average age of players of all game types

## Create DataFrame from RDD

SPARKSESSION	RDD	DATAFRAME
<code>createDataFrame(rdd)</code>	<code>toDF()</code>	<code>toDF(*cols)</code>
<code>createDataFrame(dataList)</code>	<code>toDF(*cols)</code>	
<code>createDataFrame(rowData,columns)</code>		
<code>createDataFrame(dataList,schema)</code>		

# Create DataFrame from RDD

Using toDF() function

```
[54]: columns = ["language","users_count"]  
data = [("Java", "20000"), ("Python", "100000"), ("Scala", "3000")]  
rdd = spark.sparkContext.parallelize(data)
```

```
[55]: dfFromRDD1 = rdd.toDF()  
dfFromRDD1.printSchema()  
  
root  
|-- _1: string (nullable = true)  
|-- _2: string (nullable = true)
```

```
[56]: dfFromRDD1 = rdd.toDF(columns)  
dfFromRDD1.printSchema()  
  
root  
|-- language: string (nullable = true)  
|-- users_count: string (nullable = true)
```

# Create DataFrame from RDD

## Using createDataFrame() from SparkSession

- Calling createDataFrame() from SparkSession is another way to create PySpark DataFrame manually, it takes a list object as an argument. and chain with toDF() to specify names to the columns.

```
[57]: dfFromRDD2 = spark.createDataFrame(rdd).toDF(*columns)
      dfFromRDD2.printSchema()
```

```
root
 |-- language: string (nullable = true)
 |-- users_count: string (nullable = true)
```



# Create DataFrame from List Collection

Using createDataFrame() from SparkSession

```
[67]: data = [("Java", "20000"), ("Python", "100000"), ("Scala", "3000")]
      dfFromData2 = spark.createDataFrame(data).toDF(*columns)
      dfFromData2.show()
```

```
+-----+-----+
|language|users_count|
+-----+-----+
|   Java|      20000|
| Python|     100000|
|   Scala|       3000|
+-----+-----+
```

# Create DataFrame from List Collection

Using createDataFrame() with the Row type

createDataFrame() has another signature in PySpark which takes the collection of Row type and schema for column names as arguments. To use this first we need to convert our “data” object from the list to list of Row.

```
[72]: from pyspark.sql import Row
      rowData = map(lambda x: Row(*x), data)
      dfFromData3 = spark.createDataFrame(rowData, columns)
      dfFromData3.show()
```

language	users_count
Java	20000
Python	100000
Scala	3000

# Create DataFrame from List Collection

## Create DataFrame with schema

If you wanted to specify the column names along with their data types, you should create the StructType schema first and then assign this while creating a DataFrame.

```
[78]: from pyspark.sql.types import StructType, StructField, StringType, IntegerType
data2 = [("James", "", "Smith", "36636", "M", 3000),
        ("Maria", "Anne", "Jones", "39192", "F", 4000),
        ("Jen", "Mary", "Brown", "", "F", -1)]

schema = StructType([ \
    StructField("firstname", StringType(), True), \
    StructField("middlename", StringType(), True), \
    StructField("lastname", StringType(), True), \
    StructField("id", StringType(), True), \
    StructField("gender", StringType(), True), \
    StructField("salary", IntegerType(), True) \
])

df = spark.createDataFrame(data=data2, schema=schema)
df.printSchema()
df.show()
```

```
root
|-- firstname: string (nullable = true)
|-- middlename: string (nullable = true)
|-- lastname: string (nullable = true)
|-- id: string (nullable = true)
|-- gender: string (nullable = true)
|-- salary: integer (nullable = true)
```

firstname	middlename	lastname	id	gender	salary
James		Smith	36636	M	3000
Maria	Anne	Jones	39192	F	4000
Jen	Mary	Brown		F	-1