# Introduction

## Software prerequisites:

- Java and Scala installed
- Python installed
- Apache Spark

## What is PySpark?

PySpark is a python API for spark released by Apache Spark community to support python with Spark. Using PySpark, one can easily integrate and work with RDD in python programming language too. There are numerous features that make PySpark such an amazing framework when it comes to working with huge datasets. Whether it is to perform computations on large data sets or to just analyze them, Data engineers are turning to this tool. Following are some of the said features. PySpark offers PySpark Shell which links the Python API to the spark core and initializes the Spark context. Majority of data scientists and analytics experts today use Python because of its rich library set. Integrating Python with Spark is a boon to them.

## Key features of PySpark

- **Real time computations:** Because of the in-memory processing in PySpark framework, it shows low latency
- **Polyglot:** PySpark framework is compatible with various languages like Scala, Java, Python and R, which makes it one of the most preferable frameworks for processing huge datasets
- **Caching and disk persistence:** PySpark framework provides powerful caching and very good disk persistence
- **Fast processing:** PySpark framework is way faster than other traditional frameworks for big data processing
- **Works well with RDD:** Python programming language is dynamically typed which helps when working with RDD.

# SparkContext

- SparkContext is the entry point to any spark functionality. When we run any Spark application, a driver program starts, which has the main function and your SparkContext gets initiated here. The driver program then runs the operations inside the executors on worker nodes.
- In PySpark, SparkContext is available as sc by default, so creating a new SparkContext will give an error.
- The following code block has the details of a PySpark class and the parameters, which a SparkContext can take.

```
class pyspark.SparkContext (
 master = None,
 appName = None,
 sparkHome = None,
 pyFiles = None,
 environment = None,
 batchSize = 0,
 serializer = PickleSerializer(),
 conf = None,
 gateway = None,
 jsc = None,
 profiler_cls = <class 'pyspark.profiler.BasicProfiler'>)
```

- **Parameters**

  Following are the parameters of a SparkContext.

  **Master** − It is the URL of the cluster it connects to.
  **appName** − Name of your job.
  **sparkHome** − Spark installation directory.
  **pyFiles** − The .zip or .py files to send to the cluster and add to the PYTHONPATH.
  **Environment** − Worker nodes environment variables.
  **batchSize** − The number of Python objects represented as a single Java object. Set 1 to disable batching, 0 to automatically choose the batch size based on object sizes, or -1 to use an unlimited batch size.

**Serializer** − RDD serializer.
**Conf** − An object of L{SparkConf} to set all the Spark properties.
**Gateway** − Use an existing gateway and JVM, otherwise initializing a new JVM.
**JSC** − The JavaSparkContext instance.
**profiler_cls** − A class of custom Profiler used to do profiling (the default is pyspark.profiler.BasicProfiler).

- Among the above parameters, master and appname are mostly used. The first two lines of any PySpark program looks as shown below −

```
from pyspark import SparkContext
sc = SparkContext("local", "My First App")
```


*In the above figure SparkContext is created by Databricks. We can use sc = SparkContext.getOrCreate() to get and existing context or create a new one if one does not exist.*

- Then execute the following command in the terminal to run the Python file containing the PySpark Job.

```
$SPARK_HOME/bin/spark-submit python-file.py
```

# SparkConf

- To run a Spark application on the local/cluster, you need to set a few configurations and parameters, this is what SparkConf helps with. It provides configurations to run a Spark application.
- Following are some of the most commonly used attributes of SparkConf

  **set(key, value)** − To set a configuration property.
  **setMaster(value)** − To set the master URL.
  **setAppName(value)** − To set an application name.
  **get(key, defaultValue=None)** − To get a configuration value of a key.
  **setSparkHome(value)** − To set Spark installation path on worker nodes.

- The following code block has the lines, when they get added in the Python file, it sets the basic configurations for running a PySpark application.

```
from pyspark import SparkConf, SparkContext
conf=SparkConf().setAppName("PySparkApp").setMaster("spark://master:7077")
sc = SparkContext(conf=conf)
```

# PySpark - RDD

- RDD is one of the key features of Spark. It stands for Resilient Distributed Database. It's a set of elements that are divided across multiple nodes in a cluster to run parallel processing. It can also automatically recover from failures. We can create RDD but we can't make changes in that RDD, we can create new RDD from the existing one with the required changes or we can perform different kind of operations on an RDD.

- To initialize RDD following command is used

```
from pyspark import SparkContext
RDDName = sc.textFile(" path of the file to be uploaded")
```

- RDD provides various operations
    - **count()** - Number of elements in the RDD is returned.

```
rdd = sc.parallelize (
   ["scala",
   "java",
   "hadoop",
   "spark",
   "EBS",
   "spark vs hadoop",
   "pyspark",
   "Data Science"]
)
counts = rdd.count()
counts
```

▸ (1) Spark Jobs

```
Out[15]: 8
```

    - **collect()** - All the elements in the RDD are returned.

```
rdd = sc.parallelize (
   ["scala", "java", "hadoop","spark", "EBS",
    "spark vs hadoop", "pyspark","Data Science"]
)
counts = rdd.collect()
counts
```

▸ (1) Spark Jobs

```
Out[17]:
['scala',
 'java',
 'hadoop',
 'spark',
 'EBS',
 'spark vs hadoop',
 'pyspark',
 'Data Science']
```

- **top(n)** - displays the top n elements.

```
rdd = sc.parallelize (
   ["scala", "java", "hadoop","spark", "EBS",
    "spark vs hadoop", "pyspark","Data Science"]
)
counts = rdd.top(3)
counts
```

▸ (1) Spark Jobs

```
Out[18]: ['spark vs hadoop', 'spark', 'scala']
```

- **foreach(fun)** - Returns only those elements which meet the condition of the function inside foreach.

- **filter(fun)** - A new RDD is returned containing the elements, which satisfies the function inside the filter.

```
rdd = sc.parallelize (
   ["scala", "java", "hadoop","spark", "EBS",
    "spark vs hadoop", "pyspark","Data Science"]
)
words_filter = rdd.filter(lambda x: 'spark' in x)
words_filter.collect()
```

▸ (1) Spark Jobs

```
Out[21]: ['spark', 'spark vs hadoop', 'pyspark']
```

- **map(fun, preservesPartitioning = False)** - A new RDD is returned by applying a function to each element in the RDD.

```
rdd = sc.parallelize (
   ["scala", "java", "hadoop","spark", "EBS",
    "spark vs hadoop", "pyspark","Data Science"]
)
words_map = rdd.map(lambda x: (x, len(x)))
words_map.collect()
```

▸ (1) Spark Jobs

```
Out[23]:
[('scala', 5),
 ('java', 4),
 ('hadoop', 6),
 ('spark', 5),
 ('EBS', 3),
 ('spark vs hadoop', 15),
 ('pyspark', 7),
 ('Data Science', 12)]
```

Command took 0.46 seconds -- by njp62@njit.edu at 5/15/2019, 4:47:38 PM on My Cluster

- **reduce(fun) -** After performing the specified commutative and associative binary operation, the element in the RDD is returned.

```
from operator import add
nums = sc.parallelize([1, 2, 3, 4, 5])
nums.reduce(add)
```

▸ (1) Spark Jobs

```
Out[24]: 15
```

Command took 0.44 seconds -- by njp62@njit.edu at 5/15/2019, 4:50:17 PM on My Cluster

- **join(other, numPartitions = None)** - returns RDD with a pair of elements with the matching keys and all the values for that particular key.

- **cache()**-Persist this RDD with the default storage level (MEMORY_ONLY). You can also check if the RDD is cached or not.

```
rdd = sc.parallelize (
   ["scala", "java", "hadoop","spark", "EBS",
    "spark vs hadoop", "pyspark","Data Science"]
)
rdd.cache()
rdd.persist().is_cached
```

```
Out[25]: True
```

# SparkFiles and class methods

- SparkFiles is what we use when we want to upload our files in Apache Spark using SparkContext.addfile(). Thus, SparkFiles resolve the paths to files added through SparkContext.addFile().

- SparkFiles contain the following classmethods −
    - get(filename)
    - getrootdirectory()

- **get(Filename)**
    - This classmethod is used when we want to specify the path of the file that we added using SparkContext.addfile() or sc.addFile()

```
from pyspark import SparkFiles
from pyspark import SparkContext
path=os.path.join("path of the file to be uploaded")
sc.addFile(path)
SparkFiles.get(path)
```

- **getrootdirectory()**
    - It specifies the path to the root directory, which contains the file that is added through the SparkContext.addFile().

```
from pyspark import SparkFiles
from pyspark import SparkContext
path=os.path.join("path of the file to be uploaded")
sc.addFile(path)
SparkFiles.getrootdirectory(path)
```

# Serializers

- Serialization is used for performance tuning on Apache Spark. Serialization plays an important role in costly operations.PySpark supports custom serializers for performance tuning. The following two serializers are supported by PySpark

- **MarshalSerializer**
  - Serializes objects using Python's Marshal Serializer. This serializer is faster than PickleSerializer, but supports fewer data types.

    *class pyspark.MarshalSerializer*

- **PickleSerializer**
  - Serializes objects using Python's Pickle Serializer. This serializer supports nearly any Python object, but may not be as fast as more specialized serializers.

    *class pyspark.PickleSerializer*

# PySpark - MLlib

Apache Spark offers a Machine Learning API called MLlib. PySpark has this machine learning API in Python as well. It supports different kind of algorithms, which are mentioned below −

**mllib.classification:**

The Spark.mllib package offers support for various methods to perform binary classification, regression analysis and multiclass classification. Some of the most used algorithms in Classifications are Naive Bayes, Decision Tree, etc.

**mllib.clustering:**

In clustering we perform grouping of subsets of entities on the basis of some similarities in the elements or entities

**mllib.linalg:**

This algorithms offers MLlib utilities to support linear algebra

**mllib.recommendation:**

This algorithm is used for recommender systems for filling in the missing entries in any dataset

**spark.mllib:**

spark.mllib supports collaborative filtering, where Spark uses ALS(Alternating Least Squares) to predict the sets of description of users and products in order to predict missing entries