What is the PySpark 3.5 tutorial for beginners with examples?

Authored by

stats writer

June 24, 2024

The PySpark 3.5 tutorial is a comprehensive guide designed for beginners to learn the basics of PySpark, a powerful data processing and analytics framework built on top of Apache Spark. This tutorial includes step-by-step instructions, along with examples, to help beginners understand the fundamentals of PySpark and how to use it for data manipulation, analysis, and machine learning. By the end of this tutorial, beginners will have a solid understanding of PySpark and be able to apply its capabilities to their own data science projects.

PySpark Tutorial: PySpark is a powerful open-source framework built on Apache Spark, designed to simplify and accelerate large-scale data processing and analytics tasks. It offers a high-level API for Python programming language, enabling seamless integration with existing Python ecosystems.

PySpark revolutionizes traditional computing with its distributed computing model, capable of processing massive datasets across clusters of machines with remarkable speed and efficiency. Its key advantages lie in its ability to handle diverse data formats, support for complex analytics operations, and fault tolerance. By leveraging PySpark, users can unlock unparalleled scalability, faster processing times, and enhanced performance, making it an indispensable tool for modern data-driven applications.

PySpark Tutorial Introduction

In this PySpark tutorial, you'll learn the fundamentals of Spark, how to create distributed data

processing pipelines, and leverage its versatile libraries to transform and analyze large datasets efficiently with examples. I will also explain what is PySpark, its features, advantages, modules, packages, and how to use RDD & DataFrame with simple and easy examples from my working experience in Python.

All examples explained in this PySpark (Spark with Python) tutorial are basic, simple, and easy to practice for beginners who are enthusiastic to learn PySpark and advance their careers in Big Data, Machine Learning, Data Science, and Artificial intelligence.

Note: If you can't locate the PySpark examples you need on this beginner's tutorial page, I suggest utilizing the Search option in the menu bar. This website offers numerous articles in Spark, Scala, PySpark, and Python for learning purposes.

If you are working with a smaller Dataset and don't have a Spark cluster, but still want to get benefits similar to Spark DataFrame, you can use Python Pandas DataFrames. The main difference is Pandas DataFrame is not distributed and runs on a single node.

Table of Contents -

What is PySpark

PySpark is the Python API for Apache Spark. PySpark enables developers to write Spark applications using Python, providing access to Spark's rich set of features and capabilities through Python language. With its rich set of features, robust performance, and extensive ecosystem, PySpark has become a popular choice for data engineers, data scientists, and developers working with big data and distributed computing.

source: https://databricks.com/

Apache Spark is an open-source unified analytics engine used for large-scale data processing, hereafter referred it as Spark. Spark is designed to be fast, flexible, and easy to use, making it a popular choice for processing large-scale data sets. Spark runs operations on billions and trillions of data on distributed clusters 100 times faster than traditional applications.

Spark can run on **single-node machines or multi-node machines(Cluster)**. It was created to

address the **limitations of MapReduce**, by doing in-memory processing. Spark reuses data by using an in-memory cache to speed up machine learning algorithms that repeatedly call a function on the same dataset. This lowers the latency making Spark multiple times faster than MapReduce, especially when doing machine learning, and interactive analytics. Apache Spark can also process real-time streaming.

It is also a **multi-language engine,** that provides APIs (Application Programming Interfaces) and libraries for several programming languages like Java, Scala, Python, and R, allowing developers to work with Spark using the language they are most comfortable with.

Scala: Spark's primary and native language is Scala. Many of Spark's core components are written in Scala, and it provides the most extensive API for Spark.**Java:** Spark provides a Java API that allows developers to use Spark within Java applications. Java developers can access most of Spark's functionality through this API.**Python:** Spark offers a Python API, called PySpark, which is popular among data scientists and developers who prefer Python for data analysis and machine learning tasks. PySpark provides a Pythonic way to interact with Spark.**R:** Spark also offers an R API, enabling R users to work with Spark data and perform distributed data analysis using their familiar R language.

Who uses PySpark?

PySpark is very well used in the Data Science and Machine Learning community as there are many widely used data science libraries written in Python including NumPy, and TensorFlow. Also used due to its efficient processing of large datasets. PySpark has been used by many organizations like Walmart, Trivago, Sanofi, Runtastic, and many more.

Additionally, for development, you can use the Anaconda distribution (widely used in the Machine Learning community). The Anaconda distribution is a comprehensive platform for data science and machine learning tasks. It includes a collection of popular open-source software packages and libraries, such as Python, Jupyter Notebooks, NumPy, pandas, scikit-learn, and many others.

In real-time, PySpark has been used a lot in the machine learning and data scientists community; thanks to vast Python machine learning libraries. In this PySpark tutorial for beginners, I have explained several topics that cover vast concepts of this framework.

How to run Pandas DataFrame on Apache Spark (PySpark)?

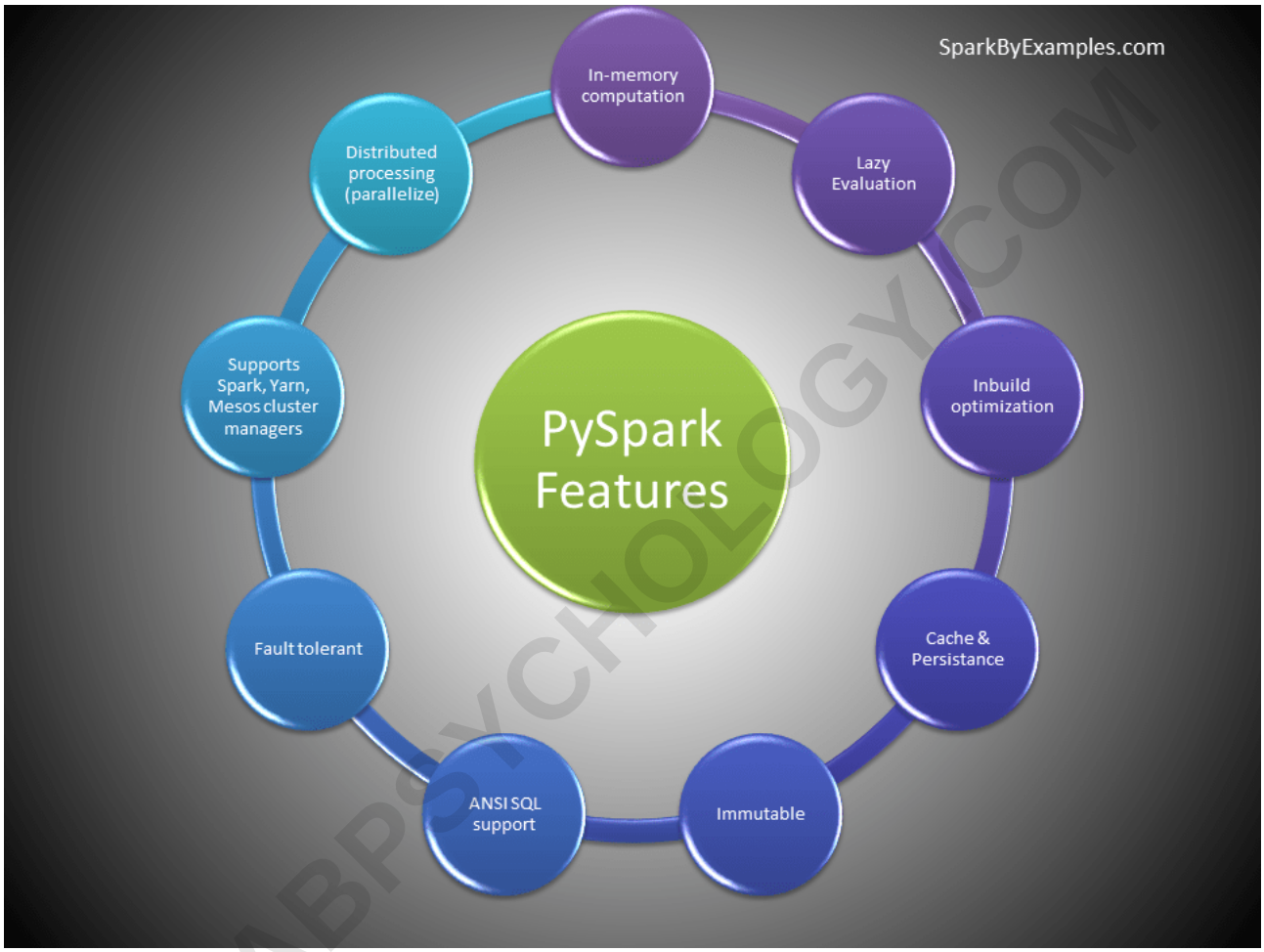
What Version of Python PySpark Supports

PySpark 3.5 supports Python versions 3.8 and newer, along with R version 3.5, Java versions 8, 11, and 17, as well as Scala versions 2.12 and 2.13, and later. However, it's worth mentioning that

starting from Spark 3.5.0, support for Java 8 versions before 8u371 has been deprecated.

PySpark Features & Advantages

The following are the main features of PySpark.



PySpark Features

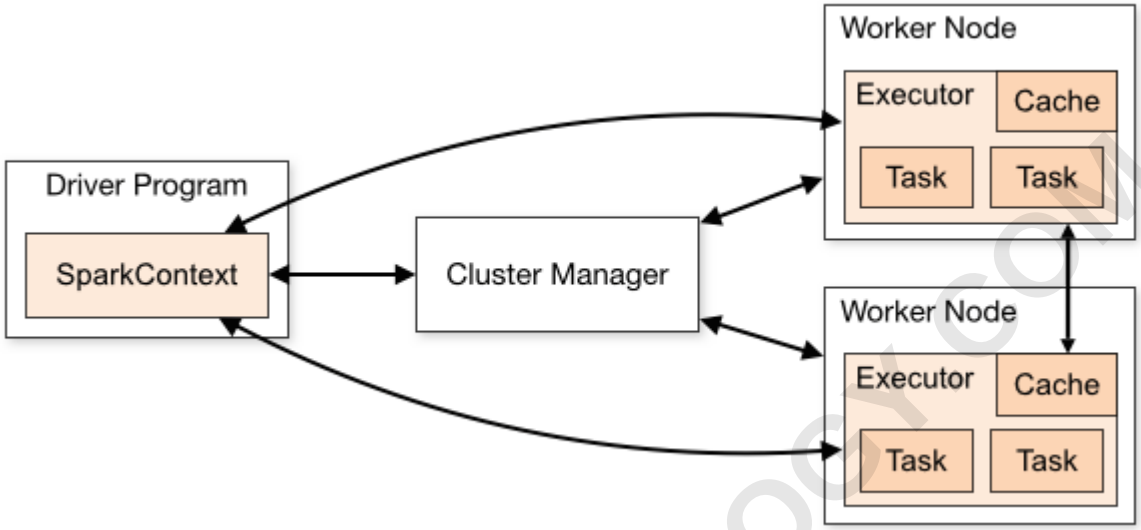
PySpark Advantages

The most important advantages of using PySpark include:

PySpark Architecture

PySpark architecture consists of a driver program that coordinates tasks and interacts with a cluster manager to allocate resources. The driver communicates with worker nodes, where tasks are executed within an executor's JVM. SparkContext manages the execution environment, while the DataFrame API enables high-level abstraction for data manipulation. SparkSession provides a

unified entry point for Spark functionality. Underneath, the cluster manager oversees resource allocation and task scheduling across nodes, facilitating parallel computation for processing large-scale data efficiently.



source : https://spark.apache.org/

To get in-depth knowledge of PySpark Architecture, read the following topics.

Cluster Managers

Cluster managers in PySpark are responsible for resource allocation and task scheduling across nodes in a distributed computing environment. Here's a brief overview of the different types:

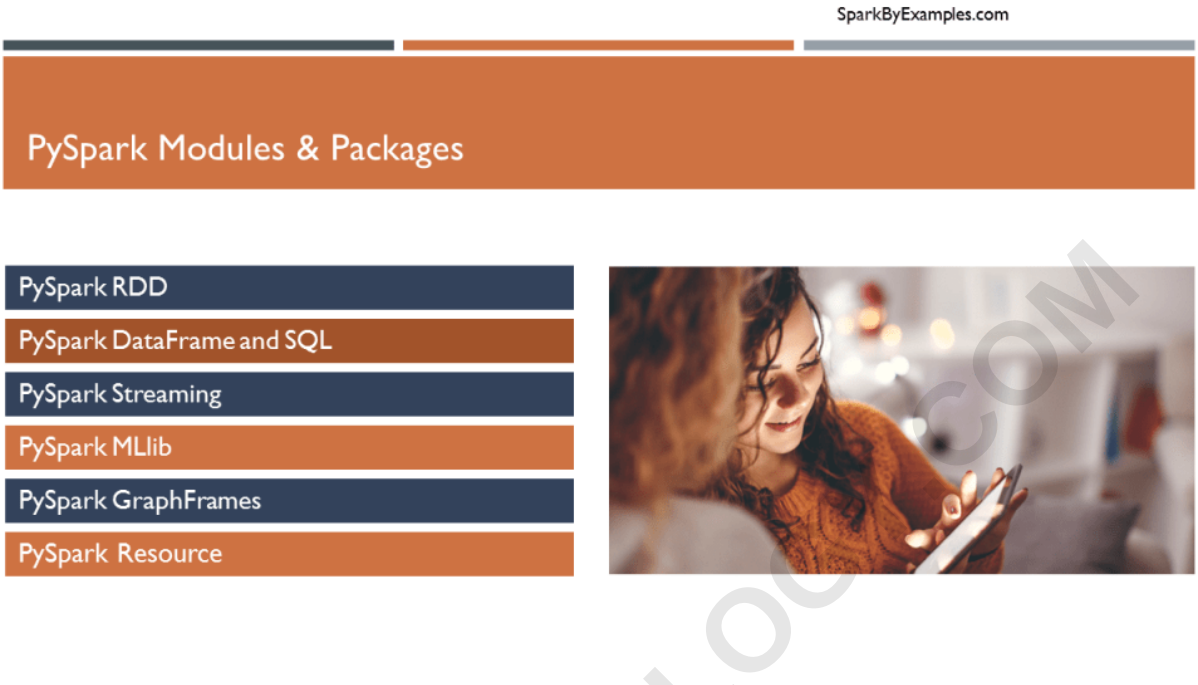
local - "local" is a special value used for the `master` parameter when initializing a SparkContext or SparkSession. When you specify `local` as the master, it means that Spark will run in local mode, utilizing only a single JVM (Java Virtual Machine) on the local machine where your Python script is executed. This mode is primarily used for development, testing, and debugging purposes

Each cluster manager type offers unique features and benefits, catering to different deployment scenarios and infrastructure requirements. The choice of cluster manager depends on factors such as scalability, resource isolation, integration with existing infrastructure, and ease of management.

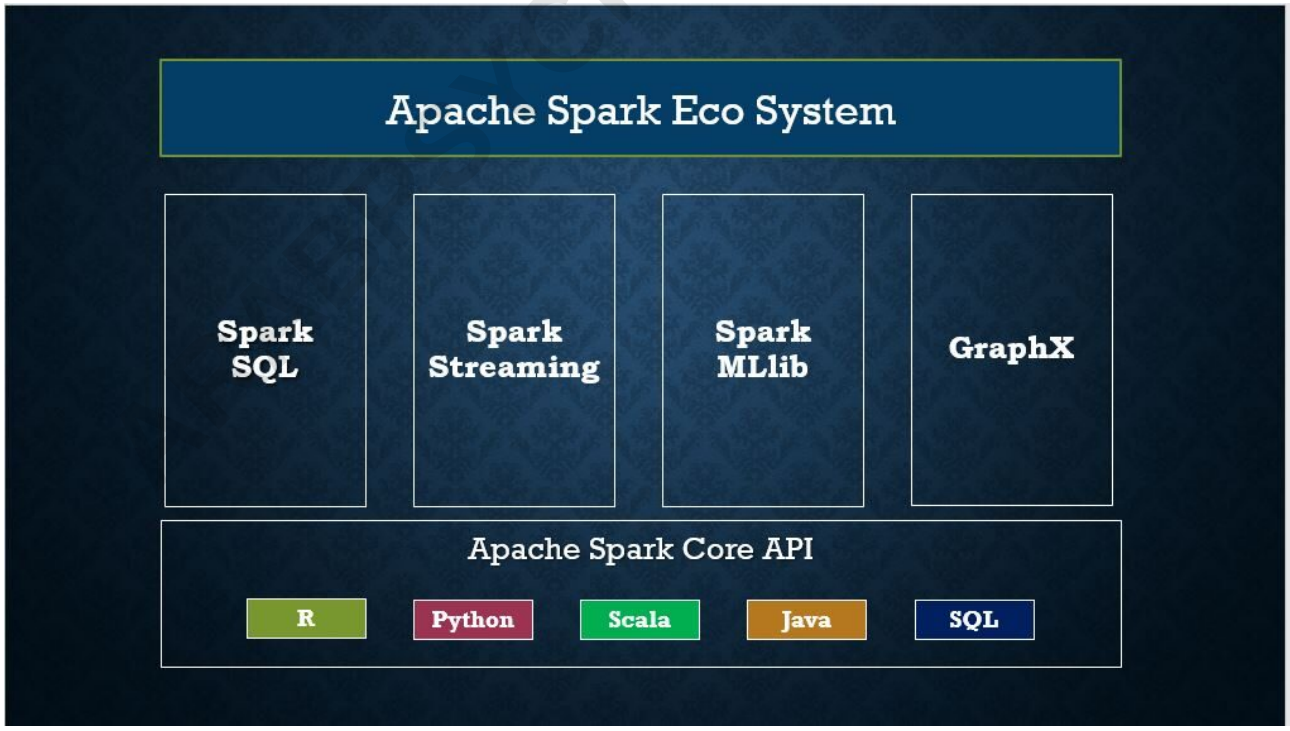
PySpark Modules & Packages

In Apache Spark, the PySpark module enables Python developers to interact with Spark, leveraging its powerful distributed computing capabilities. It provides a Python API that exposes Spark's functionality, allowing users to write Spark applications using Python programming

language.



Modules & packages



Besides these, if you want to use third-party libraries, you can find them at https://spark-packages.org/ . This page is kind of a repository of all third-party libraries.

Download & Install PySpark

Follow the below steps to install PySpark on the Anaconda distribution on Windows.

PySpark Install on Mac

Install Python or Anaconda distribution

I would recommend using Anaconda to Install Python as it is popular and used by the Machine Learning and Data Science community. Follow the instructions to Install Anaconda Distribution and Jupyter Notebook. Alternatively, you can also Python from Python.org.

Install Java

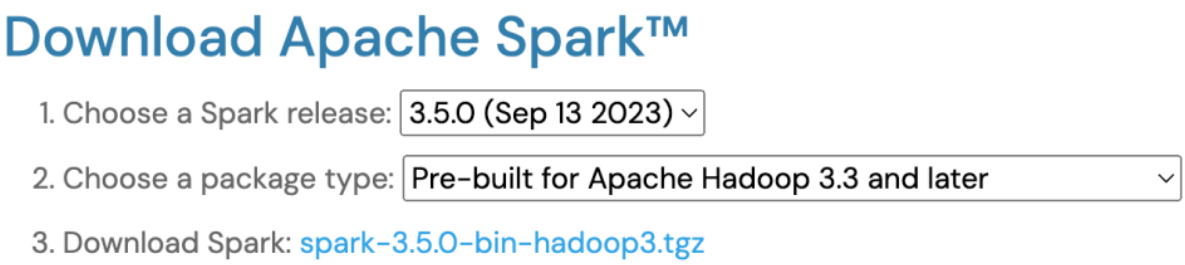
PySpark is built on top of the Spark framework, which itself is implemented in Scala and runs on the Java Virtual Machine (JVM). Hence, you need to install a compatible Java version for your Spark version. You can download and Install Java from Oracle or openJDK from openlogic.com.

Once JDK installation is complete, set `JAVA_HOME` and `PATH` variable. I have installed Java at location `c:appsJavaopenJDK17`; hence, you see I am setting the path below.

```
# Download JDK 17
JAVA_HOME = c:appsJavaopenjdk17
PATH = %PATH%;c:appsJavaopenjdk17bin
```

Install Apache Spark

Go to the Spark Download page, choose the Spark version you want to use, and then choose the package type. The URL on point 3 changes to the selected version. Click on the link from point 3 to download.



Once the binary package downloaded, untar the binary using 7zip or any extract tool and copy the

folder `spark-3.5.0-bin-hadoop3` to `c:apps`

Set the below environment variables.

```
# Set windows environment variables
SPARK_HOME = C:appsspark-3.5.0-bin-hadoop3
HADOOP_HOME = C:appsspark-3.5.0-bin-hadoop3
PATH = %PATH%;C:appsspark-3.0.5-bin-hadoop3bin
```

Install winutils.exe

Hadoop is not natively supported on Windows, but Spark expects certain Hadoop components, including the Hadoop Distributed File System (HDFS), to be present to run. Winutils provides a set of native libraries and executables that emulate Hadoop's file system API on Windows, allowing Spark to interact with the file system correctly. Without winutils, Spark may encounter errors or fail to perform file system operations on Windows platforms.

Download the appropriate version of winutils.exe from the Winutils GitHub repository and place it into the %SPARK_HOME%bin directory.

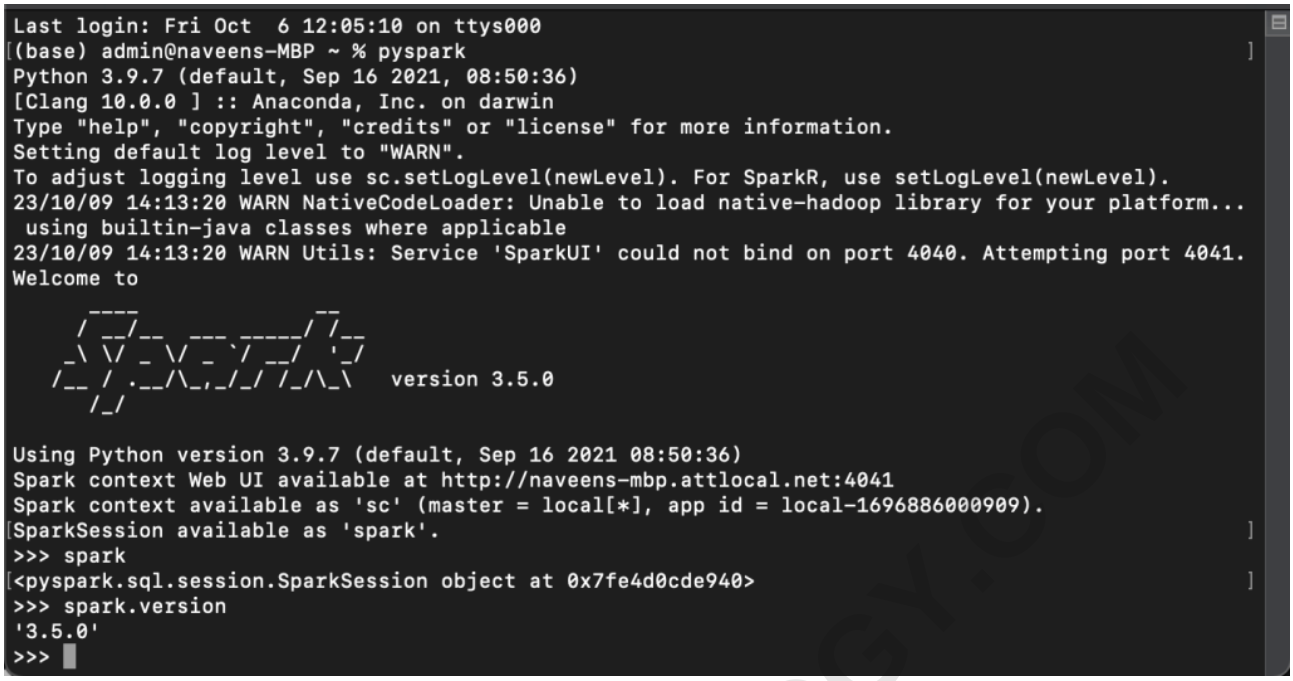
Validate PySpark Install

To validate if the PySpark has been installed correctly. Open the command prompt and type `pyspark` command to run the PySpark shell.

The shell is an interactive environment for running PySpark code. It is a CLI tool that provides a Python interpreter with access to Spark functionalities, enabling users to execute commands, perform data manipulations, and analyze results interactively.

```
# Run pyspark shell
$SPARK_HOME/sbin/pyspark
```

You should see the below output.

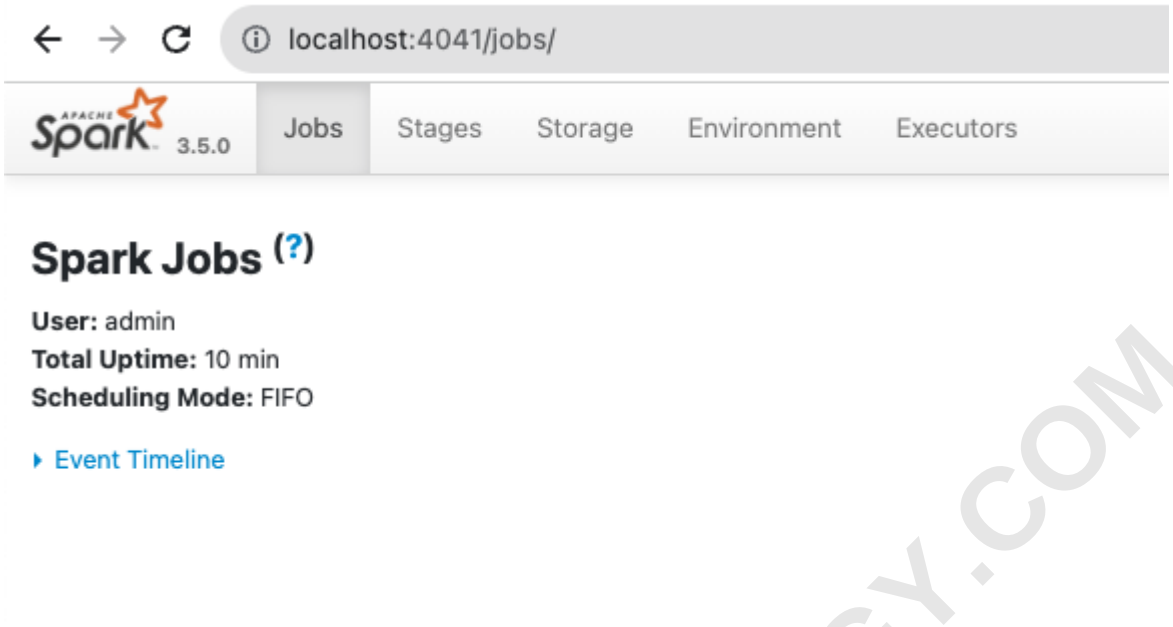


Shell creates a Spark context web UI, which can be accessed by default from http://localhost:4040.

Note: Spark Web UI typically uses port 4040 by default. However, if port 4040 is already in use, Spark will automatically attempt the next available port (4041, 4042, and so on). You can access the Spark Web UI in a web browser by navigating to `http://localhost:4040` (or the respective port number if it's different).

Spark Web UI

The Spark Web UI is a user interface provided to monitor and debug Spark applications. It displays information about job progress, task execution, resource utilization, and system metrics, allowing users to optimize performance, diagnose issues, and gain insights into their Spark jobs.



Access Spark History Server

The Spark History Server stores information about completed Spark applications (spark-submit, spark-shell), including logs, metrics, and event timelines. It allows users to view detailed information about past job executions, such as tasks, stages, and configurations, through a web-based user interface.

It serves as a centralized repository for historical job data, promoting transparency, reproducibility, and continuous improvement in Spark application development and operations. Before you start the history server, make sure you set the below config on `spark-defaults.conf`

```
spark.eventLog.enabled true
spark.history.fs.logDirectory file:///c:/logs/path
```

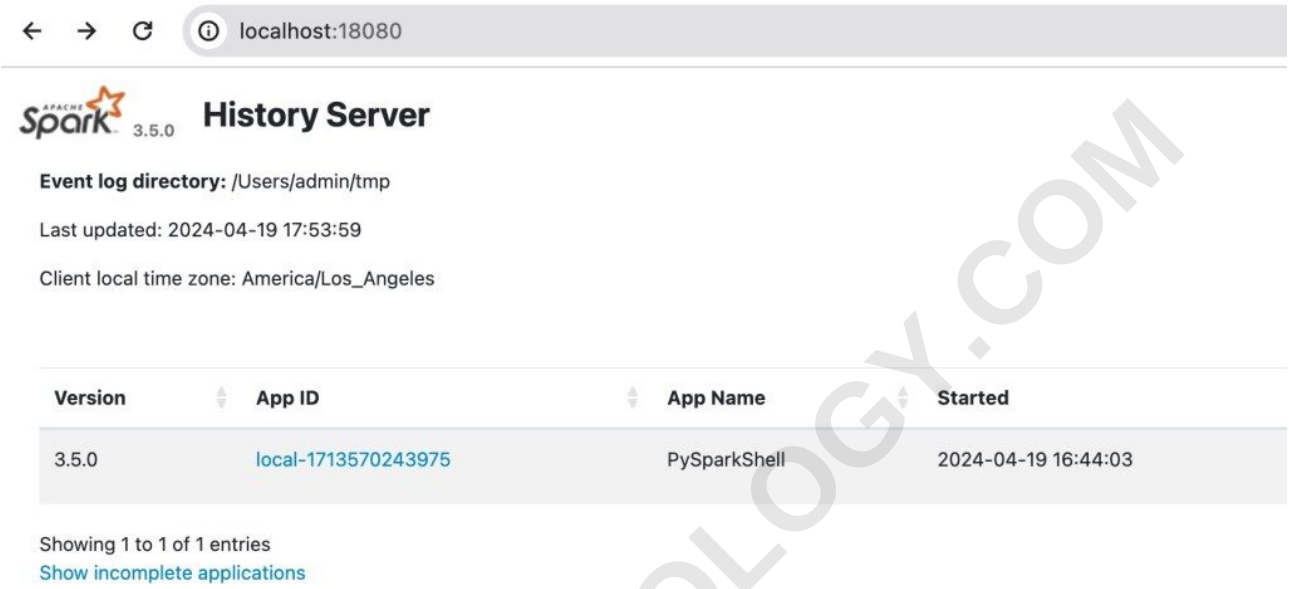
On Mac or Linux, run `start-history-server.sh` to start the history server.

```
# Start history server on linux/mac
$SPARK_HOME/sbin/start-history-server.sh
```

On Windows, this command is not available hence, run the below command to start in Windows.

```
# Start history server on windows
$SPARK_HOME/bin/spark-class.cmd org.apache.spark.deploy.history.HistoryServer
```

You can access the History server by accessing `http://localhost:18080/`



Spark History Server

It will list all the application IDs you ran in the past. By clicking on each ID, you will get its details.

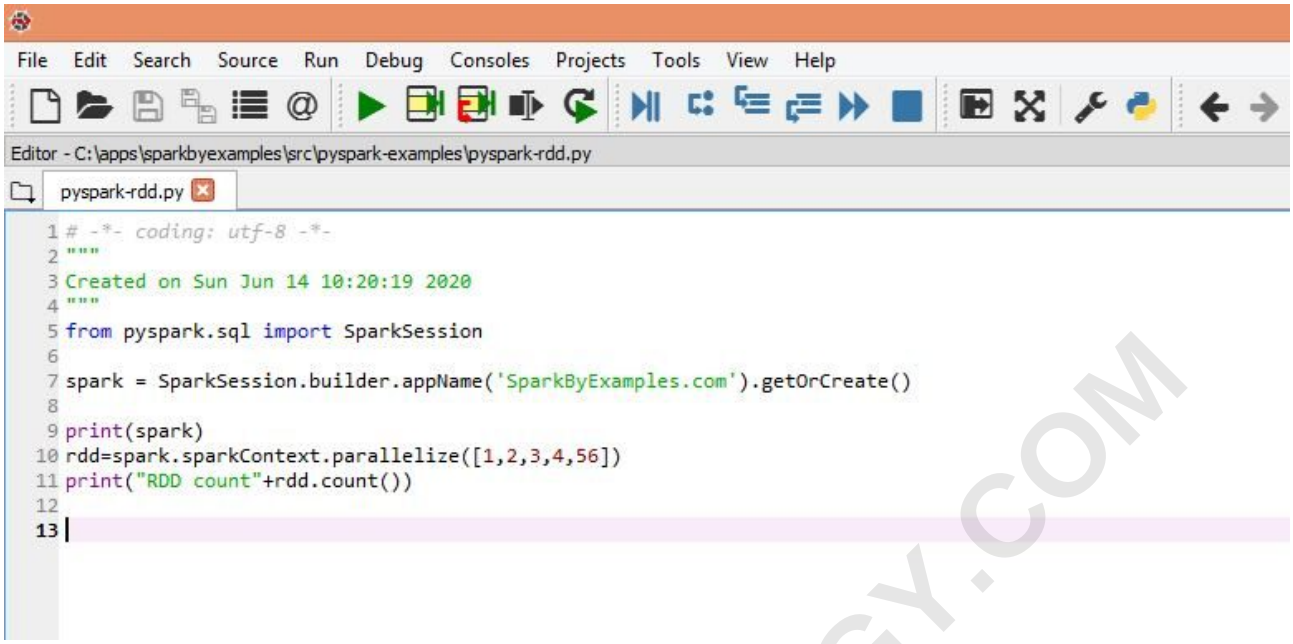
Spyder IDE & Jupyter Notebook

To create PySpark applications, you would need an IDE like Visual Studio Code, PyCharm, Spyder, etc. In this tutorial, I chose to use Spyder IDE and Jupyter Notebook to run PySpark applications. Follow Install PySpark with Anaconda & Jupyter

Once PySpark installation completes, set the following environment variable.

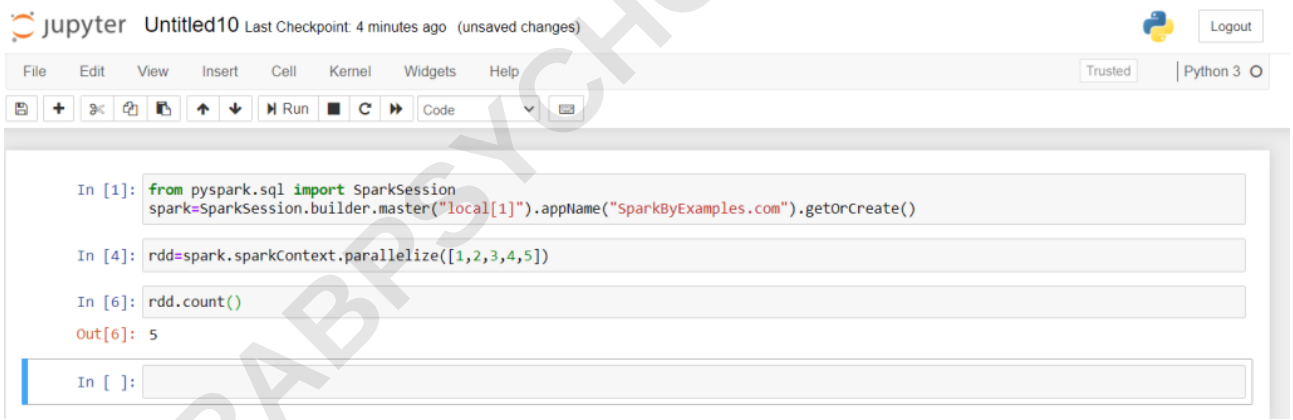
```
# Set environment variable
PYTHONPATH   =>   %SPARK_HOME%/python;$SPARK_HOME/python/lib/py4j-0.10.9-src.zip;%PYTHONPATH%
```

In Spyder IDE, run the following program. You should see 5 in output. This creates an RDD and gets you the number of elements in the RDD.



Run PySpark on Spyder IDE

Now, let's start the Jupyter Notebook



PySpark statements running on Jupyter Interface

PySpark RDD - Resilient Distributed Dataset

PySpark RDD (Resilient Distributed Dataset) is a fundamental data structure that is fault-tolerant, immutable, and distributed collections of objects. RDDs are immutable, meaning they cannot be changed once created. Any transformation on an RDD results in a new RDD. Each dataset in RDD is divided into logical partitions, which can be computed on different nodes of the cluster.

RDD Creation

In order to create an RDD, first, you need to create a SparkSession which is an entry point to the PySpark application. SparkSession can be created using a `builder()` or `newSession()` methods of the SparkSession.

Spark session internally creates a `sparkContext` variable of `SparkContext`. You can create multiple SparkSession objects but only one SparkContext per JVM. In case you want to create another new SparkContext, you should stop the existing Sparkcontext (using `stop()`) before creating a new one.

Let's create an RDD from a text file using `textFile()` function of the SparkContext.

```
# Import SparkSession
from pyspark.sql import SparkSession
```

```
# Create SparkSession
spark = SparkSession.builder
.master("local")
.appName(arabpsychology.com)
.getOrCreate()

# Create RDD from external Data source
rdd2 = spark.sparkContext.textFile("/path/test.txt")
```

Once you have an RDD, you can perform transformation and action operations. Any operation you perform on RDD runs in parallel.

RDD Operations

You can perform two types of operations on RDD; Transformations and Actions.

RDD transformations in PySpark are lazy operations and they execute only when an action is called on RDD.

Transformation operations are `map`, `filter`, `flatMap`, `groupByKey`, `reduceByKey`, `join`, `union`, `sortByKey`, `distinct`, `sample`, `mapPartitions`, and `aggregateByKey`. These functions transform RDDs by applying computations in a distributed manner across a cluster of machines and return a new RDD

RDD actions in PySpark trigger computations and return results to the Spark driver. Key actions

include `collect`, `count`, `take`, `reduce`, `foreach`, `first`, `takeOrdered`, `takeSample`, `countByKey`, `saveAsTextFile`, `saveAsSequenceFile`, `saveAsObjectFile`, `foreachPartition`, `collectAsMap`, `aggregate`, and `fold`. These actions initiate execution and materialize RDD data. Remember any RDD operation that returns non RDD is considered as an action.

PySpark DataFrame Tutorial

A DataFrame is a distributed dataset comprising data arranged in rows and columns with named attributes. It shares similarities with relational database tables or R/Python data frames but incorporates sophisticated optimizations.

If you come from a Python background, I would assume you already know what Pandas DataFrame is. PySpark DataFrame is mostly similar to Pandas DataFrame, with the exception that DataFrames are distributed in the cluster (meaning the data in data frames are stored in different machines in a cluster), and any operations in PySpark execute in parallel on all machines, whereas Panda Dataframe stores and operates on a single machine.

If you have no Python background, don't worry and you can learn about it from Python Pandas Tutorial.

Is PySpark faster than pandas?

PySpark is a distributed computing framework well-suited for processing large-scale datasets that exceed the memory capacity of a single machine. It can leverage parallel processing across a cluster of machines, enabling faster computations on massive datasets.

On the other hand, pandas, being a single-machine library, is optimized for smaller to medium-sized datasets that can fit into memory. It typically performs well for data manipulation and analysis tasks on small to medium datasets.

To know more read at Pandas DataFrame vs PySpark Differences with Examples.

Creating DataFrame

Using a list is one of the simplest ways to create a DataFrame. If you already have an RDD, you can easily convert it to DataFrame. Use `createDataFrame()` from the SparkSession to create a DataFrame.

```
# Create DataFrame
data =
```

columns =
df = spark.createDataFrame(data=data, schema = columns)

Since DataFrame is a tabular format that has names and data types in columns, use `df.printSchema()` to get the schema of the DataFrame.

To display the DataFrame use `df.show()` which shows the 20 rows by default.

```
# Output:
+---------+----------+--------+----------+------+------+
|firstname|middlename|lastname|dob |gender|salary|
+---------+----------+--------+----------+------+------+
|James | |Smith |1991-04-01|M |3000 |
|Michael |Rose | |2000-05-19|M |4000 |
|Robert | |Williams|1978-09-05|M |4000 |
|Maria |Anne |Jones |1967-12-01|F |4000 |
|Jen |Mary |Brown |1980-02-17|F |-1 |
+---------+----------+--------+----------+------+------+
```

DataFrame Operations

Similar to RDD, Transformations and Actions operations are also available in DataFrame. Below are some examples to learn more about them.

Using External Data Sources

In real-time applications, Data Frames are created from external sources, such as files from the local system, HDFS, S3 Azure, HBase, MySQL table, etc.

Supported file formats

Apache Spark, by default, supports a rich set of APIs to read and write several file formats.

For example, to read a CSV file, use the following.

```
# Create DataFrame from CSV file
df = spark.read.csv("/tmp/resources/zipcodes.csv")
df.printSchema()
```

Following are some resources to learn how to read and write to external data sources

DataFrame Examples

Following are several PySpark examples that can help you understand more about DataFrame.

PySpark SQL

PySpark SQL is a module in Spark that provides a higher-level abstraction for working with structured data and can be used SQL queries.

SQL enables you to write SQL queries against structured data, leveraging standard SQL syntax and semantics. This familiarity with SQL allows users with SQL proficiency to transition to Spark for data processing tasks easily.

First, you should create a temporary table or view on DataFrame to use SQL queries. Once the table is created, you can be accessed throughout the SparkSession using `sql()`.

These tables and views are scoped to the SparkSession that created them. Once the SparkSession is terminated, either by closing the Spark application or ending the Spark session explicitly, the temporary views are removed from memory.

To run SQL queries, use `sql()` method of the SparkSession object. Note that this function returns a DataFrame.

```
# Create temporary table
df.createOrReplaceTempView("PERSON_DATA")
```

```
# Run SQL query
df2 = spark.sql("SELECT * from PERSON_DATA")
df2.printSchema()
df2.show()
```

Use `group by` clause to run aggregate queries.

```
# Using groupby
groupDF = spark.sql("SELECT gender, count(*) from PERSON_DATA group by gender")
groupDF.show()
```

This yields the below output

```
# Output:
+------+--------+
|gender|count(1)|
+------+--------+
| F| 2|
| M| 3|
+------+--------+
```

PySpark with English SDK

Below is a snippet of how to use English SDK using GenAI to write SQL queries. For details on how to install and use it, refer to PySpark AI (pyspark-ai) - English SDK Comprehensive Guide

# Rename columns using AI
df2 = df.ai.transform("rename column name from firstname to first_name and lastname to last_name")
df2.printSchema()
df2.show()

PySpark Streaming Tutorial

Spark Streaming is a real-time data processing framework in Apache Spark that enables developers to process and analyze streaming data from various sources like file system folders, TCP sockets, S3, Flume, Kafka, Twitter, and Amazon Kinesis in near real-time. It allows you to ingest continuous streams of data, such as log files, sensor data, social media feeds, and more, and perform real-time analytics on them.

source: https://spark.apache.org/

Streaming from TCP Socket

To create a Spark Streaming application that reads data from a TCP socket, you can use the `readStream.format("socket")` from Spark session object. Use the `option()` to specify the TCP host and port number to read.

```
# Spark streaming from socket
df = spark.readStream
.format("socket")
.option("host","localhost")
.option("port","9090")
.load()
```

Spark reads the data from the socket and represents it in a "value" column of DataFrame. `df.printSchema()` outputs

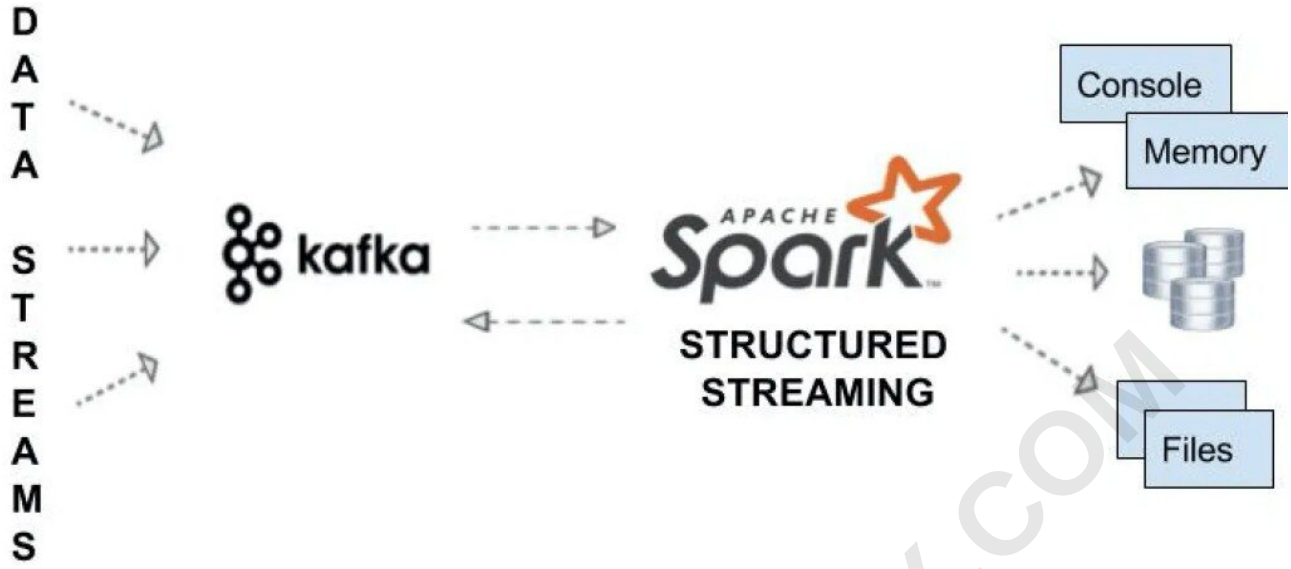
```
# Output:
root
|-- value: string (nullable = true)
```

After processing, you can stream the dataframe to the console. In real-time, we ideally stream it to either Kafka, database e.t.c

```
# Write to console
query = count.writeStream
.format("console")
.outputMode("complete")
.start()
.awaitTermination()
```

Streaming from Kafka

Using Spark Streaming we can read from Kafka topic and write to Kafka topic in TEXT, CSV, AVRO and JSON formats




```
# Stream from Kafka
df = spark.readStream
.format("kafka")
.option("kafka.bootstrap.servers", "192.168.1.100:9092")
.option("subscribe", "json_topic")
.option("startingOffsets", "earliest") // From starting
.load()
```

Write a message to another topic in Kafka using `writeStream()`

```
# Stream write to kafka
df.selectExpr("CAST(id AS STRING) AS key", "to_json(struct(*)) AS value")
.writeStream
.format("kafka")
.outputMode("append")
.option("kafka.bootstrap.servers", "192.168.1.100:9092")
.option("topic", "josn_data_topic")
.start()
.awaitTermination()
```

PySpark MLlib Tutorial

PySpark MLlib is Apache Spark's scalable machine learning library, offering a suite of algorithms and tools for building, training, and deploying machine learning models. It provides

implementations of popular algorithms for classification, regression, clustering, collaborative filtering, and more.

MLlib is designed for distributed computing, allowing it to handle large-scale datasets across clusters efficiently. It offers both high-level APIs for ease of use and low-level APIs for fine-grained control over model training and evaluation.

MLlib seamlessly integrates with Spark's ecosystem, enabling end-to-end machine learning workflows, including data preprocessing, feature engineering, model training, and deployment in production environments.

Here's a simple example using Spark MLlib in Python to train a linear regression model:

```
# Import
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorAssembler

# Sample training data
data =
df = spark.createDataFrame(data, )

# Define a feature vector assembler
assembler = VectorAssembler(inputCols=, outputCol="features_vec")

# Transform the DataFrame with the feature vector assembler
df = assembler.transform(df)

# Create a LinearRegression model
lr = LinearRegression(featuresCol="features_vec", labelCol="label")

# Fit the model to the training data
model = lr.fit(df)

# Print the coefficients and intercept of the model
print("Coefficients: " + str(model.coefficients))
print("Intercept: " + str(model.intercept))

# Stop the SparkSession
spark.stop()
```

This is a simple example to demonstrate the usage of Spark MLlib for linear regression. In practice, you would typically use larger datasets and more complex models for real-world machine learning

tasks.

PySpark GraphFrames

PySpark GraphFrames were introduced since Spark 3.0 to enable Graphs on Data Frames. Prior to 3.0, Spark had a GraphX library that supported only RDD.

Spark GraphFrames is a graph processing library built on top of Apache Spark that allows developers to work with graph data structures in a distributed and scalable manner. In simple terms, GraphFrames enables you to analyze and manipulate relationships between entities, represented as vertices (nodes) and edges (connections), using the power of Spark's distributed computing engine.

To make GraphFrames work, you need to install `graphframes` library. And, download the graphframes.jar from the Maven repository and upload it to Python.

```
# Install graphframes
pip install graphframes
```

Example

```
# Import
from graphframes import GraphFrame
```

```
# Define vertices DataFrame
vertices = spark.createDataFrame(, )

# Define edges DataFrame
edges = spark.createDataFrame(, )

# Create a GraphFrame
g = GraphFrame(vertices, edges)

# Query the graph
g.vertices.show()
g.edges.show()

# Find the shortest path between two vertices
results = g.shortestPaths(landmarks=)
results.select("id", "distances").show()
```

```
# Stop the SparkSession
spark.stop()
```

Conclusion

Apache Spark is a powerful open-source data processing framework that provides high-speed, distributed computing for big data analytics. In this PySpark tutorial for beginners, I hope you have learned the fundamentals of PySpark, how to create distributed data processing pipelines, and how to leverage its versatile libraries to transform and analyze large datasets efficiently.

Related Articles

References

Below are some of the articles/tutorials I've referred to complete this Tutorial.

Naveen

Naveen (NNK) is a Data Engineer with 20+ years of experience in transforming data into actionable insights. Over the years, He has honed his expertise in designing, implementing, and maintaining data pipelines with frameworks like Apache Spark, PySpark, Pandas, R, Hive and Machine Learning. Naveen journey in the field of data engineering has been a continuous learning, innovation, and a strong commitment to data integrity. In this blog, he shares his experiences with the data as he come across. Follow Naveen @ LinkedIn LinkedIn