**DAY 9 (17-06-2025)**

**PySpark - Big Data Processing with Python and Apache Spark**

**Introduction to Apache Spark**

**Apache Spark** is an open-source, distributed computing engine designed for fast and general-purpose big data processing. It can handle massive amounts of data across a cluster of machines.

* **Speed**: In-memory computation makes Spark faster than traditional systems like Hadoop MapReduce.
* **Scalability**: Designed to process data across hundreds of machines in parallel.
* **Language Support**: Originally written in Scala, Spark supports Scala, Java, Python (via PySpark), and R.

**What is PySpark?**

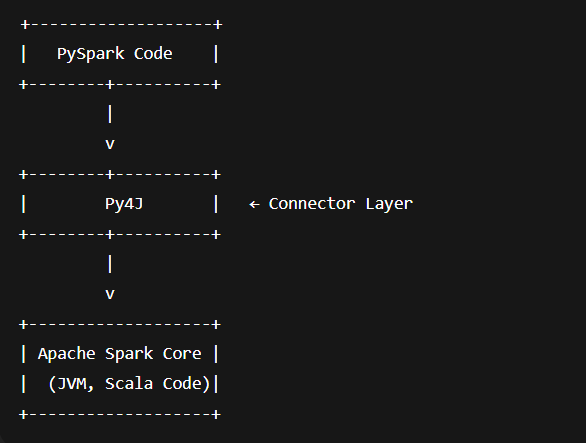
**PySpark** is the Python API for Apache Spark. It allows developers to harness the full power of Spark from Python code.

* Enables Python developers to interact with Spark’s core capabilities.
* Internally uses **Py4J**, which bridges Python and the JVM (Java Virtual Machine).

**PySpark Architecture**

**Working Mechanism:**

1. Python code is written in PySpark.
2. **Py4J** translates the Python commands and sends them to the JVM.
3. Spark (written in Scala) executes the logic on the cluster.



**Key Characteristics of RDDs:**

| **Feature** | **Description** |
| --- | --- |
| Immutability | Once created, RDDs cannot be modified. Transformations create new RDDs. |
| In-Memory Storage | Intermediate data is cached in memory for faster computation. |
| Lazy Evaluation | Transformations are not executed until an action is triggered. |
| Fault Tolerance | Lost partitions are recomputed using the transformation lineage. |
| Partitioning | RDDs are divided into partitions and distributed across nodes. |
| Persistence | Frequently accessed RDDs can be cached using persist() or cache(). |
| Granular Control | Supports both coarse-grained (entire dataset) and fine-grained (individual elements) transformations. |

**RDD Operations in Spark:**

**Transformations:**

* Return a **new RDD** and are **lazy** (executed only when an action is called).
* Do not modify the original RDD.

Examples:

* map()
* filter()
* flatMap()
* groupByKey()
* sortByKey()

**Types of Transformations:**

| **Type** | **Description** | **Examples** |
| --- | --- | --- |
| Narrow | Data needed to compute one partition lives in a single parent partition. | map(), filter() |
| Wide | Data required to compute one partition lives in multiple parent partitions. Involves shuffling. | groupByKey(), reduceByKey() |

**Actions**

* **Trigger the execution of transformations.**
* **Return results to the driver program or store them in a file.**

**Examples:**

* **collect()**
* **count()**
* **first()**
* **take()**
* **reduce()**
* **saveAsTextFile()**

**Working with Data in PySpark (Using DataFrames in Databricks Serverless)**

**Why Not RDDs?**

In Databricks **serverless compute**, you **cannot use**:

sc.parallelize()

spark.sparkContext.parallelize()

This is because **direct access to the JVM (Java Virtual Machine)** is restricted in serverless environments for security and scalability reasons. These methods rely on low-level APIs (SparkContext) which are disabled.

**Solution**: we can use DataFrames, which are high-level, optimized APIs supported on all Databricks cluster types.

**What is a DataFrame?**

A **DataFrame** in PySpark is a **distributed collection of data** organized into **named columns**, similar to a table in a relational database or a pandas DataFrame in Python.

* **Structured**: DataFrames have schema (column names and types).
* **Optimized**: Spark automatically optimizes execution using its Catalyst engine.
* **Powerful**: Supports SQL queries, complex transformations, and data integration.

Step 1:

Creating a DataFrame from a List:

# Define some sample numbers as a list of tuples

data = [(10,), (20,), (30,), (40,), (50,)]

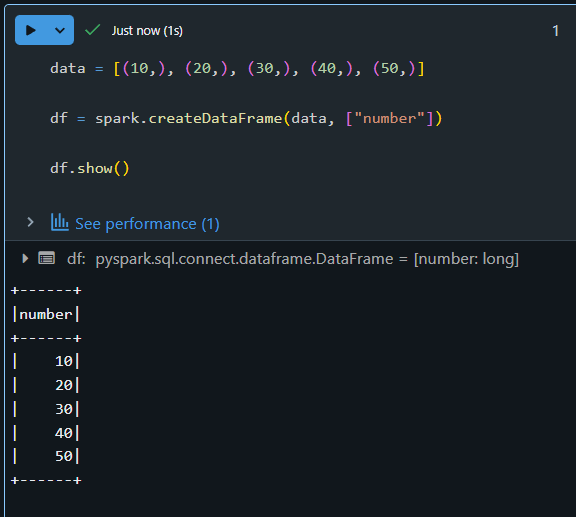
# Convert the list into a DataFrame with a column named "number"

df = spark.createDataFrame(data, ["number"])

# Display the DataFrame in tabular format

df.show()

Output:



**Explanation:**

* data = [(10,), (20,), ...]: This is a list of tuples, where each tuple represents a row.
* spark.createDataFrame(data, ["number"]): Converts the list into a distributed DataFrame with a column name.
* .show(): Displays the content in a table-like structure.

Step 2:

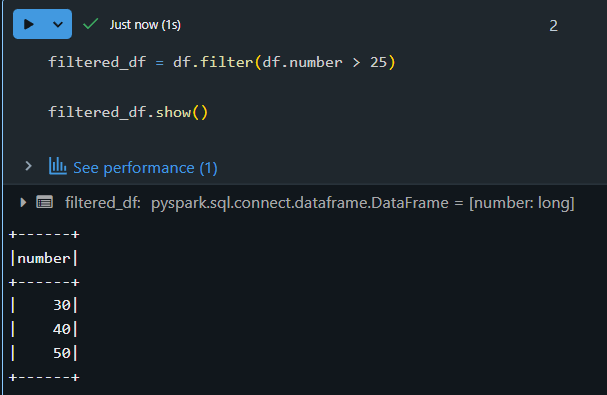
Filtering the Data (Like RDD’s filter):

# Keep only numbers greater than 25

filtered\_df = df.filter(df.number > 25)

# Display filtered results

filtered\_df.show()

OUTPUT:  


Step 3:

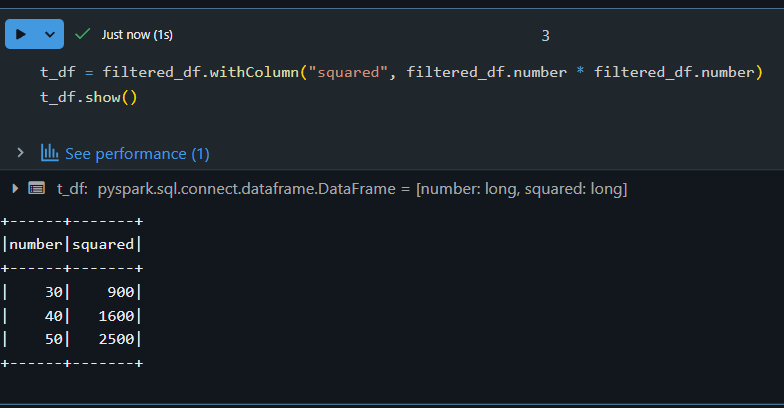
Transforming Data (Like RDD’s map):

# Create a new column "squared" by squaring the existing values

transformed\_df = filtered\_df.withColumn("squared", filtered\_df.number \* filtered\_df.number)

# Show final result

transformed\_df.show()

OUTPUT:  


**Explanation:**

* withColumn("squared", ...): Adds a new column named "squared" which contains the square of each number.
* This simulates rdd.map(lambda x: x \* x) in RDD.

Step 4:

Collecting Results to Driver:

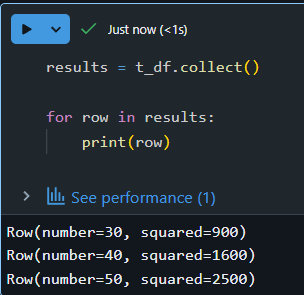
# Collect results to driver node as Python objects

results = transformed\_df.collect()

# Display each row

for row in results:

print(row)

OUTPUT:  


**Explanation:**

* .collect(): Brings all rows to the driver (like rdd.collect()).
* Looping through the collected rows lets you see each record in Python format.

**Step 5:**

**Saving Data to File System:**

# Save transformed data to a CSV in Databricks FileStore

transformed\_df.write.mode("overwrite").csv("/FileStore/output/transformed\_data")

**Explanation:**

* .write.csv(...): Writes the DataFrame as CSV files.
* mode("overwrite"): Replaces folder if it already exists.
* This replaces rdd.saveAsTextFile() in RDD.

we can view this data in the Databricks UI under **Data → DBFS → FileStore → output → transformed\_data**.

Summary Table: RDD vs DataFrame Equivalents

| **Action** | **RDD Syntax**  **(Not supported on serverless)** | **DataFrame Syntax**  **(Supported)** |
| --- | --- | --- |
| Create in memory | sc.parallelize([1,2]) | spark.createDataFrame([(1,), (2,)], ["col"]) |
| Filter data | rdd.filter(lambda x: x > 5) | df.filter(df.col > 5) |
| Map/transformation | rdd.map(lambda x: x \* x) | df.withColumn("squared", df.col \* df.col) |
| Show output | rdd.collect() | df.collect() / df.show() |
| Save to file | rdd.saveAsTextFile("path") | df.write.csv("path") |

1. **DataFrames** are the recommended approach in modern PySpark.
2. They are **faster**, **easier**, and **compatible** with Databricks serverless and cloud environments.
3. You can chain multiple operations just like with RDDs, but with better performance and readability.

RDD creation in in spark:

In Spark, RDDs (Resilient Distributed Datasets) can be created in a few ways: by parallelizing an existing collection, loading data from external storage, or transforming existing RDDs. Here's a breakdown:

1. Parallelizing an Existing Collection:

* This method involves taking a collection (like a list or array) in your driver program and distributing it across the cluster using SparkContext.parallelize().
* Example (Python):

Python

data = [1, 2, 3, 4, 5]

rdd = sc.parallelize(data)

* This is useful for quick prototyping and testing, but not ideal for large datasets where the entire dataset needs to reside on one machine initially.

2. Loading from External Storage:

Spark can read data from various sources like text files, CSV files, JSON files, and more, creating an RDD for each.

Examples:

* + Text file: rdd = sc.textFile("path/to/file.txt")
  + CSV file: (using a library like spark-csv or a custom function)
  + HDFS, HBase, or any Hadoop InputFormat supported data source

This is the most common way to work with large datasets in Spark.

3. Transforming Existing RDDs:

* RDDs can be created by applying transformations (like map, filter, flatMap) to existing RDDs.
* Example:

Python

numbers = sc.parallelize([1, 2, 3, 4, 5])

squared\_numbers = numbers.map(lambda x: x \* x)

* This allows for building complex data pipelines by chaining operations on RDDs.

**Converting Spark RDD to DataFrame and Dataset**

A screenshot of a computer

AI-generated content may be incorrect.

Spark provides 3 main abstractions to work with it. First, we will provide you with a holistic view of all of them in one place. Second, we will explore each option with examples.

[RDD](https://spark.apache.org/docs/latest/rdd-programming-guide.html) (Resilient Distributed Dataset). The main approach to work with unstructured data. Pretty similar to a distributed collection that is not always typed.

[Datasets](https://spark.apache.org/docs/latest/sql-programming-guide.html#datasets-and-dataframes). The main approach to work with semi-structured and structured data. Typed distributed collection, type-safety at a compile time, strong typing, lambda functions.

[DataFrames](https://spark.apache.org/docs/latest/sql-programming-guide.html#datasets-and-dataframes). It is the Dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. Think about it as a table in a relational database.

**Creating a SparkSession in PySpark:**

The following code initializes a **SparkSession**, which is the entry point for working with DataFrames and the Spark SQL API in PySpark:

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Spark DataFrames").getOrCreate()

**Line-by-line Breakdown:**

1. **from pyspark.sql import SparkSession**
   * This imports the SparkSession class from PySpark’s SQL module. SparkSession is the main entry point to interact with Spark’s DataFrame and SQL functionality.
2. **SparkSession.builder**
   * This is a builder pattern to configure and create a SparkSession object.
3. **.appName("Spark DataFrames")**
   * Sets the name of the Spark application. This name appears in the Spark UI and logs, useful for monitoring and debugging.
4. **.getOrCreate()**
   * This either retrieves an existing SparkSession or creates a new one if none exists. It ensures only one SparkSession is active per application.

**Creating and Displaying a DataFrame in PySpark**

The following code creates a DataFrame using PySpark and displays its content:

df = spark.createDataFrame(

[(1, 2, 3, 'a b c'),

(4, 5, 6, 'd e f'),

(7, 8, 9, 'g h i')],

['id', 'name', 'salary', 'department']

)

df.show()

**Line-by-line Breakdown:**

1. **spark.createDataFrame(...)**
   * This method creates a DataFrame from a list of tuples.
   * Each tuple represents a row of data.
2. **[(1,2,3,'a b c'), (4,5,6,'d e f'), (7,8,9,'g h i')]**
   * A list of rows. Each row contains 4 elements: integers and a string.
3. **['id', 'name', 'salary', 'department']**
   * Specifies the column names for the DataFrame.
4. **df.show()**
   * Displays the contents of the DataFrame in a tabular format.

**Output:**

+---+----+------+----------+

| id|name|salary|department|

+---+----+------+----------+

| 1| 2| 3| a b c|

| 4| 5| 6| d e f|

| 7| 8| 9| g h i|

+---+----+------+----------+

**Importing the Data**

data = spark.read.csv('/Volumes/dataset1/default/people/sales\_200\_rows.csv', header=True)

data.show()

data.printSchema()

Changing the Datatype

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

data\_schema = [StructField('order\_id', IntegerType(), True),

StructField('customer\_id', IntegerType(), True),

StructField('order\_date', StringType(), True),

StructField('product', StringType(),True),

StructField('category', StringType(),True),

StructField('quantity', StringType(),True),

StructField('price\_per\_unit', IntegerType(),True)]

defines a custom schema using StructField and data types from PySpark to precisely control how each column in a CSV file is interpreted.

Each StructField specifies the column name, data type, and whether null values are allowed.

Changing into SQL:

data.createOrReplaceTempView('people')

%sql

select \* from people