10 Important terminologies in Spark

Spark is written in **Scala and scala is primarily a JVM-based programming language**. The Scala compiler generates Java bytecode that runs on the Java Virtual Machine (JVM). So spark executors and driver are **JVM processes.**

What is JVM

The **Java Virtual Machine (JVM)** is a program that:

- Runs **Java bytecode**
- Manages memory
- Manages threads
- Provides portability across OS

Java code does **not run directly on the OS**.

It runs inside the JVM.

Flow when you write spark code using Java / Scala :

```
Java / Scala sourcecode
  ↓ compile
Java bytecode (.class)
  ↓
JVM
  ↓
Operating System
```

Same bytecode runs on:

- Linux

- Windows
- Mac

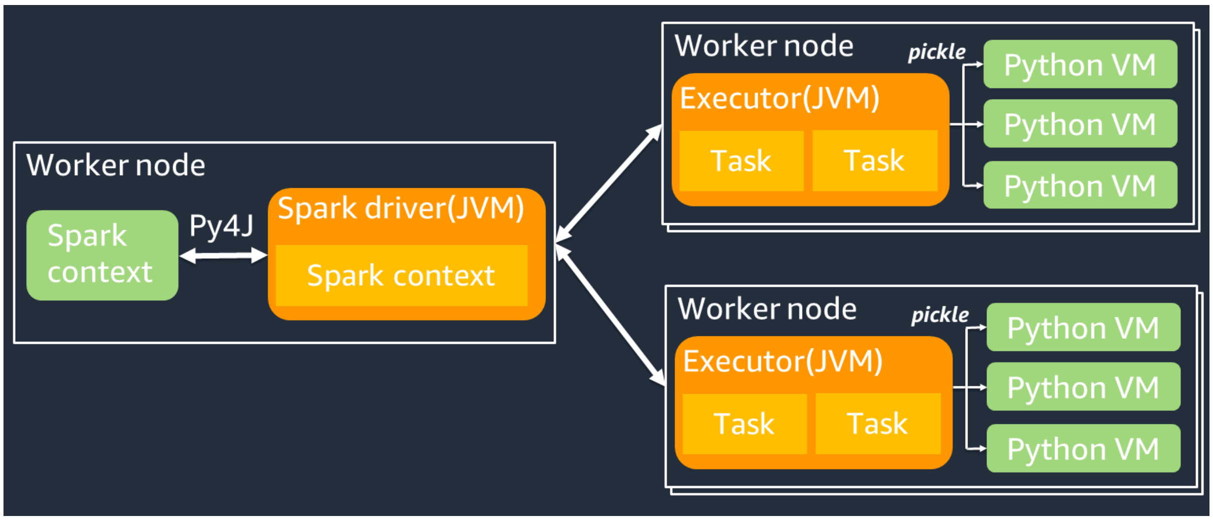
Because JVM abstracts the OS.

Flow when you write spark code using python (pyspark) :

```
Python (PySpark)
↔ Py4J ↔ (acts as a bridge)
Spark JVM
↓
OS
```

Py4J is a core component integrated into PySpark (the Python API for Apache Spark). Spark is primarily written in Scala and Java, which run on the JVM. Py4J facilitates the communication between the Python driver program and the Java Virtual Machine processes in the Spark cluster, allowing Python developers to use Spark's distributed computing capabilities seamlessly.

It establishes a bridge between the two languages using a socket connection, making it possible for Python API code to call Java methods.



In Spark you always have **multiple JVMs**.

Driver JVM

- Runs your Spark application logic
- Creates DAG, stages, tasks
- Talks to cluster manager

Executor JVMs

- Run tasks
- Process data
- Cache data
- Do shuffles

Each executor is **one JVM process**.

Inside that one JVM:
Multiple threads run
Each thread executes one task
Number of concurrent tasks = executor cores

What is garbage collection

Garbage Collection is the process of:

> Automatically finding memory that is no longer being used by a program and freeing it, so it can be reused.

In simple words:

- Your program creates objects
- Some objects are no longer needed
- GC cleans them up for you

Without GC:

- Memory would keep growing

- Program would eventually crash

Why GC exists at all

Programs allocate memory like this:

```
create object → use it → forget it
```

But the OS does **not know** when your program is done with an object.

So someone must:

- Track object usage
- Decide when an object is “dead”
- Free that memory

That “someone” is the **runtime**, via GC.

Role of Garbage Collection (GC) in Spark

> GC in Spark is responsible for freeing JVM memory used by temporary objects created during job execution so executors don’t run out of memory.

That’s the core role.

Why GC is critical in Spark

Spark runs long-lived JVM processes (driver and executors).

During a job, Spark continuously creates:

- Task metadata objects
- Shuffle buffers
- Intermediate rows
- Temporary aggregation objects
- Internal Spark objects

Once a task finishes, **most of these objects are useless**.

GC’s job is to:

- Detect they are no longer needed
- Free that memory
- Make space for the next tasks

Without GC:

- Memory would keep growing
- Executors would crash with OOM

Where GC runs in Spark

GC runs in:

- **Driver JVM**
- **Executor JVMs**

Most performance impact is from **executor GC**, not driver.

What triggers GC in Spark

GC is triggered when:

- JVM heap starts filling up
- Many temporary objects are created
- Shuffle or aggregation creates lots of intermediate data
- Python UDFs or heavy transformations create extra objects

GC is **not triggered by Spark explicitly**

It is triggered by **memory pressure**.

What happens if GC works well

- Executors keep enough free memory
- Tasks run smoothly
- No long pauses
- Stable throughput

Spark job looks healthy.

What happens if GC pressure is high

This is where problems appear.

Symptoms:

- Long task durations
- Executors appear “slow”
- High **JVM GC Time** in Spark UI
- Frequent task retries
- Sometimes executor OOM

In extreme cases:

- Full GC pauses
- Executor heartbeat timeouts
- Job failure

What is serialization

Serialization means:

> Converting an in-memory object into a stream of bytes so it can be sent or stored.

Deserialization means:

> Converting that byte stream back into an in-memory object.

Why serialization is needed at all

Computers cannot:

- Send memory pointers across processes
- Share objects across machines

So whenever data must:

- Move over network
- Move across process boundary
- Be written to disk

It must become **bytes**.

Simple real-world analogy

Object in memory:

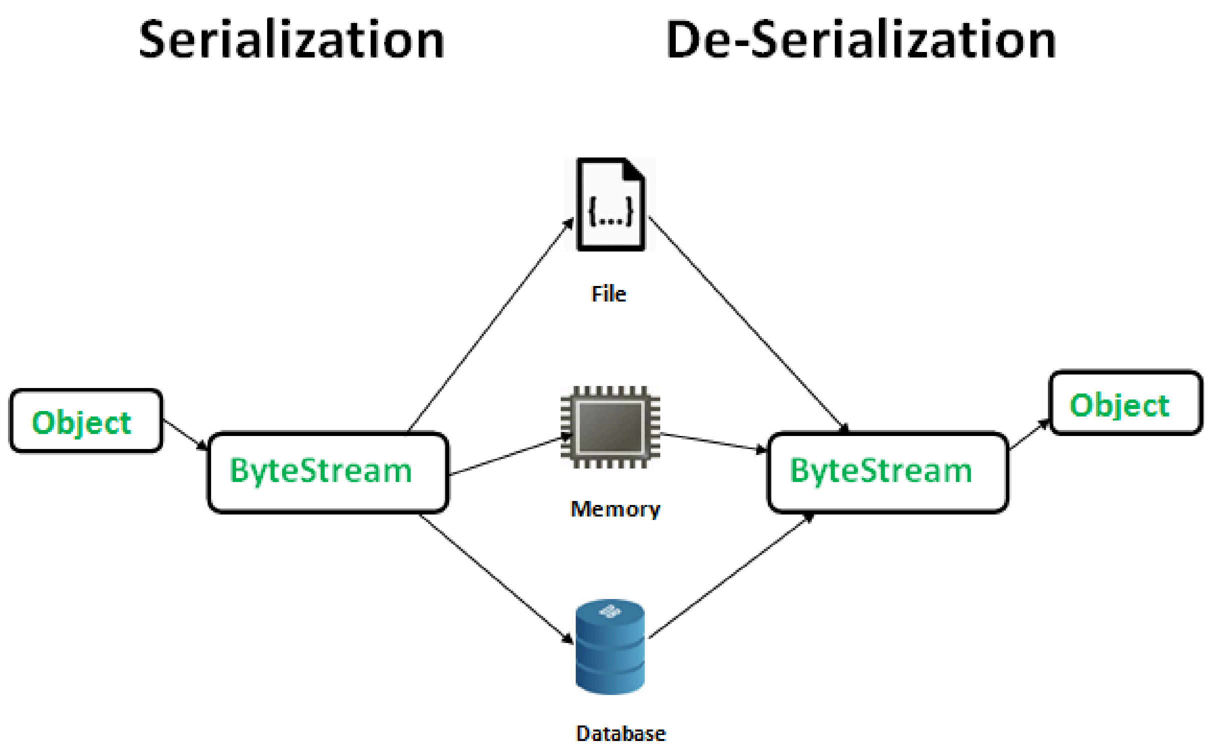
```
{name:"India", runs:350}
```

Serialized form:

```
010101011001010101...
```

Receiver:

- Reads bytes
- Reconstructs the object



```
from pyspark.sql.functions import *
orders_schema = 'order_id int,customer_id int,product_id int,unit_price floa
t,order_date date,order_status string,state string,quantity int'
df = spark.read.format("csv").schema(orders_schema).option("header" , "t
rue").load("/data/orders_1gb.csv")
df.filter(col("state")=="Assam").write.format("noop").mode("overwrite").sa
ve()
```

In Spark execution

For every task:

```
Driver JVM
   └── serialize Taskobject
         └── send to Executor
Executor JVM
   └── deserialize Taskobject   ←this is what you as task de serialization tim
```

```
e
  └── execute task logic

```

This happens **before reading even a single row**.

What Spark does internally

1. Driver builds the execution plan
2. Driver creates **many tasks**
3. Each task contains:
    - Query plan
    - Filter expression
    - Schema
    - File split info
    - Python closures (because this is PySpark)
4. Driver serializes each task
5. Executor deserializes each task

That step 5 is **Task Deserialization Time**.

In Spark, **serialization specifically means**:

> Converting an in-memory Spark object into bytes so it can be sent between Spark processes (executors, driver, Python).

Examples:

- Executor → Executor (shuffle)
- JVM → Python (UDF)
- Driver → Executor (task metadata)

Why Python UDFs are slow :

> Python UDFs are slow because Spark has to serialize data out of the JVM, execute the logic in Python, then deserialize it back into the JVM.

Normal PySpark DataFrame execution (fast)

When you use built in functions:

```
df.filter(col("qty") >10)
```

What happens:

- All logic runs **inside the Spark JVM**
- Data never leaves the JVM
- No serialization
- Spark can optimize the plan

Result: **Fast**

What changes when you use a Python UDF

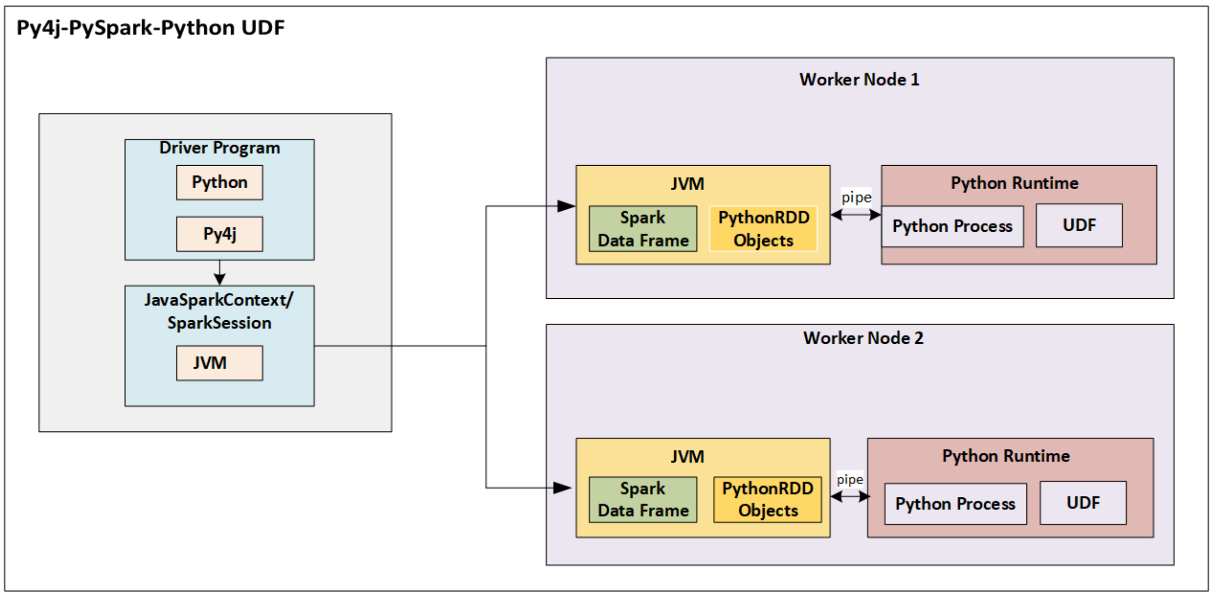
```
df.withColumn("x", my_udf(col("qty")))
```

Spark cannot run this logic inside the JVM.

So Spark must:

1. **Serialize** data from the JVM
2. Send it to a **Python process**
3. **Deserialize** it in Python
4. Run your Python function
5. **Serialize** the result in Python
6. Send it back to the JVM
7. **Deserialize** it again

This happens for every task (often row by row).



Why this makes Python UDFs slow

1. Serialization cost

Converting data to bytes and back is expensive.

2. Cross language boundary

Data moves between JVM and Python processes.

3. No Spark optimization

Spark treats the UDF as a black box.

4. Python execution cost

Python execution is slower than JVM execution.

Simple rule to remember

```
Spark built-infunctions → JVM only → fastest
Python UDF          → JVM ↔ Python → slow
```

Summary

> Python UDFs are slow because Spark must serialize data out of the JVM, execute Python logic, and deserialize the results back, breaking JVM-only execution.

Demo:

```
from pyspark.sql.functions import *
orders_schema = 'order_id int,customer_id int,product_id int,unit_price floa
t,order_date date,order_status string,state string,quantity int'
df = spark.read.format("csv").schema(orders_schema).option("header" , "t
rue").load("/data/orders_1gb.csv")
# df.filter(col("state")=="Assam").groupBy("order_status").agg(count("*").a
lias("no_of_records")).write.format("noop").mode("overwrite").save()
def get_square(x,y):
   return x*y

double_udf = udf(get_square)
df.filter(col("state")=="Assam").withColumn("double_quantity", double_udf
("quantity","unit_price")).write.format("noop").mode("overwrite").save()
```

To avoid the performance overhead, the recommendation is to implement UDF in Java/Scala that will be utilizing JVM itself as the environment for execution, which eliminates the need for serialization and network trip

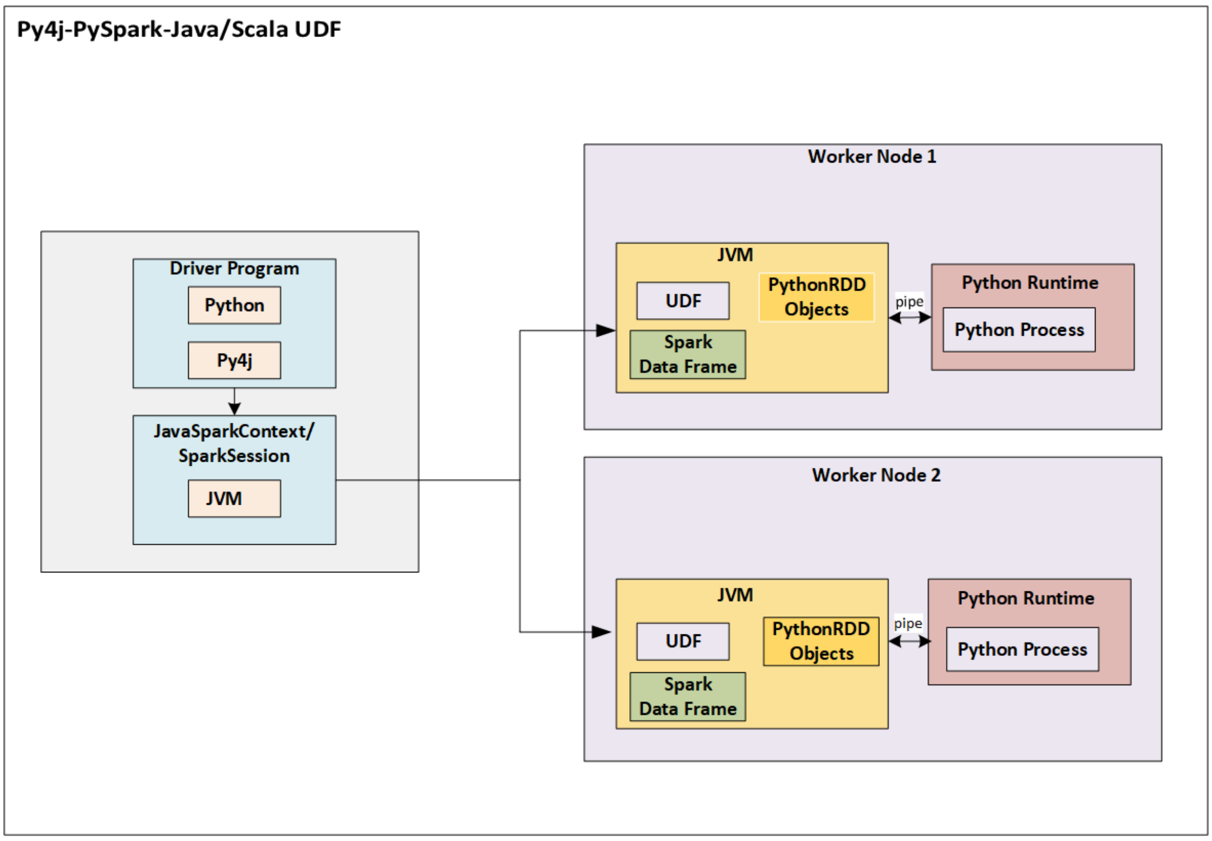


Image source : https://dzone.com/articles/pyspark-java-udf-integration-1