# NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

## AI HARDWARE AND TOOLS WORKSHOP (CACSC19)

# LAB ASSIGNMENT-4 (SPARK)

**By:**

1. **Kumar Gaurav(2021UCA1801)**
2. **Utkarsh Gupta(2021UCA1833)**
3. **Aditya Singh(2021UCA1862)**

**Branch: COMPUTER SCIENCE WITH ARTIFICIAL INTELLIGENCE - 1**

## TASK 1: Explore RDD in spark

In Spark, RDD stands for **Resilient Distributed Dataset**.

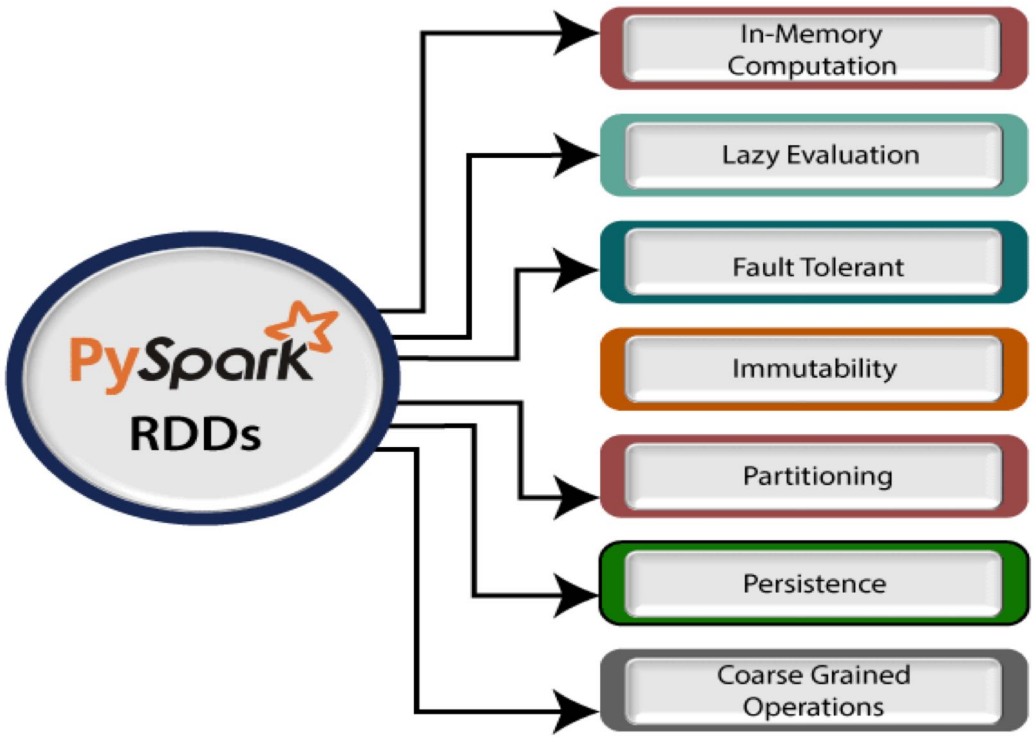
It's the fundamental data structure of Spark which represents an immutable, distributed collection of objects that can be operated on in parallel.
RDDs are fault-tolerant, meaning they can automatically recover from failures.

Some key points about RDDs in Spark:

1. Immutable: RDDs are immutable, meaning once created, they cannot be changed. However, you can transform one RDD into another through various operations.
2. Distributed: RDDs are distributed across multiple nodes in a cluster, enabling parallel processing. Each RDD is divided into multiple partitions, with each partition stored on a different node.
3. Resilient: RDDs are resilient to failures. If a partition of an RDD is lost due to a node failure, Spark can recompute the lost partition using the lineage information.
4. Lazy Evaluation: Spark uses lazy evaluation, meaning transformations on RDDs are not executed immediately. Instead, Spark builds up a directed acyclic graph (DAG) of transformations, and actions trigger the execution of this DAG.
5. Type Aware: RDDs can hold any type of Python, Java, or Scala objects. However, it's beneficial to use primitive types or specialized classes for better performance.
6. **Two types of operations:**
    - Transformations: Transformations create a new RDD from an existing one. Examples include map, filter, flatMap, etc. Transformations are lazy, so they don't compute the result immediately.
    - Actions: Actions return a value to the driver program or write data to an external storage system. Examples include reduce, collect, saveAsTextFile, etc. Actions trigger the execution of the DAG.

7. Persistence: RDDs can be persisted in memory or on disk to avoid recomputation, especially for iterative algorithms or when an RDD is reused multiple times.
8. Lineage: RDDs maintain information about how they were derived from other RDDs (lineage). This lineage information is used to recompute lost partitions in case of failures.
9. Parallel Execution: Spark automatically parallelizes operations on RDDs across the cluster, optimizing data processing and utilizing available resources efficiently.

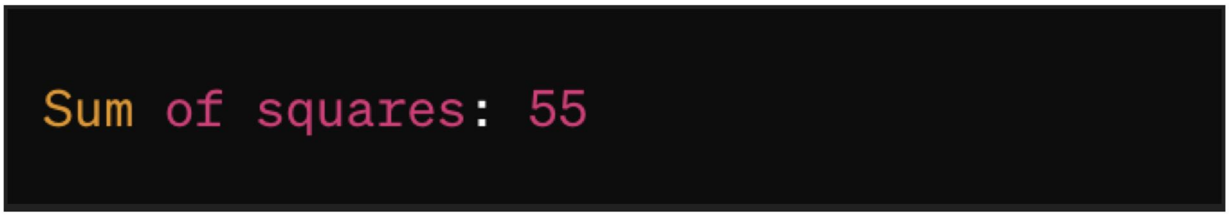
Overall, RDDs provide a powerful abstraction for distributed data processing in Spark, enabling users to perform complex computations on large datasets in a fault-tolerant and efficient manner. However, with the introduction of DataFrame and Dataset APIs, RDDs are now considered a lower-level API, and these higher-level abstractions are often preferred for most Spark applications due to their optimizations and ease of use.



**EXAMPLE**:Using RDDs in PySpark to calculate the sum of squares of a list of numbers.

```
from pyspark import SparkContext

# Initialize SparkContext

sc = SparkContext("local", "RDDExample")

# Create an RDD from a list of numbers

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

numbers_rdd = sc.parallelize(numbers)

# Square each number

squared_rdd = numbers_rdd.map(lambda x: x * x)

# Calculate the sum of squares

sum_of_squares = squared_rdd.reduce(lambda x, y: x + y)

# Output the result

print("Sum of squares:", sum_of_squares)

# Stop SparkContext

sc.stop()
```

OUTPUT:



```
Sum of squares: 55
```

Explanation:

1. Initialize SparkContext: We start by initializing a SparkContext object. This is the entry point to any Spark functionality in our Python program. We set the master to "local" to run Spark in local mode, and "RDDExample" is the application name.

2. Create an RDD: We create an RDD named `numbers_rdd` from a list of numbers `[1, 2, 3, 4, 5]` using the `parallelize` method. This distributes the data across multiple partitions.

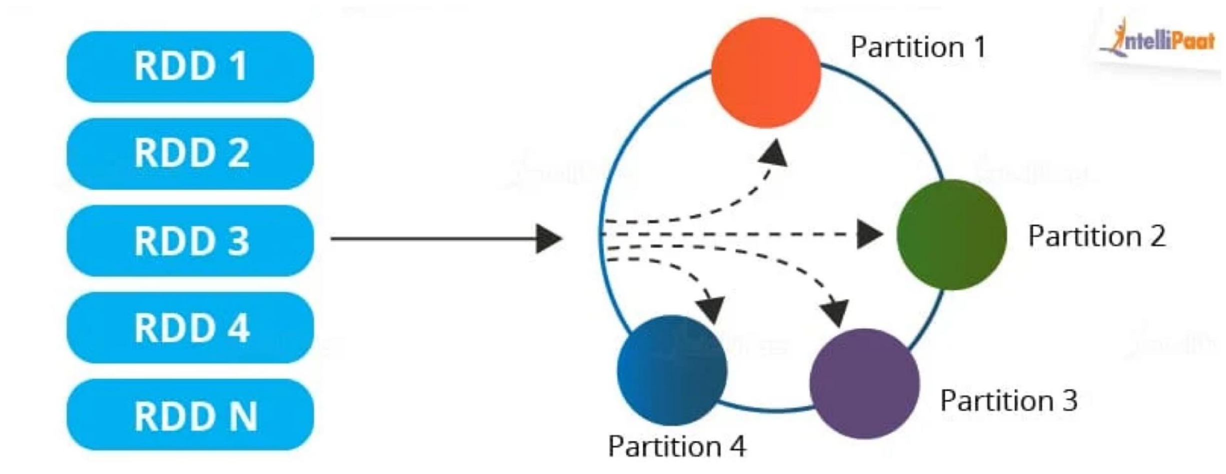
3. Map Transformation: We apply the `map` transformation to `numbers_rdd` to square each number in the RDD. The lambda function `lambda x: x * x` squares each element.

4. Reduce Action: We use the `reduce` action to calculate the sum of squares. The `reduce` function takes a lambda function that defines the operation to be performed. In this case, it's a simple addition operation `lambda x, y: x + y`.

5. Output the Result: We print the sum of squares calculated.

6. Stop SparkContext: Finally, we stop the SparkContext to release resources.

In this example, we used RDD transformations (map) and actions (reduce) to perform parallel computation on a list of numbers. The RDD abstraction allows us to distribute the computation across multiple nodes in a cluster, making it scalable and efficient for large dataset



Task 2: In PySpark, create a program that reads a CSV file containing sales data, performs data cleaning by handling missing values and removing duplicates, calculates the total sales amount for each product, and finally, outputs the results to a new CSV file. Ensure to use transformations and actions in your PySpark script

Dataset Link:
https://drive.google.com/file/d/1yy90qb8vk7Ft9VpE4e1HaOT3ESd3z0hy/view?usp=sharing

## Product sales results CSV file:

**https://drive.google.com/file/d/19ah_5Oz9GTjGUXPHXiXaRsqtyEDl4-rM/view?usp=sharing**

```
pip install pyspark


# IMPORTING NECESSARY DEPENDENCIES


from pyspark.sql import SparkSession


from pyspark.sql.functions import col, sum


#Initializing a Spark Session using PySpark library


#SparkSession.builder: This is a builder pattern used to configure and create a SparkSession.

#.appName("SalesDataAnalysis"): to set the name of the Spark application to "SalesDataAnal-ysis".
#.getOrCreate(): This method retrieves an existing SparkSession if it exists or creates a new one if none exists.


spark = SparkSession.builder \
    .appName("SalesDataAnalysis") \
    .getOrCreate()


# Read CSV file into DataFrame
sales_df = spark.read.csv("/content/sales_data_sample.csv", header=True, inferSchema=True)


sales_df
```

```
DataFrame[ORDERNUMBER: int, QUANTITYORDERED: int, PRICEEACH: double, ORDERLINENUMBER: int, SALES: double,
ORDERDATE: string, STATUS: string, QTR_ID: int, MONTH_ID: int, YEAR_ID: int, PRODUCTLINE: string, MSRP: int,
PRODUCTCODE: string, CUSTOMERNAME: string, PHONE: string, ADDRESSLINE1: string, ADDRESSLINE2: string, CITY: string,
STATE: string, POSTALCODE: string, COUNTRY: string, TERRITORY: string, CONTACTLASTNAME: string, CONTACTFIRSTNAME:
string, DEALSIZE: string]
```

```
#Performing data cleaning: handling missing values and remov- ing duplicates


cleaned_sales_df = sales_df.dropDuplicates().na.drop()


 cleaned_sales_df
```

```
DataFrame[ORDERNUMBER: int, QUANTITYORDERED: int, PRICEEACH: double, ORDERLINENUMBER: int, SALES: double,
ORDERDATE: string, STATUS: string, QTR_ID: int, MONTH_ID: int, YEAR_ID: int, PRODUCTLINE: string, MSRP: int,
PRODUCTCODE: string, CUSTOMERNAME: string, PHONE: string, ADDRESSLINE1: string, ADDRESSLINE2: string, CITY: string,
STATE: string, POSTALCODE: string, COUNTRY: string, TERRITORY: string, CONTACTLASTNAME: string, CONTACTFIRSTNAME:
string, DEALSIZE: string]
```

```
#The "PRODUCTCODE" column is being used to aggregate the cleaned_sales_df DataFrame, which shows the cleaned sales data

#Groups of rows with identical values in the "PRODUCTCODE" column are produced by the groupBy("PRODUCTCODE") function. T

#The grouped data is then aggregated using the agg() method. Sum("SALES") in agg() determines the total of the "SALES" c

#Lastly, a new name "TotalSalesAmount" is assigned to the aggregated column, which represents the total sales amount for


# Handle missing values (replace nulls with 0)
sales_df = sales_df.na.fill(0)


# Remove duplicates
sales_df = sales_df.dropDuplicates()


sales_df.printSchema()
```

```
root
 |-- ORDERNUMBER: integer (nullable = true)
 |-- QUANTITYORDERED: integer (nullable = true)
 |-- PRICEEACH: double (nullable = false)
 |-- ORDERLINENUMBER: integer (nullable = true)
 |-- SALES: double (nullable = false)
 |-- ORDERDATE: string (nullable = true)
 |-- STATUS: string (nullable = true)
 |-- QTR_ID: integer (nullable = true)
 |-- MONTH_ID: integer (nullable = true)
 |-- YEAR_ID: integer (nullable = true)
 |-- PRODUCTLINE: string (nullable = true)
 |-- MSRP: integer (nullable = true)
 |-- PRODUCTCODE: string (nullable = true)
 |-- CUSTOMERNAME: string (nullable = true)
 |-- PHONE: string (nullable = true)
 |-- ADDRESSLINE1: string (nullable = true)
 |-- ADDRESSLINE2: string (nullable = true)
 |-- CITY: string (nullable = true)
 |-- STATE: string (nullable = true)
 |-- POSTALCODE: string (nullable = true)
 |-- COUNTRY: string (nullable = true)
 |-- TERRITORY: string (nullable = true)
 |-- CONTACTLASTNAME: string (nullable = true)
 |-- CONTACTFIRSTNAME: string (nullable = true)
 |-- DEALSIZE: string (nullable = true)
```

```
# Calculate total sales amount for each product
product_sales_df = cleaned_sales_df.groupBy("PRODUCTCODE").agg(sum("SALES"). alias("TotalSalesAmount"))
```

```
product_sales_df
```

```
DataFrame[PRODUCTCODE: string, TotalSalesAmount: double]
```

```
#Final result is stored in product_sales_results
```

```
# Output the results to a new CSV file
product_sales_df.coalesce(1).write.csv("/content/product_sales_results.csv", mode="overwrite", header=True)
```

```
product_sales_df.show()
```

```
+-----------+------------------+
|PRODUCTCODE|  TotalSalesAmount|
+-----------+------------------+
|   S18_4600|12936.099999999999|
|   S18_1749|           11174.1|
|   S12_3891|          12547.32|
|   S18_2248|           3931.64|
|   S32_1268| 4706.639999999999|
|   S12_1099|            5019.9|
|   S18_2795|          19255.22|
|   S24_1937|           9554.31|
|   S32_3522|          12700.04|
|   S18_1097|          12626.71|
|   S12_1666|          18616.09|
|   S24_3969|            3647.1|
|   S24_4048|          10691.41|
|   S24_1578|           8652.03|
|   S18_3320|10743.779999999999|
|   S18_3136|16425.280000000002|
|   S32_2509|           3873.24|
|   S24_2887|          18576.34|
|   S18_4409|12215.029999999999|
|   S10_4757|           1201.25|
+-----------+------------------+
only showing top 20 rows
```

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