Exploring RDD in Spark

What is RDD in Spark?

An RDD (Resilient Distributed Dataset) is a core data structure in Apache Spark, forming its backbone since its inception. It represents an immutable, fault-tolerant collection of elements that can be processed in parallel across a cluster of machines. RDDs serve as the fundamental building blocks in Spark, upon which newer data structures like datasets and data frames are constructed.

RDDs are designed for distributed computing, dividing the dataset into logical partitions. This logical partitioning enables efficient and scalable processing by distributing different data segments across different nodes within the cluster. RDDs can be created from various data sources, such as Hadoop Distributed File System (HDFS) or local file systems, and can also be derived from existing RDDs through transformations.

Being the core abstraction in Spark, RDDs encompass a wide range of operations, including transformations (such as map, filter, and reduce) and actions (like count and collect). These operations allow users to perform complex data manipulations and computations on RDDs. RDDs provide fault tolerance by keeping track of the lineage information necessary to reconstruct lost partitions.

In summary, RDDs serve as the foundational data structure in Spark, enabling distributed processing and fault tolerance. They are integral to achieving efficient and scalable data processing in Apache Spark.

When to use RDDs?

RDD is preferred to use when you want to apply low-level transformations and actions. It gives you a greater handle and control over your data. RDDs can be used when the data is highly unstructured such as media or text streams. RDDs are used when you want to add functional programming constructs rather than domain-specific expressions. RDDs are used in the situation where the schema is not applied.

Features of Spark RDD

Spark RDD possesses the following features.

Immutability

The important fact about RDD is, it is immutable. You cannot change the state of RDD. If you want to change the state of RDD, you need to create a copy of the existing RDD and perform your required operations. Hence, the required RDD can be retrieved at any time.

In-memory computation

Data stored in a disk takes much time to load and process. Spark supports in-memory computation which stores data in RAM instead of disk. Hence, the computation power of Spark is highly increased.

Lazy evaluation

Transformations in RDDs are implemented using lazy operations. In lazy evaluation, the results are not computed immediately. It will generate the results, only when the action is triggered. Thus, the performance of the program is increased.

Fault-tolerant

As I said earlier, once you perform any operations in an existing RDD, a new copy of that RDD is created, and the operations are performed on the newly created RDD. Thus, any lost data can be recovered easily and recreated. This feature makes Spark RDD fault-tolerant.

Partitioning

Data items in RDDs are usually huge. This data is partitioned and send across different nodes for distributed computing.

Persistence

Intermediate results generated by RDD are stored to make the computation easy. It makes the process optimized.

Grained operation

Spark RDD offers two types of grained operations namely coarse-grained and fine-grained. The coarse-grained operation allows us to transform the whole dataset while the fine-grained operation allows us to transform individual elements in the dataset.

Operations of RDD

Two operations can be applied in RDD. One is transformation. And another one in action.

Transformations

Transformations are the processes that you perform on an RDD to get a result which is also an RDD. The example would be applying functions such as filter(), union(), map(), flatMap(), distinct(), reduceByKey(), mapPartitions(), sortBy() that would create an another resultant RDD. Lazy evaluation is applied in the creation of RDD.

Actions

Actions return results to the driver program or write it in a storage and kick off a computation. Some examples are count(), first(), collect(), take(), countByKey(), collectAsMap(), and reduce().

Transformations will always return RDD whereas actions return some other data type.

Practical demo of RDD operations

Let's take a practical look at some of the RDD operations. To practice Apache Spark, you need to install Cloudera virtual environment.

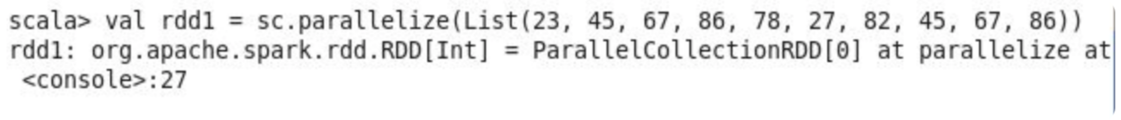
Create RDD

First, let's create an RDD using parallelize() method which is the simplest method.

```
val rdd1 = sc.parallelize(List(23, 45, 67, 86, 78, 27, 82, 45, 67, 86))
```

Here, sc denotes SparkContext

and each element is copied to form RDD.



```
scala> val rdd1 = sc.parallelize(List(23, 45, 67, 86, 78, 27, 82, 45, 67, 86))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
 <console>:27
```

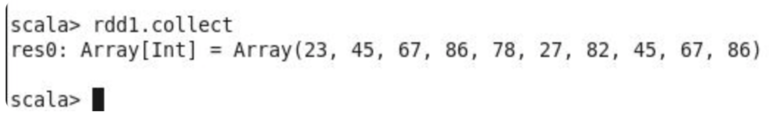
Read Result

We can read the result generated by RDD by using the collect operation.

```
rdd1.collect
```

The results are shown

here.

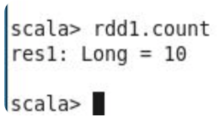


```
scala> rdd1.collect
res0: Array[Int] = Array(23, 45, 67, 86, 78, 27, 82, 45, 67, 86)

scala> 
```

Count

The count action is used to get the total number of elements present in the particular RDD.

```
rdd1.count
```



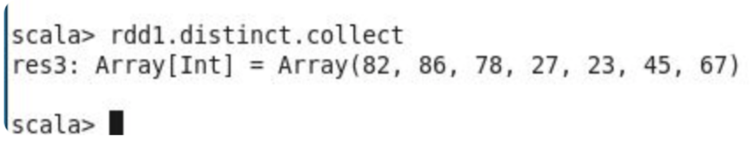
```
scala> rdd1.count
res1: Long = 10

scala> 
```

There are 10 elements in rdd1.

Distinct

Distinct is a type of transformation that is used to get the unique elements in the RDD.

```
rdd1.distinct.collect
```



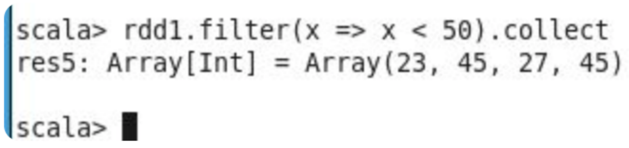
```
scala> rdd1.distinct.collect
res3: Array[Int] = Array(82, 86, 78, 27, 23, 45, 67)

scala> █
```

The distinct elements are displayed.

Filter

Filter transformation creates a new dataset by selecting the elements according to the given condition.

```
rdd1.filter(x => x < 50).collect
```



```
scala> rdd1.filter(x => x < 50).collect
res5: Array[Int] = Array(23, 45, 27, 45)

scala> █
```

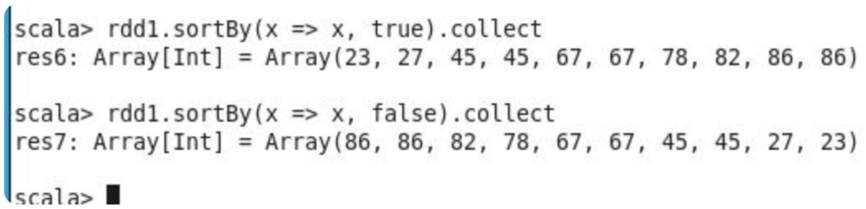
Here, the elements which are less than 50 are displayed.

sortBy

sortBy operation is used to arrange the elements in ascending order when the condition is true and in descending order when the condition is false.

```
rdd1.sortBy(x => x, true).collect
```

```
rdd1.sortBy(x => x, false).collect
```

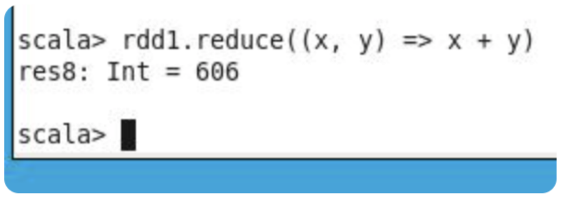


```
scala> rdd1.sortBy(x => x, true).collect
res6: Array[Int] = Array(23, 27, 45, 45, 67, 67, 78, 82, 86, 86)

scala> rdd1.sortBy(x => x, false).collect
res7: Array[Int] = Array(86, 86, 82, 78, 67, 67, 45, 45, 27, 23)

scala>
```

Reduce

Reduce action is used to summarize the RDD based on the given formula.

```
rdd1.reduce((x, y) => x + y)
```



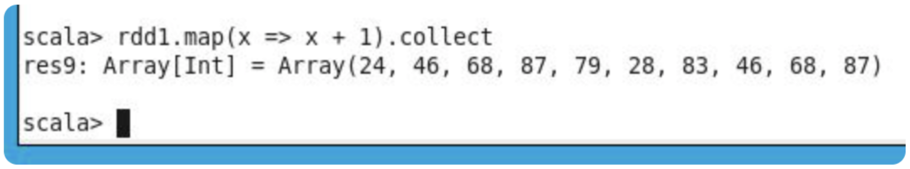
```
scala> rdd1.reduce((x, y) => x + y)
res8: Int = 606

scala>
```

Here, each element is added and the total sum is printed.

Map

Map transformation processes each element in the RDD according to the given condition and creates a new RDD.

```
rdd1.map(x => x + 1).collect
```



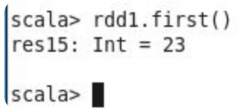
```
scala> rdd1.map(x => x + 1).collect
res9: Array[Int] = Array(24, 46, 68, 87, 79, 28, 83, 46, 68, 87)

scala> 
```

Here, each element is incremented once.

First

First is a type of action that always returns the first element of the RDD.

```
rdd1.first()
```



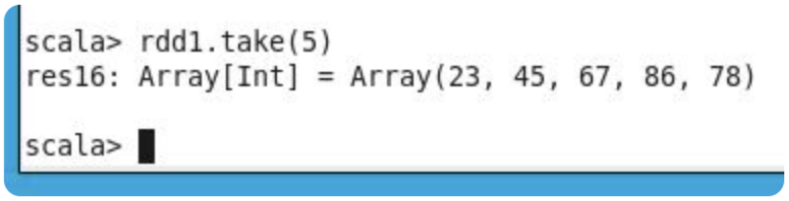
```
scala> rdd1.first()
res15: Int = 23

scala> 
```

Here, the first element in rdd1 is 23.

Take

Take action returns the first n elements in the RDD.

```
rdd1.take(5)
```



```
scala> rdd1.take(5)
res16: Array[Int] = Array(23, 45, 67, 86, 78)

scala> 
```

Here, the first 5 elements are displayed.

When you do any transformations, only copies of existing RDDs are created and the initially created RDD doesn’t change. This is because RDDs are immutable. This feature makes RDDs fault-tolerant and the lost data can also be recovered easily.

How to create RDD?

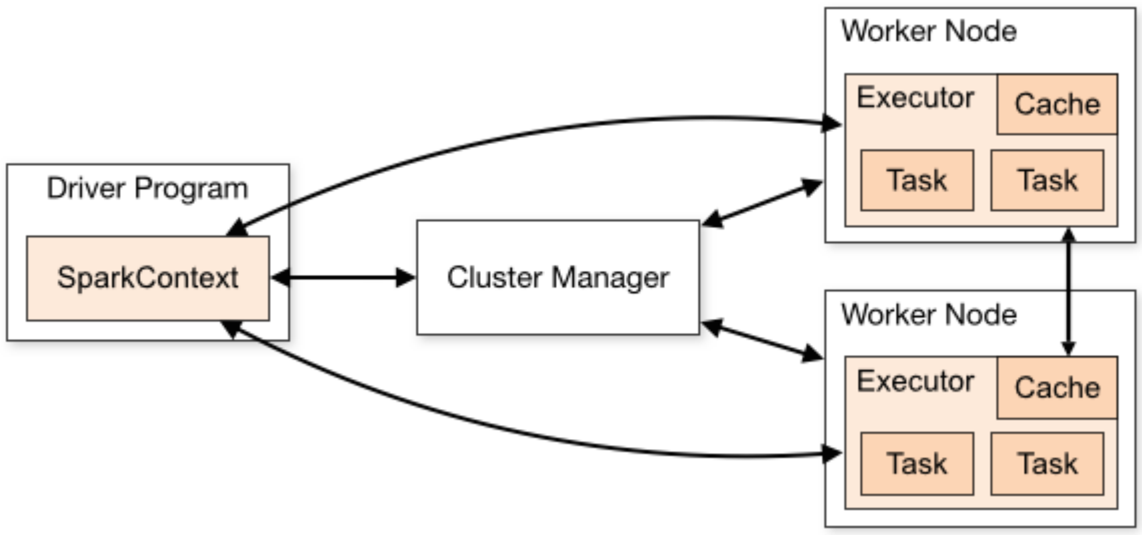
In Apache Spark, RDDs can be created in three ways.

- Parallelize method by which already existing collection can be used in the driver program.
- By referencing a dataset that is present in an external storage system such as HDFS, HBase.
- New RDDs can be created from an existing RDD.

Exploring Spark

What is Apache Spark?

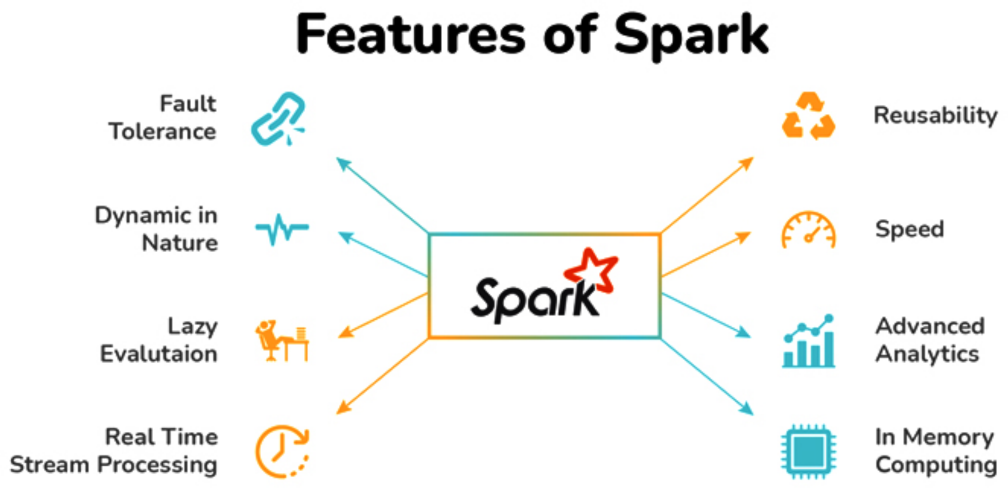
Apache Architecture, Source: Apache Spark



Apache Spark is a sort of engine which helps in operating and executing the **data analysis, data engineering**, and **machine learning** tasks both in the cloud as well as on a local machine, and for that, it can either use a single machine or the **clusters** i.e **distributed system**.

Features of Apache Spark

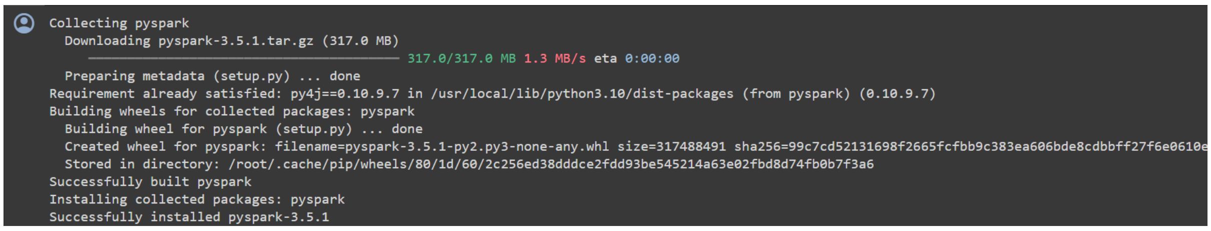
We already have some relevant tools available in the market which can perform the data engineering tasks so in this section we will discuss why we should choose Apache Spark over its other alternatives.



1. **Streaming data**: When we say streaming the data it is in the form of `batch streaming` and in this key feature Apache Spark will be able to stream our data in **real-time** by using our preferred programming language.
2. **Increasing Data science scalability**: Apache Spark is one of the widely used engines for scalable computing and to perform Data science task which requires high computational power Apache Spark should be the first choice.
3. **Handling Big Data projects**: As previously mentioned that it has high computational power so for that reason it can handle Big data projects in **cloud computing** as well using the **distributed systems/clusters** when working with the cloud, not on the local machines.

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.

```
pip install pyspark
```



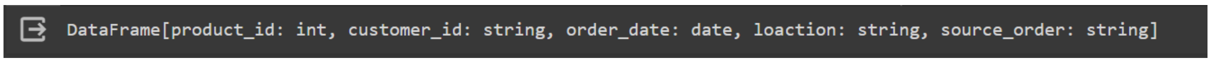
```
Collecting pyspark
  Downloading pyspark-3.5.1.tar.gz (317.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 317.0/317.0 MB 1.3 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.10/dist-packages (from pyspark) (0.10.9.7)
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.5.1-py2.py3-none-any.whl size=317488491 sha256=99c7cd52131698f2665fcfbb9c383ea606bde8cdbbff27f6e0610e
  Stored in directory: /root/.cache/pip/wheels/80/1d/60/2c256ed38dddce2fdd93be545214a63e02fbd8d74fb0b7f3a6
Successfully built pyspark
Installing collected packages: pyspark
Successfully installed pyspark-3.5.1
```

Sales Data Frame

```
from pyspark.sql.types import StructType, StructField, Intege

schema=StructType([
    StructField("product_id", IntegerType(), True),
    StructField("customer_id", StringType(), True),
    StructField("order_date", DateType(), True),
    StructField("loaction", StringType(), True),
    StructField("source_order", StringType(), True)
])
```

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder \
    .appName("aiht_project") \
    .getOrCreate()

sales_df = spark.read.format("csv").option("inferschema", "tr

display(sales_df)
```

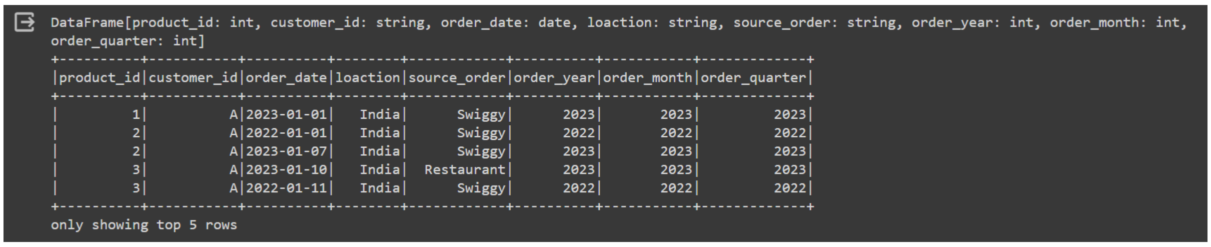


```
DataFrame[product_id: int, customer_id: string, order_date: date, loaction: string, source_order: string]
```

Deriving year

```
from pyspark.sql.functions import year

sales_df = sales_df.withColumn("order_year", year(sales_df.or
```

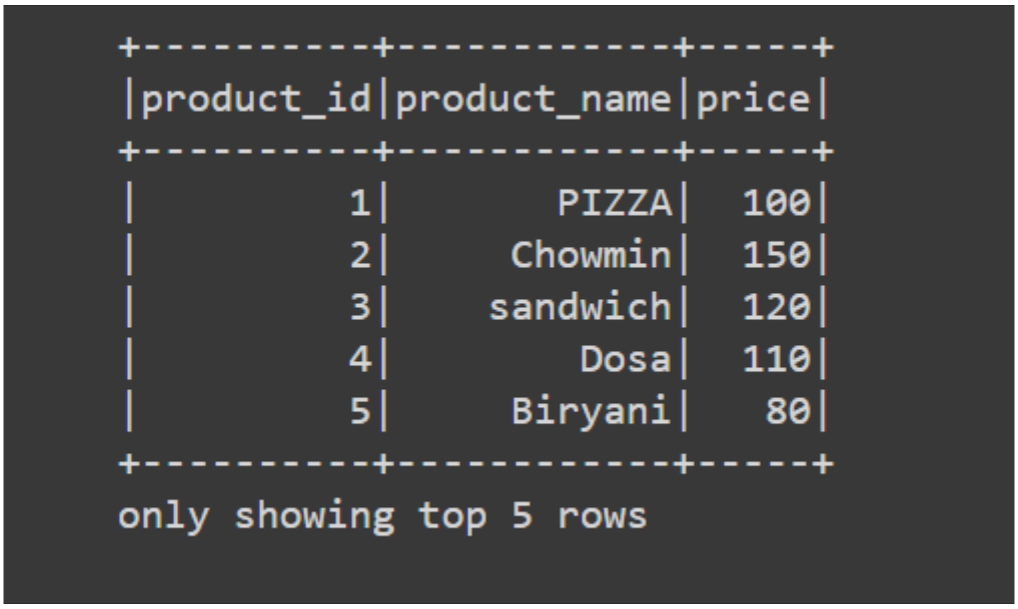
```
display(sales_df)
sales_df.show(n=5)
```



```
DataFrame[product_id: int, customer_id: string, order_date: date, loaction: string, source_order: string, order_year: int, order_month: int,
order_quarter: int]
+----------+-----------+----------+--------+------------+----------+-----------+-------------+
|product_id|customer_id|order_date|loaction|source_order|order_year|order_month|order_quarter|
+----------+-----------+----------+--------+------------+----------+-----------+-------------+
|         1|          A|2023-01-01|   India|      Swiggy|      2023|       2023|         2023|
|         2|          A|2022-01-01|   India|      Swiggy|      2022|       2022|         2022|
|         2|          A|2023-01-07|   India|      Swiggy|      2023|       2023|         2023|
|         3|          A|2023-01-10|   India|  Restaurant|      2023|       2023|         2023|
|         3|          A|2022-01-11|   India|      Swiggy|      2022|       2022|         2022|
+----------+-----------+----------+--------+------------+----------+-----------+-------------+
only showing top 5 rows
```

Menu Data Frame

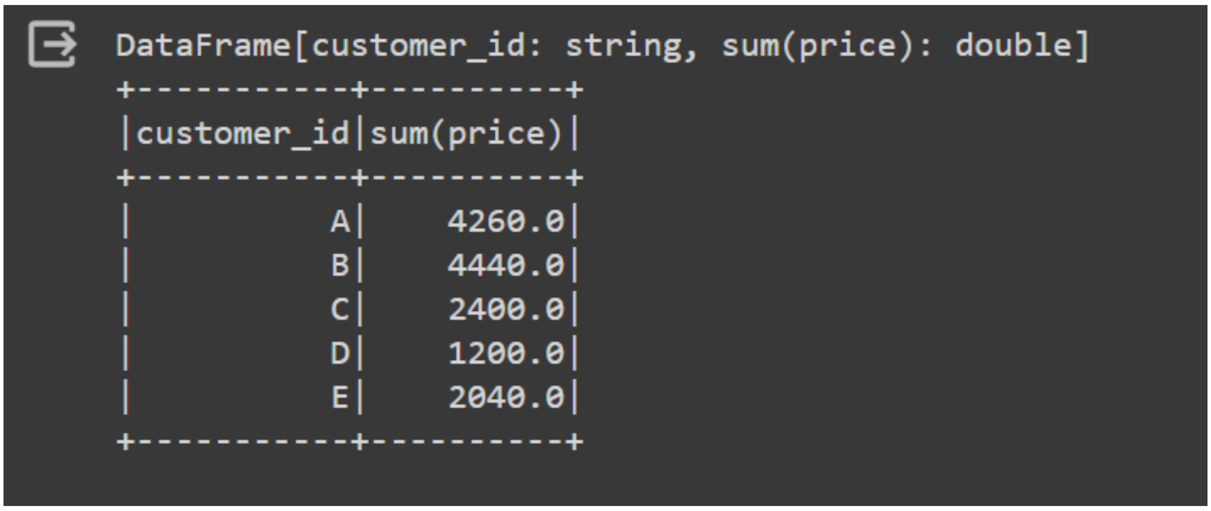
```
from pyspark.sq.types import StructType, StructField, Integer

schema=StructType([
    StructField("product_id", IntegerType(), True),
    StructField("product_name", StringType(), True),
    StructField("price", StringType(), True),

])

menu_df = spark.read.format("csv").option("inferschema", "tru
menu_df.show(n=5)
```



```
+----------+------------+-----+
|product_id|product_name|price|
+----------+------------+-----+
|         1|       PIZZA|  100|
|         2|     Chowmin|  150|
|         3|    sandwich|  120|
|         4|        Dosa|  110|
|         5|     Biryani|   80|
+----------+------------+-----+
only showing top 5 rows
```

Total amount spent by each customer

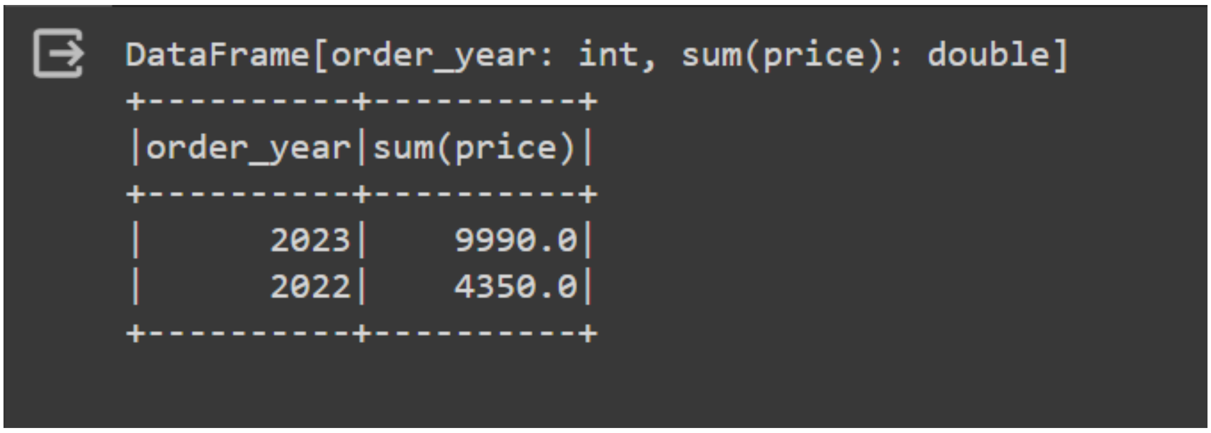
```
total_amount_spent =(sales_df.join(menu_df, 'product_id').gro
display(total_amount_spent)
total_amount_spent.show(n=5)
```



```
DataFrame[customer_id: string, sum(price): double]
+-----------+----------+
|customer_id|sum(price)|
+-----------+----------+
|          A|    4260.0|
|          B|    4440.0|
|          C|    2400.0|
|          D|    1200.0|
|          E|    2040.0|
+-----------+----------+
```

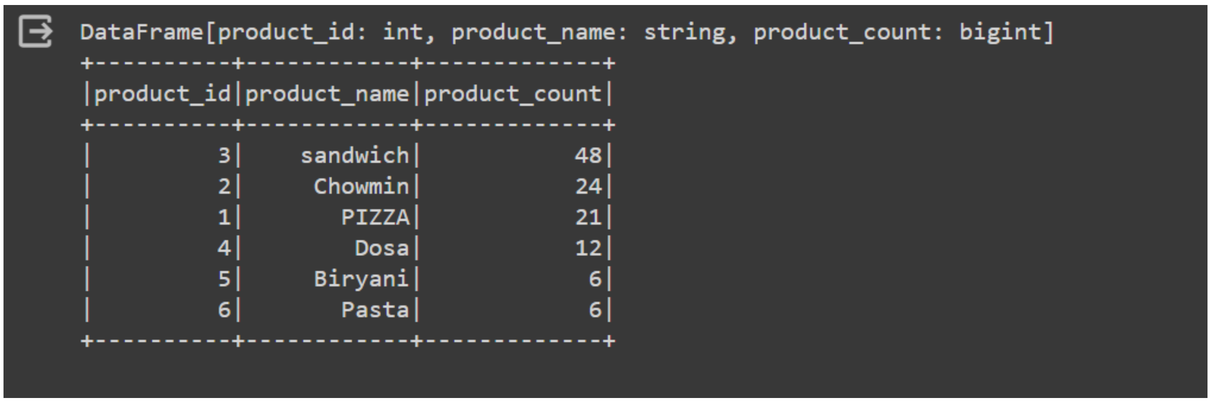
Total amount of sales in each month

```
df1 =(sales_df.join(menu_df, 'product_id').groupBy('order_yea
display(df1)
df1.show(n=2)
```


```
DataFrame[order_year: int, sum(price): double]
+----------+----------+
|order_year|sum(price)|
+----------+----------+
|      2023|    9990.0|
|      2022|    4350.0|
+----------+----------+
```

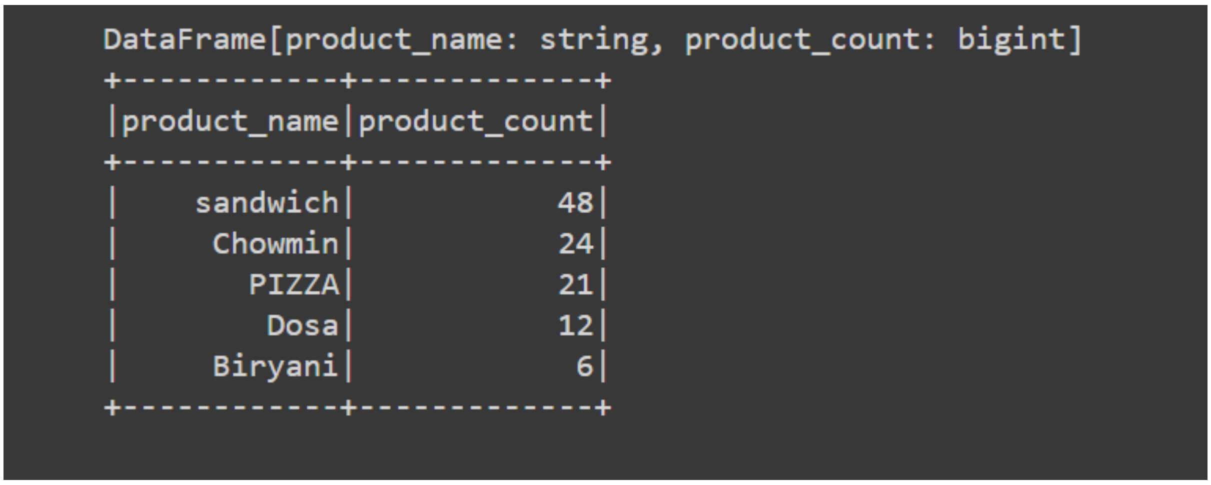
How many times each product purchased

```
from pyspark.sql.functions import count
most_df = (sales_df.join(menu_df,'product_id').groupBy('produ

           )
display(most_df)
most_df.show(n=6)
```


```
DataFrame[product_id: int, product_name: string, product_count: bigint]
+----------+------------+-------------+
|product_id|product_name|product_count|
+----------+------------+-------------+
|         3|    sandwich|           48|
|         2|     Chowmin|           24|
|         1|       PIZZA|           21|
|         4|        Dosa|           12|
|         5|     Biryani|            6|
|         6|       Pasta|            6|
+----------+------------+-------------+
```

Top 5 ordered items

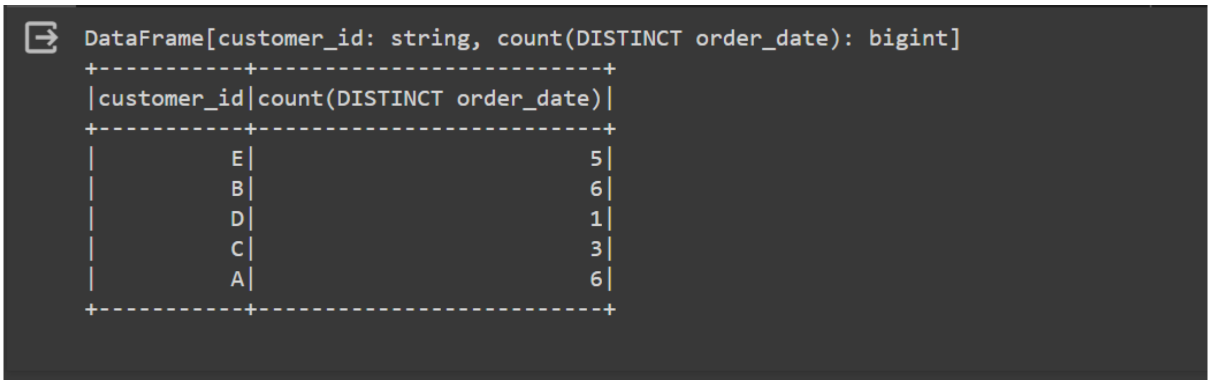
```
from pyspark.sql.functions import count
most_df = (sales_df.join(menu_df,'product_id').groupBy('produ
)

display(most_df)
most_df.show(n=5)
```



```
DataFrame[product_name: string, product_count: bigint]
+------------+-------------+
|product_name|product_count|
+------------+-------------+
|    sandwich|           48|
|     Chowmin|           24|
|       PIZZA|           21|
|        Dosa|           12|
|     Biryani|            6|
+------------+-------------+
```

Frequency of customer visited to restaurant

```
from pyspark.sql.functions import countDistinct

freq = (sales_df.filter(sales_df.source_order=='Restaurant').

      )
display(freq)
freq.show(n=5)
```



```
DataFrame[customer_id: string, count(DISTINCT order_date): bigint]
+-----------+--------------------------+
|customer_id|count(DISTINCT order_date)|
+-----------+--------------------------+
|          E|                         5|
|          B|                         6|
|          D|                         1|
|          C|                         3|
|          A|                         6|
+-----------+--------------------------+
```