Top 50 PySpark Interview Questions and Answers

PySpark Dataframe Interview Questions

Q1. What's the difference between an RDD, a DataFrame, and a DataSet?

RDD-

- It is Spark's structural square. RDDs contain all datasets and dataframes.
- If a similar arrangement of data needs to be calculated again, RDDs can be efficiently reserved.
- It's useful when you need to do low-level transformations, operations, and control on a dataset.
- It's more commonly used to alter data with functional programming structures than with domain-specific expressions.

DataFrame-

- It allows the structure, i.e., lines and segments, to be seen. You can think of it as a database table.
- Optimized Execution Plan- The catalyst analyzer is used to create query plans.
- One of the limitations of dataframes is Compile Time Wellbeing, i.e., when the structure of information is unknown, no control of information is possible.
- Also, if you're working on Python, start with DataFrames and then switch to RDDs if you need more flexibility.

DataSet (A subset of DataFrames)-

- It has the best encoding component and, unlike information edges, it enables time security in an organized manner.
- If you want a greater level of type safety at compile-time, or if you want typed JVM objects, Dataset is the way to go.

- Also, you can leverage datasets in situations where you are looking for a chance to take advantage of Catalyst optimization or even when you are trying to benefit from Tungsten's fast code generation.

Q2. How can you create a DataFrame a) using existing RDD, and b) from a CSV file?

Here's how we can create DataFrame using existing RDDs-

The toDF() function of PySpark RDD is used to construct a DataFrame from an existing RDD. The DataFrame is constructed with the default column names "_1" and "_2" to represent the two columns because RDD lacks columns.

dfFromRDD1 = rdd.toDF()

dfFromRDD1.printSchema()

Here, the printSchema() method gives you a database schema without column names-

root

|-- _1: string (nullable = true)

|-- _2: string (nullable = true)

Use the toDF() function with column names as parameters to pass column names to the DataFrame, as shown below.-

columns = ["language","users_count"]

dfFromRDD1 = rdd.toDF(columns)

dfFromRDD1.printSchema()

The above code snippet gives you the database schema with the column names-

root

|-- language: string (nullable = true)

|-- users: string (nullable = true)

Q3. Explain the use of StructType and StructField classes in PySpark with examples.

The StructType and StructField classes in PySpark are used to define the schema to the DataFrame and create complex columns such as nested struct, array, and map columns. StructType is a collection of StructField objects that determines column name, column data type, field nullability, and metadata.

- PySpark imports the StructType class from pyspark.sql.types to describe the DataFrame's structure. The DataFrame's printSchema() function displays StructType columns as "struct."
- To define the columns, PySpark offers the pyspark.sql.types import StructField class, which has the column name (String), column type (DataType), nullable column (Boolean), and metadata (MetaData).

Example showing the use of StructType and StructField classes in PySpark-

```
import pyspark

from pyspark.sql import SparkSession

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

spark = SparkSession.builder.master("local[1]") \

                    .appName('ProjectPro') \

                    .getOrCreate()

data = [("James","","William","36636","M",3000),

   ("Michael","Smith","","40288","M",4000),
```

```
    ("Robert","","Dawson","42114","M",4000),
    ("Maria","Jones","39192","F",4000)
  ]
schema = StructType([ \
    StructField("firstname",StringType(),True), \
    StructField("middlename",StringType(),True), \
    StructField("lastname",StringType(),True), \
    StructField("id", StringType(), True), \
    StructField("gender", StringType(), True), \
    StructField("salary", IntegerType(), True) \
  ])
df = spark.createDataFrame(data=data,schema=schema)
df.printSchema()
df.show(truncate=False)
```

Q5. What are the different ways to handle row duplication in a PySpark DataFrame?

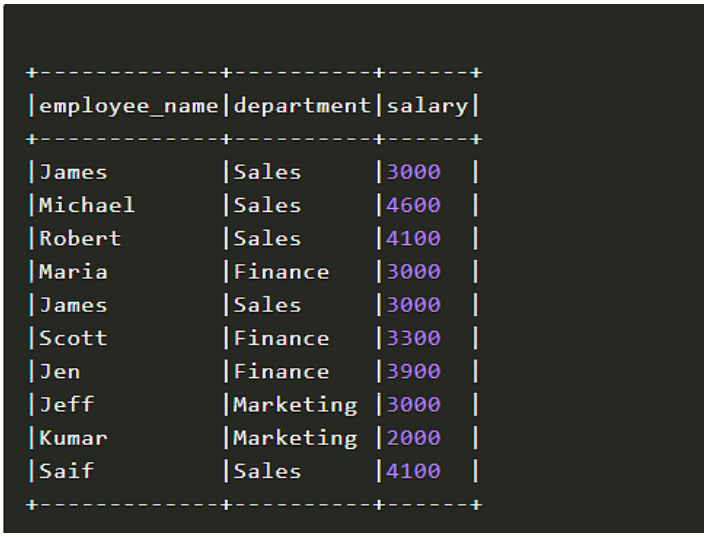
There are two ways to handle row duplication in PySpark dataframes. The distinct() function in PySpark is used to drop/remove duplicate rows (all columns) from a DataFrame, while dropDuplicates() is used to drop rows based on one or more columns.

Here's an example showing how to utilize the distinct() and dropDuplicates() methods-

First, we need to create a sample dataframe.

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import expr
spark = SparkSession.builder.appName('ProjectPro).getOrCreate()
data = [("James", "Sales", 3000), \
    ("Michael", "Sales", 4600), \
    ("Robert", "Sales", 4100), \
    ("Maria", "Finance", 3000), \
    ("James", "Sales", 3000), \
    ("Scott", "Finance", 3300), \
    ("Jen", "Finance", 3900), \
    ("Jeff", "Marketing", 3000), \
    ("Kumar", "Marketing", 2000), \
    ("Saif", "Sales", 4100) \
  ]
column= ["employee_name", "department", "salary"]
df = spark.createDataFrame(data = data, schema = column)
df.printSchema()
df.show(truncate=False)
```

Output-



```
+-------------+----------+------+
|employee_name|department|salary|
+-------------+----------+------+
|James        |Sales     |3000  |
|Michael      |Sales     |4600  |
|Robert       |Sales     |4100  |
|Maria        |Finance   |3000  |
|James        |Sales     |3000  |
|Scott        |Finance   |3300  |
|Jen          |Finance   |3900  |
|Jeff         |Marketing |3000  |
|Kumar        |Marketing |2000  |
|Saif         |Sales     |4100  |
+-------------+----------+------+
```

The record with the employer name Robert contains duplicate rows in the table above. As we can see, there are two rows with duplicate values in all fields and four rows with duplicate values in the department and salary columns.

Below is the entire code for removing duplicate rows-

```
import pyspark

from pyspark.sql import SparkSession

from pyspark.sql.functions import expr

spark = SparkSession.builder.appName('ProjectPro').getOrCreate()

data = [("James", "Sales", 3000), \

    ("Michael", "Sales", 4600), \

    ("Robert", "Sales", 4100), \

    ("Maria", "Finance", 3000), \

    ("James", "Sales", 3000), \

    ("Scott", "Finance", 3300), \

    ("Jen", "Finance", 3900), \
```

```
    ("Jeff", "Marketing", 3000), \
    ("Kumar", "Marketing", 2000), \
    ("Saif", "Sales", 4100) \
  ]
column= ["employee_name", "department", "salary"]
df = spark.createDataFrame(data = data, schema = column)
df.printSchema()
df.show(truncate=False)


#Distinct
distinctDF = df.distinct()
print("Distinct count: "+str(distinctDF.count()))
distinctDF.show(truncate=False)
#Drop duplicates
df2 = df.dropDuplicates()
print("Distinct count: "+str(df2.count()))
df2.show(truncate=False)
#Drop duplicates on selected columns
dropDisDF = df.dropDuplicates(["department","salary"])
print("Distinct count of department salary :
"+str(dropDisDF.count()))
dropDisDF.show(truncate=False)
```

}

Q6. Explain PySpark UDF with the help of an example.

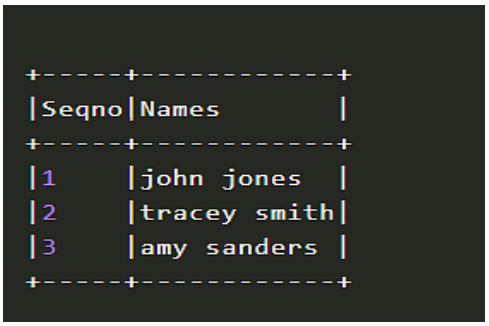
The most important aspect of Spark SQL & DataFrame is PySpark UDF (i.e., User Defined Function), which is used to expand PySpark's built-in capabilities. UDFs in PySpark work similarly to UDFs in conventional databases. We write a Python function and wrap it in PySpark SQL udf() or register it as udf and use it on DataFrame and SQL, respectively, in the case of PySpark.

Example of how we can create a UDF-

1. First, we need to create a sample dataframe.

```
spark = SparkSession.builder.appName('ProjectPro').getOrCreate()

column = ["Seqno","Name"]

data = [("1", "john jones"),

    ("2", "tracey smith"),

    ("3", "amy sanders")]

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

df.show(truncate=False)
```

Output-



2. The next step is creating a Python function. The code below generates the convertCase() method, which accepts a string parameter and turns every word's initial letter to a capital letter.

```
def convertCase(str):

    resStr=""

    arr = str.split(" ")

    for x in arr:

       resStr= resStr + x[0:1].upper() + x[1:len(x)] + " "

    return resStr
```

3. The final step is converting a Python function to a PySpark UDF.

By passing the function to PySpark SQL udf(), we can convert the convertCase() function to UDF(). The org.apache.spark.sql.functions.udf package contains this function. Before we use this package, we must first import it.

The org.apache.spark.sql.expressions.UserDefinedFunction class object is returned by the PySpark SQL udf() function.

```
""" Converting function to UDF """

convertUDF = udf(lambda z: convertCase(z),StringType())
```

Q7. Discuss the map() transformation in PySpark DataFrame with the help of an example.

PySpark map or the map() function is an RDD transformation that generates a new RDD by applying 'lambda', which is the transformation function, to each RDD/DataFrame element. RDD map() transformations are used to perform complex operations such as adding a column, changing a column,

converting data, and so on. Map transformations always produce the same number of records as the input.

Example of map() transformation in PySpark-

- First, we must create an RDD using the list of records.

```
spark = SparkSession.builder.appName("Map transformation PySpark").getOrCreate()

records = ["Project","Gutenberg’s","Alice’s","Adventures",

"in","Wonderland","Project","Gutenberg’s","Adventures",

"in","Wonderland","Project","Gutenberg’s"]

rdd=spark.sparkContext.parallelize(records)
```

- The map() syntax is-

```
map(f, preservesPartitioning=False)
```

- We are adding a new element having value 1 for each element in this PySpark map() example, and the output of the RDD is PairRDDFunctions, which has key-value pairs, where we have a word (String type) as Key and 1 (Int type) as Value.

```
rdd2=rdd.map(lambda x: (x,1))

for element in rdd2.collect():

    print(element)
```

Output-

```
('Project', 1)
('Gutenberg's', 1)
('Alice's', 1)
('Adventures', 1)
('in', 1)
('Wonderland', 1)
('Project', 1)
('Gutenberg's', 1)
('Adventures', 1)
('in', 1)
('Wonderland', 1)
('Project', 1)
('Gutenberg's', 1)
```

Q8. What do you mean by 'joins' in PySpark DataFrame? What are the different types of joins?

Joins in PySpark are used to join two DataFrames together, and by linking them together, one may join several DataFrames. INNER Join, LEFT OUTER Join, RIGHT OUTER Join, LEFT ANTI Join, LEFT SEMI Join, CROSS Join, and SELF Join are among the SQL join types it supports.

PySpark Join syntax is-

join(self, other, on=None, how=None)

The join() procedure accepts the following parameters and returns a DataFrame-

'other': The join's right side;

'on': the join column's name;

'how': default inner (Options are inner, cross, outer, full, full outer, left, left outer, right, right outer, left semi, and left anti.)

Types of Join in PySpark DataFrame-

Join String	Equivalent SQL Join
inner	INNER JOIN
outer, full, fullouter, full_outer	FULL OUTER JOIN
left, leftouter, left_outer	LEFT JOIN
right, rightouter, right_outer	RIGHT JOIN
cross	
anti, leftanti, left_anti	
semi, leftsemi, left_semi	

Q9. What is PySpark ArrayType? Explain with an example.

PySpark ArrayType is a collection data type that extends PySpark's DataType class, which is the superclass for all kinds. The types of items in all ArrayType elements should be the same. The ArraType() method may be used to construct an instance of an ArrayType. It accepts two arguments: valueType and one optional argument valueContainsNull, which specifies whether a value can accept null and is set to True by default. valueType should extend the DataType class in PySpark.

```
from pyspark.sql.types import StringType, ArrayType

arrayCol = ArrayType(StringType(),False)
```

Q10. What do you understand by PySpark Partition?

Using one or more partition keys, PySpark partitions a large dataset into smaller parts. When we build a DataFrame from a file or table, PySpark creates the DataFrame in memory with a specific number of divisions based on specified criteria. Transformations on partitioned data run quicker since each partition's transformations are executed in parallel. Partitioning in memory (DataFrame) and partitioning on disc (File system) are both supported by PySpark.

Q11. What is meant by PySpark MapType? How can you create a MapType using StructType?

PySpark MapType accepts two mandatory parameters- keyType and valueType, and one optional boolean argument valueContainsNull.

Here's how to create a MapType with PySpark StructType and StructField. The StructType() accepts a list of StructFields, each of which takes a fieldname and a value type.

```
from pyspark.sql.types import StructField, StructType, StringType, MapType

schema = StructType([

    StructField('name', StringType(), True),

    StructField('properties', MapType(StringType(),StringType()),True)

])
```

Now, using the preceding StructType structure, let's construct a DataFrame-

```
spark= SparkSession.builder.appName('PySpark StructType StructField').getOrCreate()

dataDictionary = [

        ('James',{'hair':'black','eye':'brown'}),

        ('Michael',{'hair':'brown','eye':None}),

        ('Robert',{'hair':'red','eye':'black'}),
```

('Washington',{'hair':'grey','eye':'grey'}),

('Jefferson',{'hair':'brown','eye':''})

]

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

df.printSchema()

df.show(truncate=False)

Output-

```
root
 |-- Name: string (nullable = true)
 |-- properties: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)

+----------+-----------------------------+
|Name      |properties                   |
+----------+-----------------------------+
|James     |[eye -> brown, hair -> black]|
|Michael   |[eye ->, hair -> brown]      |
|Robert    |[eye -> black, hair -> red]  |
|Washington|[eye -> grey, hair -> grey]  |
|Jefferson |[eye -> , hair -> brown]     |
+----------+-----------------------------+
```

Q12. How can PySpark DataFrame be converted to Pandas DataFrame?

First, you need to learn the difference between the PySpark and Pandas. The key difference between Pandas and PySpark is that PySpark's operations are quicker than Pandas' because of its distributed nature and parallel execution over several cores and computers.

In other words, pandas use a single node to do operations, whereas PySpark uses several computers.

You'll need to transfer the data back to Pandas DataFrame after processing it in PySpark so that you can use it in Machine Learning apps or other Python programs.

Below are the steps to convert PySpark DataFrame into Pandas DataFrame-

1. You have to start by creating a PySpark DataFrame first.

```
spark = SparkSession.builder.appName('Spark Dataframe to Pandas PySpark').getOrCreate()

SampleData = [("Ravi","","Gupta","36636","M",70000),
        ("Ram","Aggarwal","","40288","M",80000),
        ("Shyam","","Shinde","42114","",500000),
        ("Sarla","Priya","Gupta","39192","F",600000),
        ("Monica","Garg","Brown","","F",0)]

DataColumns = ["first_name","middle_name","last_name","dob","gender","salary"]

PysparkDF = spark.createDataFrame(data = SampleData, schema = DataColumns)
PysparkDF.printSchema()
PysparkDF.show(truncate=False)
```

Output-

```
root
 |-- first_name: string (nullable = true)
 |-- middle_name: string (nullable = true)
 |-- last_name: string (nullable = true)
 |-- dob: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: long (nullable = true)

+----------+-----------+---------+-----+------+------+
|first_name|middle_name|last_name|dob  |gender|salary|
+----------+-----------+---------+-----+------+------+
|Ravi      |           |Gupta    |36636|M     |70000 |
|Ram       |Aggarwal   |         |40288|M     |80000 |
|Shyam     |           |Shinde   |42114|      |500000|
|Sarla     |Priya      |Gupta    |39192|F     |600000|
|Monica    |Garg       |Brown    |     |F     |0     |
+----------+-----------+---------+-----+------+------+
```

2. The next step is to convert this PySpark dataframe into Pandas dataframe.

To convert a PySpark DataFrame to a Python Pandas DataFrame, use the toPandas() function. toPandas() gathers all records in a PySpark DataFrame and delivers them to the driver software; it should only be used on a short percentage of the data. When using a bigger dataset, the application fails due to a memory error.

```
# Converting dataframe to pandas
PandasDF = PysparkDF.toPandas()
print(PandasDF)
```

Output-

first_name	middle_name	last_name	dob	gender	salary
Ravi		Gupta	36636	M	70000
Ram	Aggarwal		40288	M	80000
Shyam		Shinde	42114		500000
Sarla	Priya	Gupta	39192	F	600000
Monica	Garg	Brown		F	0

Q13. With the help of an example, show how to employ PySpark ArrayType.

PySpark ArrayType is a data type for collections that extends PySpark's DataType class. The types of items in all ArrayType elements should be the same.

The ArraType() method may be used to construct an instance of an ArrayType. It accepts two arguments: valueType and one optional argument valueContainsNull, which specifies whether a value can accept null and is set to True by default. valueType should extend the DataType class in PySpark.

from pyspark.sql.types import StringType, ArrayType

arrayCol = ArrayType(StringType(),False)

The above example generates a string array that does not allow null values.

Q14. What is the function of PySpark's pivot() method?

The pivot() method in PySpark is used to rotate/transpose data from one column into many Dataframe columns and back using the unpivot() function (). Pivot() is an aggregation in which the values of one of the grouping columns are transposed into separate columns containing different data.

To get started, let's make a PySpark DataFrame.

```
import pyspark

from pyspark.sql import SparkSession

from pyspark.sql.functions import expr

#Create spark session

data = [("Banana",1000,"USA"), ("Carrots",1500,"USA"),
("Beans",1600,"USA"), \

    ("Orange",2000,"USA"),("Orange",2000,"USA"),("Banana",4
00,"China"), \

    ("Carrots",1200,"China"),("Beans",1500,"China"),("Orange",
4000,"China"), \

    ("Banana",2000,"Canada"),("Carrots",2000,"Canada"),("Bea
ns",2000,"Mexico")]

columns= ["Product","Amount","Country"]

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

df.printSchema()

df.show(truncate=False)
```

Output-

```
root
 |-- Product: string (nullable = true)
 |-- Amount: long (nullable = true)
 |-- Country: string (nullable = true)

+-------+------+-------+
|Product|Amount|Country|
+-------+------+-------+
|Banana |1000  |USA    |
|Carrots|1500  |USA    |
|Beans  |1600  |USA    |
|Orange |2000  |USA    |
|Orange |2000  |USA    |
|Banana |400   |China  |
|Carrots|1200  |China  |
|Beans  |1500  |China  |
|Orange |4000  |China  |
|Banana |2000  |Canada |
|Carrots|2000  |Canada |
|Beans  |2000  |Mexico |
+-------+------+-------+
```

To determine the entire amount of each product's exports to each nation, we'll group by Product, pivot by Country, and sum by Amount.

pivotDF = df.groupBy("Product").pivot("Country").sum("Amount")

pivotDF.printSchema()

pivotDF.show(truncate=False)

This will convert the nations from DataFrame rows to columns, resulting in the output seen below. Wherever data is missing, it is assumed to be null by default.

```
root
 |-- Product: string (nullable = true)
 |-- Canada: long (nullable = true)
 |-- China: long (nullable = true)
 |-- Mexico: long (nullable = true)
 |-- USA: long (nullable = true)

+-------+------+-----+------+----+
|Product|Canada|China|Mexico|USA |
+-------+------+-----+------+----+
|Orange |null  |4000 |null  |4000|
|Beans  |null  |1500 |2000  |1600|
|Banana |2000  |400  |null  |1000|
|Carrots|2000  |1200 |null  |1500|
+-------+------+-----+------+----+
```

Q15. In PySpark, how do you generate broadcast variables? Give an example.

Broadcast variables in PySpark are read-only shared variables that are stored and accessible on all nodes in a cluster so that processes may access or use them. Instead of sending this information with each job, PySpark uses efficient broadcast algorithms to distribute broadcast variables among workers, lowering communication costs.

The broadcast(v) function of the SparkContext class is used to generate a PySpark Broadcast. This method accepts the broadcast parameter v.

Generating broadcast in PySpark Shell:

broadcastVariable = sc.broadcast(Array(0, 1, 2, 3))

broadcastVariable.value

PySpark RDD Broadcast variable example

spark=SparkSession.builder.appName('SparkByExample.com')
.getOrCreate()

states = {"NY":"New York", "CA":"California", "FL":"Florida"}

```
broadcastStates = spark.sparkContext.broadcast(states)

data = [("James","Smith","USA","CA"),

    ("Michael","Rose","USA","NY"),

    ("Robert","Williams","USA","CA"),

    ("Maria","Jones","USA","FL")

  ]

rdd = spark.sparkContext.parallelize(data)

def state_convert(code):

    return broadcastState.value[code]

res = rdd.map(lambda a:
(a[0],a[1],a[2],state_convert(a{3]))).collect()

print(res)
```

PySpark DataFrame Broadcast variable example

```
spark=SparkSession.builder.appName('PySpark broadcast
variable').getOrCreate()

states = {"NY":"New York", "CA":"California", "FL":"Florida"}

broadcastStates = spark.sparkContext.broadcast(states)

data = [("James","Smith","USA","CA"),

    ("Michael","Rose","USA","NY"),

    ("Robert","William","USA","CA"),

    ("Maria","Jones","USA","FL")

  ]
```

```
columns = ["firstname","lastname","country","state"]

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

df.printSchema()

df.show(truncate=False)

def state_convert(code):

    return broadcastState.value[code]

res = df.rdd.map(lambda a:
(a[0],a[1],a[2],state_convert(a[3]))).toDF(column)

res.show(truncate=False)
```

PySpark Coding Interview Questions

Q1. You have a cluster of ten nodes with each node having 24 CPU cores. The following code works, but it may crash on huge data sets, or at the very least, it may not take advantage of the cluster's full processing capabilities. Which aspect is the most difficult to alter, and how would you go about doing so?

def cal(sparkSession: SparkSession): Unit = { val NumNode = 10 val userActivityRdd: RDD[UserActivity] = readUserActivityData(sparkSession) . repartition(NumNode) val result = userActivityRdd .map(e => (e.userId, 1L)) . reduceByKey(_ + _) result .take(1000) }

The repartition command creates ten partitions regardless of how many of them were loaded. On large datasets, they might get fairly huge, and they'll almost certainly outgrow the RAM allotted to a single executor.

In addition, each executor can only have one partition. This means that just ten of the 240 executors are engaged (10 nodes with 24 cores, each running one executor).

If the number is set exceptionally high, the scheduler's cost in handling the partition grows, lowering performance. It may even exceed the execution time in some circumstances, especially for extremely tiny partitions.

The optimal number of partitions is between two and three times the number of executors. In the given scenario, 600 = 10 × 24 x 2.5 divisions would be appropriate.

Q2. Explain the following code and what output it will yield- case class User(uId: Long, uName: String) case class UserActivity(uId: Long, activityTypeId: Int, timestampEpochSec: Long) val LoginActivityTypeId = 0 val LogoutActivityTypeId = 1 private def readUserData(sparkSession: SparkSession): RDD[User] = { sparkSession.sparkContext.parallelize(Array(User(1, "Doe, John"), User(2, "Doe, Jane"), User(3, "X, Mr."))) } private def readUserActivityData(sparkSession: SparkSession): RDD[UserActivity] = { sparkSession.sparkContext.parallelize(Array(UserActivity(1, LoginActivityTypeId, 1514764800L), UserActivity(2, LoginActivityTypeId, 1514808000L), UserActivity(1, LogoutActivityTypeId, 1514829600L), UserActivity(1, LoginActivityTypeId, 1514894400L))) } def calculate(sparkSession: SparkSession): Unit = { val userRdd: RDD[(Long, User)] = readUserData(sparkSession).map(e => (e.userId, e)) val userActivityRdd: RDD[(Long, UserActivity)] = readUserActivityData(sparkSession).map(e => (e.userId, e)) val result = userRdd .leftOuterJoin(userActivityRdd) .filter(e => e._2._2.isDefined && e._2._2.get.activityTypeId == LoginActivityTypeId) .map(e => (e._2._1.uName, e._2._2.get.timestampEpochSec)) .reduceByKey((a, b) => if (a < b) a else b) result .foreach(e => println(s"${e._1}: ${e._2}")) }

The primary function, **calculate**, reads two pieces of data. (They are given in this case from a constant inline data

structure that is transformed to a distributed dataset using parallelize.) Each of them is transformed into a tuple by the map, which consists of a userId and the item itself. To combine the two datasets, the userId is utilised.

All users' login actions are filtered out of the combined dataset. The uName and the event timestamp are then combined to make a tuple.

This is eventually reduced down to merely the initial login record per user, which is then sent to the console.

The following will be the yielded output-

Doe, John: 1514764800

Doe, Jane: 1514808000

Q3. The code below generates two dataframes with the following structure: DF1: uId, uName DF2: uId, pageId, timestamp, eventType. Join the two dataframes using code and count the number of events per uName. It should only output for users who have events in the format uName; totalEventCount.
def calculate(sparkSession: SparkSession): Unit = { val UIdColName = "uId" val UNameColName = "uName" val CountColName = "totalEventCount" val userRdd: DataFrame = readUserData(sparkSession) val userActivityRdd: DataFrame = readUserActivityData(sparkSession) val res = userRdd .repartition(col(UIdColName)) // ?????????????? . select(col(UNameColName))// ?????????????? result.show() }
This is how the code looks:

```
def calculate(sparkSession: SparkSession): Unit = {
 val UIdColName = "uId"
 val UNameColName = "uName"
 val CountColName = "totalEventCount"
 val userRdd: DataFrame = readUserData(sparkSession)
 val userActivityRdd: DataFrame =
readUserActivityData(sparkSession)
 val result = userRdd
   .repartition(col(UIdColName))
   .join(userActivityRdd, UIdColName)
   .select(col(UNameColName))
   .groupBy(UNameColName)
   .count()
   .withColumnRenamed("count", CountColName)
 result.show()
}
```

Q4. Please indicate which parts of the following code will run on the master and which parts will run on each worker node.

```
val formatter: DateTimeFormatter = DateTimeFormatter.ofPattern("yyyy/MM") def getEventCountOnWeekdaysPerMonth(data: RDD[(LocalDateTime, Long)]): Array[(String, Long)] = { val res = data .filter(e => e._1.getDayOfWeek.getValue < DayOfWeek.SATURDAY.getValue) . map(mapDateTime2Date) . reduceByKey(_ + _) . collect() result . map(e => (e._1.format(formatter), e._2)) } private def mapDateTime2Date(v: (LocalDateTime, Long)): (LocalDate, Long) = { (v._1.toLocalDate.withDayOfMonth(1), v._2) }
```

The driver application is responsible for calling this function. The DAG is defined by the assignment to the result value, as well as its execution, which is initiated by the collect() operation. The worker nodes handle all of this (including the logic of the method mapDateTime2Date). Because the result value that is gathered on the master is an array, the map performed on this value is also performed on the master.

Q5. What are the elements used by the GraphX library, and how are they generated from an RDD? To determine page rankings, fill in the following code-
def calculate(sparkSession: SparkSession): Unit = { val pageRdd: RDD[(???, Page)] = readPageData(sparkSession) . map(e => (e.pageId, e)) . cache() val pageReferenceRdd: RDD[???[PageReference]] = readPageReferenceData(sparkSession) val graph = Graph(pageRdd, pageReferenceRdd) val PageRankTolerance = 0.005 val ranks = graph.??? ranks.take(1000).foreach(print) } The output yielded will be a list of tuples: (1,1.453795159509l907) (2,0.773102420245404 8) (3,0.773102420245404 8)
Vertex, and Edge objects are supplied to the Graph object as RDDs of type RDD[VertexId, VT] and RDD[Edge[ET]] respectively (where VT and ET are any user-defined types associated with a given Vertex or Edge). For Edge type, the constructor is Edge[ET](srcId: VertexId, dstId: VertexId, attr: ET). VertexId is just an alias for Long.

PySpark Interview Questions for Data Engineer

Q1. Under what scenarios are Client and Cluster modes used for deployment?

- Cluster mode should be utilized for deployment if the client computers are not near the cluster. This is done to prevent

the network delay that would occur in Client mode while communicating between executors. In case of Client mode, if the machine goes offline, the entire operation is lost.

- Client mode can be utilized for deployment if the client computer is located within the cluster. There will be no network latency concerns because the computer is part of the cluster, and the cluster's maintenance is already taken care of, so there is no need to be concerned in the event of a failure.

Q2.How is Apache Spark different from MapReduce?

MapReduce	**Apache Spark**
Only batch-wise data processing is done using MapReduce.	Apache Spark can handle data in both real-time and batch mode.
The data is stored in HDFS (Hadoop Distributed File System), which takes a long time to retrieve.	Spark saves data in memory (RAM), making data retrieval quicker and faster when needed.
MapReduce is a high-latency framework since it is heavily reliant on disc.	Spark is a low-latency computation platform because it offers in-memory data storage and caching.

Q2. Write a spark program to check whether a given keyword exists in a huge text file or not?

```
def keywordExists(line):
  if (line.find(“my_keyword”) > -1):
    return 1
  return 0
lines = sparkContext.textFile(“sample_file.txt”);
isExist = lines.map(keywordExists);
```

```
sum=isExist.reduce(sum);

print("Found" if sum>0 else "Not Found")
```

Q3. What is meant by Executor Memory in PySpark?

Spark executors have the same fixed core count and heap size as the applications created in Spark. The heap size relates to the memory used by the Spark executor, which is controlled by the *-executor-memory* flag's property *spark.executor.memory*. On each worker node where Spark operates, one executor is assigned to it. The executor memory is a measurement of the memory utilized by the application's worker node.

Q4. List some of the functions of SparkCore.

The core engine for large-scale distributed and parallel data processing is SparkCore. The distributed execution engine in the Spark core provides APIs in Java, Python, and Scala for constructing distributed ETL applications.

Memory management, task monitoring, fault tolerance, storage system interactions, work scheduling, and support for all fundamental I/O activities are all performed by Spark Core. Additional libraries on top of Spark Core enable a variety of SQL, streaming, and machine learning applications.

They are in charge of:

- Fault Recovery
- Interactions between memory management and storage systems
- Monitoring, scheduling, and distributing jobs
- Fundamental I/O functions

Q5. What are some of the drawbacks of incorporating Spark into applications?

Despite the fact that Spark is a strong data processing engine, there are certain drawbacks to utilizing it in applications.

- When compared to MapReduce or Hadoop, Spark consumes greater storage space, which may cause memory-related issues.
- Spark can be a constraint for cost-effective large data processing since it uses "in-memory" calculations.
- When working in cluster mode, files on the path of the local filesystem must be available at the same place on all worker nodes, as the task execution shuffles across different worker nodes based on resource availability. All worker nodes must copy the files, or a separate network-mounted file-sharing system must be installed.

Q6. How can data transfers be kept to a minimum while using PySpark?

The process of shuffling corresponds to data transfers. Spark applications run quicker and more reliably when these transfers are minimized. There are quite a number of approaches that may be used to reduce them. They are as follows:

- Using broadcast variables improves the efficiency of joining big and small RDDs.
- Accumulators are used to update variable values in a parallel manner during execution.
- Another popular method is to prevent operations that cause these reshuffles.

Q7. What are Sparse Vectors? What distinguishes them from dense vectors?

Sparse vectors are made up of two parallel arrays, one for indexing and the other for storing values. These vectors are used to save space by storing non-zero values. E.g.- *val sparseVec: Vector = Vectors.sparse(5, Array(0, 4), Array(1.0, 2.0))*

The vector in the above example is of size 5, but the non-zero values are only found at indices 0 and 4.

When there are just a few non-zero values, sparse vectors come in handy. If there are just a few zero values, dense vectors should be used instead of sparse vectors, as sparse vectors would create indexing overhead, which might affect performance.

The following is an example of a dense vector:

val denseVec = Vectors.dense(4405d,260100d,400d,5.0,4.0,198.0,9070d,1.0,1.0,2.0,0.0)

The usage of sparse or dense vectors has no effect on the outcomes of calculations, but when they are used incorrectly, they have an influence on the amount of memory needed and the calculation time.

Q8. What role does Caching play in Spark Streaming?

The partition of a data stream's contents into batches of X seconds, known as DStreams, is the basis of Spark Streaming. These DStreams allow developers to cache data in memory, which may be particularly handy if the data from a DStream is utilized several times. The cache() function or the persist() method with proper persistence settings can be used to cache data. For input streams receiving data through networks such as Kafka, Flume, and others, the default persistence level setting is configured to achieve data replication on two nodes to achieve fault tolerance.

- Cache method-

val cacheDf = dframe.cache()

- Persist method-

val persistDf = dframe.persist(StorageLevel.MEMORY_ONLY)

The following are the key benefits of caching:

- Cost-effectiveness: Because Spark calculations are costly, caching aids in data reuse, which leads to reuse computations, lowering the cost of operations.
- Time-saving: By reusing computations, we may save a lot of time.
- More Jobs Achieved: Worker nodes may perform/execute more jobs by reducing computation execution time.

Q9. What API does PySpark utilize to implement graphs?

Spark RDD is extended with a robust API called GraphX, which supports graphs and graph-based calculations. The Resilient Distributed Property Graph is an enhanced property of Spark RDD that is a directed multi-graph with many parallel edges. User-defined characteristics are associated with each edge and vertex. Multiple connections between the same set of vertices are shown by the existence of parallel edges. GraphX offers a collection of operators that can allow graph computing, such as subgraph, mapReduceTriplets, joinVertices, and so on. It also offers a wide number of graph builders and algorithms for making graph analytics chores easier.

Q10. What is meant by Piping in PySpark?

According to the UNIX Standard Streams, Apache Spark supports the pipe() function on RDDs, which allows you to assemble distinct portions of jobs that can use any language. The RDD transformation may be created using the pipe() function, and it can be used to read each element of the RDD as a String. These may be altered as needed, and the results can be presented as Strings.

Q11. What are the various levels of persistence that exist in PySpark?

Spark automatically saves intermediate data from various shuffle processes. However, it is advised to use the RDD's **persist()** function. There are many levels of persistence for storing RDDs on memory, disc, or both, with varying levels

of replication. The following are the persistence levels available in Spark:

- MEMORY ONLY: This is the default persistence level, and it's used to save RDDs on the JVM as deserialized Java objects. In the event that the RDDs are too large to fit in memory, the partitions are not cached and must be recomputed as needed.
- MEMORY AND DISK: On the JVM, the RDDs are saved as deserialized Java objects. In the event that memory is inadequate, partitions that do not fit in memory will be kept on disc, and data will be retrieved from the drive as needed.
- MEMORY ONLY SER: The RDD is stored as One Byte per partition serialized Java Objects.
- DISK ONLY: RDD partitions are only saved on disc.
- OFF HEAP: This level is similar to MEMORY ONLY SER, except that the data is saved in off-heap memory.

The persist() function has the following syntax for employing persistence levels:

df.persist(StorageLevel.)

Q12. What steps are involved in calculating the executor memory?

Suppose you have the following details regarding the cluster:

No. of nodes = 10

No. of cores in each node = 15 cores

RAM of each node = 61GB

We use the following method to determine the number of cores:

No. of cores = How many concurrent tasks the executor can handle.

As a rule of thumb, 5 is the best value.

Hence, we use the following method to determine the number of executors:

No. of executors = No. of cores/Concurrent Task

= 15/5

= 3

No. of executors = No. of nodes * No. of executors in each node

= 10 * 3

= 30 executors per Spark job

Q13. Do we have a checkpoint feature in Apache Spark?

Yes, there is an API for checkpoints in Spark. The practice of checkpointing makes streaming apps more immune to errors. We can store the data and metadata in a checkpointing directory. If there's a failure, the spark may retrieve this data and resume where it left off.

In Spark, checkpointing may be used for the following data categories-

1. **Metadata checkpointing:** Metadata rmeans information about information. It refers to storing metadata in a fault-tolerant storage system such as HDFS. You can consider configurations, DStream actions, and unfinished batches as types of metadata.
2. **Data checkpointing:** Because some of the stateful operations demand it, we save the RDD to secure storage. The RDD for the next batch is defined by the RDDs from previous batches in this case.

Q14. In Spark, how would you calculate the total number of unique words?

1. Open the text file in RDD mode:

sc.textFile("hdfs://Hadoop/user/sample_file.txt");

2. A function that converts each line into words:

def toWords(line):

return line.split();

3. As a flatMap transformation, run the toWords function on each item of the RDD in Spark:

words = line.flatMap(toWords);

4. Create a (key,value) pair for each word:

def toTuple(word):

return (word, 1);

wordTuple = words.map(toTuple);

5. Run the reduceByKey() command:

def sum(x, y):

return x+y:

counts = wordsTuple.reduceByKey(sum)

6. Print:

counts.collect()

Q15. List some of the benefits of using PySpark.

PySpark is a specialized in-memory distributed processing engine that enables you to handle data in a distributed fashion effectively.

- PySpark-based programs are 100 times quicker than traditional apps.
- You can learn a lot by utilizing PySpark for data intake processes. PySpark can handle data from Hadoop HDFS, Amazon S3, and a variety of other file systems.
- Through the use of Streaming and Kafka, PySpark is also utilized to process real-time data.
- You can use PySpark streaming to swap data between the file system and the socket.
- PySpark contains machine learning and graph libraries by chance.

PySpark Data Science Interview Questions

Q1. What distinguishes Apache Spark from other programming languages?

- **High Data Processing Speed:** By decreasing read-write operations to disc, Apache Spark aids in achieving a very high data processing speed. When doing in-memory computations, the speed is about 100 times quicker, and when performing disc computations, the speed is 10 times faster.
- **Dynamic in nature:** Spark's dynamic nature comes from 80 high-level operators, making developing parallel applications a breeze.
- **In-memory Computing Ability:** Spark's in-memory computing capability, which is enabled by its DAG execution engine, boosts data processing speed. This also allows for data caching, which reduces the time it takes to retrieve data from the disc.
- **Fault Tolerance:** RDD is used by Spark to support fault tolerance. Spark RDDs are abstractions that are meant to

accommodate worker node failures while ensuring that no data is lost.
- **Stream Processing:** Spark offers real-time stream processing. The difficulty with the previous MapReduce architecture was that it could only handle data that had already been created.

Q2. Explain RDDs in detail.

Resilient Distribution Datasets (RDD) are a collection of fault-tolerant functional units that may run simultaneously. RDDs are data fragments that are maintained in memory and spread across several nodes. In an RDD, all partitioned data is distributed and consistent.

There are two types of RDDs available:

1. **Hadoop datasets-** Those datasets that apply a function to each file record in the Hadoop Distributed File System (HDFS) or another file storage system.
2. **Parallelized Collections-** Existing RDDs that operate in parallel with each other.

Q3. Mention some of the major advantages and disadvantages of PySpark.

Some of the major advantages of using PySpark are-

- Writing parallelized code is effortless.
- Keeps track of synchronization points and errors.
- Has a lot of useful built-in algorithms.

Some of the disadvantages of using PySpark are-

- Managing an issue with MapReduce may be difficult at times.
- It is inefficient when compared to alternative programming paradigms.

Q4. Explain the profilers which we use in PySpark.

PySpark allows you to create custom profiles that may be used to build predictive models. In general, profilers are calculated using the minimum and maximum values of each column. It is utilized as a valuable data review tool to ensure that the data is accurate and appropriate for future usage.

The following methods should be defined or inherited for a custom profiler-

- profile- this is identical to the system profile.
- add- this is a command that allows us to add a profile to an existing accumulated profile.
- dump- saves all of the profiles to a path.
- stats- returns the stats that have been gathered.

Q5. List some recommended practices for making your PySpark data science workflows better.

- **Avoid dictionaries**: If you use Python data types like dictionaries, your code might not be able to run in a distributed manner. Consider adding another column to a dataframe that may be used as a filter instead of utilizing keys to index entries in a dictionary. This proposal also applies to Python types that aren't distributable in PySpark, such as lists.
- **Limit the use of Pandas**: using toPandas causes all data to be loaded into memory on the driver node, preventing operations from being run in a distributed manner. When data has previously been aggregated, and you wish to utilize conventional Python plotting tools, this method is appropriate, but it should not be used for larger dataframes.
- **Minimize eager operations**: It's best to avoid eager operations that draw whole dataframes into memory if you want your pipeline to be as scalable as possible. Reading in CSVs, for example, is an eager activity, thus I stage the

dataframe to S3 as Parquet before utilizing it in further pipeline steps.

Advanced PySpark Interview Questions and Answers

Q1. Discuss PySpark SQL in detail.

- PySpark SQL is a structured data library for Spark. PySpark SQL, in contrast to the PySpark RDD API, offers additional detail about the data structure and operations. It comes with a programming paradigm- 'DataFrame.'
- A DataFrame is an immutable distributed columnar data collection. DataFrames can process huge amounts of organized data (such as relational databases) and semi-structured data (JavaScript Object Notation or JSON).
- After creating a dataframe, you can interact with data using SQL syntax/queries.
- The first step in using PySpark SQL is to use the *createOrReplaceTempView()* function to create a temporary table on DataFrame. The table is available throughout SparkSession via the sql() method. You can delete the temporary table by ending the SparkSession.
- Example of PySpark SQL-

```
import findspark

findspark.init()

import pyspark

from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

df = spark.sql("'select 'spark' as hello '")

df.show()
```

Q2. Explain the different persistence levels in PySpark.

Persisting (or caching) a dataset in memory is one of PySpark's most essential features. The different levels of persistence in PySpark are as follows-

Level	Purpose
MEMORY_ONLY	This level stores deserialized Java objects in the JVM. It is the default persistence level in PySpark.
MEMORY_AND_DISK	This level stores RDD as deserialized Java objects. If the RDD is too large to reside in memory, it saves the partitions that don't fit on the disk and reads them as needed.
MEMORY_ONLY_SER	It stores RDD in the form of serialized Java objects. Although this level saves more space in the case of fast serializers, it demands more CPU capacity to read the RDD.
MEMORY_AND_DISK_SER	This level acts similar to MEMORY ONLY SER, except instead of recomputing partitions on the fly each time they're needed, it stores them on disk.
DISK_ONLY	It only saves RDD partitions on the disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	These levels function the same as others. They copy each partition on two cluster nodes.
OFF_HEAP	This level requires off-heap memory to store RDD.

Q3. What do you mean by checkpointing in PySpark?

- A streaming application must be available 24 hours a day, seven days a week, and must be resistant to errors external to the application code (e.g., system failures, JVM crashes, etc.).

- The process of checkpointing makes streaming applications more tolerant of failures. You can save the data and metadata to a checkpointing directory.
- Checkpointing can be of two types- Metadata checkpointing and Data checkpointing.
- Metadata checkpointing allows you to save the information that defines the streaming computation to a fault-tolerant storage system like HDFS. This helps to recover data from the failure of the streaming application's driver node.
- Data checkpointing entails saving the created RDDs to a secure location. Several stateful computations combining data from different batches require this type of checkpoint.

Q4. In PySpark, how would you determine the total number of unique words?

- Open the text file in RDD mode:

sc.textFile("hdfs://Hadoop/user/test_file.txt");

- Write a function that converts each line into a single word:

def toWords(line):

return line.split();

- Run the toWords function on each member of the RDD in Spark:
  words = line.flatMap(toWords);
- Generate a (key, value) for each word:

def toTuple(word):

return (word, 1);

wordTuple = words.map(toTuple);

- Run the reduceByKey() command:

def sum(x, y):

return x+y:

counts = wordsTuple.reduceByKey(sum)

- Print it out:

counts.collect()

Q5. Explain PySpark Streaming. How do you use the TCP/IP Protocol to stream data

- Spark Streaming is a feature of the core Spark API that allows for scalable, high-throughput, and fault-tolerant live data stream processing.
- It entails data ingestion from various sources, including Kafka, Kinesis, TCP connections, and data processing with complicated algorithms using high-level functions like map, reduce, join, and window.

- Furthermore, it can write data to filesystems, databases, and live dashboards.

We can use the readStream.format("socket") method of the Spark session object for reading data from a TCP socket and specifying the streaming source host and port as parameters, as illustrated in the code below:

```
from pyspark import SparkContext

from pyspark.streaming import StreamingContext

sc = SparkContext("local[2]", "NetworkWordCount")

ssc = StreamingContext(sc, 1)

lines = ssc.socketTextStream("localhost", 9999)
```

Q6.What do you understand by Lineage Graph in PySpark?

- The Spark lineage graph is a collection of RDD dependencies. There are separate lineage graphs for each Spark application.
- The lineage graph recompiles RDDs on-demand and restores lost data from persisted RDDs.
- An RDD lineage graph helps you to construct a new RDD or restore data from a lost persisted RDD.
- It's created by applying modifications to the RDD and generating a consistent execution plan.

Q7. Outline some of the features of PySpark SQL.

- User-Defined Functions- To extend the Spark functions, you can define your own column-based transformations.
- Standard JDBC/ODBC Connectivity- Spark SQL libraries allow you to connect to Spark SQL using regular JDBC/ODBC connections and run queries (table operations) on structured data.
- Data Transformations- For transformations, Spark's RDD API offers the highest quality performance. Spark takes advantage of this functionality by converting SQL queries to RDDs for transformations.

- Performance- Due to its in-memory processing, Spark SQL outperforms Hadoop by allowing for more iterations over datasets.
- Relational Processing- Spark brought relational processing capabilities to its functional programming capabilities with the advent of SQL.

Q8. Define the role of Catalyst Optimizer in PySpark.

- Apache Spark relies heavily on the Catalyst optimizer. It improves structural queries expressed in SQL or via the DataFrame/Dataset APIs, reducing program runtime and cutting costs.
- The Spark Catalyst optimizer supports both rule-based and cost-based optimization.
- Rule-based optimization involves a set of rules to define how to execute the query.
- Cost-based optimization involves developing several plans using rules and then calculating their costs.
- Catalyst optimizer also handles various Big data challenges like semistructured data and advanced analytics.

Q9. Mention the various operators in PySpark GraphX.

- Property Operators- These operators create a new graph with the user-defined map function modifying the vertex or edge characteristics. In these operators, the graph structure is unaltered. This is a significant feature of these operators since it allows the generated graph to maintain the original graph's structural indices.
- Structural Operators- GraphX currently only supports a few widely used structural operators. The reverse operator creates a new graph with reversed edge directions. The subgraph operator returns a graph with just the vertices and edges that meet the vertex predicate. The mask operator creates a subgraph by returning a graph with all of the vertices and edges found in the input graph. The groupEdges operator merges parallel edges.

- Join Operators- The join operators allow you to join data from external collections (RDDs) to existing graphs. For example, you might want to combine new user attributes with an existing graph or pull vertex properties from one graph into another.

Q10. Consider the following scenario: you have a large text file. How will you use PySpark to see if a specific keyword exists?

lines = sc.textFile("hdfs://Hadoop/user/test_file.txt");

```
def isFound(line):

if line.find("my_keyword") > -1

return 1

return 0

foundBits = lines.map(isFound);

sum = foundBits.reduce(sum);

if sum > 0:

print "Found"

else:

print "Not Found";
```

PySpark Practice Problems | Scenario Based Interview Questions and Answers

Q1. The given file has a delimiter ~|. How will you load it as a spark DataFrame?

Important: Instead of using sparkContext(sc), use sparkSession (spark).

Name ~|Age

Azarudeen, Shahul~|25

Michel, Clarke ~|26

Virat, Kohli ~|28

Andrew, Simond ~|37

George, Bush~|59

Flintoff, David ~|12

Answer- import findspark

findspark.init()

from pyspark.sql import Sparksession, types

spark =
Sparksession.builder.master("local").appliame("scenario
based")\

-getorcreate()

sc=spark.sparkContext

dfaspark.read.text("input.csv")

df.show(truncate=0)

header=df.first()[0]

schema=header.split(-')

df_imput=df.filter(df['value'] l= header).rdd.map(lambda x: x[0].
split('-|')).toDF (schema)

df_input.show(truncate=0)

Q2. How will you merge two files – File1 and File2 – into a single DataFrame if they have different schemas?

File -1:

Name|Age

Azarudeen, Shahul|25

Michel, Clarke|26

Virat, Kohli|28

Andrew, Simond|37

File -2:

Name|Age|Gender

Rabindra, Tagore |32|Male

Madona, Laure | 59|Female

Flintoff, David|12|Male

Ammie, James| 20|Female

Answer- import findspark

findspark.init()

from pyspark.sql import SparkSession, types

spark = SparkSession.builder.master("local").appName('Modes of Dataframereader')\

.getorCreate()

sc=spark.sparkContext

df1=spark.read.option("delimiter","|").csv('input.csv')

```
df2=spark.read.option("delimiter","|").csv("input2.csv",header=True)

from pyspark.sql.functions import lit

df_add=df1.withColumn("Gender",lit("null"))

df_add. union(df2).show()
```

For the Union-

```
from pyspark.sql.types import *

schema=StructType(
    [
        StructField("Name",StringType(), True),
        StructField("Age",StringType(), True),
        StructField("Gender",StringType(),True),
    ]
)

df3=spark.read.option("delimiter","|").csv("input.csv",header=True, schema=schema)

df4=spark.read.option("delimiter","|").csv("input2.csv", header=True, schema=schema)

df3.union(df4).show()
```

Q3. Examine the following file, which contains some corrupt/bad data. What will you do with such data, and how will you import them into a Spark Dataframe?

Emp_no, Emp_name, Department

101, Murugan, HealthCare

Invalid Entry, Description: Bad Record entry

102, Kannan, Finance

103, Mani, IT

Connection lost, Description: Poor Connection

104, Pavan, HR

Bad Record, Description: Corrupt record

Answer-

```
import findspark

findspark.init()

from pyspark. sql import Sparksession, types

spark = Sparksession.builder.master("local").appName(
"Modes of Dataframereader')\

.getorcreate()

sc=spark. sparkContext

from pyspark.sql.types import *

schm structiype([

structField("col_1",stringType(), True),

StructField("col_2",stringType(), True),

structrield("col",stringtype(), True),

])
```

```
df=spark.read.option("mode",
"DROPMALFORMED").csv('input1.csv', header=True,
schema=schm)

df. show()
```

Q4. Consider a file containing an Education column that includes an array of elements, as shown below. Using Spark Dataframe, convert each element in the array to a record.

Name| Age | EducaÈ›ion

Azar|25| MBA,BE,HSC

Hari|32|

Kumar|35|ME,BE,Diploma

Answer-

```
import findspark

findspark.init()

from pyspark.sql import SparkSession, types

spark =
SparkSession.builder.master("local").appName('scenario
based')\

.getorCreate()

sc=spark.sparkContext

in_df=spark.read.option("delimiter","|").csv("input4.csv",
header-True)

in_df.show()

from pyspark.sql.functions import posexplode_outer, split
```

```
in_df.withColumn("Qualification",
explode_outer(split("Education",",","))).show()

in_df.select("*",
posexplode_outer(split("Education",",","))).withColumnRenamed
("col", "Qualification").withColumnRenamed ("pos",
"Index").drop("Education").show()
```

Capgemini PySpark Interview Questions

Q1. What are SparkFiles in Pyspark?

PySpark provides the reliability needed to upload our files to Apache Spark. This is accomplished by using sc.addFile, where 'sc' stands for SparkContext. We use SparkFiles.net to acquire the directory path.

We use the following methods in SparkFiles to resolve the path to the files added using SparkContext.addFile():

- get(filename),
- getrootdirectory()

Q2. What is SparkConf in PySpark? List a few attributes of SparkConf.

SparkConf aids in the setup and settings needed to execute a spark application locally or in a cluster. To put it another way, it offers settings for running a Spark application. The following are some of SparkConf's most important features:

- **set(key, value):** This attribute aids in the configuration property setting.
- **setSparkHome(value):** This feature allows you to specify the directory where Spark will be installed on worker nodes.
- **setAppName(value):** This element is used to specify the name of the application.
- **setMaster(value):** The master URL may be set using this property.

- **get(key, defaultValue=None):** This attribute aids in the retrieval of a key's configuration value.

Q3. What is the key difference between list and tuple?

The primary difference between lists and tuples is that lists are mutable, but tuples are immutable.

When a Python object may be edited, it is considered to be a mutable data type. Immutable data types, on the other hand, cannot be changed.

Here's an example of how to change an item list into a tuple-

list_num[3] = 7

print(list_num)

tup_num[3] = 7

Output:

[1,2,5,7]

Traceback (most recent call last):

File "python", line 6, in

TypeError: 'tuple' object doesnot support item assignment

We assigned 7 to list_num at index 3 in this code, and 7 is found at index 3 in the output. However, we set 7 to tup_num at index 3, but the result returned a type error. Because of their immutable nature, we can't change tuples.

Q4. What do you understand by errors and exceptions in Python?

There are two types of errors in Python: syntax errors and exceptions.

Syntax errors are frequently referred to as parsing errors. Errors are flaws in a program that might cause it to crash or terminate unexpectedly. When a parser detects an error, it repeats the offending line and then shows an arrow pointing to the line's beginning.

Exceptions arise in a program when the usual flow of the program is disrupted by an external event. Even if the program's syntax is accurate, there is a potential that an error will be detected during execution; nevertheless, this error is an exception. ZeroDivisionError, TypeError, and NameError are some instances of exceptions.

Q5. What are the most significant changes between the Python API (PySpark) and Apache Spark?

PySpark is a Python API created and distributed by the Apache Spark organization to make working with Spark easier for Python programmers. Scala is the programming language used by Apache Spark. It can communicate with other languages like Java, R, and Python.

Also, because Scala is a compile-time, type-safe language, Apache Spark has several capabilities that PySpark does not, one of which includes Datasets. Datasets are a highly typed collection of domain-specific objects that may be used to execute concurrent calculations.

Q6. Define SparkSession in PySpark. Write code to create SparkSession in PySpark

Spark 2.0 includes a new class called SparkSession (pyspark.sql import SparkSession). Prior to the 2.0 release, SparkSession was a unified class for all of the many contexts we had (SQLContext and HiveContext, etc). Since version 2.0, SparkSession may replace SQLContext, HiveContext, and other contexts specified before version 2.0. It's a way to get into the core PySpark technology and construct PySpark RDDs and DataFrames programmatically. Spark is the default object in

pyspark-shell, and it may be generated programmatically with SparkSession.

In PySpark, we must use the builder pattern function builder() to construct SparkSession programmatically (in a.py file), as detailed below. The getOrCreate() function retrieves an already existing SparkSession or creates a new SparkSession if none exists.

```
spark=SparkSession.builder.master("local[1]") \
                  .appName('ProjectPro') \
                  .getOrCreate()
```

Q7. Suppose you encounter the following error message while running PySpark commands on Linux- ImportError: No module named py4j.java_gateway How will you resolve it?

Py4J is a Java library integrated into PySpark that allows Python to actively communicate with JVM instances. Py4J is a necessary module for the PySpark application to execute, and it may be found in the $SPARK_HOME/python/lib/py4j-*-src.zip directory.

To execute the PySpark application after installing Spark, set the Py4j module to the PYTHONPATH environment variable. We'll get an ImportError: No module named py4j.java_gateway error if we don't set this module to env.

So, here's how this error can be resolved-

```
export SPARK_HOME=/Users/abc/apps/spark-3.0.0-bin-hadoop2.7

export PYTHONPATH=$SPARK_HOME/python:$SPARK_HOME/pyth
```

on/build:$SPARK_HOME/python/lib/py4j-0.10.9-src.zip:$PYTHONPATH

Put these in *.bashrc* file and re-load it using *source ~/.bashrc*

The py4j module version changes depending on the PySpark version we're using; to configure this version correctly, follow the steps below:

export PYTHONPATH=${SPARK_HOME}/python/:$(echo ${SPARK_HOME}/python/lib/py4j-*-src.zip):${PYTHONPATH}

Use the pip show command to see the PySpark location's path- *pip show pyspark*

Use the environment variables listed below to fix the problem on Windows-

set SPARK_HOME=C:\apps\opt\spark-3.0.0-bin-hadoop2.7

set HADOOP_HOME=%SPARK_HOME%

set PYTHONPATH=%SPARK_HOME%/python;%SPARK_HOME%/python/lib/py4j-0.10.9-src.zip;%PYTHONPATH%

Q8. Suppose you get an error- NameError: Name 'Spark' is not Defined while using spark. createDataFrame(), but there are no errors while using the same in Spark or PySpark shell. Why?

Spark shell, PySpark shell, and Databricks all have the SparkSession object 'spark' by default. However, if we are creating a Spark/PySpark application in a.py file, we must manually create a SparkSession object by using builder to resolve NameError: Name 'Spark' is not Defined.

 # Import PySpark

import pyspark

```
from pyspark.sql import SparkSession

#Create SparkSession

spark = SparkSession.builder

                .master("local[1]")

                .appName("SparkByExamples.com")

                .getOrCreate()
```

If you get the error message 'No module named pyspark', try using findspark instead-

```
#Install findspark

pip install findspark

# Import findspark

import findspark

findspark.init()

#import pyspark

import pyspark

from pyspark.sql import SparkSession
```

Q9. What are the various types of Cluster Managers in PySpark?

Spark supports the following cluster managers:

- **Standalone**- a simple cluster manager that comes with Spark and makes setting up a cluster easier.
- **Apache Mesos**- Mesos is a cluster manager that can also run Hadoop MapReduce and PySpark applications.
- **Hadoop YARN**- It is the Hadoop 2 resource management.

- **Kubernetes**- an open-source framework for automating containerized application deployment, scaling, and administration.
- **local** – not exactly a cluster manager, but it's worth mentioning because we use "local" for master() to run Spark on our laptop/computer.

Q10. Explain how Apache Spark Streaming works with receivers.

Receivers are unique objects in Apache Spark Streaming whose sole purpose is to consume data from various data sources and then move it to Spark. By streaming contexts as long-running tasks on various executors, we can generate receiver objects.

There are two different kinds of receivers which are as follows:

- Reliable receiver: When data is received and copied properly in Apache Spark Storage, this receiver validates data sources.
- Unreliable receiver: When receiving or replicating data in Apache Spark Storage, these receivers do not recognize data sources.