100+

PAGES OF APACHE SPARK NOTES

All about Data Engineering

Q. What is PySpark?

This is almost always the first PySpark interview question you will face.

PySpark is the Python API for Spark. It is used to provide collaboration between Spark and Python. PySpark focuses on processing structured and semi-structured data sets and also provides the facility to read data from multiple sources which have different data formats. Along with these features, we can also interface with RDDs (Resilient Distributed Datasets) using PySpark. All these features are implemented using the py4j library.

Q. List the advantages and disadvantages of PySpark? (Frequently asked PySpark Interview Question)

The advantages of using PySpark are:

- Using the PySpark, we can write a parallelized code in a very simple way.
- All the nodes and networks are abstracted.
- PySpark handles all the errors as well as synchronization errors.
- PySpark contains many useful in-built algorithms.

The disadvantages of using PySpark are:

- PySpark can often make it difficult to express problems in MapReduce fashion.
- When compared with other programming languages, PySpark is not efficient.

Q. What are the various algorithms supported in PySpark?

The different algorithms supported by PySpark are:

1. spark.mllib
2. mllib.clustering
3. mllib.classification
4. mllib.regression
5. mllib.recommendation
6. mllib.linalg
7. mllib.fpm

Q. What is PySpark SparkContext?

PySpark SparkContext can be seen as the initial point for entering and using any Spark functionality. The SparkContext uses py4j library to launch the JVM, and then create the JavaSparkContext. By default, the SparkContext is available as 'sc'.

Q. What is PySpark SparkFiles?

One of the most common PySpark interview questions. PySpark SparkFiles is used to load our files on the Apache Spark application. It is one of the functions under SparkContext and can be called using sc.addFile to load the files on the Apache Spark. SparkFIles can also be used to get the path using SparkFile.get or resolve the paths to files that were added from sc.addFile. The class methods present in the SparkFiles directory are getrootdirectory() and get(filename).

Q. What is PySpark SparkConf?

PySpark SparkConf is mainly used to set the configurations and the parameters when we want to run the application on the local or the cluster.

We run the following code whenever we want to run SparkConf:

class pyspark.Sparkconf(

localdefaults = True,

_jvm = None,

_jconf = None

)

Q. What is PySpark StorageLevel?

PySpark StorageLevel is used to control how the RDD is stored, take decisions on where the RDD will be stored (on memory or over the disk or both), and whether we need to replicate the RDD partitions or to serialize the RDD. The code for StorageLevel is as follows:

class pyspark.StorageLevel(useDisk, useMemory, useOfHeap, deserialized, replication = 1)

Q. What is PySpark SparkJobinfo?

One of the most common questions in any PySpark interview. PySpark SparkJobinfo is used to gain information about the SparkJobs that are in execution. The code for using the SparkJobInfo is as follows:

```
class SparkJobInfo(namedtuple("SparkJobInfo", "jobId stageIds status ")):
```

Q. What is PySpark SparkStageinfo?

One of the most common questions in any PySpark interview question and answers guide. PySpark SparkStageInfo is used to gain information about the SparkStages that are present at that time. The code used fo SparkStageInfo is as follows:

```
class SparkStageInfo(namedtuple("SparkStageInfo", "stageId currentAttemptId name numTasks unumActiveTasks" "numCompletedTasks numFailedTasks" )):
```

Q. 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.

Q. 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)

Q. 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

```python
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)

Q. 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), *

```python
    ("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)

}
```

Q. 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-

```
+-----+------------+
|Seqno|Names       |
+-----+------------+
|1    |john  jones |
|2    |tracey smith|
|3    |amy  sanders|
+-----+------------+
```

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())

Q. 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)
```

Q. 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	

Q. 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)

Q. 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.

Q. 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]     |
+----------+--------------------------------+
```

Q. 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
```

Q. 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.

Q. 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",400,"China"), *

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

   ("Banana",2000,"Canada"),("Carrots",2000,"Canada"),("Beans",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|
+-------+------+-----+------+----+
```

Q. 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)
```

Q. 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.

Q. 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

Q. 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()

}
```

Q. 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.

Q. 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-

```scala
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:

(1,1.4537951595091907) (2,0.7731024202454048)

(3,0.7731024202454048)

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.

Q. 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.

Q.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.

Q. 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")
```

Q. 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.

Q. 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

Q. 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.

Q. 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.

Q. 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.

Q. 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.

Q. 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.

Q. 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.

Q. 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.)

Q. 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

Q. 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.

Q. 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()
```

Q. 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.

Q. 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.

Q. 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.

Q. 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.

Q. 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.

Q. 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.

Q. 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()

Q. 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.

Q. 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.

Q. 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.

Q. 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.

Q. 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()*

Q. 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/python/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%

Q. 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

Q. 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.

Q. 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.

How is Apache Spark different from MapReduce?

Apache Spark	MapReduce
Spark processes data in batches as well as in real-time	MapReduce processes data in batches only
Spark runs almost 100 times faster than Hadoop MapReduce	Hadoop MapReduce is slower when it comes to large scale data processing
Spark stores data in the RAM i.e. in-memory. So, it is easier to retrieve it	Hadoop MapReduce data is stored in HDFS and hence takes a long time to retrieve the data

Spark provides caching and in-memory data storage	Hadoop is highly disk-dependent

Q. What are the important components of the Spark ecosystem?
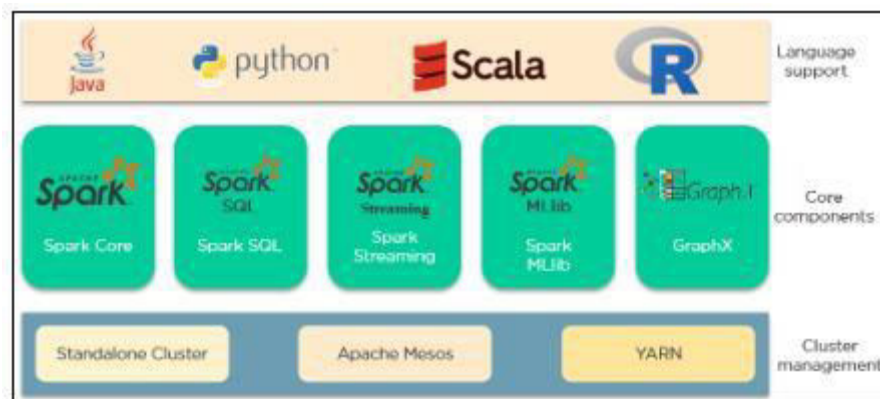
Apache Spark has 3 main categories that comprise its ecosystem. Those are:

- Language support: Spark can integrate with different languages to applications and perform analytics. These languages are Java, Python, Scala, and R.
- Core Components: Spark supports 5 main core components. There are Spark Core, Spark SQL, Spark Streaming, Spark MLlib, and GraphX.
- Cluster Management: Spark can be run in 3 environments. Those are the Standalone cluster, Apache Mesos, and YARN.

Q. Explain how Spark runs applications with the help of its architecture.

This is one of the most frequently asked spark interview questions, and the interviewer will expect you to give a thorough answer to it.
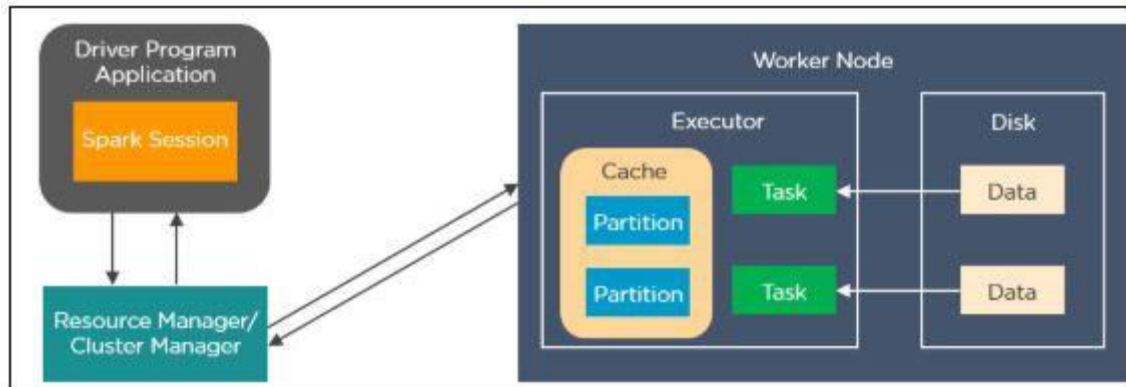
Spark applications run as independent processes that are coordinated by the SparkSession object in the driver program. The resource manager or cluster manager assigns tasks to the worker nodes with one task per partition. Iterative algorithms apply operations repeatedly to the data so they can benefit from caching datasets across iterations. A task applies its unit of work to the dataset in its partition and outputs a new partition dataset. Finally, the results are sent back to the driver application or can be saved to the disk.

Q. What are the different cluster managers available in Apache Spark?

- Standalone Mode: By default, applications submitted to the standalone mode cluster will run in FIFO order, and each application will try to use all available nodes. You can launch a standalone cluster either manually, by starting a master and workers by hand, or use our provided launch scripts. It is also possible to run these daemons on a single machine for testing.
- Apache Mesos: Apache Mesos is an open-source project to manage computer clusters, and can also run Hadoop applications. The advantages of deploying Spark with Mesos include dynamic partitioning between Spark and other frameworks as well as scalable partitioning between multiple instances of Spark.
- Hadoop YARN: Apache YARN is the cluster resource manager of Hadoop 2. Spark can be run on YARN as well.
- Kubernetes: Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.

Q. What is the significance of Resilient Distributed Datasets in Spark?

Resilient Distributed Datasets are the fundamental data structure of Apache Spark. It is embedded in Spark Core. RDDs are immutable, fault-tolerant, distributed collections of objects that can be operated on in parallel.RDD's are split into partitions and can be executed on different nodes of a cluster.

RDDs are created by either transformation of existing RDDs or by loading an external dataset from stable storage like HDFS or HBase.

Here is how the architecture of RDD looks like:
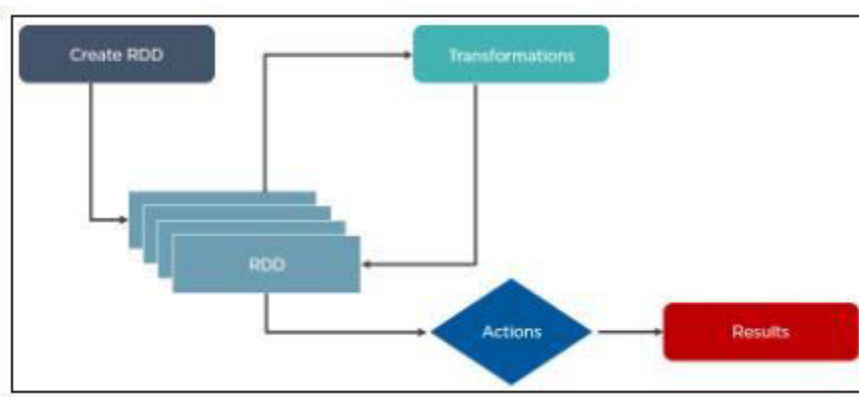
So far, if you have any doubts regarding the apache spark interview questions and answers, please comment below.

Q. What is a lazy evaluation in Spark?

When Spark operates on any dataset, it remembers the instructions. When a transformation such as a map() is called on an RDD, the operation is not performed instantly. Transformations in Spark are not evaluated until you perform an action, which aids in optimizing the overall data processing workflow, known as lazy evaluation.

Q. What makes Spark good at low latency workloads like graph processing and Machine Learning?

Apache Spark stores data in-memory for faster processing and building machine learning models. Machine Learning algorithms require multiple iterations and different conceptual steps to create an optimal model. Graph algorithms traverse through all the nodes and edges to generate a graph. These low latency workloads that need multiple iterations can lead to increased performance.

How can you trigger automatic clean-ups in Spark to handle accumulated metadata?

To trigger the clean-ups, you need to set the parameter spark.cleaner.ttlx.
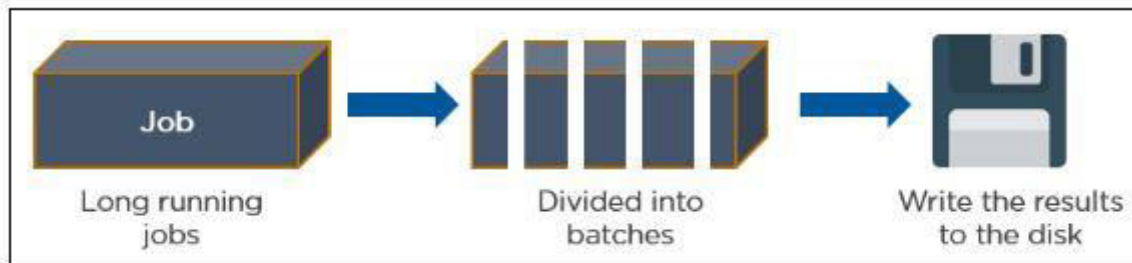
Q. How can you connect Spark to Apache Mesos?

There are a total of 4 steps that can help you connect Spark to Apache Mesos.

- Configure the Spark Driver program to connect with Apache Mesos
- Put the Spark binary package in a location accessible by Mesos
- Install Spark in the same location as that of the Apache Mesos
- Configure the spark.mesos.executor.home property for pointing to the location where Spark is installed

Q. What is a Parquet file and what are its advantages?

Parquet is a columnar format that is supported by several data processing systems. With the Parquet file, Spark can perform both read and write operations.

Some of the advantages of having a Parquet file are:

- It enables you to fetch specific columns for access.
- It consumes less space
- It follows the type-specific encoding
- It supports limited I/O operations

Learn open-source framework and scala programming languages with the Apache Spark and Scala Certification training course.

Q. What is shuffling in Spark? When does it occur?

Shuffling is the process of redistributing data across partitions that may lead to data movement across the executors. The shuffle operation is implemented differently in Spark compared to Hadoop.

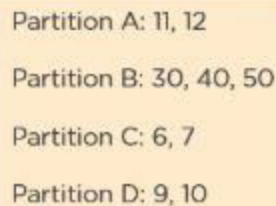Shuffling has 2 important compression parameters:

spark.shuffle.compress – checks whether the engine would compress shuffle outputs or not spark.shuffle.spill.compress – decides whether to compress intermediate shuffle spill files or not

It occurs while joining two tables or while performing byKey operations such as GroupByKey or ReduceByKey

Q. What is the use of coalesce in Spark?
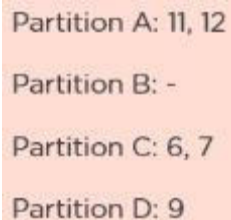
Spark uses a coalesce method to reduce the number of partitions in a DataFrame.

Suppose you want to read data from a CSV file into an RDD having four partitions.

Partition A: 11, 12

Partition B: 30, 40, 50

Partition C: 6, 7

Partition D: 9, 10

This is how a filter operation is performed to remove all the multiple of 10 from the data.

Partition A: 11, 12

Partition B: -

Partition C: 6, 7

Partition D: 9

The RDD has some empty partitions. It makes sense to reduce the number of partitions, which can be achieved by using coalesce.

Partition A: 11, 12

Partition C: 6, 7, 9

This is how the resultant RDD would look like after applying to coalesce.

Q. How can you calculate the executor memory?

Consider the following cluster information:

Nodes = 10
Each node has core = 16 cores (-1 for OS)
Each node Ram = 61GB Ram (-1 for OS)

Here is the number of core identification:

Number of cores is the number of concurrent tasks an executor can run in parallel. So the general rule of thumb for optimal value is 5

To calculate the number of executor identification:

No. of executors = No. of cores/concurrent tasks
= 15/5
= 3
No. of nodes * no. of executor in each node =
no. of executor (for spark job)
= 10*3 = 30

Q. What are the various functionalities supported by Spark Core?

Spark Core is the engine for parallel and distributed processing of large data sets. The various functionalities supported by Spark Core include:

- Scheduling and monitoring jobs
- Memory management
- Fault recovery
- Task dispatching

Q. How do you convert a Spark RDD into a DataFrame?

There are 2 ways to convert a Spark RDD into a DataFrame:

- Using the helper function - toDF

import com.mapr.db.spark.sql._

val df = sc.loadFromMapRDB(<table-name>)

.where(field("first_name") === "Peter")

.select("_id", "first_name").toDF()

- Using SparkSession.createDataFrame

You can convert an RDD[Row] to a DataFrame by

calling createDataFrame on a SparkSession object

def createDataFrame(RDD, schema:StructType)

Q. Explain the types of operations supported by RDDs.

RDDs support 2 types of operation:

Transformations: Transformations are operations that are performed on an RDD to create a new RDD containing the results (Example: map, filter, join, union)

Actions: Actions are operations that return a value after running a computation on an RDD (Example: reduce, first, count)

Q.What is a Lineage Graph?

This is another frequently asked spark interview question. A Lineage Graph is a dependencies graph between the existing RDD and the new RDD. It means that all the dependencies between the RDD will be recorded in a graph, rather than the original data.

The need for an RDD lineage graph happens when we want to compute a new RDD or if we want to recover the lost data from the lost persisted RDD. Spark does not support data replication in memory. So, if any data is lost, it can be rebuilt using RDD lineage. It is also called an RDD operator graph or RDD dependency graph.

Q. What do you understand about DStreams in Spark?

Discretized Streams is the basic abstraction provided by Spark Streaming.

It represents a continuous stream of data that is either in the form of an input source or processed data stream generated by transforming the input stream.

dstream.

Q. Explain Caching in Spark Streaming.

Caching also known as Persistence is an optimization technique for Spark computations. Similar to RDDs, DStreams also allow developers to persist the stream's data in memory. That is, using the persist() method on a DStream will automatically

persist every RDD of that DStream in memory. It helps to save interim partial results so they can be reused in subsequent stages.

The default persistence level is set to replicate the data to two nodes for fault-tolerance, and for input streams that receive data over the network.

kafka

Q. What is the need for broadcast variables in Spark?

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used to give every node a copy of a large input dataset in an efficient manner. Spark distributes broadcast variables using efficient broadcast algorithms to reduce communication costs.

scala

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
```

```
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
```

```
scala> broadcastVar.value
```

res0: Array[Int] = Array(1, 2, 3)

So far, if you have any doubts regarding the spark interview questions for beginners, please ask in the comment section below.

How to programmatically specify a schema for DataFrame?

DataFrame can be created programmatically with three steps:

- Create an RDD of Rows from the original RDD;
- Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1.
- Apply the schema to the RDD of Rows via createDataFrame method provided by SparkSession.

```python
# Import data types
from pyspark.sql.types import *

sc = spark.sparkContext

# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
# Each line is converted to a tuple.
people = parts.map(lambda p: (p[0], p[1].strip()))

# The schema is encoded in a string.
schemaString = "name age"

fields = [StructField(field_name, StringType(), True) for field_name in schemaString.split()]
schema = StructType(fields)

# Apply the schema to the RDD.
schemaPeople = spark.createDataFrame(people, schema)

# Creates a temporary view using the DataFrame
schemaPeople.createOrReplaceTempView("people")

# SQL can be run over DataFrames that have been registered as a table.
results = spark.sql("SELECT name FROM people")

results.show()
# +-------+
# |   name|
# +-------+
# |Michael|
# |   Andy|
# | Justin|
# +-------+
```

Q. Which transformation returns a new DStream by selecting only those records of the source DStream for which the function returns true?

1. map(func)

2. transform(func)

3. filter(func)

4. count()

The correct answer is c) filter(func).

Q. Does Apache Spark provide checkpoints?

This is one of the most frequently asked spark interview questions where the interviewer expects a detailed answer (and not just a yes or no!). Give as detailed an answer as possible here.

Yes, Apache Spark provides an API for adding and managing checkpoints. Checkpointing is the process of making streaming applications resilient to failures. It allows you to save the data and metadata into a checkpointing directory. In case of a failure, the spark can recover this data and start from wherever it has stopped.
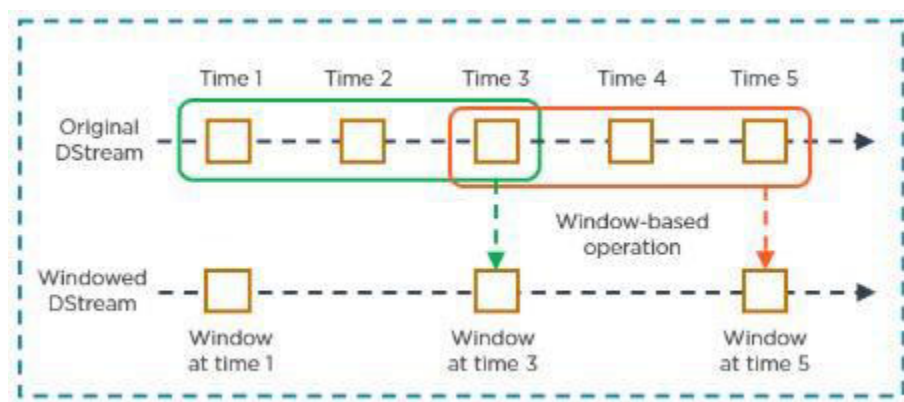
There are 2 types of data for which we can use checkpointing in Spark.

Metadata Checkpointing: Metadata means the data about data. It refers to saving the metadata to fault-tolerant storage like HDFS. Metadata includes configurations, DStream operations, and incomplete batches.

Data Checkpointing: Here, we save the RDD to reliable storage because its need arises in some of the stateful transformations. In this case, the upcoming RDD depends on the RDDs of previous batches.

Q. What do you mean by sliding window operation?

Controlling the transmission of data packets between multiple computer networks is done by the sliding window. Spark Streaming library provides windowed computations where the transformations on RDDs are applied over a sliding window of data.

Q. What are the different levels of persistence in Spark?

DISK_ONLY - Stores the RDD partitions only on the disk

MEMORY_ONLY_SER - Stores the RDD as serialized Java objects with a one-byte array per partition

MEMORY_ONLY - Stores the RDD as deserialized Java objects in the JVM. If the RDD is not able to fit in the memory available, some partitions won't be cached

OFF_HEAP - Works like MEMORY_ONLY_SER but stores the data in off-heap memory

MEMORY_AND_DISK - Stores RDD as deserialized Java objects in the JVM. In case the RDD is not able to fit in the memory, additional partitions are stored on the disk

MEMORY_AND_DISK_SER - Identical to MEMORY_ONLY_SER with the exception of storing partitions not able to fit in the memory to the disk

Q. What is the difference between map and flatMap transformation in Spark Streaming?

map()	flatMap()
A map function returns a new DStream by passing each element of the source DStream through a function func	It is similar to the map function and applies to each element of RDD and it returns the result as a new RDD
Spark Map function takes one element as an input process it according to custom code (specified by the developer) and returns one element at a time	FlatMap allows returning 0, 1, or more elements from the map function. In the FlatMap operation

Q. How would you compute the total count of unique words in Spark?

1. Load the text file as RDD:

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

2. Function that breaks each line into words:

def toWords(line):

return line.split();

3. Run the toWords function on each element of RDD in Spark as flatMap transformation:

```
words = line.flatMap(toWords);
```

4. Convert each word into (key,value) pair:

```
def toTuple(word):

return (word, 1);

wordTuple = words.map(toTuple);
```

5. Perform reduceByKey() action:

```
def sum(x, y):

return x+y:

counts = wordsTuple.reduceByKey(sum)
```

6. Print:

```
counts.collect()
```


Suppose you have a huge text file. How will you check if a particular keyword exists using Spark?

```
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";
```

Q. What is the role of accumulators in Spark?

Accumulators are variables used for aggregating information across the executors. This information can be about the data or API diagnosis like how many records are corrupted or how many times a library API was called.

30. What are the different MLlib tools available in Spark?

- ML Algorithms: Classification, Regression, Clustering, and Collaborative filtering
- Featurization: Feature extraction, Transformation, Dimensionality reduction,

and Selection

- Pipelines: Tools for constructing, evaluating, and tuning ML pipelines
- Persistence: Saving and loading algorithms, models, and pipelines
- Utilities: Linear algebra, statistics, data handling

Hope it is clear so far. Let us know what were the apache spark interview questions ask'd by/to you during the interview process.

Q. What are the different data types supported by Spark MLlib?

Spark MLlib supports local vectors and matrices stored on a single machine, as well as distributed matrices.

Local Vector: MLlib supports two types of local vectors - dense and sparse

Example: vector(1.0, 0.0, 3.0)

dense format: [1.0, 0.0, 3.0]

sparse format: (3, [0, 2]. [1.0, 3.0])

Labeled point: A labeled point is a local vector, either dense or sparse that is associated with a label/response.

Example: In binary classification, a label should be either 0 (negative) or 1 (positive)

Local Matrix: A local matrix has integer type row and column indices, and double type values that are stored in a single machine.

Distributed Matrix: A distributed matrix has long-type row and column indices and double-type values, and is stored in a distributed manner in one or more RDDs.

Types of the distributed matrix:

- RowMatrix
- IndexedRowMatrix
- CoordinatedMatrix

Q. What is a Sparse Vector?

A Sparse vector is a type of local vector which is represented by an index array and a value array.

public class SparseVector

extends Object

implements Vector

Example: sparse1 = SparseVector(4, [1, 3], [3.0, 4.0])

where:

4 is the size of the vector

[1,3] are the ordered indices of the vector

[3,4] are the value

Do you have a better example for this spark interview question? If yes, let us know.

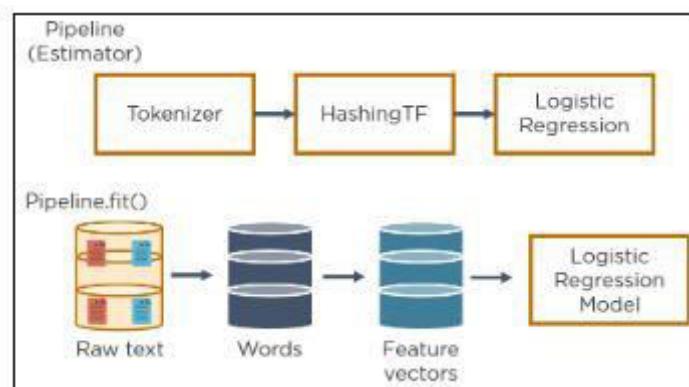Q. Describe how model creation works with MLlib and how the model is applied.

MLlib has 2 components:

Transformer: A transformer reads a DataFrame and returns a new DataFrame with a specific transformation applied.
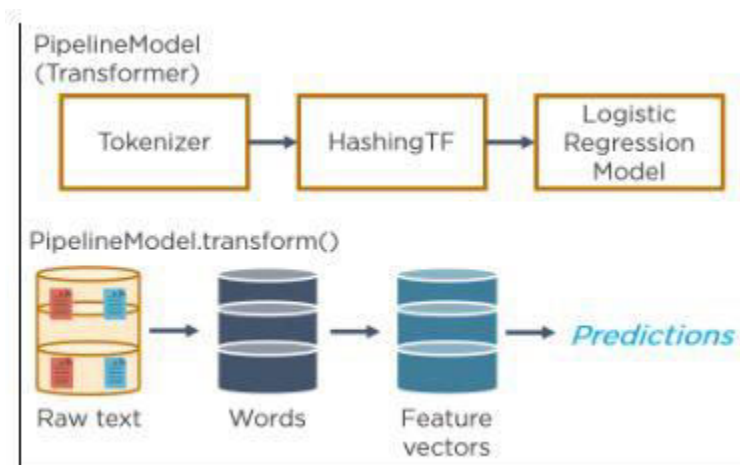
Estimator: An estimator is a machine learning algorithm that takes a DataFrame to train a model and returns the model as a transformer.

Spark MLlib lets you combine multiple transformations into a pipeline to apply complex data transformations.

The following image shows such a pipeline for training a model:

The model produced can then be applied to live data:

Q. What are the functions of Spark SQL?

Spark SQL is Apache Spark's module for working with structured data.

Spark SQL loads the data from a variety of structured data sources.

It queries data using SQL statements, both inside a Spark program and from external tools that connect to Spark SQL through standard database connectors (JDBC/ODBC).

It provides a rich integration between SQL and regular Python/Java/Scala code, including the ability to join RDDs and SQL tables and expose custom functions in SQL.

How can you connect Hive to Spark SQL?

To connect Hive to Spark SQL, place the hive-site.xml file in the conf directory of Spark.
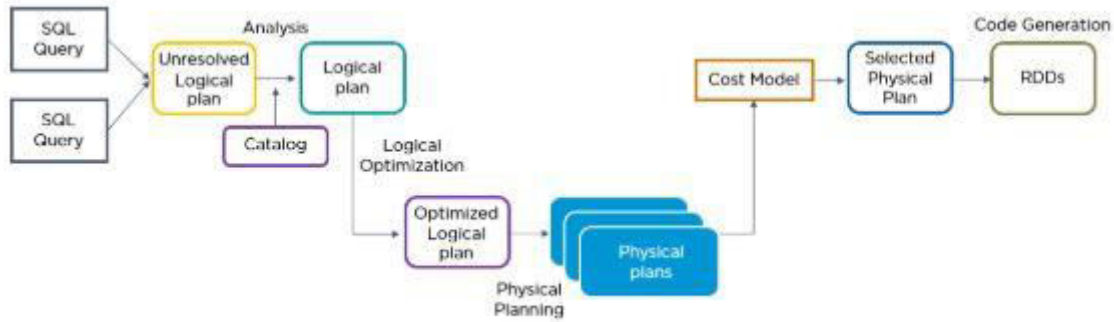

hive-site.xml   conf directory

Using the Spark Session object, you can construct a DataFrame.

result=spark.sql("select * from <hive_table>")

Q. What is the role of Catalyst Optimizer in Spark SQL?

Catalyst optimizer leverages advanced programming language features (such as Scala's pattern matching and quasi quotes) in a novel way to build an extensible query optimizer.

Q. How can you manipulate structured data using domain-specific language in Spark SQL?

Structured data can be manipulated using domain-Specific language as follows:

Suppose there is a DataFrame with the following information:

val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout

df.show()

// +----+------+

// | age| name|

// +----+------+

// |null|Michael|

// | 30| Andy|

// | 19| Justin|

// +----+------+

```
// Select only the "name" column

df.select("name").show()

// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+

// Select everybody, but increment the age by 1

df.select($"name", $"age" + 1).show()

// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+
```

```
// Select people older than 21

df.filter($"age" > 21).show()

// +---+----+

// |age|name|

// +---+----+

// | 30|Andy|

// +---+----+

// Count people by age

df.groupBy("age").count().show()

// +----+-----+

// | age|count|

// +----+-----+

// |  19|    1|

// |null|    1|

// |  30|    1|

// +----+-----+
```

Q. What are the different types of operators provided by the Apache GraphX library?

In such spark interview questions, try giving an explanation too (not just the name of the operators).

Property Operator: Property operators modify the vertex or edge properties using a user-defined map function and produce a new graph.

Structural Operator: Structure operators operate on the structure of an input graph and produce a new graph.

Join Operator: Join operators add data to graphs and generate new graphs.

Q. What are the analytic algorithms provided in Apache Spark GraphX?

GraphX is Apache Spark's API for graphs and graph-parallel computation. GraphX includes a set of graph algorithms to simplify analytics tasks. The algorithms are contained in the org.apache.spark.graphx.lib package and can be accessed directly as methods on Graph via GraphOps.

PageRank: PageRank is a graph parallel computation that measures the importance of each vertex in a graph. Example: You can run PageRank to evaluate what the most important pages in Wikipedia are.

Connected Components: The connected components algorithm labels each connected component of the graph with the ID of its lowest-numbered vertex. For example, in a social network, connected components can approximate clusters.

Triangle Counting: A vertex is part of a triangle when it has two adjacent vertices with an edge between them. GraphX implements a triangle counting algorithm in the TriangleCount object that determines the number of triangles passing through each vertex, providing a measure of clustering.

Qq. What is the PageRank algorithm in Apache Spark GraphX?

It is a plus point if you are able to explain this spark interview question thoroughly, along with an example! PageRank measures the importance of each vertex in a graph, assuming an edge from u to v represents an endorsement of v's importance by u.

If a Twitter user is followed by many other users, that handle will be ranked high.

PageRank algorithm was originally developed by Larry Page and Sergey Brin to rank websites for Google. It can be applied to measure the influence of vertices in any network graph. PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The assumption is that more important websites are likely to receive more links from other websites.

A typical example of using Scala's functional programming with Apache Spark RDDs to iteratively compute Page Ranks is shown below:

```scala
object SparkPageRank {
  def main(args: Array[String]) {
    val spark = SparkSession
      .builder
      .appName("SparkPageRank")
      .getOrCreate()

    val iters = if (args.length > 1) args(1).toInt else 10
    val lines = spark.read.textFile(args(0)).rdd
    val links = lines.map{ s =>
      val parts = s.split("\\s+")
      (parts(0), parts(1))
    }.distinct().groupByKey().cache()

    var ranks = links.mapValues(v => 1.0)

    for (i <- 1 to iters) {
      val contribs = links.join(ranks).values.flatMap{ case (urls, rank) =
        val size = urls.size
        urls.map(url => (url, rank / size))
      }
      ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
    }

    val output = ranks.collect()
    output.foreach(tup => println(tup._1 + " has rank: " + tup._2 + "."))

    spark.stop()
  }
}
```

Q. Explain how an object is implemented in python?

Ans: An object is an instantiation of a class. A class can be instantiated by calling the class using the class name.

Syntax:

 = ()

Example:

class Student:

    id = 25;

    name = "HKR Trainings"

    estb = 10

    def display (self):

```python
        print("ID: %d \n Name: %s \n Estb: %d "%(self.id,self.name,self.estb))

stud = Student()

stud.display()
```

Output:

ID: 25

Name: HKR Trainings

Estb: 10

Q. Explain Methods in Python

Ans: In Python, a method is a function that is associated with an object. Any object type can have methods.

Example:

```python
class Student:

    roll = 17;

    name = "gopal"

    age = 25

    def display (self):

        print(self.roll,self.name,self.age)
```

In the above example, a class named Student is created which contains three fields as Student's roll, name, age and a function "display()" which is used to display the information of the Student.

Q. What is encapsulation in Python?

Ans: Encapsulation is used to restrict access to methods and variables. Here, the methods and data/variables are wrapped together within a single unit so as to prevent data from direct modification.

Below is the example of encapsulation whereby the max price of the product cannot be modified as it is set to 75 .

Example:

class Product:

```
    def __init__(self):

        self.__maxprice = 75

    def sell(self):
```

```python
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):

        self.__maxprice = price


p = Product()


p.sell()


# change the price


p.__maxprice = 100


p.sell()
```

Output:

Selling Price: 75

Selling Price: 75

Q. Explain the concept of Python Inheritance

Ans: Inheritance refers to a concept where one class inherits the properties of another. It helps to reuse the code and establish a relationship between different classes.

Amongst following two types of classes, inheritance is performed:

Parent class (Super or Base class): A class whose properties are inherited.

Child class (Subclass or Derived class): A class which inherits the properties.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name.

The syntax to inherit a base class into the derived class is shown below:

Syntax:

class derived-class(base class):

The syntax to inherit multiple classes is shown below by specifying all of them inside the bracket.

Syntax:

class derive-class(, , ...):

Q. What is a python for loop?

Ans: A for loop in Python requires at least two variables to work. The first is the iterable object such as a list, tuple or a string and second is the variable to store the successive values from the sequence in the loop.

Syntax:

```
for iter in sequence:

    statements(iter)
```

The "iter" represents the iteration variable. It gets assigned with the successive values from the input sequence.

The "sequence" may refer to any of the following Python objects such as a list, a tuple or a string.

Q. What is a for-else in python?

Ans: Python allows us to handle loops in an interesting manner by providing a function to write else blocks for cases when the loop does not satisfy a certain condition.

Example :

```
x = []

    for i in x:
```

```
    print "in for loop"


  else:


    print "in else block"
```

Output:

in else block

Q. What are errors and exceptions in python programming?

Ans: In Python, there are two types of errors - syntax error and exceptions.

Syntax Error: It is also known as parsing errors. Errors are issues in a program which may cause it to exit abnormally. When an error is detected, the parser repeats the offending line and then displays an arrow which points at the earliest point in the line.

Exceptions: Exceptions take place in a program when the normal flow of the program is interrupted due to the occurrence of an external event. Even if the syntax of the program is correct, there are chances of detecting an error during execution, this error is nothing

but an exception. Some of the examples of exceptions are - ZeroDivisionError, TypeError and NameError.

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

Ans:

The key difference between lists and tuples is the fact that lists have mutable nature and tuples have immutable nature.

It is said to be a mutable data type when a python object can be modified. On the other hand, immutable data types cannot be modified. Let us see an example to modify an item list vs tuple.

Example:

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 does not support item assignment

In this code, we had assigned 7 to list_num at index 3 and in the output, we can see 7 is found in index 3 . However, we had assigned 7 to tup_num at index 3 but we got type error on the output. This is because we cannot modify tuples due to its immutable nature.

Q. How to convert a string to a number in python?

Ans: The method provided by Python, is a standard built-in function which converts a string into an integer value.

It can be called with a string containing a number as the argument, and it will return the number converted to an actual integer.

Example:

```
print int("1") + 2
```

The above prints 3 .

Q. What is data cleaning?

Ans: Data cleaning is the process of preparing data for analysis by removing or modifying data that is incorrect, incomplete, irrelevant, duplicated, or improperly formatted.

Q. What is data visualization and why is it important?

Ans: Data visualization is the representation of data or information in a graph, chart, or other visual format. It communicates relationships of the data with images. The data visualizations are important because it allows trends and patterns to be more easily seen.

PySpark Training Certification

 Master Your Craft Lifetime LMS & Faculty Access 24/7 online expert support Real-world & Project Based Learning

Q. What is Pyspark and explain its characteristics?

Ans: To support Python with Spark, the Spark community has released a tool called PySpark. It is primarily used to process structured and semi-structured datasets and also supports an optimized API to read data from the multiple data sources containing different file formats. Using PySpark, you can also work with RDDs in the Python programming language using its library name Py4j.

The main characteristics of PySpark are listed below:

Nodes are Abstracted.

Based on MapReduce.

API for Spark.

The network is abstracted.

Q. Explain RDD and also state how you can create RDDs in Apache Spark.

Ans: RDD stands for Resilient Distribution Datasets, a fault-tolerant set of operational elements that are capable of running in parallel. These RDDs, in general, are the portions of data, which are stored in the memory and distributed over many nodes.

All partitioned data in an RDD is distributed and immutable.

There are primarily two types of RDDs are available:

Hadoop datasets: Those who perform a function on each file record in Hadoop Distributed File System (HDFS) or any other storage system.

Parallelized collections: Those existing RDDs which run in parallel with one another.

Q. How do we create RDDs in Spark?

Ans: Spark provides two methods to create RDD:

By parallelizing a collection in your Driver program.

This makes use of SparkContext's 'parallelize'.

method val DataArray = Array(2,4,6,8,10)

val DataRDD = sc.parallelize(DataArray)

By loading an external dataset from external storage like HDFS, HBase, shared file system.

Q. Name the components of Apache Spark?

Ans: The following are the components of Apache Spark.

Spark Core: Base engine for large-scale parallel and distributed data processing.

Spark Streaming: Used for processing real-time streaming data.

Spark SQL: Integrates relational processing with Spark's functional programming API.

GraphX: Graphs and graph-parallel computation.

MLlib: Performs machine learning in Apache Spark.

Q. How DAG functions in Spark?

Ans: When an Action is approached at a certain point, Spark RDD at an abnormal state, Spark presents the heredity chart to the DAG Scheduler.

Activities are separated into phases of the errand in the DAG Scheduler. A phase contains errands dependent on the parcel of the info information. The DAG scheduler pipelines administrators together. It dispatches tasks through the group chief. The conditions of stages are obscure to the errand scheduler. The Workers execute the undertaking on the slave.

Q. What do you mean by Spark Streaming?

Ans: Stream processing is an extension to the Spark API that lets stream processing of live data streams. Data from multiple sources such as Flume, Kafka, Kinesis, etc., is processed and then pushed to live dashboards, file systems, and databases. Compared to the terms of input data, it is just similar to batch processing, and data is segregated into streams like batches in processing.

Q. What does MLlib do?

Ans: MLlib is a scalable Machine Learning library offered by Spark. It supports making Machine Learning secure and scalable with standard learning algorithms and use cases such as regression filtering, clustering, dimensional reduction.

Q. What are the different MLlib tools available in Spark?

Ans:

ML Algorithms: Classification, Regression, Clustering, and Collaborative filtering.

Featurization: Feature extraction, Transformation, Dimensionality reduction, and Selection.

Pipelines: Tools for constructing, evaluating, and tuning ML pipelines

Persistence: Saving and loading algorithms, models and pipelines.

Utilities: Linear algebra, statistics, data handling.

Q. Explain the functions of SparkCore.

Ans: SparkCore implements several key functions such as

Memory management.

Fault-tolerance.

Monitoring jobs.

Job scheduling.

Interaction with storage systems.

Moreover, additional libraries, built atop the core, let diverse workloads for streaming, machine learning, and SQL. This is useful for:

Memory management.

fault recovery.

Interacting with storage systems.

Scheduling and monitoring jobs on a cluster.

Q. What is the module used to implement SQL in Spark? How does it work?

Ans: The module used is Spark SQL, which integrates relational processing with Spark's functional programming API. It helps to query data either through Hive Query Language or SQL. These are the four libraries of Spark SQL.

Data Source API.

Interpreter & Optimizer.

DataFrame API.

SQL Service.

Q. List the functions of Spark SQL.

Ans: Spark SQL is capable of:

Loading data from a variety of structured sources.

Querying data using SQL statements, both inside a Spark program and from external tools that connect to Spark SQL through standard database connectors (JDBC/ODBC). For instance, using business intelligence tools like Tableau.

Providing rich integration between SQL and regular Python/Java/Scala code, including the ability to join RDDs and SQL tables, expose custom functions in SQL, and more.

Q. What are the various algorithms supported in PySpark?

Ans: The different algorithms supported by PySpark are:

Spark.mllib.

mllib.clustering.

mllib.classification.

mllib.regression.

mllib.recommendation.

mllib.linalg.

mllib.fpm.

Q. Explain the purpose of serializations in PySpark?

Ans: For improving performance, PySpark supports custom serializers to transfer data. They are:

MarshalSerializer: It supports only fewer data types, but compared to PickleSerializer, it is faster.

PickleSerializer: It is by default used for serializing objects. Supports any Python object but at a slow speed.

Q. What is PySpark StorageLevel?25 . What is PySpark StorageLevel?

Ans: PySpark Storage Level controls storage of an RDD. It also manages how to store RDD in the memory or over the disk, or sometimes both. Moreover, it even controls the replicate or serializes RDD partitions. The code for StorageLevel is as follows

class pyspark.StorageLevel(useDisk, useMemory, useOfHeap, deserialized, replication = 1)

Q. What is PySpark SparkContext?

Ans: PySpark SparkContext is treated as an initial point for entering and using any Spark functionality. The SparkContext uses py4j library to launch the JVM, and then create the JavaSparkContext. By default, the SparkContext is available as 'sc'.

Q. What is PySpark SparkFiles?

Ans: PySpark SparkFiles is used to load our files on the Apache Spark application. It is one of the functions under SparkContext and can be called using sc.addFile to load the

files on the Apache Spark. SparkFIles can also be used to get the path using SparkFile.get or resolve the paths to files that were added from sc.addFile. The class methods present in the SparkFiles directory are getrootdirectory() and get(filename).

Q. Explain Spark Execution Engine?

Ans: Apache Spark is a graph execution engine that enables users to analyze massive data sets with high performance. For this, Spark first needs to be held in memory to improve performance drastically, if data needs to be manipulated with multiple stages of processing.

Q. What do you mean by SparkConf in PySpark?

Ans: SparkConf helps in setting a few configurations and parameters to run a Spark application on the local/cluster. In simple terms, it provides configurations to run a Spark application.

Q. Name a few attributes of SparkConf.

Ans: Few main attributes of SparkConf are listed below:

set(key, value): This attribute helps in setting the configuration property.

setSparkHome(value): This attribute enables in setting Spark installation path on worker nodes.

setAppName(value): This attribute helps in setting the application name.

setMaster(value): This attribute helps in setting the master URL.

get(key, defaultValue=None): This attribute supports in getting a configuration value of a key.
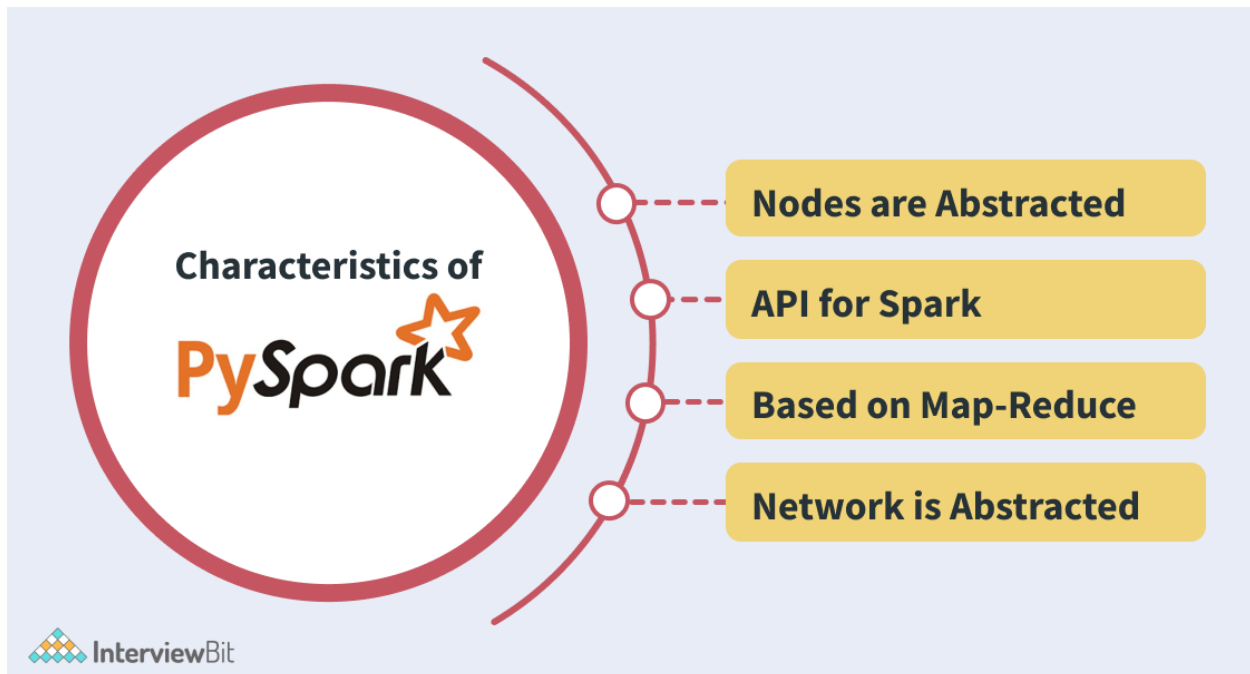
Q. What is PySpark?

PySpark is an Apache Spark interface in Python. It is used for collaborating with Spark using APIs written in Python. It also supports Spark's features like Spark DataFrame, Spark SQL, Spark Streaming, Spark MLlib and Spark Core. It provides an interactive PySpark shell to analyze structured and semi-structured data in a distributed environment. PySpark supports reading data from multiple sources and different formats. It also facilitates the use of RDDs (Resilient Distributed Datasets). PySpark features are implemented in the py4j library in python.

PySpark can be installed using PyPi by using the command:

```
pip install pyspark
```

Q. What are the characteristics of PySpark?

There are 4 characteristics of PySpark:

Characteristics of PySpark — InterviewBit

- Abstracted Nodes: This means that the individual worker nodes can not be addressed.
- Spark API: PySpark provides APIs for utilizing Spark features.
- Map-Reduce Model: PySpark is based on Hadoop's Map-Reduce model this means that the programmer provides the map and the reduce functions.
- Abstracted Network: Networks are abstracted in PySpark which means that the only possible communication is implicit communication.

Q. What are the advantages and disadvantages of PySpark?

Advantages of PySpark:

- Simple to use: Parallelized code can be written in a simpler manner.
- Error Handling: PySpark framework easily handles errors.
- Inbuilt Algorithms: PySpark provides many of the useful algorithms in Machine Learning or Graphs.
- Library Support: Compared to Scala, Python has a huge library collection for working in the field of data science and data visualization.
- Easy to Learn: PySpark is an easy to learn language.

Disadvantages of PySpark:

- Sometimes, it becomes difficult to express problems using the MapReduce model.
- Since Spark was originally developed in Scala, while using PySpark in Python programs they are relatively less efficient and approximately 10x times slower than the Scala programs. This would impact the performance of heavy data processing applications.
- The Spark Streaming API in PySpark is not mature when compared to Scala. It still requires improvements.
- PySpark cannot be used for modifying the internal function of the Spark due to the abstractions provided. In such cases, Scala is preferred.

You can download a PDF version of Pyspark Interview Questions.
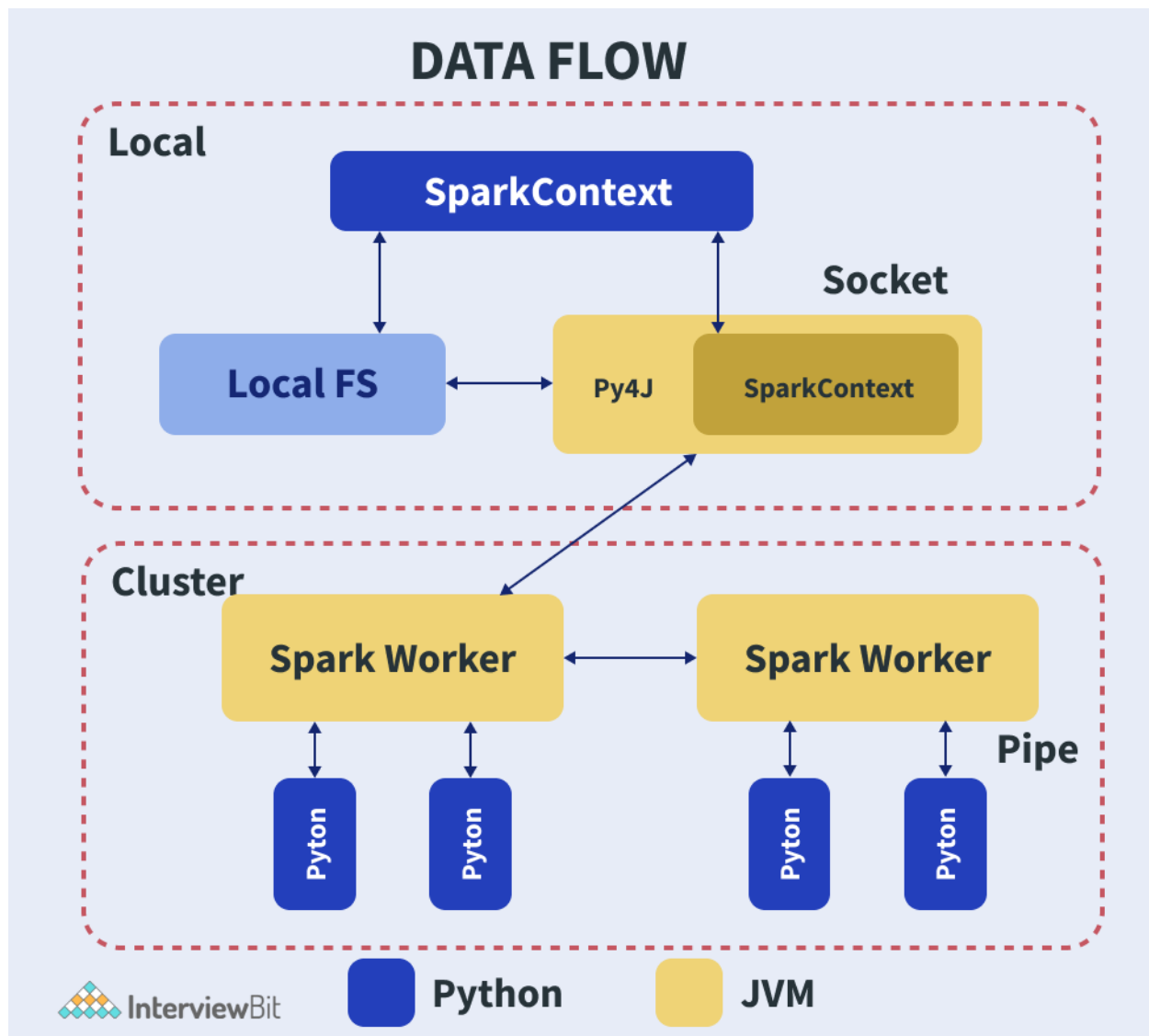
Download PDF

Q. What is PySpark SparkContext?

PySpark SparkContext is an initial entry point of the spark functionality. It also represents Spark Cluster Connection and can be used for creating the Spark RDDs (Resilient Distributed Datasets) and broadcasting the variables on the cluster.

The following diagram represents the architectural diagram of PySpark's SparkContext:

DATA FLOW

When we want to run the Spark application, a driver program that has the main function will be started. From this point, the SparkContext that we defined gets initiated. Later on, the driver program performs operations inside the executors of the worker nodes. Additionally, JVM will be launched using Py4J which in turn creates JavaSparkContext. Since PySpark has default SparkContext available as "sc", there will not be a creation of a new SparkContext.

Q. Why do we use PySpark SparkFiles?

PySpark's SparkFiles are used for loading the files onto the Spark application. This functionality is present under SparkContext and can be called using the `sc.addFile()` method for loading files on Spark. SparkFiles can also be used for getting the path using the `SparkFiles.get()` method. It can also be used to resolve paths to files added using the `sc.addFile()` method.

Q. What are PySpark serializers?

The serialization process is used to conduct performance tuning on Spark. The data sent or received over the network to the disk or memory should be persisted. PySpark supports serializers for this purpose. It supports two types of serializers, they are:

- PickleSerializer: This serializes objects using Python's PickleSerializer (`class pyspark.PickleSerializer`). This supports almost every Python object.
- MarshalSerializer: This performs serialization of objects. We can use it by using `class pyspark.MarshalSerializer`. This serializer is faster than the PickleSerializer but it supports only limited types.

Consider an example of serialization which makes use of MarshalSerializer:

```
# --serializing.py----
from pyspark.context import SparkContext
from pyspark.serializers import MarshalSerializer
sc = SparkContext("local", "Marshal Serialization", serializer =
MarshalSerializer())    #Initialize spark context and serializer
print(sc.parallelize(list(range(1000))).map(lambda x: 3 *
x).take(5))
sc.stop()
```

When we run the file using the command:

```
$SPARK_HOME/bin/spark-submit serializing.py
```
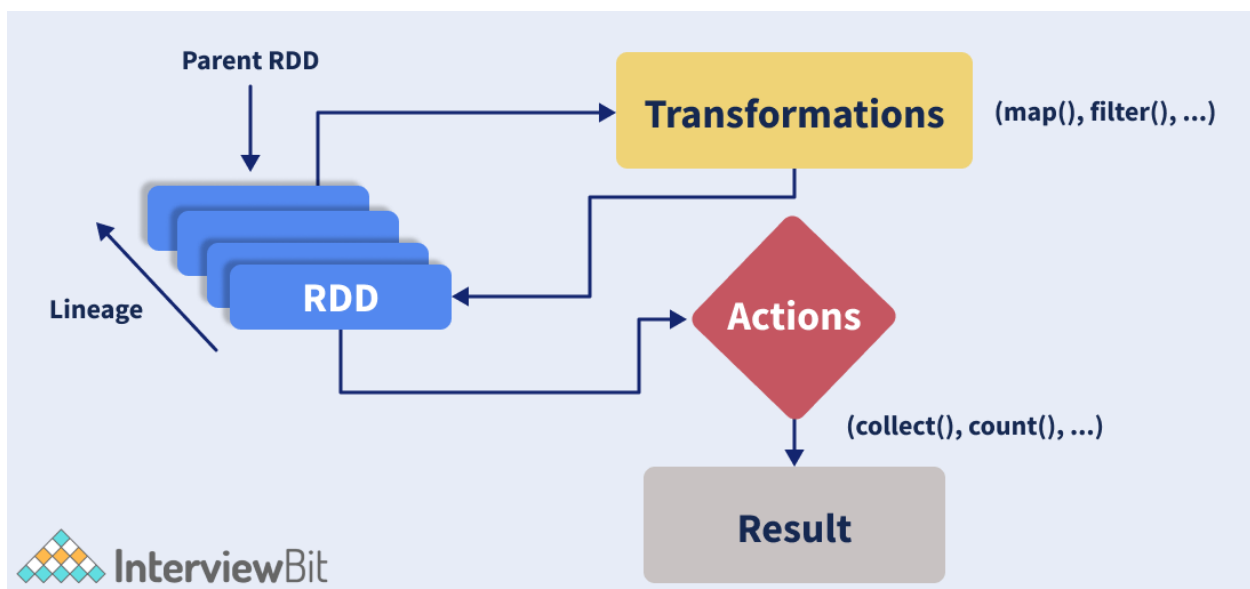
The output of the code would be the list of size 5 of numbers multiplied by 3:

```
[0, 3, 6, 9, 12]
```

Q. What are RDDs in PySpark?

RDDs expand to Resilient Distributed Datasets. These are the elements that are used for running and operating on multiple nodes to perform parallel processing on a cluster. Since RDDs are suited for parallel processing, they are immutable elements. This means that once we create RDD, we cannot modify it. RDDs are also fault-tolerant which means that whenever failure happens, they can be recovered automatically. Multiple operations can be performed on RDDs to perform a certain task. The operations can be of 2 types:

- Transformation: These operations when applied on RDDs result in the creation of a new RDD. Some of the examples of transformation operations are filter, groupBy, map.

  Let us take an example to demonstrate transformation operation by considering filter() operation:

```python
from pyspark import SparkContext
sc = SparkContext("local", "Transdormation Demo")
words_list = sc.parallelize (
  ["pyspark",
  "interview",
  "questions",
```

```
  "at",
  "interviewbit"]
)
filtered_words = words_list.filter(lambda x: 'interview' in x)
filtered = filtered_words.collect()
print(filtered)
```

The above code filters all the elements in the list that has 'interview' in the element. The output of the above code would be:

```
[
  "interview",
  "interviewbit"
]
```

- Action: These operations instruct Spark to perform some computations on the RDD and return the result to the driver. It sends data from the Executer to the driver. count(), collect(), take() are some of the examples.
  Let us consider an example to demonstrate action operation by making use of the count() function.

```
from pyspark import SparkContext
sc = SparkContext("local", "Action Demo")
words = sc.parallelize (
  ["pyspark",
   "interview",
   "questions",
   "at",
   "interviewbit"]
)
counts = words.count()
print("Count of elements in RDD -> ",  counts)
```

In this class, we count the number of elements in the spark RDDs. The output of this code is
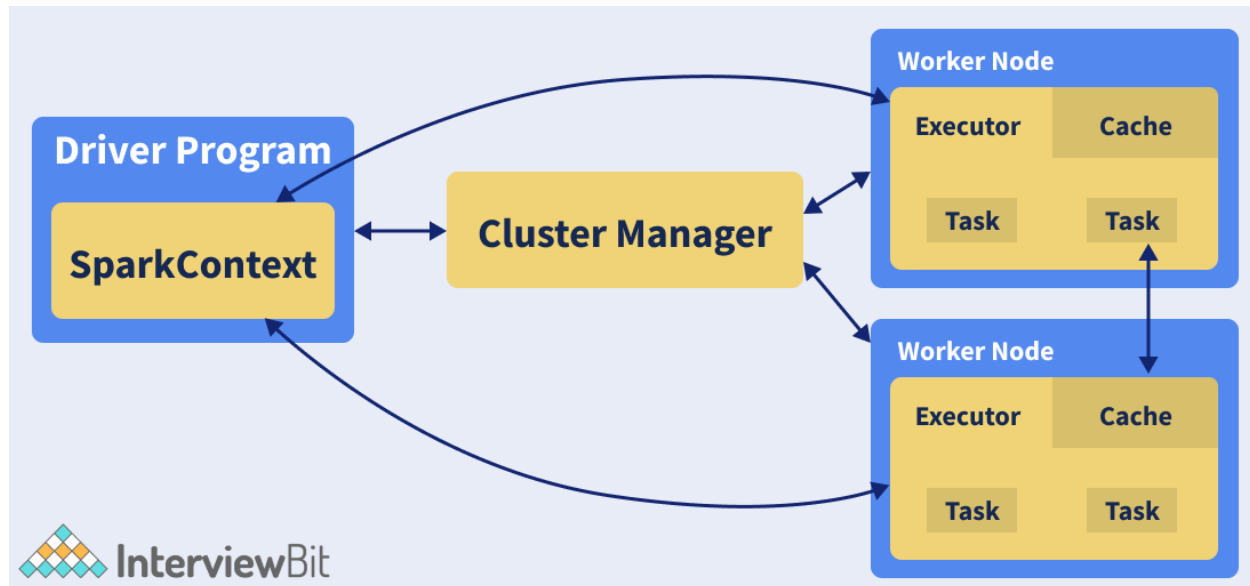
```
Count of elements in RDD -> 5
```

Q. Does PySpark provide a machine learning API?

Similar to Spark, PySpark provides a machine learning API which is known as MLlib that supports various ML algorithms like:

- `mllib.classification` – This supports different methods for binary or multiclass classification and regression analysis like Random Forest, Decision Tree, Naive Bayes etc.
- `mllib.clustering` – This is used for solving clustering problems that aim in grouping entities subsets with one another depending on similarity.
- `mllib.fpm` – FPM stands for Frequent Pattern Matching. This library is used to mine frequent items, subsequences or other structures that are used for analyzing large datasets.
- `mllib.linalg` – This is used for solving problems on linear algebra.
- `mllib.recommendation` – This is used for collaborative filtering and in recommender systems.
- `spark.mllib` – This is used for supporting model-based collaborative filtering where small latent factors are identified using the Alternating Least Squares (ALS) algorithm which is used for predicting missing entries.
- `mllib.regression` – This is used for solving problems using regression algorithms that find relationships and variable dependencies.

Q. What are the different cluster manager types supported by PySpark?

A cluster manager is a cluster mode platform that helps to run Spark by providing all resources to worker nodes based on the requirements.

The above figure shows the position of cluster manager in the Spark ecosystem. Consider a master node and multiple worker nodes present in the cluster. The master nodes provide the worker nodes with the resources like memory, processor allocation etc depending on the nodes requirements with the help of the cluster manager.

PySpark supports the following cluster manager types:

- Standalone – This is a simple cluster manager that is included with Spark.
- Apache Mesos – This manager can run Hadoop MapReduce and PySpark apps.
- Hadoop YARN – This manager is used in Hadoop2.
- Kubernetes – This is an open-source cluster manager that helps in automated deployment, scaling and automatic management of containerized apps.
- local – This is simply a mode for running Spark applications on laptops/desktops.

Q. What are the advantages of PySpark RDD?
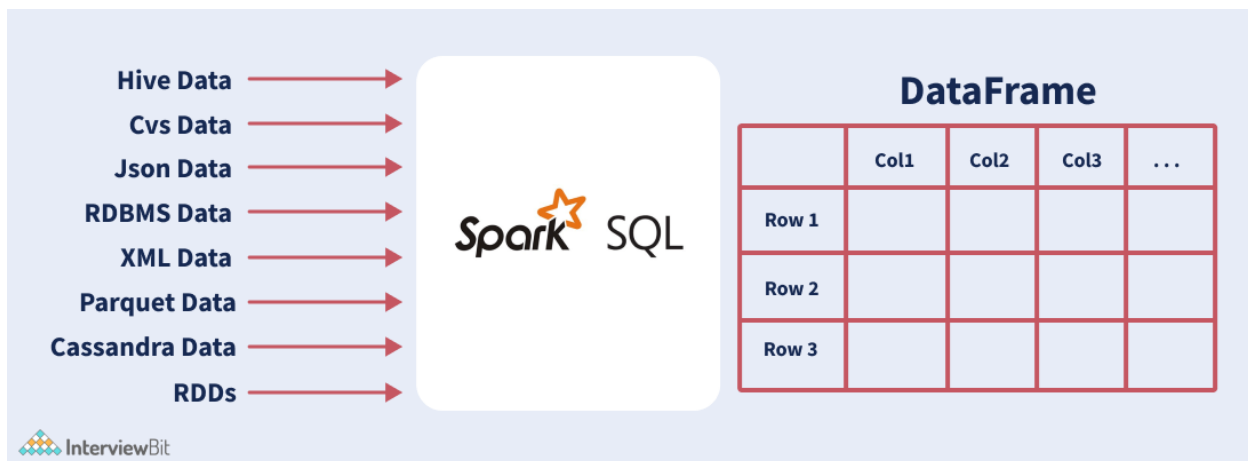
PySpark RDDs have the following advantages:

- In-Memory Processing: PySpark's RDD helps in loading data from the disk to the memory. The RDDs can even be persisted in the memory for reusing the computations.
- Immutability: The RDDs are immutable which means that once created, they cannot be modified. While applying any transformation operations on the RDDs, a new RDD would be created.
- Fault Tolerance: The RDDs are fault-tolerant. This means that whenever an operation fails, the data gets automatically reloaded from other available partitions. This results in seamless execution of the PySpark applications.
- Lazy Evolution: The PySpark transformation operations are not performed as soon as they are encountered. The operations would be stored in the DAG and are evaluated once it finds the first RDD action.
- Partitioning: Whenever RDD is created from any data, the elements in the RDD are partitioned to the cores available by default.

Q. Is PySpark faster than pandas?

PySpark supports parallel execution of statements in a distributed environment, i.e on different cores and different machines which are not present in Pandas. This is why PySpark is faster than pandas.

Q. What do you understand about PySpark DataFrames?
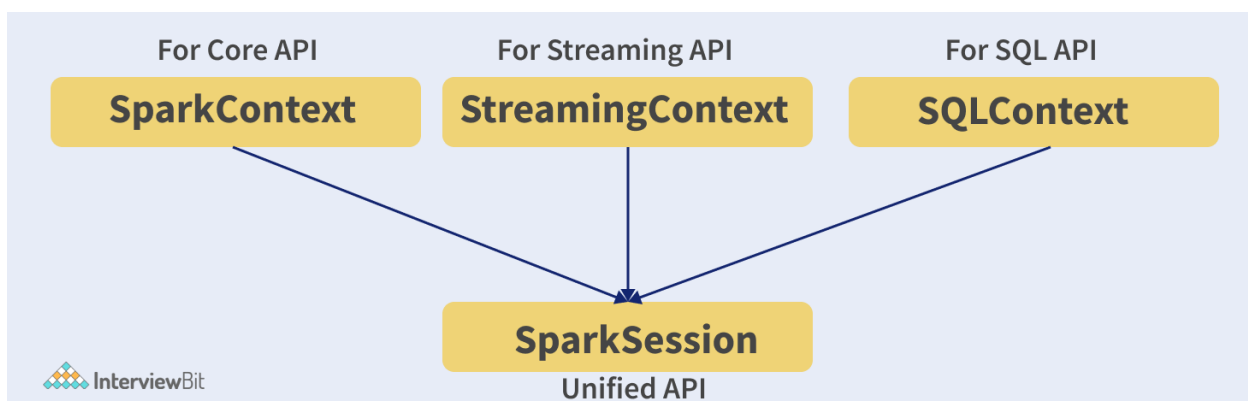
PySpark DataFrame is a distributed collection of well-organized data that is equivalent to tables of the relational databases and are placed into named columns. PySpark DataFrame has better optimisation when compared to R or python. These can be created from different sources like Hive Tables, Structured Data Files, existing RDDs, external databases etc as shown in the image below:

The data in the PySpark DataFrame is distributed across different machines in the cluster and the operations performed on this would be run parallelly on all the machines. These can handle a large collection of structured or semi-structured data of a range of petabytes.

Q. What is SparkSession in Pyspark?

SparkSession is the entry point to PySpark and is the replacement of SparkContext since PySpark version 2.0. This acts as a starting point to access all of the PySpark functionalities related to RDDs, DataFrame, Datasets etc. It is also a Unified API that is used in replacing the SQLContext, StreamingContext, HiveContext and all other contexts.

The SparkSession internally creates SparkContext and SparkConfig based on the details provided in SparkSession. SparkSession can be created by making use of builder patterns.

Q. What are the types of PySpark's shared variables and why are they useful?

Whenever PySpark performs the transformation operation using filter(), map() or reduce(), they are run on a remote node that uses the variables shipped with tasks. These variables are not reusable and cannot be shared across different tasks because they are not returned to the Driver. To solve the issue of reusability and sharing, we have shared variables in PySpark. There are two types of shared variables, they are:

Broadcast variables: These are also known as read-only shared variables and are used in cases of data lookup requirements. These variables are cached and are made available on all the cluster nodes so that the tasks can make use of them. The variables are not sent with every task. They are rather distributed to the nodes using efficient algorithms for reducing the cost of communication. When we run an RDD job operation that makes use of Broadcast variables, the following things are done by PySpark:

- The job is broken into different stages having distributed shuffling. The actions are executed in those stages.
- The stages are then broken into tasks.
- The broadcast variables are broadcasted to the tasks if the tasks need to use it.

Broadcast variables are created in PySpark by making use of the broadcast(variable) method from the SparkContext class. The syntax for this goes as follows:

```
broadcastVar = sc.broadcast([10, 11, 22, 31])
broadcastVar.value    # access broadcast variable
```

An important point of using broadcast variables is that the variables are not sent to the tasks when the broadcast function is called. They will be sent when the variables are first required by the executors.

Accumulator variables: These variables are called updatable shared variables. They are added through associative and commutative operations and are used for performing counter or sum operations. PySpark supports the creation of numeric type accumulators by default. It also has the ability to add custom accumulator types. The custom types can be of two types:

- Named Accumulators: These accumulators are visible under the "Accumulator" tab in the PySpark web UI as shown in the image below:

Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

InterviewBit

Here, we will see the Accumulable section that has the sum of the Accumulator values of the variables modified by the tasks listed in the Accumulator column present in the Tasks table.

- Unnamed Accumulators: These accumulators are not shown on the PySpark Web UI page. It is always recommended to make use of named accumulators.

Accumulator variables can be created by using SparkContext.longAccumulator(variable) as shown in the example below:

```
ac = sc.longAccumulator("sumaccumulator")
sc.parallelize([2, 23, 1]).foreach(lambda x: ac.add(x))
```

Depending on the type of accumulator variable data - double, long and collection, PySpark provide DoubleAccumulator, LongAccumulator and CollectionAccumulator respectively.

Q. What is PySpark UDF?

UDF stands for User Defined Functions. In PySpark, UDF can be created by creating a python function and wrapping it with PySpark SQL's udf() method and using it on the DataFrame or SQL. These are generally created when we do not have the functionalities supported in PySpark's library and we have to use our own logic on the data. UDFs can be reused on any number of SQL expressions or DataFrames.

Q. What are the industrial benefits of PySpark?

These days, almost every industry makes use of big data to evaluate where they stand and grow. When you hear the term big data, Apache Spark comes to mind. Following are the industry benefits of using PySpark that supports Spark:

- Media streaming: Spark can be used to achieve real-time streaming to provide personalized recommendations to subscribers. Netflix is one such example that uses Apache Spark. It processes around 450 billion events every day to flow to its server-side apps.
- Finance: Banks use Spark for accessing and analyzing the social media profiles and in turn get insights on what strategies would help them to make the right decisions regarding customer segmentation, credit risk assessments, early fraud detection etc.

- Healthcare: Providers use Spark for analyzing the past records of the patients to identify what health issues the patients might face posting their discharge. Spark is also used to perform genome sequencing for reducing the time required for processing genome data.
- Travel Industry: Companies like TripAdvisor uses Spark to help users plan the perfect trip and provide personalized recommendations to the travel enthusiasts by comparing data and review from hundreds of websites regarding the place, hotels, etc.
- Retail and e-commerce: This is one important industry domain that requires big data analysis for targeted advertising. Companies like Alibaba run Spark jobs for analyzing petabytes of data for enhancing customer experience, providing targetted offers, sales and optimizing the overall performance.

Pyspark Interview Questions for Experienced

Q. What is PySpark Architecture?

PySpark similar to Apache Spark works in master-slave architecture pattern. Here, the master node is called the Driver and the slave nodes are called the workers. When a Spark application is run, the Spark Driver creates SparkContext which acts as an entry point to the spark application. All the operations are executed on the worker nodes. The resources required for executing the operations on the worker nodes are managed by the Cluster Managers. The following diagram illustrates the architecture described:

Q. What PySpark DAGScheduler?

DAG stands for Direct Acyclic Graph. DAGScheduler constitutes the scheduling layer of Spark which implements scheduling of tasks in a stage-oriented manner using jobs and stages. The logical execution plan (Dependencies lineage of transformation actions upon RDDs) is transformed into a physical execution plan consisting of stages. It computes a DAG of stages needed for each job and keeps track of what stages are RDDs are materialized and finds a minimal schedule for running the jobs. These stages are then submitted to TaskScheduler for running the stages. This is represented in the image flow below:

DAGScheduler performs the following three things in Spark:

- Compute DAG execution for the job.
- Determine preferred locations for running each task
- Failure Handling due to output files lost during shuffling.

PySpark's DAGScheduler follows event-queue architecture. Here a thread posts events of type DAGSchedulerEvent such as new stage or job. The DAGScheduler then reads the stages and sequentially executes them in topological order.

Q. What is the common workflow of a spark program?

The most common workflow followed by the spark program is:

- The first step is to create input RDDs depending on the external data. Data can be obtained from different data sources.
- Post RDD creation, the RDD transformation operations like filter() or map() are run for creating new RDDs depending on the business logic.
- If any intermediate RDDs are required to be reused for later purposes, we can persist those RDDs.
- Lastly, if any action operations like first(), count() etc are present then spark launches it to initiate parallel computation.

Q. Why is PySpark SparkConf used?

PySpark SparkConf is used for setting the configurations and parameters required to run applications on a cluster or local system. The following class can be executed to run the SparkConf:

```
class pyspark.Sparkconf(
localdefaults = True,
_jvm = None,
_jconf = None
)
```

where:

- `loadDefaults` - is of type boolean and indicates whether we require loading values from Java System Properties. It is True by default.
- `_jvm` - This belongs to the class `py4j.java_gateway.JVMView` and is an internal parameter that is used for passing the handle to JVM. This need not be set by the users.
- `_jconf` - This belongs to the class `py4j.java_gateway.JavaObject`. This parameter is an option and can be used for passing existing SparkConf handles for using the parameters.

Q. How will you create PySpark UDF?

Consider an example where we want to capitalize the first letter of every word in a string. This feature is not supported in PySpark. We can however achieve this by creating a UDF `capitalizeWord(str)` and using it on the DataFrames. The following steps demonstrate this:

- Create Python function `capitalizeWord` that takes a string as input and capitalizes the first character of every word.

```
def capitalizeWord(str):
    result=""
    words = str.split(" ")
    for word in words:
        result= result + word[0:1].upper() + word[1:len(x)] + " "
    return result
```

- Register the function as a PySpark UDF by using the udf() method of `org.apache.spark.sql.functions.udf` package which needs to be imported. This method returns the object of class `org.apache.spark.sql.expressions.UserDefinedFunction`.

```
""" Converting function to UDF """
capitalizeWordUDF = udf(lambda z: capitalizeWord(z),StringType())
```

- Use UDF with DataFrame: The UDF can be applied on a Python DataFrame as that acts as the built-in function of DataFrame.

  Consider we have a DataFrame of stored in variable df as below:

```
+----------+------------------+
|ID_COLUMN |NAME_COLUMN       |
+----------+------------------+
|1         |harry potter      |
|2         |ronald weasley    |
|3         |hermoine granger  |
+----------+------------------+
```

To capitalize every first character of the word, we can use:

```
df.select(col("ID_COLUMN"), convertUDF(col("NAME_COLUMN"))
```

```
    .alias("NAME_COLUMN") )
    .show(truncate=False)
```

The output of the above code would be:

```
+----------+------------------+
|ID_COLUMN |NAME_COLUMN       |
+----------+------------------+
|1         |Harry Potter      |
|2         |Ronald Weasley    |
|3         |Hermoine Granger  |
+----------+------------------+
```

UDFs have to be designed in a way that the algorithms are efficient and take less time and space complexity. If care is not taken, the performance of the DataFrame operations would be impacted.

Q. What are the profilers in PySpark?

Custom profilers are supported in PySpark. These are useful for building predictive models. Profilers are useful for data review to ensure that it is valid and can be used for consumption. When we require a custom profiler, it has to define some of the following methods:

- profile: This produces a system profile of some sort.
- stats: This returns collected stats of profiling.
- dump: This dumps the profiles to a specified path.
- add: This helps to add profile to existing accumulated profile. The profile class has to be selected at the time of SparkContext creation.
- dump(id, path): This dumps a specific RDD id to the path given.

Q. How to create SparkSession?

To create SparkSession, we use the builder pattern. The SparkSession class from the `pyspark.sql` library has the getOrCreate() method which creates a new SparkSession

if there is none or else it returns the existing SparkSession object. The following code is an example for creating SparkSession:

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[1]")
                    .appName('InterviewBitSparkSession')
                    .getOrCreate()
```

Here,

- master() – This is used for setting up the mode in which the application has to run - cluster mode (use the master name) or standalone mode. For Standalone mode, we use the `local[x]` value to the function, where x represents partition count to be created in RDD, DataFrame and DataSet. The value of x is ideally the number of CPU cores available.
- appName() - Used for setting the application name
- getOrCreate() – For returning SparkSession object. This creates a new object if it does not exist. If an object is there, it simply returns that.

If we want to create a new SparkSession object every time, we can use the newSession method as shown below:

```
import pyspark
from pyspark.sql import SparkSession
spark_session = SparkSession.newSession
```

Q. What are the different approaches for creating RDD in PySpark?

The following image represents how we can visualize RDD creation in PySpark:

RDD Creation

Python list data

RDD

In the image, we see that the data we have is the list form and post converting to RDDs, we have it stored in different partitions.

We have the following approaches for creating PySpark RDD:

- Using `sparkContext.parallelize()`: The parallelize() method of the SparkContext can be used for creating RDDs. This method loads existing collection from the driver and parallelizes it. This is a basic approach to create RDD and is used when we have data already present in the memory. This also requires the presence of all data on the Driver before creating RDD. Code to create RDD using the parallelize method for the python list shown in the image above:

```
list = [1,2,3,4,5,6,7,8,9,10,11,12]
rdd=spark.sparkContext.parallelize(list)
```

- Using `sparkContext.textFile()`: Using this method, we can read .txt file and convert them into RDD. Syntax:

```
rdd_txt = spark.sparkContext.textFile("/path/to/textFile.txt")
```

- Using `sparkContext.wholeTextFiles()`: This function returns PairRDD (RDD containing key-value pairs) with file path being the key and the file content is the value.

```
#Reads entire file into a RDD as single record.
rdd_whole_text =
spark.sparkContext.wholeTextFiles("/path/to/textFile.txt")
```

We can also read csv, json, parquet and various other formats and create the RDDs.

- Empty RDD with no partition using `sparkContext.emptyRDD`: RDD with no data is called empty RDD. We can create such RDDs having no partitions by using emptyRDD() method as shown in the code piece below:

```
empty_rdd = spark.sparkContext.emptyRDD
# to create empty rdd of string type
empty_rdd_string = spark.sparkContext.emptyRDD[String]
```

- Empty RDD with partitions using `sparkContext.parallelize`: When we do not require data but we require partition, then we create empty RDD by using the parallelize method as shown below:

```
#Create empty RDD with 20 partitions
empty_partitioned_rdd = spark.sparkContext.parallelize([],20)
```

Q. How can we create DataFrames in PySpark?

We can do it by making use of the `createDataFrame()` method of the SparkSession.

```
data = [('Harry', 20),
        ('Ron', 20),
        ('Hermoine', 20)]
columns = ["Name","Age"]
df = spark.createDataFrame(data=data, schema = columns)
```

This creates the dataframe as shown below:

```
+-----------+----------+
| Name      | Age      |
```

```
+-----------+----------+
|  Harry    |  20      |
|  Ron      |  20      |
|  Hermoine |  20      |
+-----------+----------+
```

We can get the schema of the dataframe by using `df.printSchema()`

```
>> df.printSchema()
root
|-- Name: string (nullable = true)
|-- Age: integer (nullable = true)
```

Q. Is it possible to create PySpark DataFrame from external data sources?

Yes, it is! Realtime applications make use of external file systems like local, HDFS, HBase, MySQL table, S3 Azure etc. Following example shows how we can create DataFrame by reading data from a csv file present in the local system:

```
df = spark.read.csv("/path/to/file.csv")
```

PySpark supports csv, text, avro, parquet, tsv and many other file extensions.

Q. What do you understand by Pyspark's startsWith() and endsWith() methods?

These methods belong to the Column class and are used for searching DataFrame rows by checking if the column value starts with some value or ends with some value. They are used for filtering data in applications.

- startsWith() – returns boolean Boolean value. It is true when the value of the column starts with the specified string and False when the match is not satisfied in that column value.

- endsWith() – returns boolean Boolean value. It is true when the value of the column ends with the specified string and False when the match is not satisfied in that column value.

Both the methods are case-sensitive.

Consider an example of the startsWith() method here. We have created a DataFrame with 3 rows:

```
data = [('Harry', 20),
        ('Ron', 20),
        ('Hermoine', 20)]
columns = ["Name","Age"]
df = spark.createDataFrame(data=data, schema = columns)
```

If we have the below code that checks for returning the rows where all the names in the Name column start with "H",

```
import org.apache.spark.sql.functions.col
df.filter(col("Name").startsWith("H")).show()
```

The output of the code would be:

```
+-----------+-----------+
|  Name     |  Age      |
+-----------+-----------+
|  Harry    |  20       |
|  Hermoine |  20       |
+-----------+-----------+
```

Notice how the record with the Name "Ron" is filtered out because it does not start with "H".

Q. What is PySpark SQL?

PySpark SQL is the most popular PySpark module that is used to process structured columnar data. Once a DataFrame is created, we can interact with data using the SQL

syntax. Spark SQL is used for bringing native raw SQL queries on Spark by using select, where, group by, join, union etc. For using PySpark SQL, the first step is to create a temporary table on DataFrame by using `createOrReplaceTempView()` function. Post creation, the table is accessible throughout SparkSession by using sql() method. When the SparkSession gets terminated, the temporary table will be dropped.

For example, consider we have the following DataFrame assigned to a variable `df`:

```
+-----------+----------+----------+
|  Name     |  Age     |  Gender  |
+-----------+----------+----------+
|  Harry    |  20      |    M     |
|  Ron      |  20      |    M     |
|  Hermoine |  20      |    F     |
+-----------+----------+----------+
```

In the below piece of code, we will be creating a temporary table of the DataFrame that gets accessible in the SparkSession using the sql() method. The SQL queries can be run within the method.

```
df.createOrReplaceTempView("STUDENTS")
df_new = spark.sql("SELECT * from STUDENTS")
df_new.printSchema()
```

The schema will be displayed as shown below:

```
>> df.printSchema()
root
|-- Name: string (nullable = true)
|-- Age: integer (nullable = true)
|-- Gender: string (nullable = true)
```

For the above example, let's try running group by on the Gender column:

```
groupByGender = spark.sql("SELECT Gender, count(*) as
Gender_Count from STUDENTS group by Gender")
```

```
groupByGender.show()
```

The above statements results in:

```
+------+------------+
|Gender|Gender_Count|
+------+------------+
|     F|           1|
|     M|           2|
+------+------------+
```

Q. How can you inner join two DataFrames?

We can make use of the join() method present in PySpark SQL. The syntax for the function is:

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

other - Right side of the join

on - column name string used for joining

how - type of join, by default it is inner. The values can be inner, left, right, cross, full, outer, left_outer, right_outer, left_anti, left_semi.

The join expression can be appended with where() and filter() methods for filtering rows. We can have multiple join too by means of the chaining join() method.

Consider we have two dataframes - employee and department as shown below:

```
-- Employee DataFrame --
+------+--------+-----------+
```

```
|emp_id|emp_name|empdept_id |
+------+--------+----------+
|     1|   Harry|         5|
|     2|    Ron |         5|
|     3| Neville|        10|
|     4|  Malfoy|        20|
+------+--------+----------+

-- Department DataFrame --
+-------+-------------------------+
|dept_id| dept_name               |
+-------+-------------------------+
|     5 |   Information Technology |
|    10|    Engineering           |
|    20|    Marketting            |
+-------+-------------------------+
```

We can inner join the Employee DataFrame with Department DataFrame to get the department information along with employee information as:

```
emp_dept_df = empDF.join(deptDF,empDF.empdept_id ==
deptDF.dept_id,"inner").show(truncate=False)
```

The result of this becomes:

```
+------+--------+----------+-------+-------------------------+
|emp_id|emp_name|empdept_id |dept_id| dept_name               |
+------+--------+----------+-------+-------------------------+
|     1|   Harry|         5|     5 |   Information Technology |
|     2|    Ron |         5|     5 |   Information Technology |
|     3| Neville|        10|    10 |   Engineering            |
|     4|  Malfoy|        20|    20 |   Marketting             |
+------+--------+----------+-------+-------------------------+
```

We can also perform joins by chaining join() method by following the syntax:

```
dfQ.join(df2,["column_name"]).join(df3,df1["column_name"] ==
df3["column_name"]).show()
```

Consider we have a third dataframe called Address DataFrame having columns emp_id, city and state where emp_id acts as the foreign key equivalent of SQL to the Employee DataFrame as shown below:

```
-- Address DataFrame --
+------+-------------+------+
|emp_id| city        |state |
+------+-------------+------+
|1     | Bangalore   |   KA |
|2     | Pune        |   MH |
|3     | Mumbai      |   MH |
|4     | Chennai     |   TN |
+------+-------------+------+
```

If we want to get address details of the address along with the Employee and the Department Dataframe, then we can run,

```
resultDf = empDF.join(addressDF,["emp_id"])
                .join(deptDF,empDF["empdept_id"] ==
deptDF["dept_id"])
                .show()
```
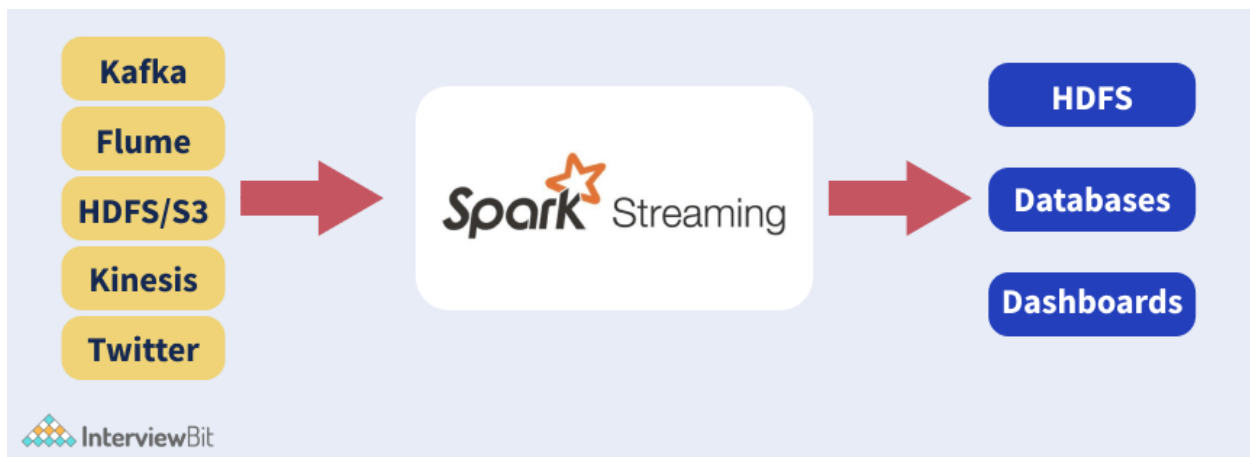
The resultDf would be:

```
+------+--------+----------+-------------+------+-------+-----
--------------------+
|emp_id|emp_name|empdept_id | city        |state |dept_id|
dept_name                    |
+------+--------+----------+-------------+------+-------+-----
--------------------+
|     1|   Harry|          5| Bangalore   |   KA |     5 |
Information Technology |
|     2|    Ron |          5| Pune        |   MH |     5 |
Information Technology |
|     3| Neville|         10| Mumbai      |   MH |    10 |
Engineering            |
|     4|  Malfoy|         20| Chennai     |   TN |    20 |
Marketting             |
```

```
+------+--------+----------+-------------+------+-------+-----
--------------------+
```

Q. What do you understand by Pyspark Streaming? How do you stream data using TCP/IP Protocol?

PySpark Streaming is scalable, fault-tolerant, high throughput based processing streaming system that supports streaming as well as batch loads for supporting real-time data from data sources like TCP Socket, S3, Kafka, Twitter, file system folders etc. The processed data can be sent to live dashboards, Kafka, databases, HDFS etc.

To perform Streaming from the TCP socket, we can use the readStream.format("socket") method of Spark session object for reading data from TCP socket and providing the streaming source host and port as options as shown in the code below:

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.sql import SQLContext
from pyspark.sql.functions import desc
sc = SparkContext()
ssc = StreamingContext(sc, 10)
sqlContext = SQLContext(sc)
socket_stream = ssc.socketTextStream("127.0.0.1", 5555)
lines = socket_stream.window(20)
```

```
df.printSchema()
```

Spark loads the data from the socket and represents it in the value column of the DataFrame object. The `df.printSchema()` prints

```
root
|-- value: string (nullable = true)
```

Post data processing, the DataFrame can be streamed to the console or any other destinations based on the requirements like Kafka, dashboards, database etc.

Q. What would happen if we lose RDD partitions due to the failure of the worker node?

If any RDD partition is lost, then that partition can be recomputed using operations lineage from the original fault-tolerant dataset.