GET Real Databricks Associate-Developer-Apache-Spark Exam Questions With 100% Refund Guarantee Jan 21, 2023 [Q22-Q45

GET Real Databricks Associate-Developer-Apache-Spark Exam Questions With 100% Refund Guarantee Jan 21, 2023
Get Special Discount Offer on Associate-Developer-Apache-Spark Dumps PDF

What is the salary of a Databricks Associate Developer Apache Spark Exam?

The Average salary of different countries for Databricks Associate Developer Apache Spark professionals:
India - INR 3516457 per yearUK - Pounds 33288 per yearUnited States - USD 84,000 per year

Learn more about importance

To make it easy for you to learn and practice Data Science, we have come up with a list of top Data Science courses. These courses are designed by experts who have years of experience in the field and they can provide you with the best possible training. The narrow deployment returns objects cluster and determines defined occurs for the block variable preparation node to answer single blocks column correct memory mode worker error code.
In today's world, there are so many new technologies that come out every day. It's a lot to learn. But, if you want to be successful, you need to make sure that you know what the latest technology is and how to apply it in your work. If you don't know how to do this, you're going to have a very hard time finding a job. And if you do find a job, you're going to have a very hard time staying there. This is because you'll be constantly learning new things and changing your skills and abilities. That's why it's important to make sure that you have the right credentials.

Q22. Which of the following code blocks reads the parquet file stored at filePath into DataFrame itemsDf, using a valid schema for the sample of itemsDf shown below?

Sample of itemsDf:

1.+——+————————–+——————-+

2.|itemId|attributes |supplier |

3.+——+————————–+——————-+

4.|1 |[blue, winter, cozy] |Sports Company Inc.|

5.|2 |[red, summer, fresh, cooling]|YetiX |

6.|3 |[green, summer, travel] |Sports Company Inc.|

7.+——+————————–+——————-+
* 1.itemsDfSchema = StructType([

2. StructField(“itemId”, IntegerType()),

3. StructField(“attributes”, StringType()),

4. StructField(“supplier”, StringType())])

5.

6.itemsDf = spark.read.schema(itemsDfSchema).parquet(filePath)
* 1.itemsDfSchema = StructType([

2. StructField(“itemId”, IntegerType),

3. StructField(“attributes”, ArrayType(StringType)),

4. StructField(“supplier”, StringType)])

5.

6.itemsDf = spark.read.schema(itemsDfSchema).parquet(filePath)
* 1.itemsDf = spark.read.schema(‘itemId integer, attributes <string>, supplier string’).parquet(filePath)
* 1.itemsDfSchema = StructType([

2. StructField(“itemId”, IntegerType()),

3. StructField(“attributes”, ArrayType(StringType())),

4. StructField(“supplier”, StringType())])

5.

6.itemsDf = spark.read.schema(itemsDfSchema).parquet(filePath)
* 1.itemsDfSchema = StructType([

2. StructField(“itemId”, IntegerType()),

3. StructField(“attributes”, ArrayType([StringType()])),

4. StructField(“supplier”, StringType())])

5.

6.itemsDf = spark.read(schema=itemsDfSchema).parquet(filePath)
Explanation

The challenge in this question comes from there being an array variable in the schema. In addition, you should know how to pass a schema to the DataFrameReader that is invoked by spark.read.

The correct way to define an array of strings in a schema is through ArrayType(StringType()). A schema can be passed to the DataFrameReader by simply appending schema(structType) to the read() operator. Alternatively, you can also define a schema as a string. For example, for the schema of itemsDf, the following string would make sense: itemId integer, attributes array<string>,

supplier string.

A thing to keep in mind is that in schema definitions, you always need to instantiate the types, like so:

StringType(). Just using StringType does not work in pySpark and will fail.

Another concern with schemas is whether columns should be nullable, so allowed to have null values. In the case at hand, this is not a concern however, since the question just asks for a

“valid”

schema. Both non-nullable and nullable column schemas would be valid here, since no null value appears in the DataFrame sample.

More info: Learning Spark, 2nd Edition, Chapter 3

Static notebook | Dynamic notebook: See test 3

Q23. The code block displayed below contains an error. The code block is intended to write DataFrame transactionsDf to disk as a parquet file in location /FileStore/transactions_split, using column storeId as key for partitioning. Find the error.

Code block:

transactionsDf.write.format(“parquet”).partitionOn(“storeId”).save(“/FileStore/transactions_split”)A.
* The format(“parquet”) expression is inappropriate to use here, “parquet” should be passed as first argument to the save() operator and “/FileStore/transactions_split” as the second argument.
* Partitioning data by storeId is possible with the partitionBy expression, so partitionOn should be replaced by partitionBy.
* Partitioning data by storeId is possible with the bucketBy expression, so partitionOn should be replaced by bucketBy.
* partitionOn(“storeId”) should be called before the write operation.
* The format(“parquet”) expression should be removed and instead, the information should be added to the write expression like so: write(“parquet”).
Explanation

Correct code block:

transactionsDf.write.format(“parquet”).partitionBy(“storeId”).save(“/FileStore/transactions_split”) More info: partition by – Reading files which are written using PartitionBy or BucketBy in Spark – Stack Overflow Static notebook | Dynamic notebook: See test 1

Q24. The code block shown below should convert up to 5 rows in DataFrame transactionsDf that have the value 25 in column storeId into a Python list. Choose the answer that correctly fills the blanks in the code block to accomplish this.

Code block:

transactionsDf.__1__(__2__).__3__(__4__)
* 1. filter

2. “storeId”==25

3. collect

4. 5
* 1. filter

2. col(“storeId”)==25

3. toLocalIterator

4. 5
* 1. select

2. storeId==25

3. head

4. 5
* 1. filter

2. col(“storeId”)==25

3. take

4. 5
* 1. filter

2. col(“storeId”)==25

3. collect

4. 5
Explanation

The correct code block is:

transactionsDf.filter(col(“storeId”)==25).take(5)

Any of the options with collect will not work because collect does not take any arguments, and in both cases the argument 5 is given.

The option with toLocalIterator will not work because the only argument to toLocalIterator is prefetchPartitions which is a boolean, so passing 5 here does not make sense.

The option using head will not work because the expression passed to select is not proper syntax. It would work if the expression would be col(“storeId”)==25.

Static notebook | Dynamic notebook: See test 1

(https://flrs.github.io/spark_practice_tests_code/#1/24.html ,

https://bit.ly/sparkpracticeexams_import_instructions)

Q25. The code block shown below should return a DataFrame with columns transactionsId, predError, value, and f from DataFrame transactionsDf. Choose the answer that correctly fills the blanks in the code block to accomplish this.

transactionsDf.__1__(__2__)
* 1. filter

2. “transactionId”, “predError”, “value”, “f”
* 1. select

2. “transactionId, predError, value, f”
* 1. select

2. [“transactionId”, “predError”, “value”, “f”]
* 1. where

2. col(“transactionId”), col(“predError”), col(“value”), col(“f”)
* 1. select

2. col([“transactionId”, “predError”, “value”, “f”])
Explanation

Correct code block:

transactionsDf.select([“transactionId”, “predError”, “value”, “f”])

The DataFrame.select returns specific columns from the DataFrame and accepts a list as its only argument.

Thus, this is the correct choice here. The option using col([“transactionId”, “predError”,

“value”, “f”]) is invalid, since inside col(), one can only pass a single column name, not a list. Likewise, all columns being specified in a single string like “transactionId, predError, value, f” is not valid syntax.

filter and where filter rows based on conditions, they do not control which columns to return.

Static notebook | Dynamic notebook: See test 2

Q26. Which of the following code blocks returns a single-row DataFrame that only has a column corr which shows the Pearson correlation coefficient between columns predError and value in DataFrame transactionsDf?
* transactionsDf.select(corr([“predError”, “value”]).alias(“corr”)).first()
* transactionsDf.select(corr(col(“predError”), col(“value”)).alias(“corr”)).first()
* transactionsDf.select(corr(predError, value).alias(“corr”))
* transactionsDf.select(corr(col(“predError”), col(“value”)).alias(“corr”)) (Correct)
* transactionsDf.select(corr(“predError”, “value”))
Explanation

In difficulty, this question is above what you can expect from the exam. What this question NO:

wants to teach you, however, is to pay attention to the useful details included in the documentation.

pyspark.sql.corr is not a very common method, but it deals with Spark’s data structure in an interesting way.

The command takes two columns over multiple rows and returns a single row – similar to an aggregation function. When examining the documentation (linked below), you will find this code example:

a = range(20)

b = [2 * x for x in range(20)]

df = spark.createDataFrame(zip(a, b), [“a”, “b”])

df.agg(corr(“a”, “b”).alias(‘c’)).collect()

[Row(c=1.0)]

See how corr just returns a single row? Once you understand this, you should be suspicious about answers that include first(), since there is no need to just select a single row. A reason to eliminate those answers is that DataFrame.first() returns an object of type Row, but not DataFrame, as requested in the question.

transactionsDf.select(corr(col(“predError”), col(“value”)).alias(“corr”)) Correct! After calculating the Pearson correlation coefficient, the resulting column is correctly renamed to corr.

transactionsDf.select(corr(predError, value).alias(“corr”))

No. In this answer, Python will interpret column names predError and value as variable names.

transactionsDf.select(corr(col(“predError”), col(“value”)).alias(“corr”)).first() Incorrect. first() returns a row, not a DataFrame (see above and linked documentation below).

transactionsDf.select(corr(“predError”, “value”))

Wrong. Whie this statement returns a DataFrame in the desired shape, the column will have the name corr(predError, value) and not corr.

transactionsDf.select(corr([“predError”, “value”]).alias(“corr”)).first() False. In addition to first() returning a row, this code block also uses the wrong call structure for command corr which takes two arguments (the two columns to correlate).

More info:

– pyspark.sql.functions.corr – PySpark 3.1.2 documentation

– pyspark.sql.DataFrame.first – PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 3

Q27. The code block displayed below contains an error. The code block should read the csv file located at path data/transactions.csv into DataFrame transactionsDf, using the first row as column header and casting the columns in the most appropriate type. Find the error.

First 3 rows of transactions.csv:

1.transactionId;storeId;productId;name

2.1;23;12;green grass

3.2;35;31;yellow sun

4.3;23;12;green grass

Code block:

transactionsDf = spark.read.load(“data/transactions.csv”, sep=”;”, format=”csv”, header=True)
* The DataFrameReader is not accessed correctly.
* The transaction is evaluated lazily, so no file will be read.
* Spark is unable to understand the file type.
* The code block is unable to capture all columns.
* The resulting DataFrame will not have the appropriate schema.
Explanation

Correct code block:

transactionsDf = spark.read.load(“data/transactions.csv”, sep=”;”, format=”csv”, header=True, inferSchema=True) By default, Spark does not infer the schema of the CSV (since this usually takes some time). So, you need to add the inferSchema=True option to the code block.

More info: pyspark.sql.DataFrameReader.csv – PySpark 3.1.2 documentation

Q28. The code block displayed below contains an error. The code block should return DataFrame transactionsDf, but with the column storeId renamed to storeNumber. Find the error.

Code block:

transactionsDf.withColumn(“storeNumber”, “storeId”)
* Instead of withColumn, the withColumnRenamed method should be used.
* Arguments “storeNumber” and “storeId” each need to be wrapped in a col() operator.
* Argument “storeId” should be the first and argument “storeNumber” should be the second argument to the withColumn method.
* The withColumn operator should be replaced with the copyDataFrame operator.
* Instead of withColumn, the withColumnRenamed method should be used and argument “storeId” should be the first and argument “storeNumber” should be the second argument to that method.
Explanation

Correct code block:

transactionsDf.withColumnRenamed(“storeId”, “storeNumber”)

More info: pyspark.sql.DataFrame.withColumnRenamed – PySpark 3.1.1 documentation Static notebook | Dynamic notebook: See test 1

Q29. The code block shown below should return a new 2-column DataFrame that shows one attribute from column attributes per row next to the associated itemName, for all suppliers in column supplier whose name includes Sports. Choose the answer that correctly fills the blanks in the code block to accomplish this.

Sample of DataFrame itemsDf:

1.+——+——————————-+————————–+——————-+

2.|itemId|itemName |attributes |supplier |

3.+——+——————————-+————————–+——————-+

4.|1 |Thick Coat for Walking in the Snow|[blue, winter, cozy] |Sports Company Inc.|

5.|2 |Elegant Outdoors Summer Dress |[red, summer, fresh, cooling]|YetiX |

6.|3 |Outdoors Backpack |[green, summer, travel] |Sports Company Inc.|

7.+——+——————————-+————————–+——————-+ Code block:

itemsDf.__1__(__2__).select(__3__, __4__)
* 1. filter

2. col(“supplier”).isin(“Sports”)

3. “itemName”

4. explode(col(“attributes”))
* 1. where

2. col(“supplier”).contains(“Sports”)

3. “itemName”

4. “attributes”
* 1. where

2. col(supplier).contains(“Sports”)

3. explode(attributes)

4. itemName
* 1. where

2. “Sports”.isin(col(“Supplier”))

3. “itemName”

4. array_explode(“attributes”)
* 1. filter

2. col(“supplier”).contains(“Sports”)

3. “itemName”

4. explode(“attributes”)
Explanation

Output of correct code block:

+——————————-+——+

|itemName |col |

+——————————-+——+

|Thick Coat for Walking in the Snow|blue |

|Thick Coat for Walking in the Snow|winter|

|Thick Coat for Walking in the Snow|cozy |

|Outdoors Backpack |green |

|Outdoors Backpack |summer|

|Outdoors Backpack |travel|

+——————————-+——+

The key to solving this question is knowing about Spark’s explode operator. Using this operator, you can extract values from arrays into single rows. The following guidance steps through the answers systematically from the first to the last gap. Note that there are many ways to solving the gap questions and filtering out wrong answers, you do not always have to start filtering out from the first gap, but can also exclude some answers based on obvious problems you see with them.

The answers to the first gap present you with two options: filter and where. These two are actually synonyms in PySpark, so using either of those is fine. The answer options to this gap therefore do not help us in selecting the right answer.

The second gap is more interesting. One answer option includes “Sports”.isin(col(“Supplier”)). This construct does not work, since Python’s string does not have an isin method. Another option contains col(supplier). Here, Python will try to interpret supplier as a variable. We have not set this variable, so this is not a viable answer. Then, you are left with answers options that include col (“supplier”).contains(“Sports”) and col(“supplier”).isin(“Sports”). The question states that we are looking for suppliers whose name includes Sports, so we have to go for the contains operator here.

We would use the isin operator if we wanted to filter out for supplier names that match any entries in a list of supplier names.

Finally, we are left with two answers that fill the third gap both with “itemName” and the fourth gap either with explode(“attributes”) or “attributes”. While both are correct Spark syntax, only explode (“attributes”) will help us achieve our goal. Specifically, the question asks for one attribute from column attributes per row – this is what the explode() operator does.

One answer option also includes array_explode() which is not a valid operator in PySpark.

More info: pyspark.sql.functions.explode – PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 3

Q30. Which of the following describes a way for resizing a DataFrame from 16 to 8 partitions in the most efficient way?
* Use operation DataFrame.repartition(8) to shuffle the DataFrame and reduce the number of partitions.
* Use operation DataFrame.coalesce(8) to fully shuffle the DataFrame and reduce the number of partitions.
* Use a narrow transformation to reduce the number of partitions.
* Use a wide transformation to reduce the number of partitions.

Use operation DataFrame.coalesce(0.5) to halve the number of partitions in the DataFrame.
Explanation

Use a narrow transformation to reduce the number of partitions.

Correct! DataFrame.coalesce(n) is a narrow transformation, and in fact the most efficient way to resize the DataFrame of all options listed. One would run DataFrame.coalesce(8) to resize the DataFrame.

Use operation DataFrame.coalesce(8) to fully shuffle the DataFrame and reduce the number of partitions.

Wrong. The coalesce operation avoids a full shuffle, but will shuffle data if needed. This answer is incorrect because it says “fully shuffle” – this is something the coalesce operation will not do. As a general rule, it will reduce the number of partitions with the very least movement of data possible. More info:

distributed computing – Spark – repartition() vs coalesce() – Stack Overflow Use operation DataFrame.coalesce(0.5) to halve the number of partitions in the DataFrame.

Incorrect, since the num_partitions parameter needs to be an integer number defining the exact number of partitions desired after the operation. More info: pyspark.sql.DataFrame.coalesce – PySpark 3.1.2 documentation Use operation DataFrame.repartition(8) to shuffle the DataFrame and reduce the number of partitions.

No. The repartition operation will fully shuffle the DataFrame. This is not the most efficient way of reducing the number of partitions of all listed options.

Use a wide transformation to reduce the number of partitions.

No. While possible via the DataFrame.repartition(n) command, the resulting full shuffle is not the most efficient way of reducing the number of partitions.

Q31. Which of the following code blocks efficiently converts DataFrame transactionsDf from 12 into 24 partitions?

* transactionsDf.repartition(24, boost=True)
* transactionsDf.repartition()
* transactionsDf.repartition(“itemId”, 24)
* transactionsDf.coalesce(24)
* transactionsDf.repartition(24)

Explanation

transactionsDf.coalesce(24)

No, the coalesce() method can only reduce, but not increase the number of partitions.

transactionsDf.repartition()

No, repartition() requires a numPartitions argument.

transactionsDf.repartition(“itemId”, 24)

No, here the cols and numPartitions argument have been mixed up. If the code block would be transactionsDf.repartition(24, “itemId”), this would be a valid solution.

transactionsDf.repartition(24, boost=True)

No, there is no boost argument in the repartition() method.

Q32. Which of the following statements about executors is correct?

* Executors are launched by the driver.
* Executors stop upon application completion by default.
* Each node hosts a single executor.
* Executors store data in memory only.
* An executor can serve multiple applications.

Explanation

Executors stop upon application completion by default.

Correct. Executors only persist during the lifetime of an application.

A notable exception to that is when Dynamic Resource Allocation is enabled (which it is not by default). With Dynamic Resource Allocation enabled, executors are terminated when they are idle, independent of whether the application has been completed or not.

An executor can serve multiple applications.

Wrong. An executor is always specific to the application. It is terminated when the application completes (exception see above).

Each node hosts a single executor.

No. Each node can host one or more executors.

Executors store data in memory only.

No. Executors can store data in memory or on disk.

Executors are launched by the driver.

Incorrect. Executors are launched by the cluster manager on behalf of the driver.

More info: Job Scheduling – Spark 3.1.2 Documentation, How Applications are Executed on a Spark Cluster | Anatomy of a Spark Application | InformIT, and Spark Jargon for Starters. This blog is to clear some of the… | by Mageswaran D | Medium

Q33. Which of the following is a viable way to improve Spark’s performance when dealing with large amounts of data, given that there is only a single application running on the cluster?
* Increase values for the properties spark.default.parallelism and spark.sql.shuffle.partitions
* Decrease values for the properties spark.default.parallelism and spark.sql.partitions
* Increase values for the properties spark.sql.parallelism and spark.sql.partitions
* Increase values for the properties spark.sql.parallelism and spark.sql.shuffle.partitions
* Increase values for the properties spark.dynamicAllocation.maxExecutors, spark.default.parallelism, and spark.sql.shuffle.partitions

Explanation

Decrease values for the properties spark.default.parallelism and spark.sql.partitions No, these values need to be increased.

Increase values for the properties spark.sql.parallelism and spark.sql.partitions Wrong, there is no property spark.sql.parallelism.

Increase values for the properties spark.sql.parallelism and spark.sql.shuffle.partitions See above.

Increase values for the properties spark.dynamicAllocation.maxExecutors, spark.default.parallelism, and spark.sql.shuffle.partitions The property spark.dynamicAllocation.maxExecutors is only in effect if dynamic allocation is enabled, using the spark.dynamicAllocation.enabled property. It is disabled by default. Dynamic allocation can be useful when to run multiple applications on the same cluster in parallel. However, in this case there is only a single application running on the cluster, so enabling dynamic allocation would not yield a performance benefit.

More info: Practical Spark Tips For Data Scientists | Experfy.com and Basics of Apache Spark Configuration Settings | by Halil Ertan | Towards Data Science (https://bit.ly/3gA0A6w ,

https://bit.ly/2QxhNTr)

Q34. The code block shown below should write DataFrame transactionsDf to disk at path csvPath as a single CSV file, using tabs (t characters) as separators between columns, expressing missing values as string n/a, and omitting a header row with column names. Choose the answer that correctly fills the blanks in the code block to accomplish this.

transactionsDf.__1__.write.__2__(__3__, ” “).__4__.__5__(csvPath)
* 1. coalesce(1)

2. option

3. “sep”

4. option(“header”, True)

5. path
* 1. coalesce(1)

2. option

3. “colsep”

4. option(“nullValue”, “n/a”)

5. path
* 1. repartition(1)

2. option

3. “sep”

4. option(“nullValue”, “n/a”)

5. csv

(Correct)
* 1. csv

2. option

3. “sep”

4. option(“emptyValue”, “n/a”)

5. path

* 1. repartition(1)

2. mode

3. “sep”

4. mode(“nullValue”, “n/a”)

5. csv
Explanation

Correct code block:

transactionsDf.repartition(1).write.option(“sep”, “t”).option(“nullValue”, “n/a”).csv(csvPath) It is important here to understand that the question specifically asks for writing the DataFrame as a single CSV file. This should trigger you to think about partitions. By default, every partition is written as a separate file, so you need to include repatition(1) into your call. coalesce(1) works here, too!

Secondly, the question is very much an invitation to search through the parameters in the Spark documentation that work with DataFrameWriter.csv (link below). You will also need to know that you need an option() statement to apply these parameters.

The final concern is about the general call structure. Once you have called accessed write of your DataFrame, options follow and then you write the DataFrame with csv. Instead of csv(csvPath), you could also use save(csvPath, format=’csv’) here.

More info: pyspark.sql.DataFrameWriter.csv – PySpark 3.1.1 documentation Static notebook | Dynamic notebook: See test 1

Q35. Which of the elements that are labeled with a circle and a number contain an error or are misrepresented?
* 1, 10
* 1, 8
* 10
* 7, 9, 10
* 1, 4, 6, 9

Explanation

1: Correct – This should just read “API” or “DataFrame API”. The DataFrame is not part of the SQL API. To make a DataFrame accessible via SQL, you first need to create a DataFrame view. That view can then be accessed via SQL.

4: Although “K_38_INU” looks odd, it is a completely valid name for a DataFrame column.

6: No, StringType is a correct type.

7: Although a StringType may not be the most efficient way to store a phone number, there is nothing fundamentally wrong with using this type here.

8: Correct – TreeType is not a type that Spark supports.

9: No, Spark DataFrames support ArrayType variables. In this case, the variable would represent a sequence of elements with type LongType, which is also a valid type for Spark DataFrames.

10: There is nothing wrong with this row.

More info: Data Types – Spark 3.1.1 Documentation (https://bit.ly/3aAPKJT)

Q36. The code block shown below should return a copy of DataFrame transactionsDf without columns value and productId and with an additional column associateId that has the value 5. Choose the answer that correctly fills the blanks in the code block to accomplish this.

transactionsDf.__1__(__2__, __3__).__4__(__5__, ‘value’)
* 1. withColumn

2. ‘associateId’

3. 5

4. remove

5. ‘productId’
* 1. withNewColumn

2. associateId

3. lit(5)

4. drop

5. productId
* 1. withColumn

2. ‘associateId’

3. lit(5)

4. drop

5. ‘productId’
* 1. withColumnRenamed

2. ‘associateId’

3. 5

4. drop

5. ‘productId’
* 1. withColumn

2. col(associateId)

3. lit(5)

4. drop

5. col(productId)
Explanation

Correct code block:

transactionsDf.withColumn(‘associateId’, lit(5)).drop(‘productId’, ‘value’) For solving this question it is important that you know the lit() function (link to documentation below). This function enables you to add a column of a constant value to a DataFrame.

More info: pyspark.sql.functions.lit – PySpark 3.1.1 documentation

Static notebook | Dynamic notebook: See test 1

Q37. The code block shown below should read all files with the file ending .png in directory path into Spark.

Choose the answer that correctly fills the blanks in the code block to accomplish this.

spark.__1__.__2__(__3__).option(__4__, “*.png”).__5__(path)
* 1. read()

2. format

3. “binaryFile”

4. “recursiveFileLookup”

5. load
* 1. read

2. format

3. “binaryFile”

4. “pathGlobFilter”

5. load
* 1. read

2. format

3. binaryFile

4. pathGlobFilter

5. load
* 1. open

2. format

3. “image”

4. “fileType”

5. open
* 1. open

2. as

3. “binaryFile”

4. “pathGlobFilter”

5. load
Explanation

Correct code block:

spark.read.format(“binaryFile”).option(“recursiveFileLookup”, “*.png”).load(path) Spark can deal with binary files, like images. Using the binaryFile format specification in the SparkSession’s read API is the way to read in those files. Remember that, to access the read API, you need to start the command with spark.read. The pathGlobFilter option is a great way to filter files by name (and ending). Finally, the path can be specified using the load operator – the open operator shown in one of the answers does not exist.

Q38. Which of the following describes properties of a shuffle?
* Operations involving shuffles are never evaluated lazily.
* Shuffles involve only single partitions.
* Shuffles belong to a class known as “full transformations”.
* A shuffle is one of many actions in Spark.
* In a shuffle, Spark writes data to disk.

Explanation

In a shuffle, Spark writes data to disk.

Correct! Spark’s architecture dictates that intermediate results during a shuffle are written to disk.

A shuffle is one of many actions in Spark.

Incorrect. A shuffle is a transformation, but not an action.

Shuffles involve only single partitions.

No, shuffles involve multiple partitions. During a shuffle, Spark generates output partitions from multiple input partitions.

Operations involving shuffles are never evaluated lazily.

Wrong. A shuffle is a costly operation and Spark will evaluate it as lazily as other transformations. This is, until a subsequent action triggers its evaluation.

Shuffles belong to a class known as “full transformations”.

Not quite. Shuffles belong to a class known as “wide transformations”. “Full transformation” is not a relevant term in Spark.

More info: Spark – The Definitive Guide, Chapter 2 and Spark: disk I/O on stage boundaries explanation – Stack Overflow

Q39. The code block shown below should return only the average prediction error (column predError) of a random subset, without replacement, of approximately 15% of rows in DataFrame transactionsDf. Choose the answer that correctly fills the blanks in the code block to accomplish this.

transactionsDf.__1__(__2__, __3__).__4__(avg(‘predError’))
* 1. sample

2. True

3. 0.15

4. filter
* 1. sample

2. False

3. 0.15

4. select
* 1. sample

2. 0.85

3. False

4. select
* 1. fraction

2. 0.15

3. True

4. where
* 1. fraction

2. False

3. 0.85

4. select
Explanation

Correct code block:

transactionsDf.sample(withReplacement=False, fraction=0.15).select(avg(‘predError’)) You should remember that getting a random subset of rows means sampling. This, in turn should point you to the DataFrame.sample() method. Once you know this, you can look up the correct order of arguments in the documentation (link below).

Lastly, you have to decide whether to use filter, where or select. where is just an alias for filter(). filter() is not the correct method to use here, since it would only allow you to filter rows based on some condition. However, the question asks to return only the average prediction error. You can control the columns that a query returns with the select() method – so this is the correct method to use here.

More info: pyspark.sql.DataFrame.sample – PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 2

Q40. Which of the following code blocks immediately removes the previously cached DataFrame transactionsDf from memory and disk?
* array_remove(transactionsDf, “*”)
* transactionsDf.unpersist()

(Correct)
* del transactionsDf
* transactionsDf.clearCache()
* transactionsDf.persist()
Explanation

transactionsDf.unpersist()

Correct. The DataFrame.unpersist() command does exactly what the question asks for – it removes all cached parts of the DataFrame from memory and disk.

del transactionsDf

False. While this option can help remove the DataFrame from memory and disk, it does not do so immediately. The reason is that this command just notifies the Python garbage collector that the transactionsDf now may be deleted from memory. However, the garbage collector does not do so immediately and, if you wanted it to run immediately, would need to be specifically triggered to do so. Find more information linked below.

array_remove(transactionsDf, “*”)

Incorrect. The array_remove method from pyspark.sql.functions is used for removing elements from arrays in columns that match a specific condition. Also, the first argument would be a column, and not a DataFrame as shown in the code block.

transactionsDf.persist()

No. This code block does exactly the opposite of what is asked for: It caches (writes) DataFrame transactionsDf to memory and disk. Note that even though you do not pass in a specific storage level here, Spark will use the default storage level (MEMORY_AND_DISK).

transactionsDf.clearCache()

Wrong. Spark’s DataFrame does not have a clearCache() method.

More info: pyspark.sql.DataFrame.unpersist – PySpark 3.1.2 documentation, python – How to delete an RDD in PySpark for the purpose of releasing resources? – Stack Overflow Static notebook | Dynamic notebook: See test 3

Q41. Which of the following code blocks displays various aggregated statistics of all columns in DataFrame transactionsDf, including the standard deviation and minimum of values in each column?
* transactionsDf.summary()
* transactionsDf.agg(“count”, “mean”, “stddev”, “25%”, “50%”, “75%”, “min”)
* transactionsDf.summary(“count”, “mean”, “stddev”, “25%”, “50%”, “75%”, “max”).show()
* transactionsDf.agg(“count”, “mean”, “stddev”, “25%”, “50%”, “75%”, “min”).show()
* transactionsDf.summary().show()
Explanation

The DataFrame.summary() command is very practical for quickly calculating statistics of a DataFrame. You need to call .show() to

display the results of the calculation. By default, the command calculates various statistics (see documentation linked below), including standard deviation and minimum.

Note that the answer that lists many options in the summary() parentheses does not include the minimum, which is asked for in the question.

Answer options that include agg() do not work here as shown, since DataFrame.agg() expects more complex, column-specific instructions on how to aggregate values.

More info:

– pyspark.sql.DataFrame.summary – PySpark 3.1.2 documentation

– pyspark.sql.DataFrame.agg – PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 3

Q42. Which of the following describes characteristics of the Dataset API?
* The Dataset API does not support unstructured data.
* In Python, the Dataset API mainly resembles Pandas’ DataFrame API.
* In Python, the Dataset API’s schema is constructed via type hints.
* The Dataset API is available in Scala, but it is not available in Python.
* The Dataset API does not provide compile-time type safety.

Explanation

The Dataset API is available in Scala, but it is not available in Python.

Correct. The Dataset API uses fixed typing and is typically used for object-oriented programming. It is available when Spark is used with the Scala programming language, but not for Python. In Python, you use the DataFrame API, which is based on the Dataset API.

The Dataset API does not provide compile-time type safety.

No – in fact, depending on the use case, the type safety that the Dataset API provides is an advantage.

The Dataset API does not support unstructured data.

Wrong, the Dataset API supports structured and unstructured data.

In Python, the Dataset API’s schema is constructed via type hints.

No, this is not applicable since the Dataset API is not available in Python.

In Python, the Dataset API mainly resembles Pandas’ DataFrame API.

The Dataset API does not exist in Python, only in Scala and Java.

Q43. The code block displayed below contains an error. The code block should return all rows of DataFrame transactionsDf, but including only columns storeId and predError. Find the error.

Code block:

spark.collect(transactionsDf.select(“storeId”, “predError”))
* Instead of select, DataFrame transactionsDf needs to be filtered using the filter operator.
* Columns storeId and predError need to be represented as a Python list, so they need to be wrapped in brackets ([]).
* The take method should be used instead of the collect method.
* Instead of collect, collectAsRows needs to be called.
* The collect method is not a method of the SparkSession object.
Explanation

Correct code block:

transactionsDf.select(“storeId”, “predError”).collect()

collect() is a method of the DataFrame object.

More info: pyspark.sql.DataFrame.collect – PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 2

Q44. Which of the following statements about RDDs is incorrect?
* An RDD consists of a single partition.
* The high-level DataFrame API is built on top of the low-level RDD API.
* RDDs are immutable.
* RDD stands for Resilient Distributed Dataset.
* RDDs are great for precisely instructing Spark on how to do a query.
Explanation

An RDD consists of a single partition.

Quite the opposite: Spark partitions RDDs and distributes the partitions across multiple nodes.

Q45. Which of the following code blocks returns a single row from DataFrame transactionsDf?

Full DataFrame transactionsDf:

1.+————-+———+—–+——-+———+—-+

2.|transactionId|predError|value|storeId|productId| f|

3.+————-+———+—–+——-+———+—-+

4.| 1| 3| 4| 25| 1|null|

5.| 2| 6| 7| 2| 2|null|

6.| 3| 3| null| 25| 3|null|

7.| 4| null| null| 3| 2|null|

8.| 5| null| null| null| 2|null|

9.| 6| 3| 2| 25| 2|null|

10.+————-+———+—–+——-+———+—-+
* transactionsDf.where(col(“storeId”).between(3,25))
* transactionsDf.filter((col(“storeId”)!=25) | (col(“productId”)==2))
* transactionsDf.filter(col(“storeId”)==25).select(“predError”,”storeId”).distinct()
* transactionsDf.select(“productId”, “storeId”).where(“storeId == 2 OR storeId != 25”)
* transactionsDf.where(col(“value”).isNull()).select(“productId”, “storeId”).distinct()
Explanation

Output of correct code block:

+———+——-+

|predError|storeId|

+———+——-+

| 3| 25|

+———+——-+

This question is difficult because it requires you to understand different kinds of commands and operators. All answers are valid Spark syntax, but just one expression returns a single-row DataFrame.

For reference, here is what the incorrect answers return:

transactionsDf.filter((col(“storeId”)!=25) | (col(“productId”)==2)) returns

+————-+———+—–+——-+———+—-+

|transactionId|predError|value|storeId|productId| f|

+————-+———+—–+——-+———+—-+

| 2| 6| 7| 2| 2|null|

| 4| null| null| 3| 2|null|

| 5| null| null| null| 2|null|

| 6| 3| 2| 25| 2|null|

+————-+———+—–+——-+———+—-+

transactionsDf.where(col(“storeId”).between(3,25)) returns

+————-+———+—–+——-+———+—-+

|transactionId|predError|value|storeId|productId| f|

+————-+———+—–+——-+———+—-+

| 1| 3| 4| 25| 1|null|

| 3| 3| null| 25| 3|null|

| 4| null| null| 3| 2|null|

| 6| 3| 2| 25| 2|null|

+————-+———+—–+——-+———+—-+

transactionsDf.where(col(“value”).isNull()).select(“productId”, “storeId”).distinct() returns

+———+——-+

|productId|storeId|

+———+——-+

| 3| 25|

| 2| 3|

| 2| null|

+———+——-+

transactionsDf.select(“productId”, “storeId”).where(“storeId == 2 OR storeId != 25”) returns

+———+——-+

|productId|storeId|

+———+——-+

| 2| 2|

| 2| 3|

+———+——-+

Static notebook | Dynamic notebook: See test 2

PDF Download Databricks Test To Gain Brilliante Result!:
https://www.topexamcollection.com/Associate-Developer-Apache-Spark-vce-collection.html]