DUMPSHQ

Free Questions for Databricks-Certified-Associate-Developer-for-Apache-Spark-3.0 by dumpshq

Shared by Abbott on 24-05-2024

Question 1

Question Type: **MultipleChoice**

Which of the following describes a narrow transformation?

Options:

A- narrow transformation is an operation in which data is exchanged across partitions.

B- A narrow transformation is a process in which data from multiple RDDs is used.

C- A narrow transformation is a process in which 32-bit float variables are cast to smaller float variables, like 16-bit or 8-bit float variables.

D- A narrow transformation is an operation in which data is exchanged across the cluster.

E- A narrow transformation is an operation in which no data is exchanged across the cluster.

Answer:

E

Explanation:

A narrow transformation is an operation in which no data is exchanged across the cluster.

Correct! In narrow transformations, no data is exchanged across the cluster, since these transformations do not require any data from outside of the partition they are applied on. Typical narrow

transformations include filter, drop, and coalesce.

A narrow transformation is an operation in which data is exchanged across partitions.

No, that would be one definition of a wide transformation, but not of a narrow transformation. Wide transformations typically cause a shuffle, in which data is exchanged across partitions, executors,

and the cluster.

A narrow transformation is an operation in which data is exchanged across the cluster.

No, see explanation just above this one.

A narrow transformation is a process in which 32-bit float variables are cast to smaller float variables, like 16-bit or 8-bit float variables.

No, type conversion has nothing to do with narrow transformations in Spark.

A narrow transformation is a process in which data from multiple RDDs is used.

No. A resilient distributed dataset (RDD) can be described as a collection of partitions. In a narrow transformation, no data is exchanged between partitions. Thus, no data is exchanged between

RDDs.

One could say though that a narrow transformation and, in fact, any transformation results in a new RDD being created. This is because a transformation results in a change to an existing RDD

(RDDs are the foundation of other Spark data structures, like DataFrames). But, since RDDs are immutable, a new RDD needs to be created to reflect the change caused by the transformation.

More info: Spark Transformation and Action: A Deep Dive | by Misbah Uddin | CodeX | Medium

Question 2

Question Type: MultipleChoice

The code block displayed below contains an error. The code block should return a DataFrame in which column predErrorAdded contains the results of Python function add_2_if_geq_3 as applied to

numeric and nullable column predError in DataFrame transactionsDf. Find the error.

Code block:

```
1. def add_2_if_geq_3(x):

2. if x is None:

3. return x

4. elif x >= 3:

5. return x+2
```

```
return x

add_2_if_geq_3_udf = udf(add_2_if_geq_3)

transactionsDf.withColumnRenamed("predErrorAdded", add_2_if_geq_3_udf(col("predError")))
```

Options:

A- The operator used to adding the column does not add column predErrorAdded to the DataFrame.

B- Instead of col('predError'), the actual DataFrame with the column needs to be passed, like so transactionsDf.predError.

C- The udf() method does not declare a return type.

D- UDFs are only available through the SQL API, but not in the Python API as shown in the code block.

E- The Python function is unable to handle null values, resulting in the code block crashing on execution.

Answer:

A

Explanation:

Correct code block:

```
def add_2_if_geq_3(x):

if x is None:

return x

elif x >= 3:

return x+2

return x

add_2_if_geq_3_udf = udf(add_2_if_geq_3)

transactionsDf.withColumn('predErrorAdded', add_2_if_geq_3_udf(col('predError'))).show()
```

Instead of withColumnRenamed, you should use the withColumn operator.

The udf() method does not declare a return type.

It is fine that the udf() method does not declare a return type, this is not a required argument. However, the default return type is StringType. This may not be the ideal return type for numeric,

nullable data -- but the code will run without specified return type nevertheless.

The Python function is unable to handle null values, resulting in the code block crashing on execution.

The Python function is able to handle null values, this is what the statement if x is None does.

UDFs are only available through the SQL API, but not in the Python API as shown in the code block.

No, they are available through the Python API. The code in the code block that concerns UDFs is correct.

Instead of col('predError'), the actual DataFrame with the column needs to be passed, like so transactionsDf.predError.

You may choose to use the transactionsDf.predError syntax, but the col('predError') syntax is fine.

Question 3

Question Type: MultipleChoice

In which order should the code blocks shown below be run in order to read a JSON file from location jsonPath into a DataFrame and return only the rows that do not have value 3 in column

productId?

1. importedDf.createOrReplaceTempView("importedDf")

2. spark.sql("SELECT * FROM importedDf WHERE productId != 3")

3. spark.sql("FILTER * FROM importedDf WHERE productId != 3")

4. importedDf = spark.read.option("format", "json").path(jsonPath)

5. importedDf = spark.read.json(jsonPath)

Options:

A- 4, 1, 2

B- 5, 1, 3

C- 5, 2

D- 4, 1, 3

E- 5, 1, 2

Answer:

E

Explanation:

Correct code block:

```
importedDf = spark.read.json(jsonPath)

importedDf.createOrReplaceTempView('importedDf')
```

spark.sql('SELECT * FROM importedDf WHERE productId != 3')

Option 5 is the only correct way listed of reading in a JSON in PySpark. The option('format', 'json') is not the correct way to tell Spark's DataFrameReader that you want to read a JSON file. You

would do this through format('json') instead. Also, you can communicate the specific path of the JSON file to the DataFramReader using the load() method, not the path() method.

In order to use a SQL command through the SparkSession spark, you first need to create a temporary view through DataFrame.createOrReplaceTempView().

The SQL statement should start with the SELECT operator. The FILTER operator SQL provides is not the correct one to use here.

Static notebook | Dynamic notebook: See test 2, Question: 59 (Databricks import instructions)

Question 4

Question Type: MultipleChoice

Which of the following code blocks returns a new DataFrame with the same columns as DataFrame transactionsDf, except for columns predError and value which should be removed?

Options:

A- transactionsDf.drop(['predError', 'value'])

B- transactionsDf.drop('predError', 'value')

C- transactionsDf.drop(col('predError'), col('value'))

D- transactionsDf.drop(predError, value)

E- transactionsDf.drop('predError & value')

Answer:

B

Explanation:

More info: pyspark.sql.DataFrame.drop --- PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 2, Question: 58 (Databricks import instructions)

Question 5

Question Type: **MultipleChoice**

The code block displayed below contains an error. The code block should count the number of rows that have a predError of either 3 or 6. Find the error.

Code block:

transactionsDf.filter(col('predError').in([3, 6])).count()

Options:

A- The number of rows cannot be determined with the count() operator.

B- Instead of filter, the select method should be used.

C- The method used on column predError is incorrect.

D- Instead of a list, the values need to be passed as single arguments to the in operator.

E- Numbers 3 and 6 need to be passed as string variables.

Answer:

C

Explanation:

Correct code block:

transactionsDf.filter(col('predError').isin([3, 6])).count()

The isin method is the correct one to use here -- the in method does not exist for the Column object.

More info: pyspark.sql.Column.isin --- PySpark 3.1.2 documentation

Question 6

Question Type: MultipleChoice

In which order should the code blocks shown below be run in order to return the number of records that are not empty in column value in the DataFrame resulting from an inner join of DataFrame

transactionsDf and itemsDf on columns productId and itemId, respectively?

1. .filter(~isnull(col('value')))

2. .count()

3. transactionsDf.join(itemsDf, col("transactionsDf.productId")==col("itemsDf.itemId"))

4. transactionsDf.join(itemsDf, transactionsDf.productId==itemsDf.itemId, how='inner')

5. .filter(col('value').isnotnull())

6. .sum(col('value'))

Options:

A- 4, 1, 2

B- 3, 1, 6

C- 3, 1, 2

D- 3, 5, 2

E- 4, 6

Answer:

A

Explanation:

Correct code block:

transactionsDf.join(itemsDf, transactionsDf.productId==itemsDf.itemId, how='inner').filter(~isnull(col('value'))).count()

Expressions col('transactionsDf.productId') and col('itemsDf.itemId') are invalid. col() does not accept the name of a DataFrame, only column names.

Static notebook | Dynamic notebook: See test 2, Question: 56 (Databricks import instructions)

Question 7

Question Type: **MultipleChoice**

Which of the following code blocks generally causes a great amount of network traffic?

Options:

A- DataFrame.select()

B- DataFrame.coalesce()

C- DataFrame.collect()

D- DataFrame.rdd.map()

E- DataFrame.count()

Answer:

C

Explanation:

DataFrame.collect() sends all data in a DataFrame from executors to the driver, so this generally causes a great amount of network traffic in comparison to the other options listed.

DataFrame.coalesce() just reduces the number of partitions and generally aims to reduce network traffic in comparison to a full shuffle.

DataFrame.select() is evaluated lazily and, unless followed by an action, does not cause significant network traffic.

DataFrame.rdd.map() is evaluated lazily, it does therefore not cause great amounts of network traffic.

DataFrame.count() is an action. While it does cause some network traffic, for the same DataFrame, collecting all data in the driver would generally be considered to cause a greater amount of

network traffic.

Question 8

Question Type: MultipleChoice

Which of the following code blocks reads in the parquet file stored at location filePath, given that all columns in the parquet file contain only whole numbers and are stored in the most appropriate

format for this kind of data?

Options:

A- 1. spark.read.schema(
2. StructType(
3. StructField('transactionId', IntegerType(), True),
4. StructField('predError', IntegerType(), True)
5.)).load(filePath)

B- 1. spark.read.schema([
2. StructField('transactionId', NumberType(), True),
3. StructField('predError', IntegerType(), True)
4.]).load(filePath)

C- 1. spark.read.schema(
2. StructType([
3. StructField('transactionId', StringType(), True),
4. StructField('predError', IntegerType(), True)]
5.)).parquet(filePath)

D- 1. spark.read.schema(
2. StructType([
3. StructField('transactionId', IntegerType(), True),
4. StructField('predError', IntegerType(), True)]
5.)).format('parquet').load(filePath)

E- 1. spark.read.schema([
2. StructField('transactionId', IntegerType(), True),
3. StructField('predError', IntegerType(), True)

4.]).load(filePath, format='parquet')

Answer:

D

Explanation:

The schema passed into schema should be of type StructType or a string, so all entries in which a list is passed are incorrect.

In addition, since all numbers are whole numbers, the IntegerType() data type is the correct option here. NumberType() is not a valid data type and StringType() would fail, since the parquet file is

stored in the 'most appropriate format for this kind of data', meaning that it is most likely an IntegerType, and Spark does not convert data types if a schema is provided.

Also note that StructType accepts only a single argument (a list of StructFields). So, passing multiple arguments is invalid.

Finally, Spark needs to know which format the file is in. However, all of the options listed are valid here, since Spark assumes parquet as a default when no file format is specifically passed.

More info: pyspark.sql.DataFrameReader.schema --- PySpark 3.1.2 documentation and StructType --- PySpark 3.1.2 documentation

Question 9

Question Type: **MultipleChoice**

Which of the following code blocks returns a DataFrame that is an inner join of DataFrame itemsDf and DataFrame transactionsDf, on columns itemId and productId, respectively and in which every

itemId just appears once?

Options:

A- itemsDf.join(transactionsDf, 'itemsDf.itemId==transactionsDf.productId').distinct('itemId')

B- itemsDf.join(transactionsDf, itemsDf.itemId==transactionsDf.productId).dropDuplicates(['itemId'])

C- itemsDf.join(transactionsDf, itemsDf.itemId==transactionsDf.productId).dropDuplicates('itemId')

D- itemsDf.join(transactionsDf, itemsDf.itemId==transactionsDf.productId, how='inner').distinct(['itemId'])

E- itemsDf.join(transactionsDf, 'itemsDf.itemId==transactionsDf.productId', how='inner').dropDuplicates(['itemId'])

Answer:

B

Explanation:

Filtering out distinct rows based on columns is achieved with the dropDuplicates method, not the distinct method which does not take any arguments.

The second argument of the join() method only accepts strings if they are column names. The SQL-like statement 'itemsDf.itemId==transactionsDf.productId' is therefore invalid.

In addition, it is not necessary to specify how='inner', since the default join type for the join command is already inner.

More info: pyspark.sql.DataFrame.join --- PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 2, Question: 53 (Databricks import instructions)

Question 10

Question Type: MultipleChoice

Which of the following code blocks produces the following output, given DataFrame transactionsDf?

Output:

1. root

2. |-- transactionId: integer (nullable = true)

3. |-- predError: integer (nullable = true)

4. |-- value: integer (nullable = true)

5. |-- storeId: integer (nullable = true)

6. |-- productId: integer (nullable = true)

7. |-- f: integer (nullable = true)

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. +-------------+---------+-----+-------+---------+----+

Options:

A- transactionsDf.schema.print()

B- transactionsDf.rdd.printSchema()

C- transactionsDf.rdd.formatSchema()

D- transactionsDf.printSchema()

E- print(transactionsDf.schema)

Answer:

D

Explanation:

The output is the typical output of a DataFrame.printSchema() call. The DataFrame's RDD representation does not have a printSchema or formatSchema method (find available methods in the RDD

documentation linked below). The output of print(transactionsDf.schema) is this: StructType(List(StructField(transactionId,IntegerType,true),StructField(predError,IntegerType,true),StructField

(value,IntegerType,true),StructField(storeId,IntegerType,true),StructField(productId,IntegerType,true),StructField(f,IntegerType,true))). It includes the same information as the nicely formatted original

output, but is not nicely formatted itself. Lastly, the DataFrame's schema attribute does not have a print() method.

More info:

- pyspark.RDD: pyspark.RDD --- PySpark 3.1.2 documentation

- DataFrame.printSchema(): pyspark.sql.DataFrame.printSchema --- PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 2, Question: 52 (Databricks import instructions)

Question 11

Question Type: MultipleChoice

The code block shown below should return a single-column DataFrame with a column named consonant_ct that, for each row, shows the number of consonants in column itemName of DataFrame

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

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.select(__1__(__2__(__3__(__4__), "a|e|i|o|u|\s", "")).__5__("consonant_ct"))

Options:

A- 1. length
2. regexp_extract
3. upper
4. col('itemName')
5. as

B- 1. size
2. regexp_replace
3. lower
4. 'itemName'
5. alias

C- 1. lower
2. regexp_replace
3. length
4. 'itemName'

5. alias

D- 1. length

2. regexp_replace

3. lower

4. col('itemName')

5. alias

E- 1. size

2. regexp_extract

3. lower

4. col('itemName')

5. alias

Answer:

D

Explanation:

Correct code block:

```
itemsDf.select(length(regexp_replace(lower(col('itemName')), 'a|e|i|o|u|\s', '')).alias('consonant_ct'))
```

Returned DataFrame:

```
+------------+
|consonant_ct|

+------------+

| 19|

| 16|

| 10|

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

This Question: tries to make you think about the string functions Spark provides and in which order they should be applied. Arguably the most difficult part, the regular expression 'a|e|i|o|u|

\s', is not a numbered blank. However, if you are not familiar with the string functions, it may be a good idea to review those before the exam.

The size operator and the length operator can easily be confused. size works on arrays, while length works on strings. Luckily, this is something you can read up about in the documentation.

The code block works by first converting all uppercase letters in column itemName into lowercase (the lower() part). Then, it replaces all vowels by 'nothing' - an empty character '' (the

regexp_replace() part). Now, only lowercase characters without spaces are included in the DataFrame. Then, per row, the length operator counts these remaining characters. Note that column itemName in itemsDf does not include any numbers or other characters, so we do not need to make any provisions for these. Finally, by using the alias() operator, we rename the resulting column to

consonant_ct.

More info:

- lower: pyspark.sql.functions.lower --- PySpark 3.1.2 documentation

- regexp_replace: pyspark.sql.functions.regexp_replace --- PySpark 3.1.2 documentation

- length: pyspark.sql.functions.length --- PySpark 3.1.2 documentation

- alias: pyspark.sql.Column.alias --- PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 2, Question: 51 (Databricks import instructions)

Question 12

Question Type: MultipleChoice

Which of the following code blocks reads all CSV files in directory filePath into a single DataFrame, with column names defined in the CSV file headers?

Content of directory filePath:

1. _SUCCESS

2. _committed_2754546451699747124

3. _started_2754546451699747124

4. part-00000-tid-2754546451699747124-10eb85bf-8d91-4dd0-b60b-2f3c02eeecaa-298-1-c000.csv.gz

5. part-00001-tid-2754546451699747124-10eb85bf-8d91-4dd0-b60b-2f3c02eeecaa-299-1-c000.csv.gz

6. part-00002-tid-2754546451699747124-10eb85bf-8d91-4dd0-b60b-2f3c02eeecaa-300-1-c000.csv.gz

7. part-00003-tid-2754546451699747124-10eb85bf-8d91-4dd0-b60b-2f3c02eeecaa-301-1-c000.csv.gz

spark.option("header",True).csv(filePath)

Options:

A- spark.read.format('csv').option('header',True).option('compression','zip').load(filePath)

B- spark.read().option('header',True).load(filePath)

C- spark.read.format('csv').option('header',True).load(filePath)

D- spark.read.load(filePath)

Answer:

C

Explanation:

The files in directory filePath are partitions of a DataFrame that have been exported using gzip compression. Spark automatically recognizes this situation and imports the CSV files as separate

partitions into a single DataFrame. It is, however, necessary to specify that Spark should load the file headers in the CSV with the header option, which is set to False by default.

20%
DISCOUNT