Databricks.Associate-Developer-Apache-Spark.v2022-06-21.q62

Exam Code:	Associate-Developer-Apache-Spark
Exam Name:	Databricks Certified Associate Developer for Apache Spark 3.0 Exam
Certification Provider:	Databricks
Free Question Number:	62
Version:	v2022-06-21
# of views:	2704
# of Questions views:	620
https://www.freepdfdumps.com/Databricks.Associate-Developer-Apache-Spark.v2022-06-21.q62.html	

NEW QUESTION: 1
The code block displayed below contains one or more errors. The code block should load parquet files at location filePath into a DataFrame, only loading those files that have been modified before 2029-03-20 05:44:46. Spark should enforce a schema according to the schema shown below. Find the error.
Schema:
1.root
2. |-- itemId: integer (nullable = true)
3. |-- attributes: array (nullable = true)
4. | |-- element: string (containsNull = true)
5. |-- supplier: string (nullable = true)
Code block:
1.schema = StructType([
2. StructType("itemId", IntegerType(), True),
3. StructType("attributes", ArrayType(StringType(), True), True),
4. StructType("supplier", StringType(), True)
5.])
6.
7.spark.read.options("modifiedBefore", "2029-03-20T05:44:46").schema(schema).load(filePath)
A. The attributes array is specified incorrectly, Spark cannot identify the file format, and the syntax of the call to Spark's DataFrameReader is incorrect.
B. Columns in the schema definition use the wrong object type and the syntax of the call to Spark's DataFrameReader is incorrect.
C. The data type of the schema is incompatible with the schema() operator and the modification date threshold is specified incorrectly.
D. Columns in the schema definition use the wrong object type, the modification date threshold is specified incorrectly, and Spark cannot identify the file format.

E. Columns in the schema are unable to handle empty values and the modification date threshold is specified incorrectly.

Answer: (SHOW ANSWER)

Explanation

Correct code block:

schema = StructType([
StructField("itemId", IntegerType(), True),
StructField("attributes", ArrayType(StringType(), True), True),
StructField("supplier", StringType(), True)
])
spark.read.options(modifiedBefore="2029-03-20T05:44:46").schema(schema).parquet(filePath) This question is more difficult than what you would encounter in the exam. In the exam, for this question type, only one error needs to be identified and not "one or multiple" as in the question.

Columns in the schema definition use the wrong object type, the modification date threshold is specified incorrectly, and Spark cannot identify the file format.

Correct! Columns in the schema definition should use the StructField type. Building a schema from pyspark.sql.types, as here using classes like StructType and StructField, is one of multiple ways of expressing a schema in Spark. A StructType always contains a list of StructFields (see documentation linked below). So, nesting StructType and StructType as shown in the question is wrong.

The modification date threshold should be specified by a keyword argument like options(modifiedBefore="2029-03-20T05:44:46") and not two consecutive non-keyword arguments as in the original code block (see documentation linked below).

Spark cannot identify the file format correctly, because either it has to be specified by using the DataFrameReader.format(), as an argument to DataFrameReader.load(), or directly by calling, for example, DataFrameReader.parquet().

Columns in the schema are unable to handle empty values and the modification date threshold is specified incorrectly.

No. If StructField would be used for the columns instead of StructType (see above), the third argument specified whether the column is nullable. The original schema shows that columns should be nullable and this is specified correctly by the third argument being True in the schema in the code block.

It is correct, however, that the modification date threshold is specified incorrectly (see above).

The attributes array is specified incorrectly, Spark cannot identify the file format, and the syntax of the call to Spark's DataFrameReader is incorrect.

Wrong. The attributes array is specified correctly, following the syntax for ArrayType (see linked documentation below). That Spark cannot identify the file format is correct, see correct answer above. In addition, the DataFrameReader is called correctly through the SparkSession spark.

Columns in the schema definition use the wrong object type and the syntax of the call to Spark's DataFrameReader is incorrect.

Incorrect, the object types in the schema definition are correct and syntax of the call to Spark's DataFrameReader is correct.

The data type of the schema is incompatible with the schema() operator and the modification date threshold is specified incorrectly.
False. The data type of the schema is StructType and an accepted data type for the DataFrameReader.schema() method. It is correct however that the modification date threshold is specified incorrectly (see correct answer above).

NEW QUESTION: 2
Which of the following describes a way for resizing a DataFrame from 16 to 8 partitions in the most efficient way?
A. Use operation DataFrame.repartition(8) to shuffle the DataFrame and reduce the number of partitions.
B. Use operation DataFrame.coalesce(8) to fully shuffle the DataFrame and reduce the number of partitions.
C. Use a narrow transformation to reduce the number of partitions.
D. 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.
Answer: (SHOW ANSWER)
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.

NEW QUESTION: 3
Which of the following code blocks returns a copy of DataFrame transactionsDf in which column productId has been renamed to productNumber?
A. transactionsDf.withColumnRenamed("productId", "productNumber")
B. transactionsDf.withColumn("productId", "productNumber")
C. transactionsDf.withColumnRenamed("productNumber", "productId")

D. transactionsDf.withColumnRenamed(col(productId), col(productNumber))
E. transactionsDf.withColumnRenamed(productId, productNumber)
Answer: A (LEAVE A REPLY)
Explanation
More info: pyspark.sql.DataFrame.withColumnRenamed - PySpark 3.1.2 documentation Static notebook | Dynamic notebook: See test 2

NEW QUESTION: 4
Which of the following code blocks prints out in how many rows the expression Inc. appears in the string-type column supplier of DataFrame itemsDf?
A. 1.counter = 0
2.
3.for index, row in itemsDf.iterrows():
4. if 'Inc.' in row['supplier']:
5. counter = counter + 1
6.
7.print(counter)
B. 1.counter = 0
2.
3.def count(x):
4. if 'Inc.' in x['supplier']:
5. counter = counter + 1
6.
7.itemsDf.foreach(count)
8.print(counter)
C. print(itemsDf.foreach(lambda x: 'Inc.' in x))
D. print(itemsDf.foreach(lambda x: 'Inc.' in x).sum())
E. 1.accum=sc.accumulator(0)
2.
3.def check_if_inc_in_supplier(row):
4. if 'Inc.' in row['supplier']:
5. accum.add(1)
6.
7.itemsDf.foreach(check_if_inc_in_supplier)
8.print(accum.value)
Answer: E (LEAVE A REPLY)
Explanation
Correct code block:
accum=sc.accumulator(0)
def check_if_inc_in_supplier(row):
if 'Inc.' in row['supplier']:

accum.add(1)
itemsDf.foreach(check_if_inc_in_supplier)
print(accum.value)
To answer this question correctly, you need to know both about the DataFrame.foreach() method and accumulators.
When Spark runs the code, it executes it on the executors. The executors do not have any information about variables outside of their scope. This is whhy simply using a Python variable counter, like in the two examples that start with counter = 0, will not work. You need to tell the executors explicitly that counter is a special shared variable, an Accumulator, which is managed by the driver and can be accessed by all executors for the purpose of adding to it.
If you have used Pandas in the past, you might be familiar with the iterrows() command. Notice that there is no such command in PySpark.
The two examples that start with print do not work, since DataFrame.foreach() does not have a return value.
More info: pyspark.sql.DataFrame.foreach - PySpark 3.1.2 documentation
Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 5
Which is the highest level in Spark's execution hierarchy?
A. Stage
B. Task
C. Executor
D. Job
E. Slot
Answer: D (LEAVE A REPLY)

NEW QUESTION: 6
In which order should the code blocks shown below be run in order to create a DataFrame that shows the mean of column predError of DataFrame transactionsDf per column storeId and productId, where productId should be either 2 or 3 and the returned DataFrame should be sorted in ascending order by column storeId, leaving out any nulls in that column?
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.+-------------+---------+-----+-------+---------+----+

1. .mean("predError")
2. .groupBy("storeId")
3. .orderBy("storeId")
4. transactionsDf.filter(transactionsDf.storeId.isNotNull())
5. .pivot("productId", [2, 3])

A. 4, 5, 2, 3, 1
B. 4, 2, 1
C. 4, 1, 5, 2, 3
D. 4, 2, 5, 1, 3
E. 4, 3, 2, 5, 1
Answer: D (LEAVE A REPLY)
Explanation
Correct code block:
transactionsDf.filter(transactionsDf.storeId.isNotNull()).groupBy("storeId").pivot("productId", [2, 3]).mean("predError").orderBy("storeId")
Output of correct code block:

```
+-------+----+----+
|storeId| 2| 3|
+-------+----+----+
| 2| 6.0|null|
| 3|null|null|
| 25| 3.0| 3.0|
+-------+----+----+
```

This question is quite convoluted and requires you to think hard about the correct order of operations.
The pivot method also makes an appearance - a method that you may not know all that much about (yet).
At the first position in all answers is code block 4, so the question is essentially just about the ordering of the remaining 4 code blocks.
The question states that the returned DataFrame should be sorted by column storeId. So, it should make sense to have code block 3 which includes the orderBy operator at the very end of the code block. This leaves you with only two answer options.
Now, it is useful to know more about the context of pivot in PySpark. A common pattern is groupBy, pivot, and then another aggregating function, like mean. In the documentation linked below you can see that pivot is a method of pyspark.sql.GroupedData - meaning that before pivoting, you have to use groupBy. The only answer option matching this requirement is the one in which code block 2 (which includes groupBy) is stated before code block 5 (which includes pivot).
More info: pyspark.sql.GroupedData.pivot - PySpark 3.1.2 documentation
Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 7
Which of the following is not a feature of Adaptive Query Execution?
A. Replace a sort merge join with a broadcast join, where appropriate.

B. Coalesce partitions to accelerate data processing.
C. Split skewed partitions into smaller partitions to avoid differences in partition processing time.
D. Reroute a query in case of an executor failure.
E. Collect runtime statistics during query execution.
Answer: D (LEAVE A REPLY)
Explanation
Reroute a query in case of an executor failure.
Correct. Although this feature exists in Spark, it is not a feature of Adaptive Query Execution. The cluster manager keeps track of executors and will work together with the driver to launch an executor and assign the workload of the failed executor to it (see also link below).
Replace a sort merge join with a broadcast join, where appropriate.
No, this is a feature of Adaptive Query Execution.
Coalesce partitions to accelerate data processing.
Wrong, Adaptive Query Execution does this.
Collect runtime statistics during query execution.
Incorrect, Adaptive Query Execution (AQE) collects these statistics to adjust query plans. This feedback loop is an essential part of accelerating queries via AQE.
Split skewed partitions into smaller partitions to avoid differences in partition processing time.
No, this is indeed a feature of Adaptive Query Execution. Find more information in the Databricks blog post linked below.
More info: Learning Spark, 2nd Edition, Chapter 12, On which way does RDD of spark finish fault-tolerance? - Stack Overflow, How to Speed up SQL Queries with Adaptive Query Execution

NEW QUESTION: 8
Which of the following describes characteristics of the Spark UI?
A. Via the Spark UI, workloads can be manually distributed across executors.
B. Via the Spark UI, stage execution speed can be modified.
C. The Scheduler tab shows how jobs that are run in parallel by multiple users are distributed across the cluster.
D. There is a place in the Spark UI that shows the property spark.executor.memory.
E. Some of the tabs in the Spark UI are named Jobs, Stages, Storage, DAGs, Executors, and SQL.
Answer: D (LEAVE A REPLY)
Explanation
There is a place in the Spark UI that shows the property spark.executor.memory.
Correct, you can see Spark properties such as spark.executor.memory in the Environment tab.
Some of the tabs in the Spark UI are named Jobs, Stages, Storage, DAGs, Executors, and SQL.
Wrong - Jobs, Stages, Storage, Executors, and SQL are all tabs in the Spark UI. DAGs can be inspected in the
"Jobs" tab in the job details or in the Stages or SQL tab, but are not a separate tab.
Via the Spark UI, workloads can be manually distributed across distributors.

No, the Spark UI is meant for inspecting the inner workings of Spark which ultimately helps understand, debug, and optimize Spark transactions.
Via the Spark UI, stage execution speed can be modified.
No, see above.
The Scheduler tab shows how jobs that are run in parallel by multiple users are distributed across the cluster.
No, there is no Scheduler tab.

NEW QUESTION: 9
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"))
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 (LEAVE A REPLY)
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

NEW QUESTION: 10
Which of the following statements about Spark's execution hierarchy is correct?
A. In Spark's execution hierarchy, a job may reach over multiple stage boundaries.
B. In Spark's execution hierarchy, manifests are one layer above jobs.
C. In Spark's execution hierarchy, a stage comprises multiple jobs.
D. In Spark's execution hierarchy, executors are the smallest unit.

E. In Spark's execution hierarchy, tasks are one layer above slots.
Answer: (SHOW ANSWER)
Explanation
In Spark's execution hierarchy, a job may reach over multiple stage boundaries.
Correct. A job is a sequence of stages, and thus may reach over multiple stage boundaries.
In Spark's execution hierarchy, tasks are one layer above slots.
Incorrect. Slots are not a part of the execution hierarchy. Tasks are the lowest layer.
In Spark's execution hierarchy, a stage comprises multiple jobs.
No. It is the other way around - a job consists of one or multiple stages.
In Spark's execution hierarchy, executors are the smallest unit.
False. Executors are not a part of the execution hierarchy. Tasks are the smallest unit!
In Spark's execution hierarchy, manifests are one layer above jobs.
Wrong. Manifests are not a part of the Spark ecosystem.

NEW QUESTION: 11
The code block shown below should return a one-column DataFrame where the column storeId is converted to string type. Choose the answer that correctly fills the blanks in the code block to accomplish this.
transactionsDf.__1__(__2__.__3__(__4__))
A. 1. select
2. col("storeId")
3. cast
4. StringType
B. 1. select
2. col("storeId")
3. as
4. StringType
C. 1. cast
2. "storeId"
3. as
4. StringType()
D. 1. select
2. col("storeId")
3. cast
4. StringType()
E. 1. select
2. storeId
3. cast
4. StringType()
Answer: D (LEAVE A REPLY)
Explanation
Correct code block:

transactionsDf.select(col("storeId").cast(StringType()))
Solving this question involves understanding that, when using types from the pyspark.sql.types such as StringType, these types need to be instantiated when using them in Spark, or, in simple words, they need to be followed by parentheses like so: StringType(). You could also use .cast("string") instead, but that option is not given here.
More info: pyspark.sql.Column.cast - PySpark 3.1.2 documentation
Static notebook | Dynamic notebook: See test 2

NEW QUESTION: 12
Which of the following code blocks can be used to save DataFrame transactionsDf to memory only, recalculating partitions that do not fit in memory when they are needed?
A. from pyspark import StorageLevel
transactionsDf.cache(StorageLevel.MEMORY_ONLY)
B. transactionsDf.cache()
C. transactionsDf.storage_level('MEMORY_ONLY')
D. transactionsDf.persist()
E. transactionsDf.clear_persist()
F. from pyspark import StorageLevel
transactionsDf.persist(StorageLevel.MEMORY_ONLY)
Answer: F (LEAVE A REPLY)
Explanation
from pyspark import StorageLevel transactionsDf.persist(StorageLevel.MEMORY_ONLY) Correct. Note that the storage level MEMORY_ONLY means that all partitions that do not fit into memory will be recomputed when they are needed.
transactionsDf.cache()
This is wrong because the default storage level of DataFrame.cache() is MEMORY_AND_DISK, meaning that partitions that do not fit into memory are stored on disk.
transactionsDf.persist()
This is wrong because the default storage level of DataFrame.persist() is MEMORY_AND_DISK.
transactionsDf.clear_persist()
Incorrect, since clear_persist() is not a method of DataFrame.
transactionsDf.storage_level('MEMORY_ONLY')
Wrong. storage_level is not a method of DataFrame.
More info: RDD Programming Guide - Spark 3.0.0 Documentation, pyspark.sql.DataFrame.persist - PySpark 3.0.0 documentation (https://bit.ly/3sxHLVC , https://bit.ly/3j2N6B9)

NEW QUESTION: 13
Which of the following options describes the responsibility of the executors in Spark?
A. The executors accept jobs from the driver, analyze those jobs, and return results to the driver.
B. The executors accept tasks from the driver, execute those tasks, and return results to the cluster manager.
C. The executors accept tasks from the driver, execute those tasks, and return results to the driver.

D. The executors accept tasks from the cluster manager, execute those tasks, and return results to the driver.
E. The executors accept jobs from the driver, plan those jobs, and return results to the cluster manager.
Answer: C (LEAVE A REPLY)
Explanation
More info: Running Spark: an overview of Spark's runtime architecture - Manning (https://bit.ly/2RPmJn9)

NEW QUESTION: 14
Which of the following code blocks creates a new 6-column DataFrame by appending the rows of the 6-column DataFrame yesterdayTransactionsDf to the rows of the 6-column DataFrame todayTransactionsDf, ignoring that both DataFrames have different column names?
A. union(todayTransactionsDf, yesterdayTransactionsDf)
B. todayTransactionsDf.unionByName(yesterdayTransactionsDf, allowMissingColumns=True)
C. todayTransactionsDf.unionByName(yesterdayTransactionsDf)
D. todayTransactionsDf.concat(yesterdayTransactionsDf)
E. todayTransactionsDf.union(yesterdayTransactionsDf)
Answer: (SHOW ANSWER)
Explanation
todayTransactionsDf.union(yesterdayTransactionsDf)
Correct. The union command appends rows of yesterdayTransactionsDf to the rows of todayTransactionsDf, ignoring that both DataFrames have different column names. The resulting DataFrame will have the column names of DataFrame todayTransactionsDf.
todayTransactionsDf.unionByName(yesterdayTransactionsDf)
No. unionByName specifically tries to match columns in the two DataFrames by name and only appends values in columns with identical names across the two DataFrames. In the form presented above, the command is a great fit for joining DataFrames that have exactly the same columns, but in a different order. In this case though, the command will fail because the two DataFrames have different columns.
todayTransactionsDf.unionByName(yesterdayTransactionsDf, allowMissingColumns=True) No. The unionByName command is described in the previous explanation. However, with the allowMissingColumns argument set to True, it is no longer an issue that the two DataFrames have different column names. Any columns that do not have a match in the other DataFrame will be filled with null where there is no value. In the case at hand, the resulting DataFrame will have 7 or more columns though, so it this command is not the right answer.
union(todayTransactionsDf, yesterdayTransactionsDf)
No, there is no union method in pyspark.sql.functions.
todayTransactionsDf.concat(yesterdayTransactionsDf)
Wrong, the DataFrame class does not have a concat method.
More info: pyspark.sql.DataFrame.union - PySpark 3.1.2 documentation,
pyspark.sql.DataFrame.unionByName - PySpark 3.1.2 documentation
Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 15

The code block displayed below contains an error. The code block should use Python method find_most_freq_letter to find the letter present most in column itemName of DataFrame itemsDf and return it in a new column most_frequent_letter. Find the error.
Code block:
1. find_most_freq_letter_udf = udf(find_most_freq_letter)
2. itemsDf.withColumn("most_frequent_letter", find_most_freq_letter("itemName"))
A. Spark is not using the UDF method correctly.
B. The UDF method is not registered correctly, since the return type is missing.
C. The "itemName" expression should be wrapped in col().
D. UDFs do not exist in PySpark.
E. Spark is not adding a column.
Answer: A (LEAVE A REPLY)
Explanation
Correct code block:
find_most_freq_letter_udf = udf(find_most_frequent_letter)
itemsDf.withColumn("most_frequent_letter", find_most_freq_letter_udf("itemName")) Spark should use the previously registered find_most_freq_letter_udf method here - but it is not doing that in the original codeblock. There, it just uses the non-UDF version of the Python method.
Note that typically, we would have to specify a return type for udf(). Except in this case, since the default return type for udf() is a string which is what we are expecting here. If we wanted to return an integer variable instead, we would have to register the Python function as UDF using find_most_freq_letter_udf = udf(find_most_freq_letter, IntegerType()).
More info: pyspark.sql.functions.udf - PySpark 3.1.1 documentation

NEW QUESTION: 16
The code block displayed below contains an error. The code block below is intended to add a column itemNameElements to DataFrame itemsDf that includes an array of all words in column itemName. Find the error.
Sample of DataFrame itemsDf:
1.+------+----------------------------------+------------------+
2.|itemId|itemName |supplier |
3.+------+----------------------------------+------------------+
4.|1 |Thick Coat for Walking in the Snow|Sports Company Inc.|
5.|2 |Elegant Outdoors Summer Dress |YetiX |
6.|3 |Outdoors Backpack |Sports Company Inc.|
7.+------+----------------------------------+------------------+
Code block:
itemsDf.withColumnRenamed("itemNameElements", split("itemName"))
itemsDf.withColumnRenamed("itemNameElements", split("itemName"))
A. All column names need to be wrapped in the col() operator.
B. Operator withColumnRenamed needs to be replaced with operator withColumn and a second argument

"," needs to be passed to the split method.
C. Operator withColumnRenamed needs to be replaced with operator withColumn and the split method needs to be replaced by the splitString method.
D. Operator withColumnRenamed needs to be replaced with operator withColumn and a second argument " " needs to be passed to the split method.
E. The expressions "itemNameElements" and split("itemName") need to be swapped.
Answer: (SHOW ANSWER)
Explanation
Correct code block:
itemsDf.withColumn("itemNameElements", split("itemName"," "))
Output of code block:
+------+--------------------------------+-----------------+--------------------------------------+
|itemId|itemName |supplier |itemNameElements |
+------+--------------------------------+-----------------+--------------------------------------+
1	Thick Coat for Walking in the Snow	Sports Company Inc.	[Thick, Coat, for, Walking, in, the, Snow]
2	Elegant Outdoors Summer Dress	YetiX	[Elegant, Outdoors, Summer, Dress]
3	Outdoors Backpack	Sports Company Inc.	[Outdoors, Backpack]
+------+--------------------------------+-----------------+--------------------------------------+ The key to solving this question is that the split method definitely needs a second argument here (also look at the link to the documentation below). Given the values in column itemName in DataFrame itemsDf, this should be a space character " ". This is the character we need to split the words in the column.
More info: pyspark.sql.functions.split - PySpark 3.1.1 documentation
Static notebook | Dynamic notebook: See test 1

NEW QUESTION: 17
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)
A. 1. coalesce(1)
2. option

3. "sep"
4. option("header", True)
5. path
B. 1. coalesce(1)
2. option
3. "colsep"
4. option("nullValue", "n/a")
5. path
C. 1. repartition(1)
2. option
3. "sep"
4. option("nullValue", "n/a")
5. csv
(Correct)
D. 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
Answer: C (LEAVE A REPLY)
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

NEW QUESTION: 18

Which of the following is the idea behind dynamic partition pruning in Spark?
A. Dynamic partition pruning is intended to skip over the data you do not need in the results of a query.
B. Dynamic partition pruning concatenates columns of similar data types to optimize join performance.
C. Dynamic partition pruning performs wide transformations on disk instead of in memory.
D. Dynamic partition pruning reoptimizes physical plans based on data types and broadcast variables.
E. Dynamic partition pruning reoptimizes query plans based on runtime statistics collected during query execution.
Answer: A (LEAVE A REPLY)
Explanation
Dynamic partition pruning reoptimizes query plans based on runtime statistics collected during query execution.
No - this is what adaptive query execution does, but not dynamic partition pruning.
Dynamic partition pruning concatenates columns of similar data types to optimize join performance.
Wrong, this answer does not make sense, especially related to dynamic partition pruning.
Dynamic partition pruning reoptimizes physical plans based on data types and broadcast variables.
It is true that dynamic partition pruning works in joins using broadcast variables. This actually happens in both the logical optimization and the physical planning stage. However, data types do not play a role for the reoptimization.
Dynamic partition pruning performs wide transformations on disk instead of in memory.
This answer does not make sense. Dynamic partition pruning is meant to accelerate Spark - performing any transformation involving disk instead of memory resources would decelerate Spark and certainly achieve the opposite effect of what dynamic partition pruning is intended for.

NEW QUESTION: 19
Which of the following code blocks sorts DataFrame transactionsDf both by column storeId in ascending and by column productId in descending order, in this priority?
A. transactionsDf.sort("storeId", asc("productId"))
B. transactionsDf.sort(col(storeId)).desc(col(productId))
C. transactionsDf.order_by(col(storeId), desc(col(productId)))
D. transactionsDf.sort("storeId", desc("productId"))
E. transactionsDf.sort("storeId").sort(desc("productId"))
Answer: D (LEAVE A REPLY)
Explanation
In this question it is important to realize that you are asked to sort transactionDf by two columns. This means that the sorting of the second column depends on the sorting of the first column.
So, any option that sorts the entire DataFrame (through chaining sort statements) will not work. The two columns need to be channeled through the same call to sort().
Also, order_by is not a valid DataFrame API method.
More info: pyspark.sql.DataFrame.sort - PySpark 3.1.2 documentation
Static notebook | Dynamic notebook: See test 2

NEW QUESTION: 20
Which of the following code blocks selects all rows from DataFrame transactionsDf in which column productId is zero or smaller or equal to 3?
A. transactionsDf.filter(productId==3 or productId<1)
B. transactionsDf.filter((col("productId")==3) or (col("productId")<1))
C. transactionsDf.filter(col("productId")==3 | col("productId")<1)
D. transactionsDf.where("productId"=3).or("productId"<1))
E. transactionsDf.filter((col("productId")==3) | (col("productId")<1))
Answer: E (LEAVE A REPLY)
Explanation
This question targets your knowledge about how to chain filtering conditions. Each filtering condition should be in parentheses. The correct operator for "or" is the pipe character (|) and not the word or. Another operator of concern is the equality operator. For the purpose of comparison, equality is expressed as two equal signs (==).
Static notebook | Dynamic notebook: See test 2

NEW QUESTION: 21
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.+-------------+---------+-----+-------+---------+----+
A. transactionsDf.schema.print()
B. transactionsDf.rdd.printSchema()
C. transactionsDf.rdd.formatSchema()
D. transactionsDf.printSchema()
E. print(transactionsDf.schema)
Answer: D (LEAVE A REPLY)
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),StructFiel
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

NEW QUESTION: 22
Which of the following statements about stages is correct?
A. Different stages in a job may be executed in parallel.
B. Stages consist of one or more jobs.
C. Stages ephemerally store transactions, before they are committed through actions.
D. Tasks in a stage may be executed by multiple machines at the same time.
E. Stages may contain multiple actions, narrow, and wide transformations.
Answer: D (LEAVE A REPLY)
Explanation
Tasks in a stage may be executed by multiple machines at the same time.
This is correct. Within a single stage, tasks do not depend on each other. Executors on multiple machines may execute tasks belonging to the same stage on the respective partitions they are holding at the same time.
Different stages in a job may be executed in parallel.
No. Different stages in a job depend on each other and cannot be executed in parallel. The nuance is that every task in a stage may be executed in parallel by multiple machines.
For example, if a job consists of Stage A and Stage B, tasks belonging to those stages may not be executed in parallel. However, tasks from Stage A may be executed on multiple machines at the same time, with each machine running it on a different partition of the same dataset. Then, afterwards, tasks from Stage B may be executed on multiple machines at the same time.
Stages may contain multiple actions, narrow, and wide transformations.
No, stages may not contain multiple wide transformations. Wide transformations mean that shuffling is required. Shuffling typically terminates a stage though, because data needs to be exchanged across the cluster. This data exchange often causes partitions to change and rearrange, making it impossible to perform tasks in parallel on the same dataset.
Stages ephemerally store transactions, before they are committed through actions.
No, this does not make sense. Stages do not "store" any data. Transactions are not "committed" in Spark.
Stages consist of one or more jobs.
No, it is the other way around: Jobs consist of one more stages.

More info: Spark: The Definitive Guide, Chapter 15.

NEW QUESTION: 23

The code block displayed below contains multiple errors. The code block should return a DataFrame that contains only columns transactionId, predError, value and storeId of DataFrame transactionsDf. Find the errors.

Code block:

transactionsDf.select([col(productId), col(f)])

Sample of 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.+-------------+---------+-----+-------+---------+----+

A. The column names should be listed directly as arguments to the operator and not as a list.

B. The select operator should be replaced by a drop operator, the column names should be listed directly as arguments to the operator and not as a list, and all column names should be expressed as strings without being wrapped in a col() operator.

C. The select operator should be replaced by a drop operator.

D. The column names should be listed directly as arguments to the operator and not as a list and following the pattern of how column names are expressed in the code block, columns productId and f should be replaced by transactionId, predError, value and storeId.

E. The select operator should be replaced by a drop operator, the column names should be listed directly as arguments to the operator and not as a list, and all col() operators should be removed.

Answer: (SHOW ANSWER)

Explanation

Correct code block: transactionsDf.drop("productId", "f")

This question requires a lot of thinking to get right. For solving it, you may take advantage of the digital notepad that is provided to you during the test. You have probably seen that the code block includes multiple errors. In the test, you are usually confronted with a code block that only contains a single error. However, since you are practicing here, this challenging multi-error question will make it easier for you to deal with single-error questions in the real exam.

The select operator should be replaced by a drop operator, the column names should be listed directly as arguments to the operator and not as a list, and all column names should be expressed as strings without being wrapped in a col() operator.

Correct! Here, you need to figure out the many, many things that are wrong with the initial code block. While the question can be solved by using a select statement, a drop statement, given the answer options, is the correct one. Then, you can read in the documentation that drop does not take a list as an argument, but just

the column names that should be dropped. Finally, the column names should be expressed as strings and not as Python variable names as in the original code block.
The column names should be listed directly as arguments to the operator and not as a list.
Incorrect. While this is a good first step and part of the correct solution (see above), this modification is insufficient to solve the question.
The column names should be listed directly as arguments to the operator and not as a list and following the pattern of how column names are expressed in the code block, columns productId and f should be replaced by transactionId, predError, value and storeId.
Wrong. If you use the same pattern as in the original code block (col(productId), col(f)), you are still making a mistake. col(productId) will trigger Python to search for the content of a variable named productId instead of telling Spark to use the column productId - for that, you need to express it as a string.
The select operator should be replaced by a drop operator, the column names should be listed directly as arguments to the operator and not as a list, and all col() operators should be removed.
No. This still leaves you with Python trying to interpret the column names as Python variables (see above).
The select operator should be replaced by a drop operator.
Wrong, this is not enough to solve the question. If you do this, you will still face problems since you are passing a Python list to drop and the column names are still interpreted as Python variables (see above).
More info: pyspark.sql.DataFrame.drop - PySpark 3.1.2 documentation
Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 24
Which of the following code blocks returns all unique values of column storeId in DataFrame transactionsDf?
A. transactionsDf["storeId"].distinct()
B. transactionsDf.select("storeId").distinct()
(Correct)
C. transactionsDf.filter("storeId").distinct()
D. transactionsDf.select(col("storeId").distinct())
E. transactionsDf.distinct("storeId")
Answer: B (LEAVE A REPLY)
Explanation
distinct() is a method of a DataFrame. Knowing this, or recognizing this from the documentation, is the key to solving this question.
More info: pyspark.sql.DataFrame.distinct - PySpark 3.1.2 documentation Static notebook | Dynamic notebook: See test 2

NEW QUESTION: 25
Which of the following code blocks shuffles DataFrame transactionsDf, which has 8 partitions, so that it has 10 partitions?
A. transactionsDf.repartition(transactionsDf.getNumPartitions()+2)
B. transactionsDf.repartition(transactionsDf.rdd.getNumPartitions()+2)
C. transactionsDf.coalesce(10)

D. transactionsDf.coalesce(transactionsDf.getNumPartitions()+2)
E. transactionsDf.repartition(transactionsDf._partitions+2)
Answer: B (LEAVE A REPLY)
Explanation
transactionsDf.repartition(transactionsDf.rdd.getNumPartitions()+2)
Correct. The repartition operator is the correct one for increasing the number of partitions. calling getNumPartitions() on DataFrame.rdd returns the current number of partitions.
transactionsDf.coalesce(10)
No, after this command transactionsDf will continue to only have 8 partitions. This is because coalesce() can only decreast the amount of partitions, but not increase it.
transactionsDf.repartition(transactionsDf.getNumPartitions()+2)
Incorrect, there is no getNumPartitions() method for the DataFrame class.
transactionsDf.coalesce(transactionsDf.getNumPartitions()+2)
Wrong, coalesce() can only be used for reducing the number of partitions and there is no getNumPartitions() method for the DataFrame class.
transactionsDf.repartition(transactionsDf._partitions+2)
No, DataFrame has no _partitions attribute. You can find out the current number of partitions of a DataFrame with the DataFrame.rdd.getNumPartitions() method.
More info: pyspark.sql.DataFrame.repartition - PySpark 3.1.2 documentation, pyspark.RDD.getNumPartitions - PySpark 3.1.2 documentation Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 26
Which of the following is the deepest level in Spark's execution hierarchy?
A. Job
B. Task
C. Executor
D. Slot
E. Stage
Answer: B (LEAVE A REPLY)
Explanation
The hierarchy is, from top to bottom: Job, Stage, Task.
Executors and slots facilitate the execution of tasks, but they are not directly part of the hierarchy. Executors are launched by the driver on worker nodes for the purpose of running a specific Spark application. Slots help Spark parallelize work. An executor can have multiple slots which enable it to process multiple tasks in parallel.

NEW QUESTION: 27
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?
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 (LEAVE A REPLY)

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

NEW QUESTION: 28

The code block displayed below contains an error. The code block should arrange the rows of DataFrame transactionsDf using information from two columns in an ordered fashion, arranging first by column value,

showing smaller numbers at the top and greater numbers at the bottom, and then by column predError, for which all values should be arranged in the inverse way of the order of items in column value. Find the error.
Code block:
transactionsDf.orderBy('value', asc_nulls_first(col('predError')))
A. Two orderBy statements with calls to the individual columns should be chained, instead of having both columns in one orderBy statement.
B. Column value should be wrapped by the col() operator.
C. Column predError should be sorted in a descending way, putting nulls last.
D. Column predError should be sorted by desc_nulls_first() instead.
E. Instead of orderBy, sort should be used.
Answer: C (LEAVE A REPLY)
Explanation
Correct code block:
transactionsDf.orderBy('value', desc_nulls_last('predError'))
Column predError should be sorted in a descending way, putting nulls last.
Correct! By default, Spark sorts ascending, putting nulls first. So, the inverse sort of the default sort is indeed desc_nulls_last.
Instead of orderBy, sort should be used.
No. DataFrame.sort() orders data per partition, it does not guarantee a global order. This is why orderBy is the more appropriate operator here.
Column value should be wrapped by the col() operator.
Incorrect. DataFrame.sort() accepts both string and Column objects.
Column predError should be sorted by desc_nulls_first() instead.
Wrong. Since Spark's default sort order matches asc_nulls_first(), nulls would have to come last when inverted.
Two orderBy statements with calls to the individual columns should be chained, instead of having both columns in one orderBy statement.
No, this would just sort the DataFrame by the very last column, but would not take information from both columns into account, as noted in the question.
More info: pyspark.sql.DataFrame.orderBy - PySpark 3.1.2 documentation, pyspark.sql.functions.desc_nulls_last - PySpark 3.1.2 documentation, sort() vs orderBy() in Spark | Towards Data Science Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 29
Which of the following code blocks returns the number of unique values in column storeId of DataFrame transactionsDf?
A. transactionsDf.select("storeId").dropDuplicates().count()
B. transactionsDf.select(count("storeId")).dropDuplicates()
C. transactionsDf.select(distinct("storeId")).count()
D. transactionsDf.dropDuplicates().agg(count("storeId"))
E. transactionsDf.distinct().select("storeId").count()

Answer: A (LEAVE A REPLY)
Explanation
transactionsDf.select("storeId").dropDuplicates().count()
Correct! After dropping all duplicates from column storeId, the remaining rows get counted, representing the number of unique values in the column.
transactionsDf.select(count("storeId")).dropDuplicates()
No. transactionsDf.select(count("storeId")) just returns a single-row DataFrame showing the number of non-null rows. dropDuplicates() does not have any effect in this context.
transactionsDf.dropDuplicates().agg(count("storeId"))
Incorrect. While transactionsDf.dropDuplicates() removes duplicate rows from transactionsDf, it does not do so taking only column storeId into consideration, but eliminates full row duplicates instead.
transactionsDf.distinct().select("storeId").count()
Wrong. transactionsDf.distinct() identifies unique rows across all columns, but not only unique rows with respect to column storeId. This may leave duplicate values in the column, making the count not represent the number of unique values in that column.
transactionsDf.select(distinct("storeId")).count()
False. There is no distinct method in pyspark.sql.functions.

NEW QUESTION: 30
Which of the following statements about executors is correct?
A. Executors are launched by the driver.
B. Executors stop upon application completion by default.
C. Each node hosts a single executor.
D. Executors store data in memory only.
E. An executor can serve multiple applications.
Answer: B (LEAVE A REPLY)
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

NEW QUESTION: 31

In which order should the code blocks shown below be run in order to assign articlesDf a DataFrame that lists all items in column attributes ordered by the number of times these items occur, from most to least often?

Sample of DataFrame articlesDf:

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

A. 1. articlesDf = articlesDf.groupby("col")
2. articlesDf = articlesDf.select(explode(col("attributes")))
3. articlesDf = articlesDf.orderBy("count").select("col")
4. articlesDf = articlesDf.sort("count",ascending=False).select("col")
5. articlesDf = articlesDf.groupby("col").count()

B. 4, 5

C. 2, 5, 3

D. 5, 2

E. 2, 3, 4

F. 2, 5, 4

Answer: (SHOW ANSWER)

Explanation

Correct code block:

articlesDf = articlesDf.select(explode(col('attributes')))
articlesDf = articlesDf.groupby('col').count()
articlesDf = articlesDf.sort('count',ascending=False).select('col')

Output of correct code block:

+-------+
| col|
+-------+
| summer|
| winter|
| blue|
| cozy|
| travel|
| fresh|

| red|
|cooling|
| green|
+-------+
Static notebook | Dynamic notebook: See test 2

NEW QUESTION: 32
The code block displayed below contains an error. The code block should combine data from DataFrames itemsDf and transactionsDf, showing all rows of DataFrame itemsDf that have a matching value in column itemId with a value in column transactionsId of DataFrame transactionsDf. Find the error.
Code block:
itemsDf.join(itemsDf.itemId==transactionsDf.transactionId)
A. The join statement is incomplete.
B. The union method should be used instead of join.
C. The join method is inappropriate.
D. The merge method should be used instead of join.
E. The join expression is malformed.
Answer: A (LEAVE A REPLY)
Explanation
Correct code block:
itemsDf.join(transactionsDf, itemsDf.itemId==transactionsDf.transactionId) The join statement is incomplete.
Correct! If you look at the documentation of DataFrame.join() (linked below), you see that the very first argument of join should be the DataFrame that should be joined with. This first argument is missing in the code block.
The join method is inappropriate.
No. By default, DataFrame.join() uses an inner join. This method is appropriate for the scenario described in the question.
The join expression is malformed.
Incorrect. The join expression itemsDf.itemId==transactionsDf.transactionId is correct syntax.
The merge method should be used instead of join.
False. There is no DataFrame.merge() method in PySpark.
The union method should be used instead of join.

Wrong. DataFrame.union() merges rows, but not columns as requested in the question.
More info: pyspark.sql.DataFrame.join - PySpark 3.1.2 documentation, pyspark.sql.DataFrame.union - PySpark 3.1.2 documentation Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 33
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)
A. 4, 1, 2
B. 5, 1, 3
C. 5, 2
D. 4, 1, 3
E. 5, 1, 2
Answer: (SHOW ANSWER)
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

NEW QUESTION: 34
Which of the following code blocks reads in the two-partition parquet file stored at filePath, making sure all columns are included exactly once even though each partition has a different schema?
Schema of first partition:
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)
Schema of second partition:
1.root
2. |-- transactionId: integer (nullable = true)
3. |-- predError: integer (nullable = true)
4. |-- value: integer (nullable = true)
5. |-- storeId: integer (nullable = true)
6. |-- rollId: integer (nullable = true)
7. |-- f: integer (nullable = true)
8. |-- tax_id: integer (nullable = false)
A. spark.read.parquet(filePath, mergeSchema='y')
B. spark.read.option("mergeSchema", "true").parquet(filePath)
C. spark.read.parquet(filePath)
D. 1.nx = 0
2.for file in dbutils.fs.ls(filePath):
3. if not file.name.endswith(".parquet"):
4. continue
5. df_temp = spark.read.parquet(file.path)
6. if nx == 0:
7. df = df_temp
8. else:
9. df = df.union(df_temp)
10. nx = nx+1
11.df
E. 1.nx = 0
2.for file in dbutils.fs.ls(filePath):
3. if not file.name.endswith(".parquet"):
4. continue
5. df_temp = spark.read.parquet(file.path)
6. if nx == 0:
7. df = df_temp
8. else:
9. df = df.join(df_temp, how="outer")
10. nx = nx+1
11.df
Answer: B (LEAVE A REPLY)
Explanation
This is a very tricky question and involves both knowledge about merging as well as schemas when reading parquet files.

spark.read.option("mergeSchema", "true").parquet(filePath)
Correct. Spark's DataFrameReader's mergeSchema option will work well here, since columns that appear in both partitions have matching data types. Note that mergeSchema would fail if one or more columns with the same name that appear in both partitions would have different data types.
spark.read.parquet(filePath)
Incorrect. While this would read in data from both partitions, only the schema in the parquet file that is read in first would be considered, so some columns that appear only in the second partition (e.g. tax_id) would be lost.
nx = 0
for file in dbutils.fs.ls(filePath):
if not file.name.endswith(".parquet"):
continue
df_temp = spark.read.parquet(file.path)
if nx == 0:
df = df_temp
else:
df = df.union(df_temp)
nx = nx+1
df
Wrong. The key idea of this solution is the DataFrame.union() command. While this command merges all data, it requires that both partitions have the exact same number of columns with identical data types.
spark.read.parquet(filePath, mergeSchema="y")
False. While using the mergeSchema option is the correct way to solve this problem and it can even be called with DataFrameReader.parquet() as in the code block, it accepts the value True as a boolean or string variable. But 'y' is not a valid option.
nx = 0
for file in dbutils.fs.ls(filePath):
if not file.name.endswith(".parquet"):
continue
df_temp = spark.read.parquet(file.path)
if nx == 0:
df = df_temp
else:
df = df.join(df_temp, how="outer")
nx = nx+1
df
No. This provokes a full outer join. While the resulting DataFrame will have all columns of both partitions, columns that appear in both partitions will be duplicated - the question says all columns that are included in the partitions should appear exactly once.
More info: Merging different schemas in Apache Spark | by Thiago Cordon | Data Arena | Medium Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 35
The code block shown below should return a DataFrame with only columns from DataFrame transactionsDf for which there is a corresponding transactionId in DataFrame itemsDf. DataFrame itemsDf is very small and much smaller than DataFrame transactionsDf. The query should be executed in an optimized way. Choose the answer that correctly fills the blanks in the code block to accomplish this.
__1__.__2__(__3__, __4__, __5__)
A. 1. transactionsDf
2. join
3. broadcast(itemsDf)
4. transactionsDf.transactionId==itemsDf.transactionId
5. "outer"
B. 1. transactionsDf
2. join
3. itemsDf
4. transactionsDf.transactionId==itemsDf.transactionId
5. "anti"
C. 1. transactionsDf
2. join
3. broadcast(itemsDf)
4. "transactionId"
5. "left_semi"
D. 1. itemsDf
2. broadcast
3. transactionsDf
4. "transactionId"
5. "left_semi"
E. 1. itemsDf
2. join
3. broadcast(transactionsDf)
4. "transactionId"
5. "left_semi"
Answer: (SHOW ANSWER)
Explanation
Correct code block:
transactionsDf.join(broadcast(itemsDf), "transactionId", "left_semi")
This question is extremely difficult and exceeds the difficulty of questions in the exam by far.
A first indication of what is asked from you here is the remark that "the query should be executed in an optimized way". You also have qualitative information about the size of itemsDf and transactionsDf. Given that itemsDf is "very small" and that the execution should be optimized, you should consider instructing Spark to perform a broadcast join, broadcasting the "very small" DataFrame itemsDf to all executors. You can

explicitly suggest this to Spark via wrapping itemsDf into a broadcast() operator. One answer option does not include this operator, so you can disregard it. Another answer option wraps the broadcast() operator around transactionsDf - the bigger of the two DataFrames. This answer option does not make sense in the optimization context and can likewise be disregarded.
When thinking about the broadcast() operator, you may also remember that it is a method of pyspark.sql.functions. One answer option, however, resolves to itemsDf.broadcast([...]). The DataFrame class has no broadcast() method, so this answer option can be eliminated as well.
All two remaining answer options resolve to transactionsDf.join([...]) in the first 2 gaps, so you will have to figure out the details of the join now. You can pick between an outer and a left semi join. An outer join would include columns from both DataFrames, where a left semi join only includes columns from the "left" table, here transactionsDf, just as asked for by the question. So, the correct answer is the one that uses the left_semi join.

NEW QUESTION: 36
Which of the following code blocks uses a schema fileSchema to read a parquet file at location filePath into a DataFrame?
A. spark.read.schema(fileSchema).format("parquet").load(filePath)
B. spark.read.schema("fileSchema").format("parquet").load(filePath)
C. spark.read().schema(fileSchema).parquet(filePath)
D. spark.read().schema(fileSchema).format(parquet).load(filePath)
E. spark.read.schema(fileSchema).open(filePath)
Answer: A (LEAVE A REPLY)
Explanation
Pay attention here to which variables are quoted. fileSchema is a variable and thus should not be in quotes. parquet is not a variable and therefore should be in quotes.
SparkSession.read (here referenced as spark.read) returns a DataFrameReader which all subsequent calls reference - the DataFrameReader is not callable, so you should not use parentheses here.
Finally, there is no open method in PySpark. The method name is load.
Static notebook | Dynamic notebook: See test 1

NEW QUESTION: 37
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'))
A. 4, 1, 2

B. 3, 1, 6
C. 3, 1, 2
D. 3, 5, 2
E. 4, 6
Answer: A (LEAVE A REPLY)
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

NEW QUESTION: 38
Which of the following code blocks returns a DataFrame showing the mean value of column "value" of DataFrame transactionsDf, grouped by its column storeId?
A. transactionsDf.groupBy(col(storeId).avg())
B. transactionsDf.groupBy("storeId").avg(col("value"))
C. transactionsDf.groupBy("storeId").agg(avg("value"))
D. transactionsDf.groupBy("storeId").agg(average("value"))
E. transactionsDf.groupBy("value").average()
Answer: C (LEAVE A REPLY)
Explanation
This question tests your knowledge about how to use the groupBy and agg pattern in Spark. Using the documentation, you can find out that there is no average() method in pyspark.sql.functions.
Static notebook | Dynamic notebook: See test 2

NEW QUESTION: 39
Which of the following is one of the big performance advantages that Spark has over Hadoop?
A. Spark achieves great performance by storing data in the DAG format, whereas Hadoop can only use parquet files.
B. Spark achieves higher resiliency for queries since, different from Hadoop, it can be deployed on Kubernetes.
C. Spark achieves great performance by storing data and performing computation in memory, whereas large jobs in Hadoop require a large amount of relatively slow disk I/O operations.
D. Spark achieves great performance by storing data in the HDFS format, whereas Hadoop can only use parquet files.
E. Spark achieves performance gains for developers by extending Hadoop's DataFrames with a user-friendly API.
Answer: C (LEAVE A REPLY)
Explanation

Spark achieves great performance by storing data in the DAG format, whereas Hadoop can only use parquet files.
Wrong, there is no "DAG format". DAG stands for "directed acyclic graph". The DAG is a means of representing computational steps in Spark. However, it is true that Hadoop does not use a DAG.
The introduction of the DAG in Spark was a result of the limitation of Hadoop's map reduce framework in which data had to be written to and read from disk continuously.
Graph DAG in Apache Spark - DataFlair
Spark achieves great performance by storing data in the HDFS format, whereas Hadoop can only use parquet files.
No. Spark can certainly store data in HDFS (as well as other formats), but this is not a key performance advantage over Hadoop. Hadoop can use multiple file formats, not only parquet.
Spark achieves higher resiliency for queries since, different from Hadoop, it can be deployed on Kubernetes.
No, resiliency is not asked for in the question. The question is about performance improvements.
Both Hadoop and Spark can be deployed on Kubernetes.
Spark achieves performance gains for developers by extending Hadoop's DataFrames with a user-friendly API.
No. DataFrames are a concept in Spark, but not in Hadoop.

NEW QUESTION: 40
Which of the following code blocks returns a 2-column DataFrame that shows the distinct values in column productId and the number of rows with that productId in DataFrame transactionsDf?
A. transactionsDf.count("productId").distinct()
B. transactionsDf.groupBy("productId").agg(col("value").count())
C. transactionsDf.count("productId")
D. transactionsDf.groupBy("productId").count()
E. transactionsDf.groupBy("productId").select(count("value"))
Answer: D (LEAVE A REPLY)
Explanation
transactionsDf.groupBy("productId").count()
Correct. This code block first groups DataFrame transactionsDf by column productId and then counts the rows in each group.
transactionsDf.groupBy("productId").select(count("value"))
Incorrect. You cannot call select on a GroupedData object (the output of a groupBy) statement.
transactionsDf.count("productId")
No. DataFrame.count() does not take any arguments.
transactionsDf.count("productId").distinct()
Wrong. Since DataFrame.count() does not take any arguments, this option cannot be right.
transactionsDf.groupBy("productId").agg(col("value").count())
False. A Column object, as returned by col("value"), does not have a count() method. You can see all available methods for Column object linked in the Spark documentation below.
More info: pyspark.sql.DataFrame.count - PySpark 3.1.2 documentation, pyspark.sql.Column - PySpark

3.1.2 documentation
Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 41
The code block displayed below contains an error. The code block should merge the rows of DataFrames transactionsDfMonday and transactionsDfTuesday into a new DataFrame, matching column names and inserting null values where column names do not appear in both DataFrames. Find the error.
Sample of DataFrame transactionsDfMonday:
1.+-------------+---------+-----+-------+---------+----+
2.|transactionId|predError|value|storeId|productId| f|
3.+-------------+---------+-----+-------+---------+----+
4.| 5| null| null| null| 2|null|
5.| 6| 3| 2| 25| 2|null|
6.+-------------+---------+-----+-------+---------+----+
Sample of DataFrame transactionsDfTuesday:
1.+-------+-------------+---------+-----+
2.|storeId|transactionId|productId|value|
3.+-------+-------------+---------+-----+
4.| 25| 1| 1| 4|
5.| 2| 2| 2| 7|
6.| 3| 4| 2| null|
7.| null| 5| 2| null|
8.+-------+-------------+---------+-----+
Code block:
sc.union([transactionsDfMonday, transactionsDfTuesday])
A. The DataFrames' RDDs need to be passed into the sc.union method instead of the DataFrame variable names.
B. Instead of union, the concat method should be used, making sure to not use its default arguments.
C. Instead of the Spark context, transactionDfMonday should be called with the join method instead of the union method, making sure to use its default arguments.
D. Instead of the Spark context, transactionDfMonday should be called with the union method.
E. Instead of the Spark context, transactionDfMonday should be called with the unionByName method instead of the union method, making sure to not use its default arguments.
Answer: E (LEAVE A REPLY)
Explanation
Correct code block:
transactionsDfMonday.unionByName(transactionsDfTuesday, True)
Output of correct code block:
+-------------+---------+-----+-------+---------+----+
|transactionId|predError|value|storeId|productId| f|
+-------------+---------+-----+-------+---------+----+

5	null	null	null	2	null
6	3	2	25	2	null
1	null	4	25	1	null
2	null	7	2	2	null
4	null	null	3	2	null
5	null	null	null	2	null
+----------+---------+-----+-------+---------+----+

For solving this question, you should be aware of the difference between the DataFrame.union() and DataFrame.unionByName() methods. The first one matches columns independent of their names, just by their order. The second one matches columns by their name (which is asked for in the question). It also has a useful optional argument, allowMissingColumns. This allows you to merge DataFrames that have different columns - just like in this example.

sc stands for SparkContext and is automatically provided when executing code on Databricks. While sc.union() allows you to join RDDs, it is not the right choice for joining DataFrames. A hint away from sc.union() is given where the question talks about joining "into a new DataFrame".

concat is a method in pyspark.sql.functions. It is great for consolidating values from different columns, but has no place when trying to join rows of multiple DataFrames.

Finally, the join method is a contender here. However, the default join defined for that method is an inner join which does not get us closer to the goal to match the two DataFrames as instructed, especially given that with the default arguments we cannot define a join condition.

More info:

- pyspark.sql.DataFrame.unionByName - PySpark 3.1.2 documentation
- pyspark.SparkContext.union - PySpark 3.1.2 documentation
- pyspark.sql.functions.concat - PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 42

Which of the following DataFrame methods is classified as a transformation?

A. DataFrame.count()

B. DataFrame.show()

C. DataFrame.select()

D. DataFrame.foreach()

E. DataFrame.first()

Answer: (SHOW ANSWER)

Explanation

DataFrame.select()

Correct, DataFrame.select() is a transformation. When the command is executed, it is evaluated lazily and returns an RDD when it is triggered by an action.

DataFrame.foreach()

Incorrect, DataFrame.foreach() is not a transformation, but an action. The intention of foreach() is to apply code to each element of a DataFrame to update accumulator variables or write the elements to external storage. The process does not return an RDD - it is an action!
DataFrame.first()
Wrong. As an action, DataFrame.first() executed immediately and returns the first row of a DataFrame.
DataFrame.count()
Incorrect. DataFrame.count() is an action and returns the number of rows in a DataFrame.
DataFrame.show()
No, DataFrame.show() is an action and displays the DataFrame upon execution of the command.

NEW QUESTION: 43
Which of the following statements about DAGs is correct?
A. DAGs help direct how Spark executors process tasks, but are a limitation to the proper execution of a query when an executor fails.
B. DAG stands for "Directing Acyclic Graph".
C. Spark strategically hides DAGs from developers, since the high degree of automation in Spark means that developers never need to consider DAG layouts.
D. In contrast to transformations, DAGs are never lazily executed.
E. DAGs can be decomposed into tasks that are executed in parallel.
Answer: (SHOW ANSWER)
Explanation
DAG stands for "Directing Acyclic Graph".
No, DAG stands for "Directed Acyclic Graph".
Spark strategically hides DAGs from developers, since the high degree of automation in Spark means that developers never need to consider DAG layouts.
No, quite the opposite. You can access DAGs through the Spark UI and they can be of great help when optimizing queries manually.
In contrast to transformations, DAGs are never lazily executed.
DAGs represent the execution plan in Spark and as such are lazily executed when the driver requests the data processed in the DAG.

NEW QUESTION: 44
Which of the following describes the characteristics of accumulators?
A. Accumulators are used to pass around lookup tables across the cluster.
B. All accumulators used in a Spark application are listed in the Spark UI.
C. Accumulators can be instantiated directly via the accumulator(n) method of the pyspark.RDD module.
D. Accumulators are immutable.
E. If an action including an accumulator fails during execution and Spark manages to restart the action and complete it successfully, only the successful attempt will be counted in the accumulator.
Answer: (SHOW ANSWER)
Explanation

If an action including an accumulator fails during execution and Spark manages to restart the action and complete it successfully, only the successful attempt will be counted in the accumulator.
Correct, when Spark tries to rerun a failed action that includes an accumulator, it will only update the accumulator if the action succeeded.
Accumulators are immutable.
No. Although accumulators behave like write-only variables towards the executors and can only be read by the driver, they are not immutable.
All accumulators used in a Spark application are listed in the Spark UI.
Incorrect. For scala, only named, but not unnamed, accumulators are listed in the Spark UI. For pySpark, no accumulators are listed in the Spark UI - this feature is not yet implemented.
Accumulators are used to pass around lookup tables across the cluster.
Wrong - this is what broadcast variables do.
Accumulators can be instantiated directly via the accumulator(n) method of the pyspark.RDD module.
Wrong, accumulators are instantiated via the accumulator(n) method of the sparkContext, for example: counter
= spark.sparkContext.accumulator(0).
More info: python - In Spark, RDDs are immutable, then how Accumulators are implemented? - Stack Overflow, apache spark - When are accumulators truly reliable? - Stack Overflow, Spark - The Definitive Guide, Chapter 14

NEW QUESTION: 45
Which of the following code blocks returns a single-column DataFrame of all entries in Python list throughputRates which contains only float-type values ?
A. spark.createDataFrame((throughputRates), FloatType)
B. spark.createDataFrame(throughputRates, FloatType)
C. spark.DataFrame(throughputRates, FloatType)
D. spark.createDataFrame(throughputRates)
E. spark.createDataFrame(throughputRates, FloatType())
Answer: (SHOW ANSWER)
Explanation
spark.createDataFrame(throughputRates, FloatType())
Correct! spark.createDataFrame is the correct operator to use here and the type FloatType() which is passed in for the command's schema argument is correctly instantiated using the parentheses.
Remember that it is essential in PySpark to instantiate types when passing them to SparkSession.createDataFrame. And, in Databricks, spark returns a SparkSession object.
spark.createDataFrame((throughputRates), FloatType)
No. While packing throughputRates in parentheses does not do anything to the execution of this command, not instantiating the FloatType with parentheses as in the previous answer will make this command fail.
spark.createDataFrame(throughputRates, FloatType)
Incorrect. Given that it does not matter whether you pass throughputRates in parentheses or not, see the explanation of the previous answer for further insights.

spark.DataFrame(throughputRates, FloatType)
Wrong. There is no SparkSession.DataFrame() method in Spark.
spark.createDataFrame(throughputRates)
False. Avoiding the schema argument will have PySpark try to infer the schema. However, as you can see in the documentation (linked below), the inference will only work if you pass in an "RDD of either Row, namedtuple, or dict" for data (the first argument to createDataFrame). But since you are passing a Python list, Spark's schema inference will fail.
More info: pyspark.sql.SparkSession.createDataFrame - PySpark 3.1.2 documentation Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 46
The code block displayed below contains an error. The code block should save DataFrame transactionsDf at path path as a parquet file, appending to any existing parquet file. Find the error.
Code block:
A. transactionsDf.format("parquet").option("mode", "append").save(path)
B. The code block is missing a reference to the DataFrameWriter.
C. save() is evaluated lazily and needs to be followed by an action.
D. The mode option should be omitted so that the command uses the default mode.
E. The code block is missing a bucketBy command that takes care of partitions.
F. Given that the DataFrame should be saved as parquet file, path is being passed to the wrong method.
Answer: (SHOW ANSWER)
Explanation
Correct code block:
transactionsDf.write.format("parquet").option("mode", "append").save(path)

NEW QUESTION: 47
Which of the following code blocks reorders the values inside the arrays in column attributes of DataFrame itemsDf from last to first one in the alphabet?
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.+------+--------------------------+------------------+
A. itemsDf.withColumn('attributes', sort_array(col('attributes').desc()))
B. itemsDf.withColumn('attributes', sort_array(desc('attributes')))
C. itemsDf.withColumn('attributes', sort(col('attributes'), asc=False))
D. itemsDf.withColumn("attributes", sort_array("attributes", asc=False))
E. itemsDf.select(sort_array("attributes"))
Answer: D (LEAVE A REPLY)
Explanation
Output of correct code block:
+------+--------------------------+------------------+
|itemId|attributes |supplier |
+------+--------------------------+------------------+
1	[winter, cozy, blue]	Sports Company Inc.
2	[summer, red, fresh, cooling]	YetiX
3	[travel, summer, green]	Sports Company Inc.
+------+--------------------------+------------------+
It can be confusing to differentiate between the different sorting functions in PySpark. In this case, a particularity about sort_array has to be considered: The sort direction is given by the second argument, not by the desc method. Luckily, this is documented in the documentation (link below). Also, for solving this question you need to understand the difference between sort and sort_array. With sort, you cannot sort values in arrays. Also, sort is a method of DataFrame, while sort_array is a method of pyspark.sql.functions.
More info: pyspark.sql.functions.sort_array - PySpark 3.1.2 documentation Static notebook | Dynamic notebook: See test 2

NEW QUESTION: 48
Which of the following code blocks reads in the JSON file stored at filePath, enforcing the schema expressed in JSON format in variable json_schema, shown in the code block below?
Code block:
1.json_schema = """
2.{"type": "struct",
3. "fields": [
4. {
5. "name": "itemId",
6. "type": "integer",
7. "nullable": true,
8. "metadata": {}
9. },
10. {
11. "name": "supplier",

12. "type": "string",
13. "nullable": true,
14. "metadata": {}
15. }
16.]
17.}
18."""
A. spark.read.json(filePath, schema=json_schema)
B. spark.read.schema(json_schema).json(filePath)
1.schema = StructType.fromJson(json.loads(json_schema))
2.spark.read.json(filePath, schema=schema)
C. spark.read.json(filePath, schema=schema_of_json(json_schema))
D. spark.read.json(filePath, schema=spark.read.json(json_schema))
Answer: (SHOW ANSWER)
Explanation
Spark provides a way to digest JSON-formatted strings as schema. However, it is not trivial to use. Although slightly above exam difficulty, this question is beneficial to your exam preparation, since it helps you to familiarize yourself with the concept of enforcing schemas on data you are reading in - a topic within the scope of the exam.
The first answer that jumps out here is the one that uses spark.read.schema instead of spark.read.json. Looking at the documentation of spark.read.schema (linked below), we notice that the operator expects types pyspark.sql.types.StructType or str as its first argument. While variable json_schema is a string, the documentation states that the str should be "a DDL-formatted string (For example col0 INT, col1 DOUBLE)". Variable json_schema does not contain a string in this type of format, so this answer option must be wrong.
With four potentially correct answers to go, we now look at the schema parameter of spark.read.json() (documentation linked below). Here, too, the schema parameter expects an input of type pyspark.sql.types.StructType or "a DDL-formatted string (For example col0 INT, col1 DOUBLE)". We already know that json_schema does not follow this format, so we should focus on how we can transform json_schema into pyspark.sql.types.StructType. Hereby, we also eliminate the option where schema=json_schema.
The option that includes schema=spark.read.json(json_schema) is also a wrong pick, since spark.read.json returns a DataFrame, and not a pyspark.sql.types.StructType type.
Ruling out the option which includes schema_of_json(json_schema) is rather difficult. The operator's documentation (linked below) states that it "[p]arses a JSON string and infers its schema in DDL format". This use case is slightly different from the case at hand: json_schema already is a schema definition, it does not make sense to "infer" a schema from it. In the documentation you can see an example use case which helps you understand the difference better. Here, you pass string '{a: 1}' to schema_of_json() and the method infers a DDL-format schema STRUCT<a: BIGINT> from it.
In our case, we may end up with the output schema of schema_of_json() describing the schema of the JSON schema, instead of using the schema itself. This is not the right answer option.

Now you may consider looking at the StructType.fromJson() method. It returns a variable of type StructType - exactly the type which the schema parameter of spark.read.json expects.
Although we could have looked at the correct answer option earlier, this explanation is kept as exhaustive as necessary to teach you how to systematically eliminate wrong answer options.
More info:
- pyspark.sql.DataFrameReader.schema - PySpark 3.1.2 documentation
- pyspark.sql.DataFrameReader.json - PySpark 3.1.2 documentation
- pyspark.sql.functions.schema_of_json - PySpark 3.1.2 documentation
Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 49
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"))
A. Instead of select, DataFrame transactionsDf needs to be filtered using the filter operator.
B. Columns storeId and predError need to be represented as a Python list, so they need to be wrapped in brackets ([]).
C. The take method should be used instead of the collect method.
D. Instead of collect, collectAsRows needs to be called.
E. The collect method is not a method of the SparkSession object.
Answer: (SHOW ANSWER)
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

NEW QUESTION: 50
Which of the following describes Spark's standalone deployment mode?
A. Standalone mode uses a single JVM to run Spark driver and executor processes.
B. Standalone mode means that the cluster does not contain the driver.
C. Standalone mode is how Spark runs on YARN and Mesos clusters.
D. Standalone mode uses only a single executor per worker per application.
E. Standalone mode is a viable solution for clusters that run multiple frameworks, not only Spark.
Answer: D (LEAVE A REPLY)
Explanation
Standalone mode uses only a single executor per worker per application.
This is correct and a limitation of Spark's standalone mode.
Standalone mode is a viable solution for clusters that run multiple frameworks.

Incorrect. A limitation of standalone mode is that Apache Spark must be the only framework running on the cluster. If you would want to run multiple frameworks on the same cluster in parallel, for example Apache Spark and Apache Flink, you would consider the YARN deployment mode.
Standalone mode uses a single JVM to run Spark driver and executor processes.
No, this is what local mode does.
Standalone mode is how Spark runs on YARN and Mesos clusters.
No. YARN and Mesos modes are two deployment modes that are different from standalone mode. These modes allow Spark to run alongside other frameworks on a cluster. When Spark is run in standalone mode, only the Spark framework can run on the cluster.
Standalone mode means that the cluster does not contain the driver.
Incorrect, the cluster does not contain the driver in client mode, but in standalone mode the driver runs on a node in the cluster.
More info: Learning Spark, 2nd Edition, Chapter 1

NEW QUESTION: 51
The code block shown below should return a copy of DataFrame transactionsDf with an added column cos. This column should have the values in column value converted to degrees and having the cosine of those converted values taken, rounded to two decimals. Choose the answer that correctly fills the blanks in the code block to accomplish this.
Code block:
transactionsDf.__1__(__2__, round(__3__(__4__(__5__)),2))
A. 1. withColumn
2. col("cos")
3. cos
4. degrees
5. transactionsDf.value
B. 1. withColumnRenamed
2. "cos"
3. cos
4. degrees
5. "transactionsDf.value"
C. 1. withColumn
2. "cos"
3. cos
4. degrees
5. transactionsDf.value
D. 1. withColumn
2. col("cos")
3. cos
4. degrees
5. col("value")

E
. 1. withColumn
2. "cos"
3. degrees
4. cos
5. col("value")
Answer: (SHOW ANSWER)
Explanation
Correct code block:
transactionsDf.withColumn("cos", round(cos(degrees(transactionsDf.value)),2)) This question is especially confusing because col, "cos" are so similar. Similar-looking answer options can also appear in the exam and, just like in this question, you need to pay attention to the details to identify what the correct answer option is.
The first answer option to throw out is the one that starts with withColumnRenamed: The question NO: speaks specifically of adding a column. The withColumnRenamed operator only renames an existing column, however, so you cannot use it here.
Next, you will have to decide what should be in gap 2, the first argument of transactionsDf.withColumn(). Looking at the documentation (linked below), you can find out that the first argument of withColumn actually needs to be a string with the name of the column to be added. So, any answer that includes col("cos") as the option for gap 2 can be disregarded.
This leaves you with two possible answers. The real difference between these two answers is where the cos and degree methods are, either in gaps 3 and 4, or vice-versa. From the question you can find out that the new column should have "the values in column value converted to degrees and having the cosine of those converted values taken". This prescribes you a clear order of operations: First, you convert values from column value to degrees and then you take the cosine of those values. So, the inner parenthesis (gap 4) should contain the degree method and then, logically, gap 3 holds the cos method. This leaves you with just one possible correct answer.
More info: pyspark.sql.DataFrame.withColumn - PySpark 3.1.2 documentation Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 52
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?
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: (SHOW ANSWER)
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

NEW QUESTION: 53
The code block shown below should return a DataFrame with all columns of DataFrame transactionsDf, but only maximum 2 rows in which column productId has at least the value 2. Choose the answer that correctly fills the blanks in the code block to accomplish this.
transactionsDf.__1__(__2__).__3__
A. 1. where
2. "productId" > 2
3. max(2)
B. 1. where
2. transactionsDf[productId] >= 2
3. limit(2)
C. 1. filter
2. productId > 2
3. max(2)
D. 1. filter
2. col("productId") >= 2
3. limit(2)
E. 1. where
2. productId >= 2
3. limit(2)
Answer: D (LEAVE A REPLY)
Explanation
Correct code block:
transactionsDf.filter(col("productId") >= 2).limit(2)
The filter and where operators in gap 1 are just aliases of one another, so you cannot use them to pick the right answer.
The column definition in gap 2 is more helpful. The DataFrame.filter() method takes an argument of type Column or str. From all possible answers, only the one including col("productId") >= 2 fits this profile, since it returns a Column type.
The answer option using "productId" > 2 is invalid, since Spark does not understand that "productId" refers to column productId. The answer option using transactionsDf[productId] >= 2 is wrong because you cannot refer to a column using square bracket notation in Spark (if you are coming from Python using Pandas, this is

something to watch out for). In all other options, productId is being referred to as a Python variable, so they are relatively easy to eliminate.
Also note that the question asks for the value in column productId being at least 2. This translates to a "greater or equal" sign (>= 2), but not a "greater" sign (> 2).
Another thing worth noting is that there is no DataFrame.max() method. If you picked any option including this, you may be confusing it with the pyspark.sql.functions.max method. The correct method to limit the amount of rows is the DataFrame.limit() method.
More info:
- pyspark.sql.DataFrame.filter - PySpark 3.1.2 documentation
- pyspark.sql.DataFrame.limit - PySpark 3.1.2 documentation
Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 54
The code block shown below should add a column itemNameBetweenSeparators to DataFrame itemsDf. The column should contain arrays of maximum 4 strings. The arrays should be composed of the values in column itemsDf which are separated at - or whitespace characters. Choose the answer that correctly fills the blanks in the code block to accomplish this.
Sample of DataFrame itemsDf:
1.+------+---------------------------------+------------------+
2.|itemId|itemName |supplier |
3.+------+---------------------------------+------------------+
4.|1 |Thick Coat for Walking in the Snow|Sports Company Inc.|
5.|2 |Elegant Outdoors Summer Dress |YetiX |
6.|3 |Outdoors Backpack |Sports Company Inc.|
7.+------+---------------------------------+------------------+
Code block:
itemsDf.__1__(__2__, __3__(__4__, "[\s\-]", __5__))
A. 1. withColumn
2. "itemNameBetweenSeparators"
3. split
4. "itemName"
5. 4
(Correct)
B. 1. withColumnRenamed
2. "itemNameBetweenSeparators"
3. split
4. "itemName"
5. 4
C. 1. withColumnRenamed
2. "itemName"
3. split

4. "itemNameBetweenSeparators"
5. 4

D. 1. withColumn
2. "itemNameBetweenSeparators"
3. split
4. "itemName"
5. 5

E. 1. withColumn
2. itemNameBetweenSeparators
3. str_split
4. "itemName"
5. 5

Answer: A (LEAVE A REPLY)

Explanation

This question deals with the parameters of Spark's split operator for strings.

To solve this question, you first need to understand the difference between DataFrame.withColumn() and DataFrame.withColumnRenamed(). The correct option here is DataFrame.withColumn() since, according to the question, we want to add a column and not rename an existing column. This leaves you with only 3 answers to consider.

The second gap should be filled with the name of the new column to be added to the DataFrame. One of the remaining answers states the column name as itemNameBetweenSeparators, while the other two state it as "itemNameBetweenSeparators". The correct option here is "itemNameBetweenSeparators", since the other option would let Python try to interpret itemNameBetweenSeparators as the name of a variable, which we have not defined. This leaves you with 2 answers to consider.

The decision boils down to how to fill gap 5. Either with 4 or with 5. The question asks for arrays of maximum four strings. The code in gap 5 relates to the limit parameter of Spark's split operator (see documentation linked below). The documentation states that "the resulting array's length will not be more than limit", meaning that we should pick the answer option with 4 as the code in the fifth gap here.

On a side note: One answer option includes a function str_split. This function does not exist in pySpark.

More info: pyspark.sql.functions.split - PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 55

Which of the following describes a shuffle?

A. A shuffle is a process that is executed during a broadcast hash join.

B. A shuffle is a process that compares data across executors.

C. A shuffle is a process that compares data across partitions.

D. A shuffle is a Spark operation that results from DataFrame.coalesce().

E. A shuffle is a process that allocates partitions to executors.

Answer: C (LEAVE A REPLY)

Explanation
A shuffle is a Spark operation that results from DataFrame.coalesce().
No. DataFrame.coalesce() does not result in a shuffle.
A shuffle is a process that allocates partitions to executors.
This is incorrect.
A shuffle is a process that is executed during a broadcast hash join.
No, broadcast hash joins avoid shuffles and yield performance benefits if at least one of the two tables is small in size (<= 10 MB by default). Broadcast hash joins can avoid shuffles because instead of exchanging partitions between executors, they broadcast a small table to all executors that then perform the rest of the join operation locally.
A shuffle is a process that compares data across executors.
No, in a shuffle, data is compared across partitions, and not executors.
More info: Spark Repartition & Coalesce - Explained (https://bit.ly/32KF7zS)

NEW QUESTION: 56
The code block displayed below contains multiple errors. The code block should remove column transactionDate from DataFrame transactionsDf and add a column transactionTimestamp in which dates that are expressed as strings in column transactionDate of DataFrame transactionsDf are converted into unix timestamps. Find the errors.
Sample of DataFrame transactionsDf:
1.+-------------+---------+-----+-------+---------+----+----------------+
2.|transactionId|predError|value|storeId|productId| f| transactionDate|
3.+-------------+---------+-----+-------+---------+----+----------------+
4.| 1| 3| 4| 25| 1|null|2020-04-26 15:35|
5.| 2| 6| 7| 2| 2|null|2020-04-13 22:01|
6.| 3| 3| null| 25| 3|null|2020-04-02 10:53|
7.+-------------+---------+-----+-------+---------+----+----------------+ Code block:
1.transactionsDf = transactionsDf.drop("transactionDate")
2.transactionsDf["transactionTimestamp"] = unix_timestamp("transactionDate", "yyyy-MM-dd")
A. Column transactionDate should be dropped after transactionTimestamp has been written. The string indicating the date format should be adjusted. The withColumn operator should be used instead of the existing column assignment. Operator to_unixtime() should be used instead of unix_timestamp().
B. Column transactionDate should be dropped after transactionTimestamp has been written. The withColumn operator should be used instead of the existing column assignment. Column transactionDate should be wrapped in a col() operator.
C. Column transactionDate should be wrapped in a col() operator.
D. The string indicating the date format should be adjusted. The withColumnReplaced operator should be used instead of the drop and assign pattern in the code block to replace column transactionDate with the new column transactionTimestamp.

E. Column transactionDate should be dropped after transactionTimestamp has been written. The string indicating the date format should be adjusted. The withColumn operator should be used instead of the existing column assignment.

Answer: E (LEAVE A REPLY)

Explanation

This question requires a lot of thinking to get right. For solving it, you may take advantage of the digital notepad that is provided to you during the test. You have probably seen that the code block includes multiple errors. In the test, you are usually confronted with a code block that only contains a single error. However, since you are practicing here, this challenging multi-error question will make it easier for you to deal with single-error questions in the real exam.

You can clearly see that column transactionDate should be dropped only after transactionTimestamp has been written. This is because to generate column transactionTimestamp, Spark needs to read the values from column transactionDate.

Values in column transactionDate in the original transactionsDf DataFrame look like 2020-04-26 15:35. So, to convert those correctly, you would have to pass yyyy-MM-dd HH:mm. In other words:

The string indicating the date format should be adjusted.

While you might be tempted to change unix_timestamp() to to_unixtime() (in line with the from_unixtime() operator), this function does not exist in Spark. unix_timestamp() is the correct operator to use here.

Also, there is no DataFrame.withColumnReplaced() operator. A similar operator that exists is DataFrame.withColumnRenamed().

Whether you use col() or not is irrelevant with unix_timestamp() - the command is fine with both.

Finally, you cannot assign a column like transactionsDf["columnName"] = ... in Spark. This is Pandas syntax (Pandas is a popular Python package for data analysis), but it is not supported in Spark.

So, you need to use Spark's DataFrame.withColumn() syntax instead.

More info: pyspark.sql.functions.unix_timestamp - PySpark 3.1.2 documentation Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 57

The code block displayed below contains an error. The code block should trigger Spark to cache DataFrame transactionsDf in executor memory where available, writing to disk where insufficient executor memory is available, in a fault-tolerant way. Find the error.

Code block:

transactionsDf.persist(StorageLevel.MEMORY_AND_DISK)

A. Caching is not supported in Spark, data are always recomputed.

B. Data caching capabilities can be accessed through the spark object, but not through the DataFrame API.

C. The storage level is inappropriate for fault-tolerant storage.

D. The code block uses the wrong operator for caching.

E. The DataFrameWriter needs to be invoked.

Answer: (SHOW ANSWER)

Explanation

The storage level is inappropriate for fault-tolerant storage.

Correct. Typically, when thinking about fault tolerance and storage levels, you would want to store redundant copies of the dataset. This can be achieved by using a storage level such as StorageLevel.MEMORY_AND_DISK_2.
The code block uses the wrong command for caching.
Wrong. In this case, DataFrame.persist() needs to be used, since this operator supports passing a storage level.
DataFrame.cache() does not support passing a storage level.
Caching is not supported in Spark, data are always recomputed.
Incorrect. Caching is an important component of Spark, since it can help to accelerate Spark programs to great extent. Caching is often a good idea for datasets that need to be accessed repeatedly.
Data caching capabilities can be accessed through the spark object, but not through the DataFrame API.
No. Caching is either accessed through DataFrame.cache() or DataFrame.persist().
The DataFrameWriter needs to be invoked.
Wrong. The DataFrameWriter can be accessed via DataFrame.write and is used to write data to external data stores, mostly on disk. Here, we find keywords such as "cache" and "executor memory" that point us away from using external data stores. We aim to save data to memory to accelerate the reading process, since reading from disk is comparatively slower. The DataFrameWriter does not write to memory, so we cannot use it here.
More info: Best practices for caching in Spark SQL | by David Vrba | Towards Data Science

NEW QUESTION: 58
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.+------+----------------------------+------------------+
A. 1.itemsDfSchema = StructType([
2. StructField("itemId", IntegerType()),
3. StructField("attributes", StringType()),
4. StructField("supplier", StringType())])
5.
6.itemsDf = spark.read.schema(itemsDfSchema).parquet(filePath)
B. 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)
C. 1.itemsDf = spark.read.schema('itemId integer, attributes <string>, supplier string').parquet(filePath)
D. 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)
E. 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)
Answer: (SHOW ANSWER)
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

NEW QUESTION: 59
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()
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 (LEAVE A REPLY)
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

NEW QUESTION: 60
Which of the following code blocks returns a DataFrame that matches the multi-column DataFrame itemsDf, except that integer column itemId has been converted into a string column?
A. itemsDf.withColumn("itemId", convert("itemId", "string"))
B. itemsDf.withColumn("itemId", col("itemId").cast("string"))
(Correct)
C. itemsDf.select(cast("itemId", "string"))
D. itemsDf.withColumn("itemId", col("itemId").convert("string"))
E. spark.cast(itemsDf, "itemId", "string")
Answer: B (LEAVE A REPLY)
Explanation
itemsDf.withColumn("itemId", col("itemId").cast("string"))
Correct. You can convert the data type of a column using the cast method of the Column class. Also note that you will have to use the withColumn method on itemsDf for replacing the existing itemId column with the new version that contains strings.
itemsDf.withColumn("itemId", col("itemId").convert("string"))
Incorrect. The Column object that col("itemId") returns does not have a convert method.
itemsDf.withColumn("itemId", convert("itemId", "string"))
Wrong. Spark's spark.sql.functions module does not have a convert method. The question is trying to mislead you by using the word "converted". Type conversion is also called "type casting". This may help you remember to look for a cast method instead of a convert method (see correct answer).
itemsDf.select(astype("itemId", "string"))
False. While astype is a method of Column (and an alias of Column.cast), it is not a method of pyspark.sql.functions (what the code block implies). In addition, the question asks to return a full DataFrame that matches the multi-column DataFrame itemsDf. Selecting just one column from itemsDf as in the code block would just return a single-column DataFrame.
spark.cast(itemsDf, "itemId", "string")
No, the Spark session (called by spark) does not have a cast method. You can find a list of all methods available for the Spark session linked in the documentation below.
More info:
- pyspark.sql.Column.cast - PySpark 3.1.2 documentation
- pyspark.sql.Column.astype - PySpark 3.1.2 documentation

- pyspark.sql.SparkSession - PySpark 3.1.2 documentation
Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 61
Which of the following code blocks returns all unique values across all values in columns value and productId in DataFrame transactionsDf in a one-column DataFrame?
A. tranactionsDf.select('value').join(transactionsDf.select('productId'), col('value')==col('productId'), 'outer')
B. transactionsDf.select(col('value'), col('productId')).agg({'*': 'count'})
C. transactionsDf.select('value', 'productId').distinct()
D. transactionsDf.select('value').union(transactionsDf.select('productId')).distinct()
E. transactionsDf.agg({'value': 'collect_set', 'productId': 'collect_set'})
Answer: (SHOW ANSWER)
Explanation
transactionsDf.select('value').union(transactionsDf.select('productId')).distinct() Correct. This code block uses a common pattern for finding the unique values across multiple columns: union and distinct. In fact, it is so common that it is even mentioned in the Spark documentation for the union command (link below).
transactionsDf.select('value', 'productId').distinct()
Wrong. This code block returns unique rows, but not unique values.
transactionsDf.agg({'value': 'collect_set', 'productId': 'collect_set'}) Incorrect. This code block will output a one-row, two-column DataFrame where each cell has an array of unique values in the respective column (even omitting any nulls).
transactionsDf.select(col('value'), col('productId')).agg({'*': 'count'}) No. This command will count the number of rows, but will not return unique values.
transactionsDf.select('value').join(transactionsDf.select('productId'), col('value')==col('productId'), 'outer')
Wrong. This command will perform an outer join of the value and productId columns. As such, it will return a two-column DataFrame. If you picked this answer, it might be a good idea for you to read up on the difference between union and join, a link is posted below.
More info: pyspark.sql.DataFrame.union - PySpark 3.1.2 documentation, sql - What is the difference between JOIN and UNION? - Stack Overflow Static notebook | Dynamic notebook: See test 3

NEW QUESTION: 62

Which of the following code blocks saves DataFrame transactionsDf in location /FileStore/transactions.csv as a CSV file and throws an error if a file already exists in the location?

A. transactionsDf.write.save("/FileStore/transactions.csv")

B. transactionsDf.write.format("csv").mode("error").path("/FileStore/transactions.csv")

C. transactionsDf.write.format("csv").mode("ignore").path("/FileStore/transactions.csv")

D. transactionsDf.write("csv").mode("error").save("/FileStore/transactions.csv")

E. transactionsDf.write.format("csv").mode("error").save("/FileStore/transactions.csv")

Answer: (SHOW ANSWER)

Explanation

Static notebook | Dynamic notebook: See test 1

(https://flrs.github.io/spark_practice_tests_code/#1/28.html ,

https://bit.ly/sparkpracticeexams_import_instructions)