DUMPSARENA

Databricks Certified Associate Developer for Apache Spark 3.0 Exam

Databricks Databricks-Certified-Associate-Developer-for-Apache-Spark-3.0

Version Demo

Total Demo Questions: 2

Total Premium Questions: 23

Buy Premium PDF

https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

Topic Break Down

Topic	No. of Questions
Topic 1, Apache Spark DataFrames	21
Topic 2, Apache Spark SQL	1
Topic 3, Apache Spark Core	1
Total	23

QUESTION NO: 1

Which of the following code blocks returns a DataFrame where columns predError and productId are removed from DataFrame transactionsDf?

Sample of 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.withColumnRemoved("predError", "productId")

B. transactionsDf.drop(["predError", "productId", "associateId"])

C. transactionsDf.drop("predError", "productId", "associateId")

D. transactionsDf.dropColumns("predError", "productId", "associateId")

E. transactionsDf.drop(col("predError", "productId"))

ANSWER: C

Explanation:

`transactionsDf.drop("predError", "productId", "associateId")` is correct because PySpark's `DataFrame.drop()` method accepts a variable number of column names as separate arguments. Supplying the strings `"predError"` and `"productId"` therefore produces a new DataFrame without those two fields; DataFrame transformations are immutable, so `transactionsDf` itself is not modified. The resulting DataFrame retains the remaining columns, such as `transactionId`, `value`, `storeId`, and `f`. The additional string `"associateId"` does not prevent the operation from succeeding: when a string column name supplied to `drop` is absent from the schema, Spark ignores it. This makes the call safe when a known optional column may or may not be present. The API defines `drop` with a variadic column argument structure, allowing multiple string names in one invocation. See the Apache Spark PySpark 3.0 DataFrame.drop API reference and the current PySpark DataFrame.drop documentation.

QUESTION NO: 2

DataFrame readingsDf contains `deviceId`, `readingTime`, and `temperature`. A developer must calculate the difference between each temperature reading and the preceding reading for the same device, ordered by `readingTime`. The first reading for each device must have a null difference. Which code block returns the required DataFrame?

A.
```
w = Window.partitionBy("deviceId").orderBy("readingTime")
readingsDf.withColumn("temperatureDifference", col("temperature") -
lead("temperature").over(w))
```

B.
```
w = Window.orderBy("readingTime")
readingsDf.withColumn("temperatureDifference", col("temperature") -
lag("temperature").over(w))
```

C.
```
w = Window.partitionBy("deviceId").orderBy("readingTime")
readingsDf.withColumn("temperatureDifference", lag("temperature").over(w) -
col("temperature"))
```

D.
```
w = Window.partitionBy("deviceId").orderBy("readingTime")
readingsDf.withColumn("temperatureDifference", col("temperature") -
lag("temperature").over(w))
```

ANSWER: D

Explanation:

`lag("temperature")` accesses the prior row within each device partition after the rows are ordered by `readingTime`. Subtracting that prior temperature from the current temperature produces the required difference. For the first row in each partition, `lag()` returns null, so the resulting difference is null.

Using `lead()` accesses the following reading instead of the preceding one. Omitting `partitionBy("deviceId")` would allow a reading from a different device to be used as the prior value.