Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5 Practice Questions

Shared by Daniel on 18-08-2026

For More Free Questions and Preparation Resources

Check the Links on Last Page

Question 1

Question Type: MultipleChoice

Given the code:

```
df = spark.read.csv("large_dataset.csv")
filtered_df = df.filter(col("error_column")
                .contains("error"))
mapped_df = filtered_df.select(split(col("timestamp"), " ").getItem(0).alias("date"),
lit(1).alias("count"))
reduced_df = mapped_df.groupBy("date").sum("count")
reduced_df.count()
reduced_df.show()
```

df = spark.read.csv("large_dataset.csv")

filtered_df = df.filter(col("error_column").contains("error"))

mapped_df = filtered_df.select(split(col("timestamp"), " ").getItem(0).alias("date"),
lit(1).alias("count"))

reduced_df = mapped_df.groupBy("date").sum("count")

reduced_df.count()

reduced_df.show()

At which point will Spark actually begin processing the data?

Options:

A- When the filter transformation is applied
B- When the count action is applied
C- When the groupBy transformation is applied
D- When the show action is applied

Answer:

B

Explanation:

Spark uses lazy evaluation. Transformations like filter, select, and groupBy only define the DAG (Directed Acyclic Graph). No execution occurs until an action is triggered.

The first action in the code is:reduced_df.count()

So Spark starts processing data at this line.

Question 2

Question Type: MultipleChoice

A data scientist has identified that some records in the user profile table contain null values in any of the fields, and such records should be removed from the dataset before processing. The schema includes fields like user_id, username, date_of_birth, created_ts, etc.

The schema of the user profile table looks like this:

```
user_id STRING,
username STRING,
full_name STRING,
date_of_birth DATE,
primary_email STRING,
created_ts TIMESTAMP,
updated_ts TIMESTAMP,
last_login_ts TIMESTAMP
```

Which block of Spark code can be used to achieve this requirement?

Options:

Options:

A- filtered_df = users_raw_df.na.drop(thresh=0)
B- filtered_df = users_raw_df.na.drop(how='all')
C- filtered_df = users_raw_df.na.drop(how='any')
D- filtered_df = users_raw_df.na.drop(how='all', thresh=None)

Answer:

C

Explanation:

.na.drop(how='any') drops any row that has at least one null value.

This is exactly what's needed when the goal is to retain only fully complete records.

Usage:CopyEdit

filtered_df = users_raw_df.na.drop(how='any')

Explanation of incorrect options:

A: thresh=0 is invalid --- thresh must be 1.

B: how='all' drops only rows where all columns are null (too lenient).

D: spark.na.drop doesn't support mixing how and thresh in that way; it's incorrect syntax.

Question 3

Question Type: MultipleChoice

18 of 55.

An engineer has two DataFrames --- df1 (small) and df2 (large). To optimize the join, the engineer uses a broadcast join:

from pyspark.sql.functions import broadcast

df_result = df2.join(broadcast(df1), on="id", how="inner")

What is the purpose of using broadcast() in this scenario?

Options:

A- It increases the partition size for df1 and df2.
B- It ensures that the join happens only when the id values are identical.
C- It reduces the number of shuffle operations by replicating the smaller DataFrame to all nodes.
D- It filters the id values before performing the join.

Answer:

C

Explanation:

A broadcast join is a type of join where the smaller DataFrame is replicated (broadcast) to all

worker nodes in the cluster. This avoids shuffling the large DataFrame across the network.

Benefits:

Eliminates shuffle for the smaller dataset.

Greatly improves performance when one side of the join is small enough to fit in memory.

Correct usage example:

df_result = df2.join(broadcast(df1), 'id')

This is a map-side join, where each executor joins its local partition of the large dataset with the broadcasted copy of the small one.

Why the other options are incorrect:

A: Broadcasting does not change partition sizes.

B: Joins always match on key equality; this is not specific to broadcast joins.

D: Broadcasting does not filter; it distributes data for faster joins.

Databricks Exam Guide (June 2025): Section ''Developing Apache Spark DataFrame/DataSet API Applications'' --- broadcast joins and partitioning strategies.

PySpark SQL Functions --- broadcast() method documentation.

==========

Question 4

Question Type: MultipleChoice

A developer runs:

```
df.write.partitionBy("color", "fruit").parquet("/path/to/output")
```

What is the result?

Options:

Options:

A- It stores all data in a single Parquet file.

B- It throws an error if there are null values in either partition column.
C- It appends new partitions to an existing Parquet file.
D- It creates separate directories for each unique combination of color and fruit.

Answer:

D

Explanation:

The partitionBy() method in Spark organizes output into subdirectories based on unique combinations of the specified columns:

e.g.

/path/to/output/color=red/fruit=apple/part-0000.parquet

/path/to/output/color=green/fruit=banana/part-0001.parquet

This improves query performance via partition pruning.

It does not consolidate into a single file.

Null values are allowed in partitions.

It does not 'append' unless .mode('append') is used.

Question 5

Question Type: MultipleChoice

An engineer has a large ORC file located at /file/test_data.orc and wants to read only specific columns to reduce memory usage.

Which code fragment will Select the columns, i.e., col1, col2, during the reading process?

Options:

A- spark.read.orc('/file/test_data.orc').filter('col1 = 'value' ').select('col2')
B- spark.read.format('orc').select('col1', 'col2').load('/file/test_data.orc')
C- spark.read.orc('/file/test_data.orc').selected('col1', 'col2')
D- spark.read.format('orc').load('/file/test_data.orc').select('col1', 'col2')

Answer:

D

Explanation:

The correct way to load specific columns from an ORC file is to first load the file using .load() and then apply .select() on the resulting DataFrame. This is valid with .read.format('orc') or the shortcut .read.orc().

df = spark.read.format('orc').load('/file/test_data.orc').select('col1', 'col2')

Why others are incorrect:

A performs selection after filtering, but doesn't match the intention to minimize memory at load.

B incorrectly tries to use .select() before .load(), which is invalid.

C uses a non-existent .selected() method.

D correctly loads and then selects.

Question 6

Question Type: MultipleChoice

47 of 55.

A data engineer has written the following code to join two DataFrames df1 and df2:

df1 = spark.read.csv("sales_data.csv")

df2 = spark.read.csv("product_data.csv")

df_joined = df1.join(df2, df1.product_id == df2.product_id)

The DataFrame df1 contains ~10 GB of sales data, and df2 contains ~8 MB of product data.

Which join strategy will Spark use?

Options:

A- Shuffle join, as the size difference between df1 and df2 is too large for a broadcast join to work efficiently.
B- Shuffle join, because AQE is not enabled, and Spark uses a static query plan.

C- Shuffle join because no broadcast hints were provided.
D- Broadcast join, as df2 is smaller than the default broadcast threshold.

Answer:

D

Explanation:

Spark automatically uses a broadcast hash join when one side of the join is small enough to fit within the broadcast threshold.

Default threshold:

spark.sql.autoBroadcastJoinThreshold = 10MB (as of Spark 3.5)

Since df2 is 8 MB, Spark automatically broadcasts it to all executors. This avoids a shuffle on the large dataset (df1) and speeds up the join.

Why the other options are incorrect:

A: 8 MB < 10 MB threshold broadcast join is efficient.

B: AQE is not required for broadcast joins; it's a static optimization.

C: Broadcast hints are optional --- Spark infers automatically.

Databricks Exam Guide (June 2025): Section ''Troubleshooting and Tuning Apache Spark DataFrame API Applications'' --- broadcast joins and optimization.

Spark SQL Join Strategies --- Broadcast hash join and shuffle join thresholds.

==========

Question 7

Question Type: MultipleChoice

A developer is trying to join two tables, sales.purchases_fct and sales.customer_dim, using the following code:

```
import pyspark.sql.functions as F

purch_df = spark.table('sales.purchases_fct')
cust_df = spark.table('sales.customer_dim').dropDuplicates(['cust_id'])

fact_df = purch_df.join(cust_df, F.col('customer_id') == F.col('cust_id'))
```

fact_df = purch_df.join(cust_df, F.col('customer_id') == F.col('custid'))

The developer has discovered that customers in the purchases_fct table that do not exist in the customer_dim table are being dropped from the joined table.

Which change should be made to the code to stop these customer records from being dropped?

Options:

A- fact_df = purch_df.join(cust_df, F.col('customer_id') == F.col('custid'), 'left')
B- fact_df = cust_df.join(purch_df, F.col('customer_id') == F.col('custid'))
C- fact_df = purch_df.join(cust_df, F.col('cust_id') == F.col('customer_id'))
D- fact_df = purch_df.join(cust_df, F.col('customer_id') == F.col('custid'), 'right_outer')

Answer:

A

Explanation:

In Spark, the default join type is an inner join, which returns only the rows with matching keys in both DataFrames. To retain all records from the left DataFrame (purch_df) and include matching records from the right DataFrame (cust_df), a left outer join should be used.

By specifying the join type as 'left', the modified code ensures that all records from purch_df are preserved, and matching records from cust_df are included. Records in purch_df without a corresponding match in cust_df will have null values for the columns from cust_df.

This approach is consistent with standard SQL join operations and is supported in PySpark's DataFrame API.

Question 8

Question Type: MultipleChoice

A data engineer needs to persist a file-based data source to a specific location. However, by

default, Spark writes to the warehouse directory (e.g., /user/hive/warehouse). To override this, the engineer must explicitly define the file path.

Which line of code ensures the data is saved to a specific location?

Options:

Options:

A- users.write(path='/some/path').saveAsTable('default_table')
B- users.write.saveAsTable('default_table').option('path', '/some/path')
C- users.write.option('path', '/some/path').saveAsTable('default_table')
D- users.write.saveAsTable('default_table', path='/some/path')

Answer:

C

Explanation:

To persist a table and specify the save path, use:

users.write.option('path', '/some/path').saveAsTable('default_table')

The .option('path', ...) must be applied before calling saveAsTable.

Option A uses invalid syntax (write(path=...)).

Option B applies .option() after .saveAsTable()---which is too late.

Option D uses incorrect syntax (no path parameter in saveAsTable).