DATABRICKS-CERTIFIED-ASSOCIAT

Q&As

Databricks Certified Associate Developer for Apache Spark 3.0

Pass Databricks DATABRICKS-CERTIFIED-ASSOCIATE-DEVELOPER-FOR-APACHE-SPARK Exam with 100% Guarantee

Free Download Real Questions & Answers **PDF** and **VCE** file from:

https://www.geekcert.com/databricks-certified-associate-developer-for-apache-spark.html

100% Passing Guarantee
100% Money Back Assurance

Following Questions and Answers are all new published by Databricks Official Exam Center

QUESTION 1

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.

Correct Answer: B

Correct code block:

transactionsDf.write.format("parquet").option("mode", "append").save(path)

QUESTION 2

Which of the following DataFrame operators is never classified as a wide transformation?

A. DataFrame.sort()

B. DataFrame.aggregate()

C. DataFrame.repartition()

D. DataFrame.select()

E. DataFrame.join()

Correct Answer: D

As a general rule: After having gone through the practice tests you probably have a good feeling for what classifies as a wide and what classifies as a narrow transformation. If you are unsure, feel free to play around in Spark and display the explanation of the Spark execution plan via DataFrame.[operation, for example sort()].explain(). If repartitioning is involved, it would count as a wide transformation. DataFrame.select() Correct! A wide transformation includes a shuffle, meaning that an input partition maps to one or more output partitions. This is expensive and causes traffic across the cluster. With the select() operation however, you pass commands to Spark that tell Spark to perform an operation on a specific slice of any partition. For this, Spark does not need to exchange data across partitions, each partition can be worked on independently. Thus, you do not cause a wide transformation. DataFrame.repartition() Incorrect. When you repartition a DataFrame, you redefine partition boundaries. Data will flow across your cluster and end up in different partitions after the repartitioning is completed. This is known as a shuffle and, in turn, is classified as a wide transformation. DataFrame.aggregate() No. When you aggregate, you may compare and summarize data across partitions. In the process, data are exchanged across the cluster, and newly formed output partitions depend on one or

more input partitions. This is a typical characteristic of a shuffle, meaning that the aggregate operation may classify as a wide transformation.

DataFrame.join() Wrong. Joining multiple DataFrames usually means that large amounts of data are exchanged across the cluster, as new partitions are formed. This is a shuffle and therefore DataFrame.join() counts as a wide transformation. DataFrame.sort() False. When sorting, Spark needs to compare many rows across all partitions to each other. This is an expensive operation, since data is exchanged across the cluster and new partitions are formed as data is reordered. This process classifies as a shuffle and, as a result, DataFrame.sort() counts as wide transformation. More info: Understanding Apache Spark Shuffle | Philipp Brunenberg

QUESTION 3

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())

Correct Answer: E

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, 55 (Databricks import instructions)

QUESTION 4

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()

Correct Answer: A

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.

QUESTION 5

Which of the following describes Spark\\'s Adaptive Query Execution?

A. Adaptive Query Execution features include dynamically coalescing shuffle partitions, dynamically injecting scan filters, and dynamically optimizing skew joins.

B. Adaptive Query Execution is enabled in Spark by default.

C. Adaptive Query Execution reoptimizes queries at execution points.

D. Adaptive Query Execution features are dynamically switching join strategies and dynamically optimizing skew joins.

E. Adaptive Query Execution applies to all kinds of queries.

Correct Answer: D

Adaptive Query Execution features include dynamically coalescing shuffle partitions, dynamically injecting scan filters, and dynamically optimizing skew joins. This is almost correct. All of these features, except for dynamically injecting scan filters, are part of Adaptive Query Execution. Dynamically injecting scan filters for join operations to limit the amount of data to be considered in a query is part of Dynamic Partition Pruning and not of Adaptive Query Execution. Adaptive Query Execution reoptimizes queries at execution points. No, Adaptive Query Execution reoptimizes queries at materialization points. Adaptive Query Execution is enabled in Spark by default. No, Adaptive Query Execution is disabled in Spark needs to be enabled through the spark.sql.adaptive.enabled property. Adaptive Query Execution applies to all kinds of queries. No, Adaptive Query Execution applies only to queries that are not streaming queries and that contain at least one exchange (typically expressed through a join, aggregate, or window operator) or one subquery. More info: How to Speed up SQL Queries with Adaptive Query Execution, Learning Spark, 2nd Edition, Chapter 12 (https://bit.ly/3tOh8M1)