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

Which of the following code blocks applies the Python function to_limit on column predError in table transactionsDf, returning a DataFrame with columns transactionId and result?

A. 1.spark.udf.register("LIMIT_FCN", to_limit) 2.spark.sql("SELECT transactionId, LIMIT_FCN(predError) AS result FROM transactionsDf")

B. 1.spark.udf.register("LIMIT_FCN", to_limit) 2.spark.sql("SELECT transactionId, LIMIT_FCN(predError) FROM transactionsDf AS result")

C. 1.spark.udf.register("LIMIT_FCN", to_limit) 2.spark.sql("SELECT transactionId, to_limit(predError) AS result FROM transactionsDf") spark.sql ("SELECT transactionId, udf(to_limit(predError)) AS result FROM transactionsDf")

D. 1.spark.udf.register(to_limit, "LIMIT_FCN") 2.spark.sql("SELECT transactionId, LIMIT_FCN(predError) AS result FROM transactionsDf")

Correct Answer: A

spark.udf.register("LIMIT_FCN", to_limit)

spark.sql("SELECT transactionId, LIMIT_FCN(predError) AS result FROM transactionsDf") Correct! First,

you have to register to_limit as UDF to use it in a sql statement. Then, you can use it under the

LIMIT_FCN name, correctly naming the resulting column result.

spark.udf.register(to_limit, "LIMIT_FCN")

spark.sql("SELECT transactionId, LIMIT_FCN(predError) AS result FROM transactionsDf") No. In this

answer, the arguments to spark.udf.register are flipped.

spark.udf.register("LIMIT_FCN", to_limit)

spark.sql("SELECT transactionId, to_limit(predError) AS result FROM transactionsDf") Wrong, this answer

does not use the registered LIMIT_FCN in the sql statement, but tries to access the to_limit method

directly. This will fail, since Spark cannot access it. spark.sql("SELECT transactionId, udf(to_limit

(predError)) AS result FROM transactionsDf") Incorrect, there is no udf method in Spark\\'s SQL.

spark.udf.register("LIMIT_FCN", to_limit)

spark.sql("SELECT transactionId, LIMIT_FCN(predError) FROM transactionsDf AS result") False. In this

answer, the column that results from applying the UDF is not correctly renamed to result.

Static notebook | Dynamic notebook: See test 3, 52 (Databricks import instructions)

QUESTION 2

Which of the following statements about data skew is incorrect?

A. Spark will not automatically optimize skew joins by default.

B. Broadcast joins are a viable way to increase join performance for skewed data over sort- merge joins.

C. In skewed DataFrames, the largest and the smallest partition consume very different amounts of memory.

D. To mitigate skew, Spark automatically disregards null values in keys when joining.

E. Salting can resolve data skew.

Correct Answer: D

To mitigate skew, Spark automatically disregards null values in keys when joining. This statement is incorrect, and thus the correct answer to the question. Joining keys that contain null values is of particular concern with regard to data skew. In real-world applications, a table may contain a great number of records that do not have a value assigned to the column used as a join key. During the join, the data is at risk of being heavily skewed. This is because all records with a null-value join key are then evaluated as a single large partition, standing in stark contrast to the potentially diverse key values (and therefore small partitions) of the non-null-key records. Spark specifically does not handle this automatically. However, there are several strategies to mitigate this problem like discarding null values temporarily, only to merge them back later (see last link below). In skewed DataFrames, the largest and the smallest partition consume very different amounts of memory. This statement is correct. In fact, having very different partition sizes is the very definition of skew. Skew can degrade Spark performance because the largest partition occupies a single executor for a long time. This blocks a Spark job and is an inefficient use of resources, since other executors that processed smaller partitions need to idle until the large partition is processed. Salting can resolve data skew. This statement is correct. The purpose of salting is to provide Spark with an opportunity to repartition data into partitions of similar size, based on a salted partitioning key. A salted partitioning key typically is a column that consists of uniformly distributed random numbers. The number of unique entries in the partitioning key column should match the number of your desired number of partitions. After repartitioning by the salted key, all partitions should have roughly the same size. Spark does not automatically optimize skew joins by default. This statement is correct. Automatic skew join optimization is a feature of Adaptive Query Execution (AQE). By default, AQE is disabled in Spark. To enable it, Spark\\'s spark.sql.adaptive.enabled configuration option needs to be set to true instead of leaving it at the default false. To automatically optimize skew joins, Spark\\'s spark.sql.adaptive.skewJoin.enabled options also needs to be set to true, which it is by default. When skew join optimization is enabled, Spark recognizes skew joins and optimizes them by splitting the bigger partitions into smaller partitions which leads to performance increases. Broadcast joins are a viable way to increase join performance for skewed data over sort- merge joins. This statement is correct. Broadcast joins can indeed help increase join performance for skewed data, under some conditions. One of the DataFrames to be joined needs to be small enough to fit into each executor\\'s memory, along a partition from the other DataFrame. If this is the case, a broadcast join increases join performance over a sort-merge join. The reason is that a sort-merge join with skewed data involves excessive shuffling. During shuffling, data is sent around the cluster, ultimately slowing down the Spark application. For skewed data, the amount of data, and thus the slowdown, is particularly big. Broadcast joins, however, help reduce shuffling data. The smaller table is directly stored on all executors, eliminating a great amount of network traffic, ultimately increasing join performance relative to the sort-merge join. It is worth noting that for optimizing skew join behavior it may make sense to manually adjust Spark\\'s spark.sql.autoBroadcastJoinThreshold configuration property if the smaller DataFrame is bigger than the 10 MB set by default. More info:

-Performance Tuning - Spark 3.0.0 Documentation

-Data Skew and Garbage Collection to Improve Spark Performance

-Section 1.2 - Joins on Skewed Data ?GitBook

QUESTION 3

The code block displayed below contains an error. The code block should write DataFrame transactionsDf

as a parquet file to location filePath after partitioning it on column storeId.

Find the error.

Code block:

transactionsDf.write.partitionOn("storeId").parquet(filePath)

A. The partitioning column as well as the file path should be passed to the write() method of DataFrame transactionsDf directly and not as appended commands as in the code block.

B. The partitionOn method should be called before the write method.

C. The operator should use the mode() option to configure the DataFrameWriter so that it replaces any existing files at location filePath.

D. Column storeId should be wrapped in a col() operator.

E. No method partitionOn() exists for the DataFrame class, partitionBy() should be used instead.

Correct Answer: E

QUESTION 4

Which of the following code blocks performs a join in which the small DataFrame transactionsDf is sent to all executors where it is joined with DataFrame itemsDf on columns storeId and itemId, respectively?

A. itemsDf.join(transactionsDf, itemsDf.itemId == transactionsDf.storeId, "right_outer")

B. itemsDf.join(transactionsDf, itemsDf.itemId == transactionsDf.storeId, "broadcast")

C. itemsDf.merge(transactionsDf, "itemsDf.itemId == transactionsDf.storeId", "broadcast")

D. itemsDf.join(broadcast(transactionsDf), itemsDf.itemId == transactionsDf.storeId)

E. itemsDf.join(transactionsDf, broadcast(itemsDf.itemId == transactionsDf.storeId))

Correct Answer: D

QUESTION 5

Which of the following code blocks returns a copy of DataFrame itemsDf where the column supplier has been renamed to manufacturer?

A. itemsDf.withColumn(["supplier", "manufacturer"])

B. itemsDf.withColumn("supplier").alias("manufacturer")

C. itemsDf.withColumnRenamed("supplier", "manufacturer")

D. itemsDf.withColumnRenamed(col("manufacturer"), col("supplier"))

E. itemsDf.withColumnsRenamed("supplier", "manufacturer")

Correct Answer: C

itemsDf.withColumnRenamed("supplier", "manufacturer") Correct! This uses the relatively trivial DataFrame method withColumnRenamed for renaming column supplier to column manufacturer.

Note that the asks for "a copy of DataFrame itemsDf". This may be confusing if you are not familiar with Spark yet. RDDs (Resilient Distributed Datasets) are the foundation of Spark DataFrames and are immutable. As such, DataFrames are immutable, too. Any command that changes anything in the DataFrame therefore necessarily returns a copy, or a new version, of it that has the changes applied.

itemsDf.withColumnsRenamed("supplier", "manufacturer") Incorrect. Spark\\'s DataFrame API does not have a withColumnsRenamed() method. itemsDf.withColumnRenamed(col("manufacturer"), col ("supplier")) No. Watch out ?although the col() method works for many methods of the DataFrame API, withColumnRenamed is not one of them. As outlined in the documentation linked below, withColumnRenamed expects strings.

itemsDf.withColumn(["supplier", "manufacturer"])

Wrong. While DataFrame.withColumn() exists in Spark, it has a different purpose than renaming columns. withColumn is typically used to add columns to DataFrames, taking the name of the new column as a first, and a Column as a second argument. Learn more via the documentation that is linked below.

itemsDf.withColumn("supplier").alias("manufacturer") No. While DataFrame.withColumn() exists, it requires 2 arguments. Furthermore, the alias() method on DataFrames would not help the cause of renaming a column much.

DataFrame.alias() can be useful in addressing the input of join statements. However, this is far outside of the scope of this question.

If you are curious nevertheless, check out the link below. More info:

pyspark.sql.DataFrame.withColumnRenamed -- PySpark 3.1.1 documentation,

pyspark.sql.DataFrame.withColumn -- PySpark 3.1.1 documentation, and pyspark.sql.DataFrame.alias --

PySpark 3.1.2 documentation (https://bit.ly/3aSB5tm , https://bit.ly/2Tv4rbE , https://bit.ly/2RbhBd2)

Static notebook | Dynamic notebook: See test 1, 31 (Databricks import instructions) (https://flrs.github.io/

spark_practice_tests_code/#1/31.html , https://bit.ly/sparkpracticeexams_import_instructions)