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.pass2lead.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 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 2

The code block displayed below contains an error. When the code block below has executed, it should have divided DataFrame transactionsDf into 14 parts, based on columns storeId and

transactionDate (in this order). Find the error.

Code block:

transactionsDf.coalesce(14, ("storeId", "transactionDate"))

A. The parentheses around the column names need to be removed and .select() needs to be appended to the code block.

B. Operator coalesce needs to be replaced by repartition, the parentheses around the column names need to be

removed, and .count() needs to be appended to the code block.

C. Operator coalesce needs to be replaced by repartition, the parentheses around the column names need to be removed, and .select() needs to be appended to the code block.

D. Operator coalesce needs to be replaced by repartition and the parentheses around the column names need to be replaced by square brackets.

E. Operator coalesce needs to be replaced by repartition.

Correct Answer: B

Correct code block:

transactionsDf.repartition(14, "storeId", "transactionDate").count() Since we do not know how many

partitions DataFrame transactionsDf has, we cannot safely use coalesce, since it would not make any

change if the current number of partitions is smaller than 14.

So, we need to use repartition.

In the Spark documentation, the call structure for repartition is shown like this:

DataFrame.repartition(numPartitions, *cols). The * operator means that any argument after numPartitions

will be

interpreted as column. Therefore, the brackets need to be removed. Finally, the specifies that after the execution the DataFrame should be divided. So, indirectly this is asking us to append an action to the code block. Since .select() is a transformation. the only possible choice here is .count(). More info: pyspark.sql.DataFrame.repartition -- PySpark 3.1.1 documentation Static notebook | Dynamic notebook: See test 1, 40 (Databricks import instructions)

QUESTION 3

Which of the following code blocks adds a column predErrorSqrt to DataFrame transactionsDf that is the square root of column predError?

A. transactionsDf.withColumn("predErrorSqrt", sqrt(predError))

B. transactionsDf.select(sqrt(predError))

C. transactionsDf.withColumn("predErrorSqrt", col("predError").sqrt())

D. transactionsDf.withColumn("predErrorSqrt", sqrt(col("predError")))

E. transactionsDf.select(sqrt("predError"))

Correct Answer: D

transactionsDf.withColumn("predErrorSqrt", sqrt(col("predError"))) Correct. The DataFrame.withColumn() operator is

used to add a new column to a DataFrame. It takes two arguments: The name of the new column (here: predErrorSqrt) and a Column expression as the new column. In PySpark, a Column expression means referring to a column using the col ("predError") command or by other means, for example by transactionsDf.predError, or even just using the column name as a string, "predError". The asks for the square root. sqrt() is a function in pyspark.sql.functions and calculates the square root. It takes a value or a Column as an input. Here it is the predError column of DataFrame transactionsDf expressed through col("predError"). transactionsDf.withColumn ("predErrorSqrt", sqrt(predError)) Incorrect. In this expression, sqrt(predError) is incorrect syntax. You cannot refer to predError in this way ?to Spark it looks as if you are trying to refer to the non-existent Python variable predError. You could pass transactionsDf.predError, col("predError") (as in the correct solution), or even just "predError" instead. transactionsDf.select(sqrt(predError)) Wrong. Here, the explanation just above this one about how to refer to predError applies. transactionsDf.select(sqrt("predError")) No. While this is correct syntax, it will return a single-column DataFrame only containing a column showing the square root of column predError. However, the asks for a column to be added to the original DataFrame transactionsDf. transactionsDf.withColumn("predErrorSqrt", col ("predError").sqrt()) No. The issue with this statement is that column col("predError") has no sqrt() method. sqrt() is a member of pyspark.sql.functions, but not of pyspark.sql.Column. More info: pyspark.sql.DataFrame.withColumn -- PySpark 3.1.2 documentation and pyspark.sql.functions.sqrt --PySpark 3.1.2 documentation Static notebook | Dynamic notebook: See test 2, 31 (Databricks import instructions)

QUESTION 4

Which of the following code blocks prints out in how many rows the expression Inc. appears in the stringtype 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)

Correct Answer: E

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 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, 22 (Databricks import instructions)

QUESTION 5

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.

Correct Answer: D