Free Questions for Databricks-Certified-Associate-Developer-for-Apache-Spark-3.0 by ebraindumps

Shared by Hoover on 29-01-2024

Question 1

Question Type: MultipleChoice

The code block displayed below contains an error. The code block should trigger Spark to cache DataFrame transactionsDf in executor memory where available, writing to disk where insufficient

executor memory is available, in a fault-tolerant way. Find the error.

Code block:

```
transactionsDf.persist(StorageLevel.MEMORY_AND_DISK)
```

Options:

A- Caching is not supported in Spark, data are always recomputed.

B- Data caching capabilities can be accessed through the spark object, but not through the DataFrame API.

C- The storage level is inappropriate for fault-tolerant storage.

D- The code block uses the wrong operator for caching.

E- The DataFrameWriter needs to be invoked.

Answer:

C

Explanation:

The storage level is inappropriate for fault-tolerant storage.

Correct. Typically, when thinking about fault tolerance and storage levels, you would want to store redundant copies of the dataset. This can be achieved by using a storage level such as

StorageLevel.MEMORY_AND_DISK_2.

The code block uses the wrong command for caching.

Wrong. In this case, DataFrame.persist() needs to be used, since this operator supports passing a storage level. DataFrame.cache() does not support passing a storage level.

Caching is not supported in Spark, data are always recomputed.

Incorrect. Caching is an important component of Spark, since it can help to accelerate Spark programs to great extent. Caching is often a good idea for datasets that need to be accessed

repeatedly.

Data caching capabilities can be accessed through the spark object, but not through the DataFrame API.

No. Caching is either accessed through DataFrame.cache() or DataFrame.persist().

The DataFrameWriter needs to be invoked.

Wrong. The DataFrameWriter can be accessed via DataFrame.write and is used to write data to external data stores, mostly on disk. Here, we find keywords such as 'cache' and 'executor

memory' that point us away from using external data stores. We aim to save data to memory to accelerate the reading process, since reading from disk is comparatively slower. The

DataFrameWriter does not write to memory, so we cannot use it here.

More info: Best practices for caching in Spark SQL | by David Vrba | Towards Data Science

Question 2

Question Type: MultipleChoice

Which of the following code blocks reads the parquet file stored at filePath into DataFrame itemsDf, using a valid schema for the sample of itemsDf shown below?

Sample of itemsDf:

1. +-------+-----------------------------+-------------------+

2. |itemId|attributes |supplier |

3. +-------+-----------------------------+-------------------+

```
|1 |[blue, winter, cozy] |Sports Company Inc.|
|2 |[red, summer, fresh, cooling]|YetiX |
|3 |[green, summer, travel] |Sports Company Inc.|
+-------+----------------------------+-------------------+
```

Options:

A-
```
itemsDfSchema = StructType([
StructField('itemId', IntegerType()),
StructField('attributes', StringType()),
StructField('supplier', StringType())])

itemsDf = spark.read.schema(itemsDfSchema).parquet(filePath)
```

B-
```
itemsDfSchema = StructType([
StructField('itemId', IntegerType),
StructField('attributes', ArrayType(StringType)),
StructField('supplier', StringType)])

itemsDf = spark.read.schema(itemsDfSchema).parquet(filePath)
```

C-
```
itemsDf = spark.read.schema('itemId integer, attributes <string>, supplier string').parquet(filePath)
```

D-
```
itemsDfSchema = StructType([
```

2. StructField('itemId', IntegerType()),
3. StructField('attributes', ArrayType(StringType())),
4. StructField('supplier', StringType())])
5.
6. itemsDf = spark.read.schema(itemsDfSchema).parquet(filePath)

E- 1. itemsDfSchema = StructType([
2. StructField('itemId', IntegerType()),
3. StructField('attributes', ArrayType([StringType()])),
4. StructField('supplier', StringType())])
5.
6. itemsDf = spark.read(schema=itemsDfSchema).parquet(filePath)

Answer:

D

Explanation:

The challenge in this Question: comes from there being an array variable in the schema. In addition, you should know how to pass a schema to the DataFrameReader that is invoked by

spark.read.

The correct way to define an array of strings in a schema is through ArrayType(StringType()). A schema can be passed to the DataFrameReader by simply appending schema(structType) to the read() operator. Alternatively, you can also define a schema as a string. For example, for the schema of itemsDf, the following string would make sense: itemId integer, attributes array<string>,

supplier string.

A thing to keep in mind is that in schema definitions, you always need to instantiate the types, like so: StringType(). Just using StringType does not work in pySpark and will fail.

Another concern with schemas is whether columns should be nullable, so allowed to have null values. In the case at hand, this is not a concern however, since the Question: just asks for a

'valid'

schema. Both non-nullable and nullable column schemas would be valid here, since no null value appears in the DataFrame sample.

More info: Learning Spark, 2nd Edition, Chapter 3

Static notebook | Dynamic notebook: See test 3, Question: 19 (Databricks import instructions)

Question 3

Question Type: MultipleChoice

Which of the following code blocks creates a new 6-column DataFrame by appending the rows of the 6-column DataFrame yesterdayTransactionsDf to the rows of the 6-column DataFrame

todayTransactionsDf, ignoring that both DataFrames have different column names?

Options:

A- union(todayTransactionsDf, yesterdayTransactionsDf)

B- todayTransactionsDf.unionByName(yesterdayTransactionsDf, allowMissingColumns=True)

C- todayTransactionsDf.unionByName(yesterdayTransactionsDf)

D- todayTransactionsDf.concat(yesterdayTransactionsDf)

E- todayTransactionsDf.union(yesterdayTransactionsDf)

Answer:

E

Explanation:

todayTransactionsDf.union(yesterdayTransactionsDf)

Correct. The union command appends rows of yesterdayTransactionsDf to the rows of todayTransactionsDf, ignoring that both DataFrames have different column names. The resulting DataFrame will have the column names of DataFrame todayTransactionsDf.

todayTransactionsDf.unionByName(yesterdayTransactionsDf)

No. unionByName specifically tries to match columns in the two DataFrames by name and only appends values in columns with identical names across the two DataFrames. In the form presented

above, the command is a great fit for joining DataFrames that have exactly the same columns, but in a different order. In this case though, the command will fail because the two DataFrames have

different columns.

todayTransactionsDf.unionByName(yesterdayTransactionsDf, allowMissingColumns=True)

No. The unionByName command is described in the previous explanation. However, with the allowMissingColumns argument set to True, it is no longer an issue that the two DataFrames have

different column names. Any columns that do not have a match in the other DataFrame will be filled with null where there is no value. In the case at hand, the resulting DataFrame will have 7 or more

columns though, so it this command is not the right answer.

union(todayTransactionsDf, yesterdayTransactionsDf)

No, there is no union method in pyspark.sql.functions.

todayTransactionsDf.concat(yesterdayTransactionsDf)

Wrong, the DataFrame class does not have a concat method.

More info: pyspark.sql.DataFrame.union --- PySpark 3.1.2 documentation, pyspark.sql.DataFrame.unionByName --- PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 3, Question: 18 (Databricks import instructions)

Question 4

Question Type: **MultipleChoice**

Which of the elements in the labeled panels represent the operation performed for broadcast variables?

Larger image

Options:

A- 2, 5

B- 3

C- 2, 3

D- 1, 2

E- 1, 3, 4

Answer:

C

Explanation:

2,3

Correct! Both panels 2 and 3 represent the operation performed for broadcast variables. While a broadcast operation may look like panel 3, with the driver being the bottleneck, it most probably

looks like panel 2.

This is because the torrent protocol sits behind Spark's broadcast implementation. In the torrent protocol, each executor will try to fetch missing broadcast variables from the driver or other nodes,

preventing the driver from being the bottleneck.

1,2

Wrong. While panel 2 may represent broadcasting, panel 1 shows bi-directional communication which does not occur in broadcast operations.

3

No. While broadcasting may materialize like shown in panel 3, its use of the torrent protocol also enables communciation as shown in panel 2 (see first explanation).

1,3,4

No. While panel 2 shows broadcasting, panel 1 shows bi-directional communication -- not a characteristic of broadcasting. Panel 4 shows uni-directional communication, but in the wrong direction.

Panel 4 resembles more an accumulator variable than a broadcast variable.

2,5

Incorrect. While panel 2 shows broadcasting, panel 5 includes bi-directional communication -- not a characteristic of broadcasting.

More info: Broadcast Join with Spark -- henning.kropponline.de

Question 5

Question Type: MultipleChoice

Which of the following is not a feature of Adaptive Query Execution?

Options:

A- Replace a sort merge join with a broadcast join, where appropriate.

B- Coalesce partitions to accelerate data processing.

C- Split skewed partitions into smaller partitions to avoid differences in partition processing time.

D- Reroute a query in case of an executor failure.

E- Collect runtime statistics during query execution.

Answer:

D

Explanation:

Reroute a query in case of an executor failure.

Correct. Although this feature exists in Spark, it is not a feature of Adaptive Query Execution. The cluster manager keeps track of executors and will work together with the driver to launch an

executor and assign the workload of the failed executor to it (see also link below).

Replace a sort merge join with a broadcast join, where appropriate.

No, this is a feature of Adaptive Query Execution.

Coalesce partitions to accelerate data processing.

Wrong, Adaptive Query Execution does this.

Collect runtime statistics during query execution.

Incorrect, Adaptive Query Execution (AQE) collects these statistics to adjust query plans. This feedback loop is an essential part of accelerating queries via AQE.

Split skewed partitions into smaller partitions to avoid differences in partition processing time.

No, this is indeed a feature of Adaptive Query Execution. Find more information in the Databricks blog post linked below.

More info: Learning Spark, 2nd Edition, Chapter 12, On which way does RDD of spark finish fault-tolerance? - Stack Overflow, How to Speed up SQL Queries with Adaptive Query Execution

Question 6

Question Type: **MultipleChoice**

Which of the following statements about storage levels is incorrect?

Options:

A- The cache operator on DataFrames is evaluated like a transformation.

B- In client mode, DataFrames cached with the MEMORY_ONLY_2 level will not be stored in the edge node's memory.

C- Caching can be undone using the DataFrame.unpersist() operator.

D- MEMORY_AND_DISK replicates cached DataFrames both on memory and disk.

E- DISK_ONLY will not use the worker node's memory.

Answer:

D

Explanation:

MEMORY_AND_DISK replicates cached DataFrames both on memory and disk.

Correct, this statement is wrong. Spark prioritizes storage in memory, and will only store data on disk that does not fit into memory.

DISK_ONLY will not use the worker node's memory.

Wrong, this statement is correct. DISK_ONLY keeps data only on the worker node's disk, but not in memory.

In client mode, DataFrames cached with the MEMORY_ONLY_2 level will not be stored in the edge node's memory.

Wrong, this statement is correct. In fact, Spark does not have a provision to cache DataFrames in the driver (which sits on the edge node in client mode). Spark caches DataFrames in the executors'

memory.

Caching can be undone using the DataFrame.unpersist() operator.

Wrong, this statement is correct. Caching, as achieved via the DataFrame.cache() or DataFrame.persist() operators can be undone using the DataFrame.unpersist() operator. This operator will

remove all of its parts from the executors' memory and disk.

The cache operator on DataFrames is evaluated like a transformation.

Wrong, this statement is correct. DataFrame.cache() is evaluated like a transformation: Through lazy evaluation. This means that after calling DataFrame.cache() the command will not have any

effect until you call a subsequent action, like DataFrame.cache().count().

More info: pyspark.sql.DataFrame.unpersist --- PySpark 3.1.2 documentation

Question 7

Question Type: MultipleChoice

Which of the following statements about reducing out-of-memory errors is incorrect?

Options:

A- Concatenating multiple string columns into a single column may guard against out-of-memory errors.

B- Reducing partition size can help against out-of-memory errors.

C- Limiting the amount of data being automatically broadcast in joins can help against out-of-memory errors.

D- Setting a limit on the maximum size of serialized data returned to the driver may help prevent out-of-memory errors.

E- Decreasing the number of cores available to each executor can help against out-of-memory errors.

Answer:

A

Explanation:

Concatenating multiple string columns into a single column may guard against out-of-memory errors.

Exactly, this is an incorrect answer! Concatenating any string columns does not reduce the size of the data, it just structures it a different way. This does little to how Spark processes the data and

definitely does not reduce out-of-memory errors.

Reducing partition size can help against out-of-memory errors.

No, this is not incorrect. Reducing partition size is a viable way to aid against out-of-memory errors, since executors need to load partitions into memory before processing them. If the executor does

not have enough memory available to do that, it will throw an out-of-memory error. Decreasing partition size can therefore be very helpful for preventing that.

Decreasing the number of cores available to each executor can help against out-of-memory errors.

No, this is not incorrect. To process a partition, this partition needs to be loaded into the memory of an executor. If you imagine that every core in every executor processes a partition, potentially in

parallel with other executors, you can imagine that memory on the machine hosting the executors fills up quite quickly. So, memory usage of executors is a concern, especially when multiple

partitions are processed at the same time. To strike a balance between performance and memory usage, decreasing the number of cores may help against out-of-memory errors.

Setting a limit on the maximum size of serialized data returned to the driver may help prevent out-of-memory errors.

No, this is not incorrect. When using commands like collect() that trigger the transmission of potentially large amounts of data from the cluster to the driver, the driver may experience out-of-memory

errors. One strategy to avoid this is to be careful about using commands like collect() that send back large amounts of data to the driver. Another strategy is setting the parameter spark.driver.maxResultSize. If data to be transmitted to the driver exceeds the threshold specified by the parameter, Spark will abort the job and therefore prevent an out-of-memory error.

Limiting the amount of data being automatically broadcast in joins can help against out-of-memory errors.

Wrong, this is not incorrect. As part of Spark's internal optimization, Spark may choose to speed up operations by broadcasting (usually relatively small) tables to executors. This broadcast is

happening from the driver, so all the broadcast tables are loaded into the driver first. If these tables are relatively big, or multiple mid-size tables are being broadcast, this may lead to an out-of-

memory error. The maximum table size for which Spark will consider broadcasting is set by the spark.sql.autoBroadcastJoinThreshold parameter.

More info: Configuration - Spark 3.1.2 Documentation and Spark OOM Error --- Closeup. Does the following look familiar when... | by Amit Singh Rathore | The Startup | Medium

Question 8

Question Type: MultipleChoice

Which of the following is a problem with using accumulators?

Options:

A- Only unnamed accumulators can be inspected in the Spark UI.

B- Only numeric values can be used in accumulators.

C- Accumulator values can only be read by the driver, but not by executors.

D- Accumulators do not obey lazy evaluation.

E- Accumulators are difficult to use for debugging because they will only be updated once, independent if a task has to be re-run due to hardware failure.

Answer:

C

Explanation:

Accumulator values can only be read by the driver, but not by executors.

Correct. So, for example, you cannot use an accumulator variable for coordinating workloads between executors. The typical, canonical, use case of an accumulator value is to report data, for

example for debugging purposes, back to the driver. For example, if you wanted to count values that match a specific condition in a UDF for debugging purposes, an accumulator provides a good

way to do that.

Only numeric values can be used in accumulators.

No. While pySpark's Accumulator only supports numeric values (think int and float), you can define accumulators for custom types via the AccumulatorParam interface (documentation linked below).

Accumulators do not obey lazy evaluation.

Incorrect -- accumulators do obey lazy evaluation. This has implications in practice: When an accumulator is encapsulated in a transformation, that accumulator will not be modified until a

subsequent action is run.

Accumulators are difficult to use for debugging because they will only be updated once, independent if a task has to be re-run due to hardware failure.

Wrong. A concern with accumulators is in fact that under certain conditions they can run for each task more than once. For example, if a hardware failure occurs during a task after an accumulator

variable has been increased but before a task has finished and Spark launches the task on a different worker in response to the failure, already executed accumulator variable increases will be

repeated.

Only unnamed accumulators can be inspected in the Spark UI.

No. Currently, in PySpark, no accumulators can be inspected in the Spark UI. In the Scala interface of Spark, only named accumulators can be inspected in the Spark UI.

More info: Aggregating Results with Spark Accumulators | Sparkour, RDD Programming Guide - Spark 3.1.2 Documentation, pyspark.Accumulator --- PySpark 3.1.2 documentation, and

pyspark.AccumulatorParam --- PySpark 3.1.2 documentation

Question 9

Question Type: MultipleChoice

Which of the following describes a valid concern about partitioning?

Options:

A- A shuffle operation returns 200 partitions if not explicitly set.

B- Decreasing the number of partitions reduces the overall runtime of narrow transformations if there are more executors available than partitions.

C- No data is exchanged between executors when coalesce() is run.

D- Short partition processing times are indicative of low skew.

E- The coalesce() method should be used to increase the number of partitions.

Answer:

A

Explanation:

A shuffle operation returns 200 partitions if not explicitly set.

Correct. 200 is the default value for the Spark property spark.sql.shuffle.partitions. This property determines how many partitions Spark uses when shuffling data for joins or aggregations.

The coalesce() method should be used to increase the number of partitions.

Incorrect. The coalesce() method can only be used to decrease the number of partitions.

Decreasing the number of partitions reduces the overall runtime of narrow transformations if there are more executors available than partitions.

No. For narrow transformations, fewer partitions usually result in a longer overall runtime, if more executors are available than partitions.

A narrow transformation does not include a shuffle, so no data need to be exchanged between executors. Shuffles are expensive and can be a bottleneck for executing Spark workloads.

Narrow transformations, however, are executed on a per-partition basis, blocking one executor per partition. So, it matters how many executors are available to perform work in parallel relative to the number of partitions. If the number of executors is greater than the number of partitions, then some executors are idle while other process the partitions. On the flip side, if the number of executors is

smaller than the number of partitions, the entire operation can only be finished after some executors have processed multiple partitions, one after the other. To minimize the overall runtime, one

would want to have the number of partitions equal to the number of executors (but not more).

So, for the scenario at hand, increasing the number of partitions reduces the overall runtime of narrow transformations if there are more executors available than partitions.

No data is exchanged between executors when coalesce() is run.

No. While coalesce() avoids a full shuffle, it may still cause a partial shuffle, resulting in data exchange between executors.

Short partition processing times are indicative of low skew.

Incorrect. Data skew means that data is distributed unevenly over the partitions of a dataset. Low skew therefore means that data is distributed evenly.

Partition processing time, the time that executors take to process partitions, can be indicative of skew if some executors take a long time to process a partition, but others do not. However, a short

processing time is not per se indicative a low skew: It may simply be short because the partition is small.

A situation indicative of low skew may be when all executors finish processing their partitions in the same timeframe. High skew may be indicated by some executors taking much longer to finish their partitions than others. But the answer does not make any comparison -- so by itself it does not provide enough information to make any assessment about skew.

More info: Spark Repartition & Coalesce - Explained and Performance Tuning - Spark 3.1.2 Documentation

Question 10

Question Type: MultipleChoice

Which of the following statements about executors is correct?

Options:

A- Executors are launched by the driver.

B- Executors stop upon application completion by default.

C- Each node hosts a single executor.

D- Executors store data in memory only.

E- An executor can serve multiple applications.

Answer:

B

Explanation:

Executors stop upon application completion by default.

Correct. Executors only persist during the lifetime of an application.

A notable exception to that is when Dynamic Resource Allocation is enabled (which it is not by default). With Dynamic Resource Allocation enabled, executors are terminated when they are idle,

independent of whether the application has been completed or not.

An executor can serve multiple applications.

Wrong. An executor is always specific to the application. It is terminated when the application completes (exception see above).

Each node hosts a single executor.

No. Each node can host one or more executors.

Executors store data in memory only.

No. Executors can store data in memory or on disk.

Executors are launched by the driver.

Incorrect. Executors are launched by the cluster manager on behalf of the driver.

More info: Job Scheduling - Spark 3.1.2 Documentation, How Applications are Executed on a Spark Cluster | Anatomy of a Spark Application | InformIT, and Spark Jargon for Starters. This blog is to

clear some of the... | by Mageswaran D | Medium