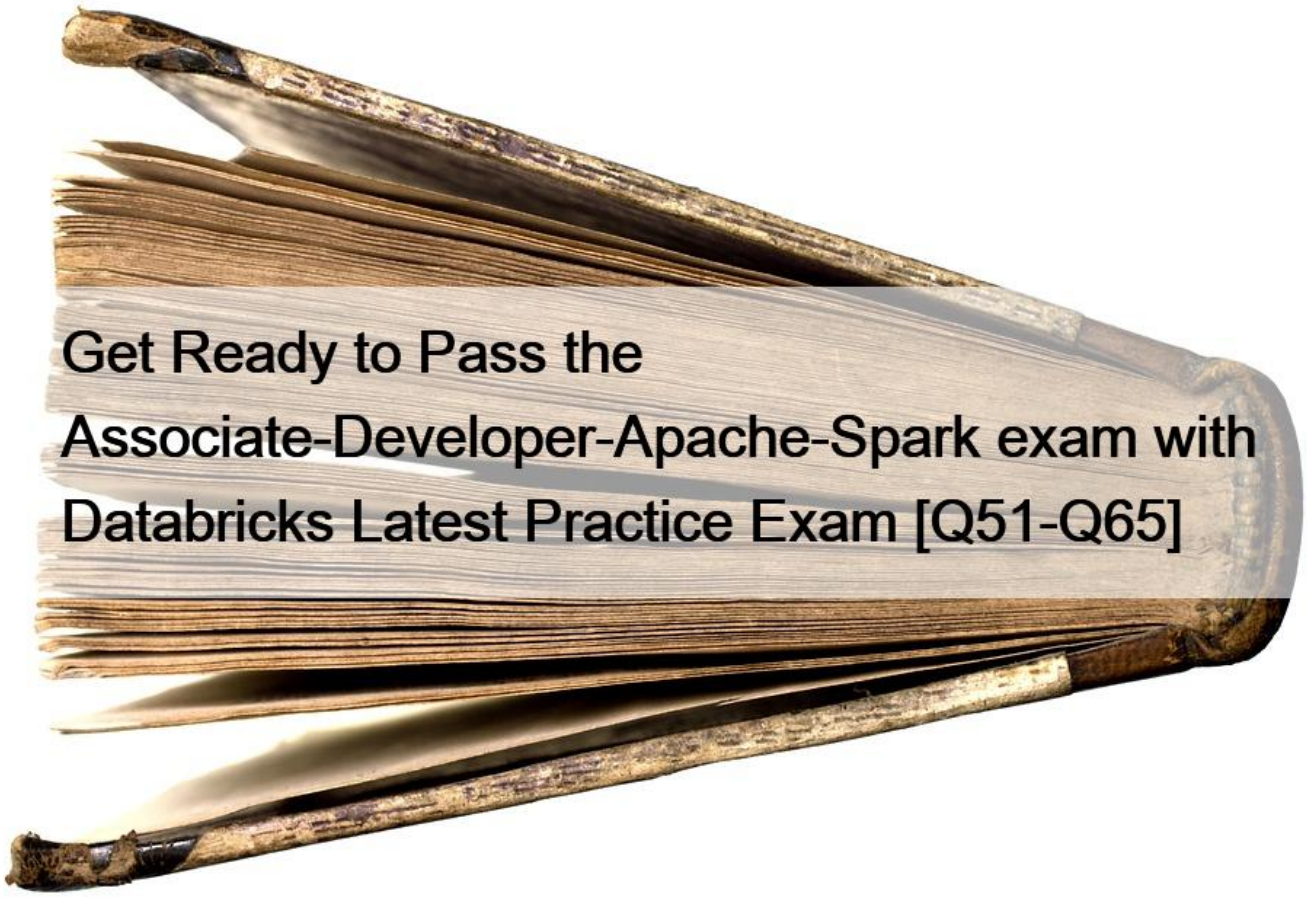
Get Ready to Pass the Associate-Developer-Apache-Spark exam with Databricks Latest Practice Exam [Q51-Q65



Get Ready to Pass the Associate-Developer-Apache-Spark exam with Databricks Latest Practice Exam Get Prepared for Your Associate-Developer-Apache-Spark Exam With Actual Databricks Study Guide! NEW QUESTION 51

Which of the following describes a difference between Spark’s cluster and client execution modes?
* In cluster mode, the cluster manager resides on a worker node, while it resides on an edge node in client mode.
* In cluster mode, executor processes run on worker nodes, while they run on gateway nodes in client mode.
* In cluster mode, the driver resides on a worker node, while it resides on an edge node in client mode.
* In cluster mode, a gateway machine hosts the driver, while it is co-located with the executor in client mode.
* In cluster mode, the Spark driver is not co-located with the cluster manager, while it is co-located in client mode.

Explanation

In cluster mode, the driver resides on a worker node, while it resides on an edge node in client mode.

Correct. The idea of Spark’s client mode is that workloads can be executed from an edge node, also known as gateway machine, from outside the cluster. The most common way to execute Spark however is in cluster mode, where the driver resides on a worker node.

In practice, in client mode, there are tight constraints about the data transfer speed relative to the data transfer speed between worker nodes in the cluster. Also, any job in that is executed in client mode will fail if the edge node fails. For these reasons, client mode is usually not used in a production environment.

In cluster mode, the cluster manager resides on a worker node, while it resides on an edge node in client execution mode.

No. In both execution modes, the cluster manager may reside on a worker node, but it does not reside on an edge node in client mode.

In cluster mode, executor processes run on worker nodes, while they run on gateway nodes in client mode.

This is incorrect. Only the driver runs on gateway nodes (also known as “edge nodes”) in client mode, but not the executor processes.

In cluster mode, the Spark driver is not co-located with the cluster manager, while it is co-located in client mode.

No, in client mode, the Spark driver is not co-located with the driver. The whole point of client mode is that the driver is outside the cluster and not associated with the resource that manages the cluster (the machine that runs the cluster manager).

In cluster mode, a gateway machine hosts the driver, while it is co-located with the executor in client mode.

No, it is exactly the opposite: There are no gateway machines in cluster mode, but in client mode, they host the driver.

NEW QUESTION 52

Which of the following is a viable way to improve Spark’s performance when dealing with large amounts of data, given that there is only a single application running on the cluster?
* Increase values for the properties spark.default.parallelism and spark.sql.shuffle.partitions
* Decrease values for the properties spark.default.parallelism and spark.sql.partitions
* Increase values for the properties spark.sql.parallelism and spark.sql.partitions
* Increase values for the properties spark.sql.parallelism and spark.sql.shuffle.partitions
* Increase values for the properties spark.dynamicAllocation.maxExecutors, spark.default.parallelism, and spark.sql.shuffle.partitions

Explanation

Decrease values for the properties spark.default.parallelism and spark.sql.partitions No, these values need to be increased.

Increase values for the properties spark.sql.parallelism and spark.sql.partitions Wrong, there is no property spark.sql.parallelism.

Increase values for the properties spark.sql.parallelism and spark.sql.shuffle.partitions See above.

Increase values for the properties spark.dynamicAllocation.maxExecutors, spark.default.parallelism, and spark.sql.shuffle.partitions The property spark.dynamicAllocation.maxExecutors is only in effect if dynamic allocation is enabled, using the spark.dynamicAllocation.enabled property. It is disabled by default. Dynamic allocation can be useful when to run multiple applications on the same cluster in parallel. However, in this case there is only a single application running on the cluster, so enabling dynamic allocation would not yield a performance benefit.

More info: Practical Spark Tips For Data Scientists | Experfy.com and Basics of Apache Spark Configuration Settings | by Halil Ertan | Towards Data Science (https://bit.ly/3gA0A6w ,

https://bit.ly/2QxhNTr)

NEW QUESTION 53

Which of the following code blocks returns a DataFrame where columns predError and productId are removed from DataFrame transactionsDf?

Sample of DataFrame transactionsDf:

1.+————-+———+—–+——-+———+—-+

2.|transactionId|predError|value|storeId|productId|f |

3.+————-+———+—–+——-+———+—-+

4.|1 |3 |4 |25 |1 |null|

5.|2 |6 |7 |2 |2 |null|

6.|3 |3 |null |25 |3 |null|

7.+————-+———+—–+——-+———+—-+
* transactionsDf.withColumnRemoved(“predError”, “productId”)
* transactionsDf.drop([“predError”, “productId”, “associateId”])
* transactionsDf.drop(“predError”, “productId”, “associateId”)
* transactionsDf.dropColumns(“predError”, “productId”, “associateId”)
* transactionsDf.drop(col(“predError”, “productId”))

Explanation

The key here is to understand that columns that are passed to DataFrame.drop() are ignored if they do not exist in the DataFrame. So, passing column name associateId to transactionsDf.drop() does not have any effect.

Passing a list to transactionsDf.drop() is not valid. The documentation (link below) shows the call structure as DataFrame.drop(*cols). The * means that all arguments that are passed to DataFrame.drop() are read as columns. However, since a list of columns, for example [“predError”,

“productId”, “associateId”] is not a column, Spark will run into an error.

More info: pyspark.sql.DataFrame.drop – PySpark 3.1.1 documentation

Static notebook | Dynamic notebook: See test 1

NEW QUESTION 54

The code block displayed below contains an error. The code block should combine data from DataFrames itemsDf and transactionsDf, showing all rows of DataFrame itemsDf that have a matching value in column itemId with a value in column

transactionsId of DataFrame transactionsDf. Find the error.

Code block:

itemsDf.join(itemsDf.itemId==transactionsDf.transactionId)
* The join statement is incomplete.
* The union method should be used instead of join.
* The join method is inappropriate.
* The merge method should be used instead of join.
* The join expression is malformed.

Explanation

Correct code block:

itemsDf.join(transactionsDf, itemsDf.itemId==transactionsDf.transactionId) The join statement is incomplete.

Correct! If you look at the documentation of DataFrame.join() (linked below), you see that the very first argument of join should be the DataFrame that should be joined with. This first argument is missing in the code block.

The join method is inappropriate.

No. By default, DataFrame.join() uses an inner join. This method is appropriate for the scenario described in the question.

The join expression is malformed.

Incorrect. The join expression itemsDf.itemId==transactionsDf.transactionId is correct syntax.

The merge method should be used instead of join.

False. There is no DataFrame.merge() method in PySpark.

The union method should be used instead of join.

Wrong. DataFrame.union() merges rows, but not columns as requested in the question.

More info: pyspark.sql.DataFrame.join – PySpark 3.1.2 documentation, pyspark.sql.DataFrame.union – PySpark 3.1.2 documentation Static notebook | Dynamic notebook: See test 3

NEW QUESTION 55

Which of the following describes the role of tasks in the Spark execution hierarchy?
* Tasks are the smallest element in the execution hierarchy.
* Within one task, the slots are the unit of work done for each partition of the data.
* Tasks are the second-smallest element in the execution hierarchy.
* Stages with narrow dependencies can be grouped into one task.
* Tasks with wide dependencies can be grouped into one stage.

Explanation

Stages with narrow dependencies can be grouped into one task.

Wrong, tasks with narrow dependencies can be grouped into one stage.

Tasks with wide dependencies can be grouped into one stage.

Wrong, since a wide transformation causes a shuffle which always marks the boundary of a stage. So, you cannot bundle multiple tasks that have wide dependencies into a stage.

Tasks are the second-smallest element in the execution hierarchy.

No, they are the smallest element in the execution hierarchy.

Within one task, the slots are the unit of work done for each partition of the data.

No, tasks are the unit of work done per partition. Slots help Spark parallelize work. An executor can have multiple slots which enable it to process multiple tasks in parallel.

NEW QUESTION 56

Which of the following code blocks returns a copy of DataFrame transactionsDf that only includes columns transactionId, storeId, productId and f?

Sample of DataFrame transactionsDf:

1.+————-+———+—–+——-+———+—-+

2.|transactionId|predError|value|storeId|productId| f|

3.+————-+———+—–+——-+———+—-+

4.| 1| 3| 4| 25| 1|null|

5.| 2| 6| 7| 2| 2|null|

6.| 3| 3| null| 25| 3|null|

7.+————-+———+—–+——-+———+—-+
* transactionsDf.drop(col(“value”), col(“predError”))
* transactionsDf.drop(“predError”, “value”)
* transactionsDf.drop(value, predError)
* transactionsDf.drop([“predError”, “value”])
* transactionsDf.drop([col(“predError”), col(“value”)])
Explanation

Output of correct code block:

+————-+——-+———+—-+

|transactionId|storeId|productId| f|

+————-+——-+———+—-+

| 1| 25| 1|null|

| 2| 2| 2|null|

| 3| 25| 3|null|

+————-+——-+———+—-+

To solve this question, you should be fmailiar with the drop() API. The order of column names does not matter

– in this question the order differs in some answers just to confuse you. Also, drop() does not take a list. The *cols operator in the documentation means that all arguments passed to drop() are interpreted as column names.

More info: pyspark.sql.DataFrame.drop – PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 2

NEW QUESTION 57

Which of the following code blocks concatenates rows of DataFrames transactionsDf and transactionsNewDf, omitting any duplicates?
* transactionsDf.concat(transactionsNewDf).unique()
* transactionsDf.union(transactionsNewDf).distinct()
* spark.union(transactionsDf, transactionsNewDf).distinct()
* transactionsDf.join(transactionsNewDf, how=”union”).distinct()
* transactionsDf.union(transactionsNewDf).unique()
Explanation

DataFrame.unique() and DataFrame.concat() do not exist and union() is not a method of the SparkSession. In addition, there is no union option for the join method in the DataFrame.join() statement.

More info: pyspark.sql.DataFrame.union – PySpark 3.1.2 documentation

Static notebook | Dynamic notebook: See test 2

NEW QUESTION 58

Which of the following code blocks reads JSON file imports.json into a DataFrame?
* spark.read().mode(“json”).path(“/FileStore/imports.json”)
* spark.read.format(“json”).path(“/FileStore/imports.json”)
* spark.read(“json”, “/FileStore/imports.json”)
* spark.read.json(“/FileStore/imports.json”)
* spark.read().json(“/FileStore/imports.json”)
Explanation

Static notebook | Dynamic notebook: See test 1

(https://flrs.github.io/spark_practice_tests_code/#1/25.html ,

https://bit.ly/sparkpracticeexams_import_instructions)

NEW QUESTION 59

Which of the following code blocks immediately removes the previously cached DataFrame transactionsDf from memory and disk?
* array_remove(transactionsDf, “*”)
* transactionsDf.unpersist()

(Correct)
* del transactionsDf
* transactionsDf.clearCache()
* transactionsDf.persist()
Explanation

transactionsDf.unpersist()

Correct. The DataFrame.unpersist() command does exactly what the question asks for – it removes all cached parts of the DataFrame from memory and disk.

del transactionsDf

False. While this option can help remove the DataFrame from memory and disk, it does not do so immediately. The reason is that this command just notifies the Python garbage collector that the transactionsDf now may be deleted from memory. However, the garbage collector does not do so immediately and, if you wanted it to run immediately, would need to be specifically triggered to do so. Find more information linked below.

array_remove(transactionsDf, “*”)

Incorrect. The array_remove method from pyspark.sql.functions is used for removing elements from arrays in columns that match a specific condition. Also, the first argument would be a column, and not a DataFrame as shown in the code block.

transactionsDf.persist()

No. This code block does exactly the opposite of what is asked for: It caches (writes) DataFrame transactionsDf to memory and disk. Note that even though you do not pass in a specific storage level here, Spark will use the default storage level (MEMORY_AND_DISK).

transactionsDf.clearCache()

Wrong. Spark’s DataFrame does not have a clearCache() method.

More info: pyspark.sql.DataFrame.unpersist – PySpark 3.1.2 documentation, python – How to delete an RDD in PySpark for the purpose of releasing resources? – Stack Overflow Static notebook | Dynamic notebook: See test 3

NEW QUESTION 60

The code block displayed below contains an error. The code block should return the average of rows in column value grouped by

unique storeId. Find the error.

Code block:

transactionsDf.agg(“storeId”).avg(“value”)
* Instead of avg(“value”), avg(col(“value”)) should be used.
* The avg(“value”) should be specified as a second argument to agg() instead of being appended to it.
* All column names should be wrapped in col() operators.
* agg should be replaced by groupBy.
* “storeId” and “value” should be swapped.

Explanation

Static notebook | Dynamic notebook: See test 1

(https://flrs.github.io/spark_practice_tests_code/#1/30.html ,

https://bit.ly/sparkpracticeexams_import_instructions)

NEW QUESTION 61

Which of the following code blocks returns a DataFrame with approximately 1,000 rows from the 10,000-row DataFrame itemsDf, without any duplicates, returning the same rows even if the code block is run twice?
* itemsDf.sampleBy(“row”, fractions={0: 0.1}, seed=82371)
* itemsDf.sample(fraction=0.1, seed=87238)
* itemsDf.sample(fraction=1000, seed=98263)
* itemsDf.sample(withReplacement=True, fraction=0.1, seed=23536)
* itemsDf.sample(fraction=0.1)

Explanation

itemsDf.sample(fraction=0.1, seed=87238)

Correct. If itemsDf has 10,000 rows, this code block returns about 1,000, since DataFrame.sample() is never guaranteed to return an exact amount of rows. To ensure you are not returning duplicates, you should leave the withReplacement parameter at False, which is the default. Since the question specifies that the same rows should be returned even if the code block is run twice, you need to specify a seed. The number passed in the seed does not matter as long as it is an integer.

itemsDf.sample(withReplacement=True, fraction=0.1, seed=23536)

Incorrect. While this code block fulfills almost all requirements, it may return duplicates. This is because withReplacement is set to True.

Here is how to understand what replacement means: Imagine you have a bucket of 10,000 numbered balls and you need to take 1,000 balls at random from the bucket (similar to the problem in the question). Now, if you would take those balls with replacement, you would take a ball, note its number, and put it back into the bucket, meaning the next time you take a ball from the bucket there would be a chance you could take the exact same ball again. If you took the balls without replacement, you would leave the ball outside the bucket and not put it back in as you take the next 999 balls.

itemsDf.sample(fraction=1000, seed=98263)

Wrong. The fraction parameter needs to have a value between 0 and 1. In this case, it should be 0.1, since

1,000/10,000 = 0.1.

itemsDf.sampleBy(“row”, fractions={0: 0.1}, seed=82371)

No, DataFrame.sampleBy() is meant for stratified sampling. This means that based on the values in a column in a DataFrame, you can draw a certain fraction of rows containing those values from the DataFrame (more details linked below). In the scenario at hand, sampleBy is not the right operator to use because you do not have any information about any column that the sampling should depend on.

itemsDf.sample(fraction=0.1)

Incorrect. This code block checks all the boxes except that it does not ensure that when you run it a second time, the exact same rows will be returned. In order to achieve this, you would have to specify a seed.

More info:

– pyspark.sql.DataFrame.sample – PySpark 3.1.2 documentation

– pyspark.sql.DataFrame.sampleBy – PySpark 3.1.2 documentation

– Types of Samplings in PySpark 3. The explanations of the sampling… | by Pinar Ersoy | Towards Data Science

NEW QUESTION 62

Which of the following describes a shuffle?
* A shuffle is a process that is executed during a broadcast hash join.
* A shuffle is a process that compares data across executors.
* A shuffle is a process that compares data across partitions.
* A shuffle is a Spark operation that results from DataFrame.coalesce().
* A shuffle is a process that allocates partitions to executors.

Explanation

A shuffle is a Spark operation that results from DataFrame.coalesce().

No. DataFrame.coalesce() does not result in a shuffle.

A shuffle is a process that allocates partitions to executors.

This is incorrect.

A shuffle is a process that is executed during a broadcast hash join.

No, broadcast hash joins avoid shuffles and yield performance benefits if at least one of the two tables is small in size (<= 10 MB by default). Broadcast hash joins can avoid shuffles because instead of exchanging partitions between executors, they broadcast a small table to all executors that then perform the rest of the join operation locally.

A shuffle is a process that compares data across executors.

No, in a shuffle, data is compared across partitions, and not executors.

More info: Spark Repartition & Coalesce – Explained (https://bit.ly/32KF7zS)

NEW QUESTION 63

Which of the following describes characteristics of the Spark UI?
* Via the Spark UI, workloads can be manually distributed across executors.
* Via the Spark UI, stage execution speed can be modified.
* The Scheduler tab shows how jobs that are run in parallel by multiple users are distributed across the cluster.
* There is a place in the Spark UI that shows the property spark.executor.memory.
* Some of the tabs in the Spark UI are named Jobs, Stages, Storage, DAGs, Executors, and SQL.

Explanation

There is a place in the Spark UI that shows the property spark.executor.memory.

Correct, you can see Spark properties such as spark.executor.memory in the Environment tab.

Some of the tabs in the Spark UI are named Jobs, Stages, Storage, DAGs, Executors, and SQL.

Wrong – Jobs, Stages, Storage, Executors, and SQL are all tabs in the Spark UI. DAGs can be inspected in the

“Jobs” tab in the job details or in the Stages or SQL tab, but are not a separate tab.

Via the Spark UI, workloads can be manually distributed across distributors.

No, the Spark UI is meant for inspecting the inner workings of Spark which ultimately helps understand, debug, and optimize Spark transactions.

Via the Spark UI, stage execution speed can be modified.

No, see above.

The Scheduler tab shows how jobs that are run in parallel by multiple users are distributed across the cluster.

No, there is no Scheduler tab.

NEW QUESTION 64

Which of the following code blocks saves DataFrame transactionsDf in location /FileStore/transactions.csv as a CSV file and throws an error if a file already exists in the location?
* transactionsDf.write.save(“/FileStore/transactions.csv”)
* transactionsDf.write.format(“csv”).mode(“error”).path(“/FileStore/transactions.csv”)
* transactionsDf.write.format(“csv”).mode(“ignore”).path(“/FileStore/transactions.csv”)
* transactionsDf.write(“csv”).mode(“error”).save(“/FileStore/transactions.csv”)
* transactionsDf.write.format(“csv”).mode(“error”).save(“/FileStore/transactions.csv”)

Explanation

Static notebook | Dynamic notebook: See test 1

(https://flrs.github.io/spark_practice_tests_code/#1/28.html ,

https://bit.ly/sparkpracticeexams_import_instructions)

NEW QUESTION 65

Which of the following code blocks returns a single-column DataFrame showing the number of words in column supplier of DataFrame itemsDf?

Sample of DataFrame itemsDf:

1.+——+————————–+——————-+

2.|itemId|attributes |supplier |

3.+——+————————–+——————-+

4.|1 |[blue, winter, cozy] |Sports Company Inc.|

5.|2 |[red, summer, fresh, cooling]|YetiX |

6.|3 |[green, summer, travel] |Sports Company Inc.|

7.+——+————————–+——————-+
* itemsDf.split(“supplier”, ” “).count()
* itemsDf.split(“supplier”, ” “).size()
* itemsDf.select(word_count(“supplier”))
* spark.select(size(split(col(supplier), ” “)))
* itemsDf.select(size(split(“supplier”, ” “)))

Explanation

Output of correct code block:

+——————————-+

|size(split(supplier, , -1))|

+——————————-+

| 3|

| 1|

| 3|

+————————-+

This question shows a typical use case for the split command: Splitting a string into words. An additional difficulty is that you are asked to count the words. Although it is tempting to use the count method here, the size method (as in: size of an array) is actually the correct one to use. Familiarize yourself with the split and the size methods using the linked documentation below.

More info:

Split method: pyspark.sql.functions.split – PySpark 3.1.2 documentation Size method: pyspark.sql.functions.size – PySpark 3.1.2 documentation Static notebook | Dynamic notebook: See test 2