Pass Your Exam With 100% Verified Associate-Developer-Apache-Spark Exam Questions [Q20-Q44

Pass Your Exam With 100% Verified Associate-Developer-Apache-Spark Exam Questions
Associate-Developer-Apache-Spark Dumps PDF - Associate-Developer-Apache-Spark Real Exam Questions Answers

Q20. Which of the following statements about the differences between actions and transformations is correct?
* Actions are evaluated lazily, while transformations are not evaluated lazily.
* Actions generate RDDs, while transformations do not.
* Actions do not send results to the driver, while transformations do.
* Actions can be queued for delayed execution, while transformations can only be processed immediately.
* Actions can trigger Adaptive Query Execution, while transformation cannot.

Explanation

Actions can trigger Adaptive Query Execution, while transformation cannot.

Correct. Adaptive Query Execution optimizes queries at runtime. Since transformations are evaluated lazily, Spark does not have any runtime information to optimize the query until an action is called. If Adaptive Query Execution is enabled, Spark will then try to optimize the query based on the feedback it gathers while it is evaluating the query.

Actions can be queued for delayed execution, while transformations can only be processed immediately.

No, there is no such concept as “delayed execution” in Spark. Actions cannot be evaluated lazily, meaning that they are executed immediately.

Actions are evaluated lazily, while transformations are not evaluated lazily.

Incorrect, it is the other way around: Transformations are evaluated lazily and actions trigger their evaluation.

Actions generate RDDs, while transformations do not.

No. Transformations change the data and, since RDDs are immutable, generate new RDDs along the way.

Actions produce outputs in Python and data types (integers, lists, text files,…) based on the RDDs, but they do not generate them.

Here is a great tip on how to differentiate actions from transformations: If an operation returns a DataFrame, Dataset, or an RDD, it is a transformation. Otherwise, it is an action.

Actions do not send results to the driver, while transformations do.

No. Actions send results to the driver. Think about running DataFrame.count(). The result of this command will return a number to the driver. Transformations, however, do not send results back to the driver. They produce RDDs that remain on the worker nodes.

More info: What is the difference between a transformation and an action in Apache Spark? | Bartosz Mikulski, How to Speed up SQL Queries with Adaptive Query Execution

Q21. Which of the following statements about Spark’s execution hierarchy is correct?
* In Spark’s execution hierarchy, a job may reach over multiple stage boundaries.
* In Spark’s execution hierarchy, manifests are one layer above jobs.
* In Spark’s execution hierarchy, a stage comprises multiple jobs.
* In Spark’s execution hierarchy, executors are the smallest unit.
* In Spark’s execution hierarchy, tasks are one layer above slots.

Explanation

In Spark’s execution hierarchy, a job may reach over multiple stage boundaries.

Correct. A job is a sequence of stages, and thus may reach over multiple stage boundaries.

In Spark’s execution hierarchy, tasks are one layer above slots.

Incorrect. Slots are not a part of the execution hierarchy. Tasks are the lowest layer.

In Spark’s execution hierarchy, a stage comprises multiple jobs.

No. It is the other way around – a job consists of one or multiple stages.

In Spark’s execution hierarchy, executors are the smallest unit.

False. Executors are not a part of the execution hierarchy. Tasks are the smallest unit!

In Spark’s execution hierarchy, manifests are one layer above jobs.

Wrong. Manifests are not a part of the Spark ecosystem.

Q22. Which of the following code blocks creates a new DataFrame with two columns season and wind_speed_ms where column season is of data type string and column wind_speed_ms is of data type double?
* spark.DataFrame({“season”: [“winter”,”summer”], “wind_speed_ms”: [4.5, 7.5]})
* spark.createDataFrame([(“summer”, 4.5), (“winter”, 7.5)], [“season”, “wind_speed_ms”])
* 1. from pyspark.sql import types as T

2. spark.createDataFrame(((“summer”, 4.5), (“winter”, 7.5)), T.StructType([T.StructField(“season”,
* CharType()), T.StructField(“season”, T.DoubleType())]))
* spark.newDataFrame([(“summer”, 4.5), (“winter”, 7.5)], [“season”, “wind_speed_ms”])
* spark.createDataFrame({“season”: [“winter”,”summer”], “wind_speed_ms”: [4.5, 7.5]})
Explanation

spark.createDataFrame([(“summer”, 4.5), (“winter”, 7.5)], [“season”, “wind_speed_ms”]) Correct. This command uses the Spark Session’s createDataFrame method to create a new DataFrame. Notice how rows, columns, and column names are passed in here: The rows are specified as a Python list. Every entry in the list is a new row. Columns are specified as Python tuples (for example (“summer”, 4.5)). Every column is one entry in the tuple.

The column names are specified as the second argument to createDataFrame(). The documentation (link below) shows that “when schema is a list of column names, the type of each column will be inferred from data” (the first argument). Since values 4.5 and 7.5 are both float variables, Spark will correctly infer the double type for column wind_speed_ms. Given that all values in column

“season” contain only strings, Spark will cast the column appropriately as string.

Find out more about SparkSession.createDataFrame() via the link below.

spark.newDataFrame([(“summer”, 4.5), (“winter”, 7.5)], [“season”, “wind_speed_ms”]) No, the SparkSession does not have a newDataFrame method.

from pyspark.sql import types as T

spark.createDataFrame(((“summer”, 4.5), (“winter”, 7.5)), T.StructType([T.StructField(“season”,

T.CharType()), T.StructField(“season”, T.DoubleType())]))

No. pyspark.sql.types does not have a CharType type. See link below for available data types in Spark.

spark.createDataFrame({“season”: [“winter”,”summer”],

“wind_speed_ms”: [4.5, 7.5]}) No, this is not correct Spark syntax. If you have considered this option to be correct, you may have some experience with Python’s pandas package, in which this would be correct syntax. To create a Spark DataFrame from a Pandas DataFrame, you can simply use spark.createDataFrame(pandasDf) where pandasDf is the Pandas DataFrame.

Find out more about Spark syntax options using the examples in the documentation for SparkSession.createDataFrame linked below.

spark.DataFrame({“season”: [“winter”,”summer”], “wind_speed_ms”: [4.5, 7.5]}) No, the Spark Session (indicated by spark in the code above) does not have a DataFrame method.

More info: pyspark.sql.SparkSession.createDataFrame – PySpark 3.1.1 documentation and Data Types – Spark 3.1.2 Documentation Static notebook | Dynamic notebook: See test 1

Q23. Which of the following describes a way for resizing a DataFrame from 16 to 8 partitions in the most efficient way?
* Use operation DataFrame.repartition(8) to shuffle the DataFrame and reduce the number of partitions.
* Use operation DataFrame.coalesce(8) to fully shuffle the DataFrame and reduce the number of partitions.
* Use a narrow transformation to reduce the number of partitions.
* Use a wide transformation to reduce the number of partitions.

Use operation DataFrame.coalesce(0.5) to halve the number of partitions in the DataFrame.
Explanation

Use a narrow transformation to reduce the number of partitions.

Correct! DataFrame.coalesce(n) is a narrow transformation, and in fact the most efficient way to resize the DataFrame of all options listed. One would run DataFrame.coalesce(8) to resize the DataFrame.

Use operation DataFrame.coalesce(8) to fully shuffle the DataFrame and reduce the number of partitions.

Wrong. The coalesce operation avoids a full shuffle, but will shuffle data if needed. This answer is incorrect because it says “fully shuffle” – this is something the coalesce operation will not do. As a general rule, it will reduce the number of partitions with the very least movement of data possible. More info:

distributed computing – Spark – repartition() vs coalesce() – Stack Overflow Use operation DataFrame.coalesce(0.5) to halve the number of partitions in the DataFrame.

Incorrect, since the num_partitions parameter needs to be an integer number defining the exact number of partitions desired after the operation. More info: pyspark.sql.DataFrame.coalesce – PySpark 3.1.2 documentation Use operation DataFrame.repartition(8) to shuffle the DataFrame and reduce the number of partitions.

No. The repartition operation will fully shuffle the DataFrame. This is not the most efficient way of reducing the number of partitions of all listed options.

Use a wide transformation to reduce the number of partitions.

No. While possible via the DataFrame.repartition(n) command, the resulting full shuffle is not the most efficient way of reducing the number of partitions.

Q24. Which of the following describes slots?
* Slots are dynamically created and destroyed in accordance with an executor’s workload.

* To optimize I/O performance, Spark stores data on disk in multiple slots.
* A Java Virtual Machine (JVM) working as an executor can be considered as a pool of slots for task execution.
* A slot is always limited to a single core.

Slots are the communication interface for executors and are used for receiving commands and sending results to the driver.
Explanation

Slots are the communication interface for executors and are used for receiving commands and sending results to the driver.

Wrong, executors communicate with the driver directly.

Slots are dynamically created and destroyed in accordance with an executor’s workload.

No, Spark does not actively create and destroy slots in accordance with the workload. Per executor, slots are made available in accordance with how many cores per executor (property spark.executor.cores) and how many CPUs per task (property spark.task.cpus) the Spark configuration calls for.

A slot is always limited to a single core.

No, a slot can span multiple cores. If a task would require multiple cores, it would have to be executed through a slot that spans multiple cores.

In Spark documentation, “core” is often used interchangeably with “thread”, although “thread” is the more accurate word. A single physical core may be able to make multiple threads available. So, it is better to say that a slot can span multiple threads.

To optimize I/O performance, Spark stores data on disk in multiple slots.

No – Spark stores data on disk in multiple partitions, not slots.

More info: Spark Architecture | Distributed Systems Architecture

Q25. Which of the following code blocks displays various aggregated statistics of all columns in DataFrame transactionsDf, including the standard deviation and minimum of values in each column?
* transactionsDf.summary()
* transactionsDf.agg(“count”, “mean”, “stddev”, “25%”, “50%”, “75%”, “min”)
* transactionsDf.summary(“count”, “mean”, “stddev”, “25%”, “50%”, “75%”, “max”).show()
* transactionsDf.agg(“count”, “mean”, “stddev”, “25%”, “50%”, “75%”, “min”).show()
* transactionsDf.summary().show()
Explanation

The DataFrame.summary() command is very practical for quickly calculating statistics of a DataFrame. You need to call .show() to display the results of the calculation. By default, the command calculates various statistics (see documentation linked below), including standard deviation and minimum.

Note that the answer that lists many options in the summary() parentheses does not include the minimum, which is asked for in the question.

Answer options that include agg() do not work here as shown, since DataFrame.agg() expects more complex, column-specific instructions on how to aggregate values.

More info:

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

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

Static notebook | Dynamic notebook: See test 3

Q26. Which of the following code blocks returns a new DataFrame with the same columns as DataFrame transactionsDf, except for columns predError and value which should be removed?
* transactionsDf.drop([“predError”, “value”])
* transactionsDf.drop(“predError”, “value”)
* transactionsDf.drop(col(“predError”), col(“value”))
* transactionsDf.drop(predError, value)
* transactionsDf.drop(“predError & value”)
Explanation

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

Static notebook | Dynamic notebook: See test 2

Q27. Which of the following describes Spark’s way of managing memory?
* Spark uses a subset of the reserved system memory.
* Storage memory is used for caching partitions derived from DataFrames.
* As a general rule for garbage collection, Spark performs better on many small objects than few big objects.
* Disabling serialization potentially greatly reduces the memory footprint of a Spark application.
* Spark’s memory usage can be divided into three categories: Execution, transaction, and storage.
Explanation

Spark’s memory usage can be divided into three categories: Execution, transaction, and storage.

No, it is either execution or storage.

As a general rule for garbage collection, Spark performs better on many small objects than few big objects.

No, Spark’s garbage collection runs faster on fewer big objects than many small objects.

Disabling serialization potentially greatly reduces the memory footprint of a Spark application.

The opposite is true – serialization reduces the memory footprint, but may impact performance in a negative way.

Spark uses a subset of the reserved system memory.

No, the reserved system memory is separate from Spark memory. Reserved memory stores Spark’s internal objects.

More info: Tuning – Spark 3.1.2 Documentation, Spark Memory Management | Distributed Systems Architecture, Learning

Spark, 2nd Edition, Chapter 7

Q28. Which of the following code blocks returns a single row from DataFrame transactionsDf?

Full 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.| 4| null| null| 3| 2|null|

8.| 5| null| null| null| 2|null|

9.| 6| 3| 2| 25| 2|null|

10.+————-+———+—–+——-+———+—-+
* transactionsDf.where(col(“storeId”).between(3,25))
* transactionsDf.filter((col(“storeId”)!=25) | (col(“productId”)==2))
* transactionsDf.filter(col(“storeId”)==25).select(“predError”,”storeId”).distinct()
* transactionsDf.select(“productId”, “storeId”).where(“storeId == 2 OR storeId != 25”)
* transactionsDf.where(col(“value”).isNull()).select(“productId”, “storeId”).distinct()
Explanation

Output of correct code block:

+———+——-+

|predError|storeId|

+———+——-+

| 3| 25|

+———+——-+

This question is difficult because it requires you to understand different kinds of commands and operators. All answers are valid Spark syntax, but just one expression returns a single-row DataFrame.

For reference, here is what the incorrect answers return:

transactionsDf.filter((col(“storeId”)!=25) | (col(“productId”)==2)) returns

+————-+———+—–+——-+———+—-+

|transactionId|predError|value|storeId|productId| f|

+————-+———+—–+——-+———+—-+

| 2| 6| 7| 2| 2|null|

| 4| null| null| 3| 2|null|

| 5| null| null| null| 2|null|

| 6| 3| 2| 25| 2|null|

+————-+———+—–+——-+———+—-+

transactionsDf.where(col(“storeId”).between(3,25)) returns

+————-+———+—–+——-+———+—-+

|transactionId|predError|value|storeId|productId| f|

+————-+———+—–+——-+———+—-+

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

| 3| 3| null| 25| 3|null|

| 4| null| null| 3| 2|null|

| 6| 3| 2| 25| 2|null|

+————-+———+—–+——-+———+—-+

transactionsDf.where(col(“value”).isNull()).select(“productId”, “storeId”).distinct() returns

+———+——-+

|productId|storeId|

+———+——-+

| 3| 25|

| 2| 3|

| 2| null|

+———+——-+

transactionsDf.select(“productId”, “storeId”).where(“storeId == 2 OR storeId != 25”) returns

+———+——-+

|productId|storeId|

+———+——-+

| 2| 2|

| 2| 3|

+———+——-+

Static notebook | Dynamic notebook: See test 2

Q29. The code block displayed below contains an error. The code block is intended to join DataFrame itemsDf with the larger DataFrame transactionsDf on column itemId. Find the error.

Code block:

transactionsDf.join(itemsDf, “itemId”, how=”broadcast”)
* The syntax is wrong, how= should be removed from the code block.
* The join method should be replaced by the broadcast method.
* Spark will only perform the broadcast operation if this behavior has been enabled on the Spark cluster.
* The larger DataFrame transactionsDf is being broadcasted, rather than the smaller DataFrame itemsDf.
* broadcast is not a valid join type.

Explanation

broadcast is not a valid join type.

Correct! The code block should read transactionsDf.join(broadcast(itemsDf), “itemId”). This would imply an inner join (this is the default in DataFrame.join()), but since the join type is not given in the question, this would be a valid choice.

The larger DataFrame transactionsDf is being broadcasted, rather than the smaller DataFrame itemsDf.

This option does not apply here, since the syntax around broadcasting is incorrect.

Spark will only perform the broadcast operation if this behavior has been enabled on the Spark cluster.

No, it is enabled by default, since the spark.sql.autoBroadcastJoinThreshold property is set to 10 MB by default. If that property would be set to -1, then broadcast joining would be disabled.

More info: Performance Tuning – Spark 3.1.1 Documentation (https://bit.ly/3gCz34r) The join method should be replaced by the broadcast method.

No, DataFrame has no broadcast() method.

The syntax is wrong, how= should be removed from the code block.

No, having the keyword argument how= is totally acceptable.

Q30. 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”))
* The parentheses around the column names need to be removed and .select() needs to be appended to the code block.
* 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.

(Correct)
* 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.
* Operator coalesce needs to be replaced by repartition and the parentheses around the column names need to be replaced by square brackets.
* Operator coalesce needs to be replaced by repartition.
Explanation

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 question specifies that after the execution the DataFrame should be divided. So, indirectly this question 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

Q31. The code block shown below should read all files with the file ending .png in directory path into Spark.

Choose the answer that correctly fills the blanks in the code block to accomplish this.

spark.__1__.__2__(__3__).option(__4__, “*.png”).__5__(path)
* 1. read()

2. format

3. “binaryFile”

4. “recursiveFileLookup”

5. load
* 1. read

2. format

3. “binaryFile”

4. “pathGlobFilter”

5. load
* 1. read

2. format

3. binaryFile

4. pathGlobFilter

5. load
* 1. open

2. format

3. “image”

4. “fileType”

5. open
* 1. open

2. as

3. “binaryFile”

4. “pathGlobFilter”

5. load
Explanation

Correct code block:

spark.read.format(“binaryFile”).option(“recursiveFileLookup”, “*.png”).load(path)
Spark can deal with binary files, like images. Using the binaryFile format specification in the SparkSession’s read API is the way to read in those files. Remember that, to access the read API, you need to start the command with spark.read. The pathGlobFilter option is a great way to filter files by name (and ending). Finally, the path can be specified using the load operator – the open operator shown in one of the answers does not exist.

Q32. The code block displayed below contains an error. The code block is intended to return all columns of DataFrame transactionsDf except for columns predError, productId, and value. Find the error.

Excerpt of DataFrame transactionsDf:

transactionsDf.select(~col(“predError”), ~col(“productId”), ~col(“value”))
* The select operator should be replaced by the drop operator and the arguments to the drop operator should be column names predError, productId and value wrapped in the col operator so they should be expressed like drop(col(predError), col(productId), col(value)).
* The select operator should be replaced with the deselect operator.
* The column names in the select operator should not be strings and wrapped in the col operator, so they should be expressed like select(~col(predError), ~col(productId), ~col(value)).
* The select operator should be replaced by the drop operator.
* The select operator should be replaced by the drop operator and the arguments to the drop operator should be column names predError, productId and value as strings.

(Correct)
Explanation

Correct code block:

transactionsDf.drop(“predError”, “productId”, “value”)

Static notebook | Dynamic notebook: See test 1

Q33. Which of the following describes characteristics of the Spark driver?
* The Spark driver requests the transformation of operations into DAG computations from the worker nodes.
* If set in the Spark configuration, Spark scales the Spark driver horizontally to improve parallel processing performance.
* The Spark driver processes partitions in an optimized, distributed fashion.
* In a non-interactive Spark application, the Spark driver automatically creates the SparkSession object.
* The Spark driver’s responsibility includes scheduling queries for execution on worker nodes.
Explanation

The Spark driver requests the transformation of operations into DAG computations from the worker nodes.

No, the Spark driver transforms operations into DAG computations itself.

If set in the Spark configuration, Spark scales the Spark driver horizontally to improve parallel processing performance.

No. There is always a single driver per application, but one or more executors.

The Spark driver processes partitions in an optimized, distributed fashion.

No, this is what executors do.

In a non-interactive Spark application, the Spark driver automatically creates the SparkSession object.

Wrong. In a non-interactive Spark application, you need to create the SparkSession object. In an interactive Spark shell, the Spark driver instantiates the object for you.

Q34. Which of the following options describes the responsibility of the executors in Spark?
* The executors accept jobs from the driver, analyze those jobs, and return results to the driver.
* The executors accept tasks from the driver, execute those tasks, and return results to the cluster manager.
* The executors accept tasks from the driver, execute those tasks, and return results to the driver.
* The executors accept tasks from the cluster manager, execute those tasks, and return results to the driver.
* The executors accept jobs from the driver, plan those jobs, and return results to the cluster manager.

Explanation

More info: Running Spark: an overview of Spark’s runtime architecture – Manning (https://bit.ly/2RPmJn9)

Q35. The code block shown below should return a copy of DataFrame transactionsDf with an added column cos.

This column should have the values in column value converted to degrees and having the cosine of those converted values taken, rounded to two decimals. Choose the answer that correctly fills the blanks in the code block to accomplish this.

Code block:

transactionsDf.__1__(__2__, round(__3__(__4__(__5__)),2))
* 1. withColumn

2. col(“cos”)

3. cos

4. degrees

5. transactionsDf.value
* 1. withColumnRenamed

2. “cos”

3. cos

4. degrees

5. “transactionsDf.value”
* 1. withColumn

2. “cos”

3. cos

4. degrees

5. transactionsDf.value
* 1. withColumn

2. col(“cos”)

3. cos

4. degrees

5. col(“value”)

E

. 1. withColumn

2. “cos”

3. degrees

4. cos

5. col(“value”)
Explanation

Correct code block:

transactionsDf.withColumn(“cos”, round(cos(degrees(transactionsDf.value)),2)) This question is especially confusing because col, “cos” are so similar. Similar-looking answer options can also appear in the exam and, just like in this question, you need to pay attention to the details to identify what the correct answer option is.

The first answer option to throw out is the one that starts with withColumnRenamed: The question NO:

speaks specifically of adding a column. The withColumnRenamed operator only renames an existing column, however, so you cannot use it here.

Next, you will have to decide what should be in gap 2, the first argument of transactionsDf.withColumn().

Looking at the documentation (linked below), you can find out that the first argument of withColumn actually needs to be a string with the name of the column to be added. So, any answer that includes col(“cos”) as the option for gap 2 can be disregarded.

This leaves you with two possible answers. The real difference between these two answers is where the cos and degree methods are, either in gaps 3 and 4, or vice-versa. From the question you can find out that the new column should have “the values in

column value converted to degrees and having the cosine of those converted values taken”. This prescribes you a clear order of operations: First, you convert values from column value to degrees and then you take the cosine of those values. So, the inner parenthesis (gap 4) should contain the degree method and then, logically, gap 3 holds the cos method. This leaves you with just one possible correct answer.

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

Q36. The code block shown below should store DataFrame transactionsDf on two different executors, utilizing the executors’ memory as much as possible, but not writing anything to disk. Choose the answer that correctly fills the blanks in the code block to accomplish this.

1.from pyspark import StorageLevel

2.transactionsDf.__1__(StorageLevel.__2__).__3__
* 1. cache

2. MEMORY_ONLY_2

3. count()
* 1. persist

2. DISK_ONLY_2

3. count()
* 1. persist

2. MEMORY_ONLY_2

3. select()
* 1. cache

2. DISK_ONLY_2

3. count()
* 1. persist

2. MEMORY_ONLY_2

3. count()
Explanation

Correct code block:

from pyspark import StorageLevel

transactionsDf.persist(StorageLevel.MEMORY_ONLY_2).count()

Only persist takes different storage levels, so any option using cache() cannot be correct. persist() is evaluated lazily, so an action needs to follow this command. select() is not an action, but count() is – so all options using select() are incorrect.

Finally, the question states that “the executors’ memory should be utilized as much as possible, but not writing anything to disk”. This points to a MEMORY_ONLY storage level. In this storage level, partitions that do not fit into memory will be recomputed when they are needed, instead of being written to disk, as with the storage option MEMORY_AND_DISK. Since the data need to be duplicated across two executors, _2 needs to be appended to the storage level.

Static notebook | Dynamic notebook: See test 2

Q37. The code block displayed below contains an error. The code block should count the number of rows that have a predError of either 3 or 6. Find the error.

Code block:

transactionsDf.filter(col(‘predError’).in([3, 6])).count()
* The number of rows cannot be determined with the count() operator.
* Instead of filter, the select method should be used.
* The method used on column predError is incorrect.
* Instead of a list, the values need to be passed as single arguments to the in operator.
* Numbers 3 and 6 need to be passed as string variables.

Explanation

Correct code block:

transactionsDf.filter(col(‘predError’).isin([3, 6])).count()

The isin method is the correct one to use here – the in method does not exist for the Column object.

More info: pyspark.sql.Column.isin – PySpark 3.1.2 documentation

Q38. In which order should the code blocks shown below be run in order to assign articlesDf a DataFrame that lists all items in column attributes ordered by the number of times these items occur, from most to least often?

Sample of DataFrame articlesDf:

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.+——+————————–+——————-+

* 1. articlesDf = articlesDf.groupby(“col”)

2. articlesDf = articlesDf.select(explode(col(“attributes”)))

3. articlesDf = articlesDf.orderBy(“count”).select(“col”)

4. articlesDf = articlesDf.sort(“count”,ascending=False).select(“col”)

5. articlesDf = articlesDf.groupby(“col”).count()
* 4, 5
* 2, 5, 3
* 5, 2
* 2, 3, 4
* 2, 5, 4
Explanation

Correct code block:

articlesDf = articlesDf.select(explode(col(‘attributes’)))

articlesDf = articlesDf.groupby(‘col’).count()

articlesDf = articlesDf.sort(‘count’,ascending=False).select(‘col’)

Output of correct code block:

+——-+

| col|

+——-+

| summer|

| winter|

| blue|

| cozy|

| travel|

| fresh|

| red|

|cooling|

| green|

+——-+

Static notebook | Dynamic notebook: See test 2

Q39. The code block shown below should write DataFrame transactionsDf to disk at path csvPath as a single CSV file, using tabs (t characters) as separators between columns, expressing missing values as string n/a, and omitting a header row with column names. Choose the answer that correctly fills the blanks in the code block to accomplish this.

transactionsDf.__1__.write.__2__(__3__, ” “).__4__.__5__(csvPath)
* 1. coalesce(1)

2. option

3. “sep”

4. option(“header”, True)

5. path
* 1. coalesce(1)

2. option

3. “colsep”

4. option(“nullValue”, “n/a”)

5. path
* 1. repartition(1)

2. option

3. “sep”

4. option(“nullValue”, “n/a”)

5. csv

(Correct)
* 1. csv

2. option

3. “sep”

4. option(“emptyValue”, “n/a”)

5. path

* 1. repartition(1)

2. mode

3. “sep”

4. mode(“nullValue”, “n/a”)

5. csv
Explanation

Correct code block:

transactionsDf.repartition(1).write.option(“sep”, “t”).option(“nullValue”, “n/a”).csv(csvPath) It is important here to understand that the question specifically asks for writing the DataFrame as a single CSV file. This should trigger you to think about partitions. By default, every partition is written as a separate file, so you need to include repatition(1) into your call. coalesce(1) works here, too!

Secondly, the question is very much an invitation to search through the parameters in the Spark documentation that work with DataFrameWriter.csv (link below). You will also need to know that you need an option() statement to apply these parameters.

The final concern is about the general call structure. Once you have called accessed write of your DataFrame, options follow and then you write the DataFrame with csv. Instead of csv(csvPath), you could also use save(csvPath, format=’csv’) here.

More info: pyspark.sql.DataFrameWriter.csv – PySpark 3.1.1 documentation Static notebook | Dynamic notebook: See test 1

Q40. 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)

Q41. Which of the following describes the conversion of a computational query into an execution plan in Spark?
* Spark uses the catalog to resolve the optimized logical plan.
* The catalog assigns specific resources to the optimized memory plan.
* The executed physical plan depends on a cost optimization from a previous stage.
* Depending on whether DataFrame API or SQL API are used, the physical plan may differ.
* The catalog assigns specific resources to the physical plan.

Explanation

The executed physical plan depends on a cost optimization from a previous stage.

Correct! Spark considers multiple physical plans on which it performs a cost analysis and selects the final physical plan in accordance with the lowest-cost outcome of that analysis. That final physical plan is then executed by Spark.

Spark uses the catalog to resolve the optimized logical plan.

No. Spark uses the catalog to resolve the unresolved logical plan, but not the optimized logical plan. Once the unresolved logical plan is resolved, it is then optimized using the Catalyst Optimizer.

The optimized logical plan is the input for physical planning.

The catalog assigns specific resources to the physical plan.

No. The catalog stores metadata, such as a list of names of columns, data types, functions, and databases.

Spark consults the catalog for resolving the references in a logical plan at the beginning of the conversion of the query into an execution plan. The result is then an optimized logical plan.

Depending on whether DataFrame API or SQL API are used, the physical plan may differ.

Wrong – the physical plan is independent of which API was used. And this is one of the great strengths of Spark!

The catalog assigns specific resources to the optimized memory plan.

There is no specific “memory plan” on the journey of a Spark computation.

More info: Spark’s Logical and Physical plans … When, Why, How and Beyond. | by Laurent Leturgez | datalex | Medium

Q42. Which of the following is a characteristic of the cluster manager?
* Each cluster manager works on a single partition of data.
* The cluster manager receives input from the driver through the SparkContext.
* The cluster manager does not exist in standalone mode.
* The cluster manager transforms jobs into DAGs.
* In client mode, the cluster manager runs on the edge node.

Explanation

The cluster manager receives input from the driver through the SparkContext.

Correct. In order for the driver to contact the cluster manager, the driver launches a SparkContext. The driver then asks the cluster manager for resources to launch executors.

In client mode, the cluster manager runs on the edge node.

No. In client mode, the cluster manager is independent of the edge node and runs in the cluster.

The cluster manager does not exist in standalone mode.

Wrong, the cluster manager exists even in standalone mode. Remember, standalone mode is an easy means to deploy Spark across a whole cluster, with some limitations. For example, in standalone mode, no other frameworks can run in parallel with Spark. The cluster manager is part of Spark in standalone deployments however and helps launch and maintain resources across the cluster.

The cluster manager transforms jobs into DAGs.

No, transforming jobs into DAGs is the task of the Spark driver.

Each cluster manager works on a single partition of data.

No. Cluster managers do not work on partitions directly. Their job is to coordinate cluster resources so that they can be requested by and allocated to Spark drivers.

More info: Introduction to Core Spark Concepts * BigData

Q43. Which of the following code blocks returns all unique values of column storeId in DataFrame transactionsDf?
* transactionsDf[“storeId”].distinct()
* transactionsDf.select(“storeId”).distinct()

(Correct)
* transactionsDf.filter(“storeId”).distinct()
* transactionsDf.select(col(“storeId”).distinct())
* transactionsDf.distinct(“storeId”)
Explanation

distinct() is a method of a DataFrame. Knowing this, or recognizing this from the documentation, is the key to solving this question.

More info: pyspark.sql.DataFrame.distinct – PySpark 3.1.2 documentation Static notebook | Dynamic notebook: See test 2

Q44. Which of the following statements about broadcast variables is correct?
* Broadcast variables are serialized with every single task.
* Broadcast variables are commonly used for tables that do not fit into memory.
* Broadcast variables are immutable.
* Broadcast variables are occasionally dynamically updated on a per-task basis.
* Broadcast variables are local to the worker node and not shared across the cluster.
Explanation

Broadcast variables are local to the worker node and not shared across the cluster.

This is wrong because broadcast variables are meant to be shared across the cluster. As such, they are never just local to the worker node, but available to all worker nodes.

Broadcast variables are commonly used for tables that do not fit into memory.

This is wrong because broadcast variables can only be broadcast because they are small and do fit into memory.

Broadcast variables are serialized with every single task.

This is wrong because they are cached on every machine in the cluster, precisely avoiding to have to be serialized with every single task.

Broadcast variables are occasionally dynamically updated on a per-task basis.

This is wrong because broadcast variables are immutable – they are never updated.

More info: Spark – The Definitive Guide, Chapter 14