Databricks

Developer-for-Apache-Spark-Python

Certified Associate Developer for Apache Spark - Python

Latest Version: 6.0

Question: 1

Which of the following statements accurately describes the role of the Spark Driver in a Spark application?

A. The Driver is responsible for distributing tasks to executors and managing data partitioning.
B. The Driver is responsible for storing the final results of the computation.
C. The Driver is the main process that coordinates the execution of the Spark application, including submitting tasks to executors, tracking their progress, and aggregating results.
D. The Driver is responsible for reading and writing data to external data sources.
E. The Driver is responsible for managing the memory allocation for executors.

Answer: C

Explanation:
The Spark Driver is the main process that coordinates the execution of the Spark application. It is responsible for submitting tasks to executors, tracking their progress, and aggregating results. The Driver is not responsible for data partitioning, storing final results, reading/writing to external sources, or managing executor memory.

Question: 2

In the context of Spark, what is the primary function of a Spark Executor?

A. Executors are responsible for converting data from one format to another, like CSV to Parquet.
B. Executors are responsible for caching data in memory for faster retrieval.
C. Executors are responsible for running tasks assigned by the Driver and processing data on the worker nodes.
D. Executors are responsible for scheduling tasks across different worker nodes.
E. Executors are responsible for managing the communication between the Driver and the worker nodes.

Answer: C

Explanation:
Spark Executors are responsible for executing tasks assigned by the Driver. They are deployed on the worker nodes and process data locally. Data conversion, caching, task scheduling, and communication management are not the primary functions of executors.

Question: 3

What is the key advantage of using Spark's Adaptive Query Execution (AQE)?

A. AQE automatically optimizes the physical execution plan based on runtime data characteristics, potentially leading to faster query execution.
B. AQE allows for dynamic allocation of resources to executors based on workload demands.
C. AQE enables the use of different data sources without requiring specific configuration changes.
D. AQE ensures data integrity by automatically validating data transformations during execution.
E. AQE eliminates the need for manual optimization of Spark queries.

Answer: A

Explanation:
Adaptive Query Execution (AQE) in Spark dynamically adjusts the physical execution plan based on runtime data characteristics. This can lead to faster query execution by making optimal decisions regarding data shuffling, partitioning, and other aspects of the query plan. While AQE can enhance performance, it does not directly address resource allocation, data source compatibility, data validation, or eliminate the need for manual optimization.

Question: 4

Which of the following scenarios would benefit the most from using Adaptive Query Execution (AQE) in Spark?

A. A query that processes a fixed dataset with a well-defined schema and predictable data distribution.
B. A query that performs simple aggregations on a small dataset, where performance is not critical.
C. A query that involves complex joins, aggregations, and data transformations on a large and potentially skewed dataset.
D. A query that reads data from a real-time data stream, where data characteristics are highly dynamic.
E. A query that performs batch processing of data with static characteristics.

Answer: C,D

Explanation:
AQE is most beneficial when dealing with complex queries on large and potentially skewed datasets, as well as queries on dynamic data streams. In these scenarios, AQE can effectively adapt to data characteristics and optimize the execution plan for improved performance. AQE is less impactful on simple queries with fixed datasets, where pre-optimization might already be sufficient.

Question: 5

How does Spark's AQE dynamically adjust the query execution plan?

A. By analyzing the data distribution and skewness during execution, AQE dynamically adjusts the number of partitions and the shuffling strategy to optimize data movement.

B. AQE dynamically adjusts the memory allocation for executors based on the data volume being processed.
C. AQE dynamically adjusts the number of worker nodes used for execution based on the workload demands.
D. AQE dynamically adjusts the data serialization format used for data exchange between executors.
E. AQE dynamically adjusts the Spark configuration settings based on the query performance metrics.

Answer: A

Explanation:
A major aspect of AQE's dynamic adjustments involves analyzing data distribution and skewness during execution. This information is used to dynamically adjust the number of partitions and the shuffling strategy, effectively optimizing data movement and potentially reducing query execution time. While AQE can influence resource allocation (like memory), it does not directly manage the number of worker nodes, serialization format, or Spark configuration settings.

Question: 6

What is the role of the Spark Shuffle Manager in Adaptive Query Execution (AQE)?

A. The Shuffle Manager is responsible for managing the data partitioning and shuffling process, allowing AQE to dynamically adjust these operations based on runtime data characteristics.
B. The Shuffle Manager is responsible for caching frequently accessed data in memory, which can improve AQE's performance.
C. The Shuffle Manager is responsible for validating data transformations during execution, ensuring data integrity for AQE.
D. The Shuffle Manager is responsible for scheduling tasks across different worker nodes, which can be dynamically adjusted by AQE.
E. The Shuffle Manager is responsible for managing the communication between the Driver and the worker nodes, which can be optimized by AQE.

Answer: A

Explanation:
The Spark Shuffle Manager plays a crucial role in AQE by managing the data partitioning and shuffling process. This allows AQE to dynamically adjust these operations based on runtime data characteristics. While the Shuffle Manager is important for data management, its role in AQE is primarily focused on data partitioning and shuffling, not caching, data validation, task scheduling, or communication management.

Question: 7

How does Spark's AQE address data skewness during query execution?

A. AQE identifies skewed data partitions and dynamically adds more executors to handle the increased workload.

B. AQE dynamically adjusts the shuffling strategy to redistribute skewed data across multiple partitions, improving load balancing.
C. AQE automatically filters out skewed data points, reducing the overall data volume and improving performance.
D. AQE applies a pre-processing step to normalize skewed data before the query execution.
E. AQE dynamically increases the memory allocation for executors handling skewed data partitions.

Answer: B

Explanation:
AQE tackles data skewness by dynamically adjusting the shuffling strategy. It identifies skewed partitions and redistributes the skewed data across multiple partitions, improving load balancing and reducing the impact of skewness on query performance. AQE does not automatically add executors, filter out data, apply pre-processing steps, or increase memory allocation as primary methods to address data skewness.

Question: 8

You have a DataFrame named 'products' with columns 'product_id', 'name', and 'price'. You want to create a new DataFrame where each row represents a pair of products, along with the difference between their prices. How would you achieve this using the Spark DataFrame API?

A.
```
from pyspark.sql.functions import col, broadcast, abs

products_joined = products.join(broadcast(products), col('product_id') != col('product_id'), 'left')
products_joined = products_joined.withColumn('price_difference', abs(col('price') - col('price')))
```
B.
```
from pyspark.sql.functions import col, broadcast, abs

products_joined = products.crossJoin(products)
products_joined = products_joined.withColumn('price_difference', abs(col('price') - col('price')))
```
C.
```
from pyspark.sql.functions import col, broadcast, abs

products_joined = products.join(products, col('product_id') != col('product_id'), 'left')
products_joined = products_joined.withColumn('price_difference', abs(col('price') - col('price')))
```
D.
```
from pyspark.sql.functions import col, broadcast, abs

products_joined = products.join(products, col('product_id') == col('product_id'), 'left')
products_joined = products_joined.withColumn('price_difference', abs(col('price') - col('price')))
```
E.
```
from pyspark.sql.functions import col, broadcast, abs

products_joined = products.join(products, col('product_id') != col('product_id'), 'inner')
products_joined = products_joined.withColumn('price_difference', abs(col('price') - col('price')))
```

Answer: B

Explanation:
The correct code snippet is Option B : ‘from pyspark.sql.functions import col, broadcast, abs products_joined = products.crossJoin(products) products_joined = products_joined.withColumn('price_difference', abs(col('price') - col('price')))’. This code uses crossJoin’ to create a new DataFrame where each row represents a pair of products from the original DataFrame. The ‘abs' function is then used to calculate the absolute difference between the prices of each product pair, creating a new column 'price_difference'. Here's why the other options are incorrect: Option A : This code uses a join with a condition where != col('product_id')'. This condition will always be true and effectively create a cross join. However, the ‘broadcast' function is unnecessary as the join condition is not on the 'product_id' column. Option C : Similar to Option A, this code uses a 'left' join with the wrong condition. The ‘broadcast function is unnecessary. Option D : This code uses an ‘inner’ join with the condition , which will only create pairs of products with the same 'product_id'. This is not what the question requires. Option E : This code uses an 'inner' join with a condition != , which is not correct as it will only join products with different IDs, potentially missing some pairs.

Question: 9

You have a DataFrame named 'df' with columns 'name', 'age', and 'city'. You want to create a new DataFrame that only includes the 'name' and 'age' columns, and rename 'age' to 'years old'. Which of the following Python code snippets accomplishes this correctly?

☐
```
df.select("name", "age").withColumnRenamed("age", "years_old")
```

☐
```
df.select("name", df.col("age").alias("years_old"))
```

☐
```
df.select(df.name, df.age.alias("years_old"))
```

☐
```
df.select(df["name"], df["age"].alias("years_old"))
```

☐
```
df.select(['name', 'age']).withColumnRenamed('age', 'years_old')
```

A. Option A
B. Option B
C. Option C
D. Option D
E. Option E

Answer: B,D

Explanation:
Both options B and D are correct. Option B uses the ‘col' function to select the 'age' column and then renames it using the ‘alias' function. Option D uses bracket notation to select the 'name' and 'age' columns, and then renames 'age' using the ‘alias' function. Option A would only select the 'name' and 'age' columns without renaming. Option C attempts to use dot notation, which is not valid for selecting columns in Spark. Option E uses bracket notation, but the selection is done in a list, which is not how Spark selects columns.

Question: 10

Given a DataFrame 'df' with columns 'name', 'age', and 'city', you want to create a new column named 'age_category' based on the following logic: - If 'age' is less than 18, assign 'Child' - If 'age' is between 18 and 65, assign 'Adult' - If 'age' is greater than 65, assign 'Senior' Which Python code snippet correctly implements this logic?

☐
```
df.withColumn("age_category", when(df.age < 18, "Child").otherwise(when(df.age > 65, "Senior").otherwise("Adult")))
```

☐
```
df.withColumn("age_category", when(df.age < 18, "Child").when(df.age > 65, "Senior").otherwise("Adult"))
```

☐
```
df.withColumn("age_category", when(df.age < 18, "Child").when(df.age >= 18 and df.age <= 65, "Adult").when(df.age > 65, "Senior"))
```

☐
```
df.withColumn("age_category", when(df.age < 18, "Child").otherwise(when(df.age >= 18 and df.age <= 65, "Adult").otherwise("Senior")))
```

☐
```
df.withColumn("age_category", when(df.age < 18, "Child").when(df.age <= 65, "Adult").otherwise("Senior"))
```

A. Option A
B. Option B
C. Option C
D. Option D
E. Option E

Answer: A,D

Explanation:
Both options A and D correctly implement the logic using the ‘when' and ‘otherwise' functions. Option B is incorrect because it does not use the 'otherwise' function, so any age greater than 65 will be assigned 'Adult'. Option C incorrectly uses multiple 'when' functions without the otherwise' function, so the last condition will be evaluated for all ages above 65. Option E is incorrect because it does not cover the case for ages greater than 65.

60 DAY MONEY BACK GUARANTEE
100%
SATISFACTION GUARANTEED