Download Free Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5 Exam PDF | PrepBolt

Don't miss out! Download the latest free Databricks Certified Associate Developer for Apache Spark 3.5 - Python exam PDF questions. Access real Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5 dumps with verified answers and boost your chances to pass your certification on the first try with PrepBolt Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5 exam pdf questions and answers.

Thank you for Downloading Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5 exam PDF Demo

https://prepbolt.com/Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5.html

Question 1

Question Type: MultipleChoice

A data engineer is running a Spark job to process a dataset of 1 TB stored in distributed storage. The cluster has 10 nodes, each with 16 CPUs. Spark UI shows:

Low number of Active Tasks

Many tasks complete in milliseconds

Fewer tasks than available CPUs

Which approach should be used to adjust the partitioning for optimal resource allocation?

Options:

A- Set the number of partitions equal to the total number of CPUs in the cluster
B- Set the number of partitions to a fixed value, such as 200
C- Set the number of partitions equal to the number of nodes in the cluster
D- Set the number of partitions by dividing the dataset size (1 TB) by a reasonable partition size, such as 128 MB

Answer:

D

Explanation:

Spark's best practice is to estimate partition count based on data volume and a reasonable partition size --- typically 128 MB to 256 MB per partition.

With 1 TB of data: 1 TB / 128 MB ~8000 partitions

This ensures that tasks are distributed across available CPUs for parallelism and that each task processes an optimal volume of data.

Option A (equal to cores) may result in partitions that are too large.

Option B (fixed 200) is arbitrary and may underutilize the cluster.

Option C (nodes) gives too few partitions (10), limiting parallelism.

Question 2

Question Type: MultipleChoice

1 of 55. A data scientist wants to ingest a directory full of plain text files so that each record in the output DataFrame contains the entire contents of a single file and the full path of the file the text was read from.

The first attempt does read the text files, but each record contains a single line. This code is shown below:

txt_path = "/datasets/raw_txt/*"

df = spark.read.text(txt_path) # one row per line by default

df = df.withColumn("file_path", input_file_name()) # add full path

Which code change can be implemented in a DataFrame that meets the data scientist's requirements?

Options:

A- Add the option wholetext to the text() function.
B- Add the option lineSep to the text() function.
C- Add the option wholetext=False to the text() function.
D- Add the option lineSep=', ' to the text() function.

Answer:

A

Explanation:

By default, the spark.read.text() method reads a text file one line per record. This means that each line in a text file becomes one row in the resulting DataFrame.

To read each file as a single record, Apache Spark provides the option wholetext, which, when set to True, causes Spark to treat the entire file contents as one single string per row.

Correct usage:

df = spark.read.option('wholetext', True).text(txt_path)

This way, each record in the DataFrame will contain the full content of one file instead of one line per record.

To also include the file path, the function input_file_name() can be used to create an additional column that stores the complete path of the file being read:

```
from pyspark.sql.functions import input_file_name

df = spark.read.option('wholetext', True).text(txt_path) \

.withColumn('file_path', input_file_name())
```

This approach satisfies both requirements from the question:

Each record holds the entire contents of a file.

Each record also contains the file path from which the text was read.

Why the other options are incorrect:

B or D (lineSep) -- The lineSep option only defines the delimiter between lines. It does not combine the entire file content into a single record.

C (wholetext=False) -- This is the default behavior, which still reads one record per line rather than per file.

Reference (Databricks Apache Spark 3.5 -- Python / Study Guide):

PySpark API Reference: DataFrameReader.text --- describes the wholetext option.

PySpark Functions: input_file_name() --- adds a column with the source file path.

Databricks Certified Associate Developer for Apache Spark Exam Guide (June 2025): Section ''Using Spark DataFrame APIs'' --- covers reading files and handling DataFrames.

Question 3

Question Type: MultipleChoice

29 of 55.

A Spark application is experiencing performance issues in client mode due to the driver being resource-constrained.

How should this issue be resolved?

Options:

A- Switch the deployment mode to cluster mode.
B- Add more executor instances to the cluster.
C- Increase the driver memory on the client machine.
D- Switch the deployment mode to local mode.

Answer:

A

Explanation:

In client mode, the driver runs on the same machine that submitted the job (often a developer's workstation). If the driver has insufficient memory or CPU, it becomes a bottleneck.

Solution: Run the job in cluster mode.

In cluster mode, the driver runs inside the cluster on a worker node, benefiting from distributed cluster resources and improved performance for large workloads.

Why the other options are incorrect:

B: Executors handle tasks, not driver overhead.

C: May help temporarily but doesn't scale; cluster mode is best practice.

D: Local mode runs everything on one JVM --- worse for large workloads.

Databricks Exam Guide (June 2025): Section ''Using Spark Connect to Deploy Applications'' --- explains client vs. cluster deployment modes.

Spark Deployment Overview --- driver behavior and resource management.

==========

Question 4

Question Type: MultipleChoice

A data engineer is running a batch processing job on a Spark cluster with the following configuration:

10 worker nodes

16 CPU cores per worker node

64 GB RAM per node

The data engineer wants to allocate four executors per node, each executor using four cores.

What is the total number of CPU cores used by the application?

Options:

A- 160
B- 64
C- 80
D- 40

Answer:

C

Explanation:

If each of the 10 nodes runs 4 executors, and each executor is assigned 4 CPU cores:

Executors per node = 4

Cores per executor = 4

Total executors = 4 * 10 = 40

Total cores = 40 executors * 4 cores = 160 cores

However, Spark uses 1 core for overhead on each node when managing multiple executors. Thus, the practical allocation is:

Total usable executors = 4 executors/node 10 nodes = 40

Total cores = 4 cores 40 executors = 160

Answe r : A --- but the question asks specifically about ''CPU cores used by the application,'' assuming all

However, if you are considering 4 executors/node 4 cores = 16 cores per node, across 10 nodes, that's 160.

Final Answe r: A

Question 5

Question Type: MultipleChoice

19 of 55. A Spark developer wants to improve the performance of an existing PySpark UDF that runs a hash function not available in the standard Spark functions library.

The existing UDF code is:

import hashlib

from pyspark.sql.types import StringType

def shake_256(raw):

return hashlib.shake_256(raw.encode()).hexdigest(20)

shake_256_udf = udf(shake_256, StringType())

The developer replaces this UDF with a Pandas UDF for better performance:

@pandas_udf(StringType())

def shake_256(raw: str) -> str:

return hashlib.shake_256(raw.encode()).hexdigest(20)

However, the developer receives this error:

TypeError: Unsupported signature: (raw: str) -> str

What should the signature of the shake_256() function be changed to in order to fix this error?

A.

def shake_256(raw: str) -> str:

B.

def shake_256(raw: [pd.Series]) -> pd.Series:

C.

def shake_256(raw: pd.Series) -> pd.Series:

D.

def shake_256(raw: [str]) -> [str]:

Options:

A- Option A
B- Option B
C- Option C
D- Option D

Answer:

C

Explanation:

Pandas UDFs (vectorized UDFs) process entire Pandas Series objects, not scalar values. Each

invocation operates on a column (Series) rather than a single value.

Correct syntax:

@pandas_udf(StringType())

def shake_256(raw: pd.Series) -> pd.Series:

return raw.apply(lambda x: hashlib.shake_256(x.encode()).hexdigest(20))

This allows Spark to apply the function in a vectorized way, improving performance significantly over traditional Python UDFs.

Why the other options are incorrect:

A/D: These define scalar functions --- not compatible with Pandas UDFs.

B: Uses an invalid type hint [pd.Series] (not a valid Python type annotation).

PySpark Pandas API --- @pandas_udf decorator and function signatures.

Databricks Exam Guide (June 2025): Section ''Using Pandas API on Apache Spark'' --- creating and invoking Pandas UDFs.

Thank You for trying Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5 PDF Demo

To try our Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5 practice exam software visit link below

https://prepbolt.com/Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5.html

Start Your Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5 Preparation

Use Coupon “SAVE50” for extra 50% discount on the purchase of Practice Test Software. Test your Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5 preparation with actual exam questions.