Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5 Exam Questions

Total Questions: 130+
Demo Questions: 25
Version: Updated for 2025

Prepared and Verified by Cert Empire – Your Trusted IT Certification Partner

For Access to the full set of Updated Questions – Visit: Databricks-Certified-Associate-Developer-for-Apache-Spark-3.5 Exam Questions by Cert Empire

Question: 1

A data scientist is working with a Spark DataFrame called customerDF that contains customer information. The DataFrame has a column named email with customer email addresses. The data scientist needs to split this column into username and domain parts. Which code snippet splits the email column into username and domain columns?

A. customerDF.select(col("email").substr(0, 5).alias("username"), col("email").substr(-5).alias("domain"))

B. customerDF.withColumn("username", split(col("email"), "@").getItem(0)) \ .withColumn("domain", split(col("email"), "@").getItem(1))

C. customerDF.withColumn("username", substringindex(col("email"), "@", 1)) \ .withColumn("domain", substringindex(col("email"), "@", -1))

D. customerDF.select(regexpreplace(col("email"), "@", "").alias("username"), regexpreplace(col("email"), "@", "").alias("domain"))

Answer:

B

Explanation:

CertEmpire

The most robust and idiomatic way to solve this problem using the Spark DataFrame API is to use the split function. This function takes the column and a delimiter ("@") as arguments and returns an array of strings. For an email like "user@domain.com", it produces "user", "domain.com". The getItem(0) method is then used to extract the first element (the username), and getItem(1) extracts the second element (the domain). The withColumn transformation is used twice to add these new values as columns to the original DataFrame. This approach correctly handles variable-length usernames and domains.

Why Incorrect Options are Wrong:

A. This option uses substr with fixed-length indices (5 characters), which is incorrect as email usernames and domains have variable lengths.
C. This option fails on edge cases. If an email address is missing the "@" delimiter, substringindex will incorrectly return the entire string for both the username and domain parts, instead of indicating a missing part (e.g., with a null).
D. This option uses regexpreplace to remove the "@" symbol, which does not split the column. It would incorrectly result in a column containing the full email address without the "@".

References:

1. Apache Spark 3.5.1 Documentation, pyspark.sql.functions.split: This function splits a string around a pattern. The documentation shows its usage as split(str, pattern, limit=-1), which is ideal for this scenario.
Source: https://spark.apache.org/docs/3.5.1/api/python/reference/pyspark.sql/api/pyspark.sql.functions.split.html
2. Apache Spark 3.5.1 Documentation, pyspark.sql.Column.getItem: This method is used to get an item at a specific ordinal from an array, which is the output of the split function.
Source: https://spark.apache.org/docs/3.5.1/api/python/reference/pyspark.sql/api/pyspark.sql.Column.getItem.html
3. Databricks SQL Language Reference, substringindex function: This documentation confirms the behavior that makes option C incorrect. It states, "If delim is not found within str, str is returned." This means for a malformed email like "user", the domain column would incorrectly be populated with "user" instead of NULL.
Source: https://docs.databricks.com/en/sql/language-manual/functions/substringindex.html

CertEmpire

Question: 2

A data engineer wants to create an external table from a JSON file located at /data/input.json with the following requirements: Create an external table named users Automatically infer schema Merge records with differing schemas Which code snippet should the engineer use? Options:

A. CREATE TABLE users USING json OPTIONS (path '/data/input.json')

B. CREATE EXTERNAL TABLE users USING json OPTIONS (path '/data/input.json')

C. CREATE EXTERNAL TABLE users USING json OPTIONS (path '/data/input.json', mergeSchema 'true')

D. CREATE EXTERNAL TABLE users USING json OPTIONS (path '/data/input.json', schemaMerge 'true')

Answer:

C

Explanation:

The solution requires creating an external table, which necessitates the CREATE EXTERNAL TABLE statement. The data source is specified with USING json. The OPTIONS clause is used to configure the data source. The path option points to the data's location. To fulfill the requirement of merging records with differing schemas, the mergeSchema option must be set to 'true'. This option instructs Spark to scan all files to infer a superset schema, accommodating schema evolution across the dataset.

CertEmpire

Why Incorrect Options are Wrong:

A. This statement creates a managed table, not an external one, because it omits the EXTERNAL keyword. The table's data would be moved into the warehouse directory.
B. This statement correctly creates an external table but fails to include the mergeSchema option, which is a specific requirement for handling differing schemas.
D. This statement uses an incorrect option name, schemaMerge. The correct and officially documented option for merging schemas in the JSON data source is mergeSchema.

References:

1. Databricks SQL Language Reference, CREATE TABLE: The documentation confirms the syntax for creating an external table using a data source: CREATE EXTERNAL TABLE tablename ... USING datasource OPTIONS (key1=val1, ...)
Source: Databricks Documentation SQL reference Data definition language CREATE TABLE.
2. Apache Spark 3.x Documentation, Data Sources, JSON Files: This official documentation lists the available options for the JSON data source. It explicitly defines the mergeSchema option.
Source: Apache Spark Documentation SQL, DataFrames and Datasets Guide Data Sources

JSON Files. Section: "Data Source Option". The documentation states: mergeSchema (default false): "If true, infers the schema from all JSON files, and merges them."
3. Databricks Documentation, Read and write JSON files: This page mirrors the Apache Spark documentation, confirming the use and behavior of the mergeSchema option within the Databricks environment.
Source: Databricks Documentation Develop and run code Work with data Read and write data Read and write JSON files. Section: "Options".

CertEmpire

Question: 3

Given this code:

```
inputStream
.withWatermark("event_time", "10 minutes")
.groupBy(window("event_time", "15 minutes"))
.count()
```

.withWatermark("eventtime", "10 minutes") .groupBy(window("eventtime", "15 minutes")) .count()
What happens to data that arrives after the watermark threshold? Options:

A. Records that arrive later than the watermark threshold (10 minutes) will automatically be included
in the aggregation if they fall within the 15-minute window.

B. Any data arriving more than 10 minutes after the watermark threshold will be ignored and not included in the aggregation.

C. Data arriving more than 10 minutes after the latest watermark will still be included in the aggregation but will be placed into the next window.

CertEmpire

D. The watermark ensures that late data arriving within 10 minutes of the latest eventtime will be processed and included in the windowed aggregation.

Answer:

B

Explanation:

The .withWatermark("eventtime", "10 minutes") operation sets a threshold for how late data can be before it is dropped. The watermark is calculated as the maximum event time seen by the engine so far, minus the specified delay (10 minutes). Any incoming record whose eventtime is older than this calculated watermark is considered too late and is subsequently dropped, meaning it will not be included in any windowed aggregation. The primary purpose of the watermark is to allow the streaming engine to bound the amount of state it needs to maintain for aggregations by discarding data that is considered too old to be relevant.

Why Incorrect Options are Wrong:

A: This is incorrect. The watermark's function is precisely to drop records that are too late, not to automatically include them.
C: This is incorrect. Late data that exceeds the watermark threshold is dropped entirely. It is not

re-assigned to a different window.

D: This describes data that is late but still arrives within the 10-minute tolerance window. The question specifically asks about data that arrives after this threshold has been passed.

References:

1. Apache Spark 3.5.0 Official Documentation, Structured Streaming Programming Guide:
Section: Handling Late Data and Watermarking
Content: "Now, consider what happens if one of the events arrives late to the application. For example, a word generated at 12:04 (i.e. event time) is received by the application at 12:11. Since the application is using 10 minutes as the threshold, the watermark is at max(12:04, 12:01) - 10 mins = 11:54. As the event time 12:04 is newer than the watermark 11:54, the application adds the event to the buffer for the window 12:00 - 12:10..... However, when the watermark is updated to 12:01, the old window 12:00 - 12:05 is older than the watermark, and any late data that belongs to that window is dropped." This illustrates that once the watermark advances past a window's boundary, late data for that window is dropped.
2. Databricks Documentation, Structured Streaming - Watermarking:
Section: How watermarking works
Content: "You can set a threshold for how long to wait for late data to arrive. This is done by setting a watermark. .... Any records that arrive after the threshold are discarded." This statement directly confirms that data arriving after the defined threshold is dropped.

CertEmpire

Question: 4

A developer runs:

```
df.write.partitionBy("color", "fruit").parquet("/path/to/output")
```

What is the result? Options:

A. It stores all data in a single Parquet file.

B. It throws an error if there are null values in either partition column.

C. It appends new partitions to an existing Parquet file.

D. It creates separate directories for each unique combination of color and fruit.

Answer:

D

Explanation:

The DataFrameWriter.partitionBy() method is used to partition the output data based on the values of one or more columns. When df.write.partitionBy("color", "fruit") is executed, Spark creates a hierarchical directory structure at the specified path. For each unique combination of values in the "color" and "fruit" columns, a separate subdirectory is created. For example, data for red apples would be in a path like .../color=red/fruit=apple/. This physical data layout on disk is a key optimization strategy, as it allows Spark to prune, or skip reading, irrelevant data directories for queries that filter on the partition columns.

CertEmpire

Why Incorrect Options are Wrong:

A. The purpose of partitionBy is to split data into multiple directories and files based on column values, not to consolidate it into a single file.
B. Spark does not throw an error for null values in partition columns. Instead, it writes them to a special default partition directory, typically named HIVEDEFAULTPARTITION.
C. The default save mode is errorifexists. To append, one must explicitly set .mode("append"). Even then, appending adds new files, it does not modify existing immutable Parquet files.

References:

1. Apache Spark 3.4.1 Documentation, pyspark.sql.DataFrameWriter.partitionBy: "Partitions the output by the given columns. When specified, the output is laid out on the file system similar to Hive's partitioning scheme. As an example, df.write.partitionBy('year', 'month').parquet(path) would create a directory structure like path/year=2016/month=1/."
Source: Apache Spark Documentation - pyspark.sql.DataFrameWriter.partitionBy
2. Databricks Documentation, Partitioning for Delta Lake: "Databricks uses partitioning to divide

your data into subdirectories based on the values of one or more columns... When you partition a table, you are telling Databricks to physically organize the data by values in one or more columns."
Source: Databricks Documentation - When to partition tables in Databricks
3. Book: Spark: The Definitive Guide by Bill Chambers and Matei Zaharia (O'Reilly Media, 2018), Chapter 9, "Partitioning": "Partitioning is a tool that allows you to separate your data into different directories based on the values in a certain column... This creates a directory structure that Spark SQL and other Spark applications can read from and use for improving performance by pruning away files that it knows do not contain relevant data." (p. 190). The chapter also discusses save modes (p. 186) and null handling in partitions.

CertEmpire

Question: 5

Which command overwrites an existing JSON file when writing a DataFrame?

A. df.write.mode("overwrite").json("path/to/file")

B. df.write.overwrite.json("path/to/file")

C. df.write.json("path/to/file", overwrite=True)

D. df.write.format("json").save("path/to/file", mode="overwrite")

Answer:

A, D

Explanation:

In Apache Spark, the DataFrameWriter API is used to save a DataFrame to an external storage system. To overwrite existing data, the save mode must be set to "overwrite".
Option A uses the standard method chaining: df.write returns a DataFrameWriter, .mode("overwrite") sets the desired save behavior, and .json("path/to/file") specifies the format and path, triggering the write operation.
Option D uses the generic .save() method. Here, .format("json") specifies the data source type, and the mode="overwrite" argument is passed directly into the .save() method, achieving the same result. Both are valid and documented ways to perform an overwrite operation.

CertEmpire

Why Incorrect Options are Wrong:

B. df.write.overwrite.json("path/to/file")
The DataFrameWriter API does not have a direct .overwrite method or attribute; the save mode is set exclusively using the .mode() method or as a parameter to .save().
C. df.write.json("path/to/file", overwrite=True)
The format-specific .json() method does not accept an overwrite boolean parameter. Save modes must be configured using the .mode() method prior to the write call.

References:

1. Apache Spark 3.5.1 Official Documentation, pyspark.sql.DataFrameWriter:
For Option A: The documentation shows the mode(saveMode) method, which sets the SaveMode, and the json(path, ...) method, which saves the content in JSON format. The standard usage is to chain these: df.write.mode('overwrite').json(path).
For Option D: The documentation for the save(path=None, format=None, mode=None, ...) method explicitly lists mode as a parameter. This confirms that df.write.format('json').save(path, mode='overwrite') is a valid syntax.
For Options B and C: The API documentation confirms the absence of a .overwrite method and an overwrite parameter within the .json() method, invalidating these options.

2. Databricks Official Documentation, "Read and write JSON files":
The documentation provides examples for writing data. The standard syntax shown for overwriting is df.write.mode("overwrite").json("/tmp/my-json-file"), which directly supports the correctness of Option A. It also explains the different save modes, including overwrite.

CertEmpire

Question: 6

Given the code:

```
df = spark.read.csv("large_dataset.csv")
filtered_df = df.filter(col("error_column")
                .contains("error"))
mapped_df = filtered_df.select(split(col("timestamp"), " ").getItem(0).alias("date"),
lit(1).alias("count"))
reduced_df = mapped_df.groupBy("date").sum("count")
reduced_df.count()
reduced_df.show()
```

df = spark.read.csv("largedataset.csv") filtereddf = df.filter(col("errorcolumn").contains("error")) mappeddf = filtereddf.select(split(col("timestamp"), " ").getItem(0).alias("date"), lit(1).alias("count")) reduceddf = mappeddf.groupBy("date").sum("count") reduceddf.count() reduceddf.show() At which point will Spark actually begin processing the data?

A. When the filter transformation is applied

B. When the count action is applied

C. When the groupBy transformation is applied

D. When the show action is applied

CertEmpire

Answer:

B

Explanation:

Apache Spark operates on the principle of lazy evaluation. It builds a Directed Acyclic Graph (DAG) of transformations (read, filter, select, groupBy) but does not execute any computation. The processing is only triggered when an action is called. In the provided code, count() is the first action invoked on the DataFrame reduceddf. This call forces Spark to execute the entire chain of preceding transformations-from reading the CSV file to filtering, selecting, and grouping-to compute the final result, which is the number of rows in the aggregated DataFrame.

Why Incorrect Options are Wrong:

A. filter is a narrow transformation. It only adds a step to the logical execution plan and does not trigger any data processing.
C. groupBy is a wide transformation that defines a shuffle operation. However, like all transformations, it is lazy and does not execute until an action is called.
D. show() is also an action that would trigger execution. However, it appears after the count() action in the code. Therefore, the computation begins when count() is called first.

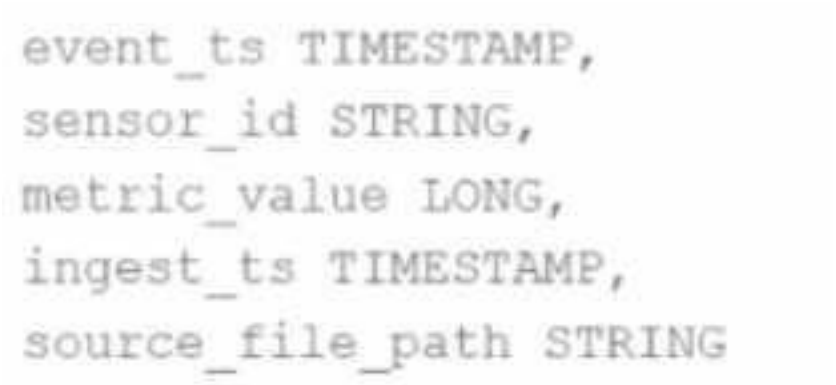
References:

1. Databricks Documentation - Apache Spark programming with Databricks: In the "Transformations" section, it states, "Transformations are lazy. Code in a notebook cell that defines a DataFrame and transformations does not run until you explicitly call an action." It lists count() and show() as common actions. This confirms that processing starts with the first action, count().
Source: Databricks Documentation, Introduction to structured data in PySpark, Section: "Transformations".
2. Apache Spark 3.4.1 Documentation - RDD Programming Guide: The fundamental concept of lazy evaluation is explained here. "All transformations in Spark are lazy, in that they do not compute their results right away... The transformations are only computed when an action requires a result to be returned to the driver program." The guide lists count() as a primary example of an action.
Source: Apache Spark Official Documentation, RDD Programming Guide, Section: "Basics" - "Lazy Evaluation".
3. Book: Spark: The Definitive Guide (by Bill Chambers and Matei Zaharia): Chapter 2, "A Gentle Introduction to Spark," explicitly details this behavior. It explains that transformations like filter() and groupBy() are lazy, and the logical plan is only executed when an action like count() is called.
Source: Chambers, B., & Zaharia, M. (2018). Spark: The Definitive Guide. O'Reilly Media, Inc. Chapter 2, "The Concept of Lazy Evaluation" (pp. 26-27).

CertEmpire

Question: 7

Given the schema:



eventts TIMESTAMP, sensorid STRING, metricvalue LONG, ingestts TIMESTAMP, sourcefilepath STRING The goal is to deduplicate based on: eventts, sensorid, and metricvalue. Options:

A. dropDuplicates on all columns (wrong criteria)

B. dropDuplicates with no arguments (removes based on all columns)

C. groupBy without aggregation (invalid use)

D. dropDuplicates on the exact matching fields

CertEmpire

Answer:

D

Explanation:

The dropDuplicates() transformation is the correct and most efficient method for this task. When a list of column names is passed as an argument to this function (e.g., df.dropDuplicates("eventts", "sensorid", "metricvalue")), Spark identifies and removes rows that have identical values only in those specified columns. The first encountered row among a set of duplicates is retained, while subsequent duplicates are discarded. This behavior precisely matches the requirement to deduplicate based on the specified three-column key.

Why Incorrect Options are Wrong:

A. dropDuplicates on all columns would incorrectly use ingestts and sourcefilepath in the deduplication criteria, failing to remove intended duplicates.
B. dropDuplicates with no arguments is functionally identical to using all columns, which is incorrect for the same reason as option A.
C. A groupBy operation is an intermediate step for aggregation. It returns a GroupedData object, not a deduplicated DataFrame, and is an incomplete operation for this goal.

References:

1. Apache Spark 3.5.1 Official Documentation, pyspark.sql.DataFrame.dropDuplicates:
Reference: "Return a new DataFrame with duplicate rows removed, optionally only considering certain columns. For a static batch DataFrame, it keeps the first row for each set of duplicates." The function signature is DataFrame.dropDuplicates(subset=None), where subset is an "optional list of column names to consider."
Location: Apache Spark API Docs pyspark.sql DataFrame API pyspark.sql.DataFrame.dropDuplicates.
2. Databricks Official Documentation, "Duplicate records":
Reference: "You can use dropDuplicates to remove duplicate rows from a DataFrame, optionally considering only a subset of columns... The following code drops duplicate rows from a DataFrame, considering only the columns 'name' and 'gender'."
Location: Databricks Documentation Apache Spark DataFrames Transformations Duplicate records.
3. University of California, Berkeley - CS 194, Data Science Courseware:
Reference: Lecture materials on Spark DataFrames often explain that df.dropDuplicates('col1', 'col2') is the standard method for removing duplicate records based on a subset of columns, contrasting it with df.distinct() which operates on all columns.
Location: Found in typical course materials for data engineering with Spark, such as UC Berkeley's Data Science curriculum resources.CertEmpire

Question: 8

An engineer has two DataFrames: df1 (small) and df2 (large). A broadcast join is used: python CopyEdit from pyspark.sql.functions import broadcast result = df2.join(broadcast(df1), on='id', how='inner') What is the purpose of using broadcast() in this scenario? Options:

A. It filters the id values before performing the join.

B. It increases the partition size for df1 and df2.

C. It reduces the number of shuffle operations by replicating the smaller DataFrame to all nodes.

D. It ensures that the join happens only when the id values are identical.

Answer:

C

Explanation:

The broadcast() function is a join hint that instructs Spark's query optimizer to use a broadcast hash join strategy. This strategy is highly efficient when joining a large DataFrame with a much smaller one. The smaller DataFrame (df1) is collected by the driver and then replicated (broadcasted) to every executor node in the cluster. Each executor can then perform the join locally against its partitions of the large DataFrame (df2). This avoids the expensive shuffle operation that would otherwise be required to redistribute the keys of the large DataFrame across the network, significantly improving performance.

CertEmpire

Why Incorrect Options are Wrong:

A. The broadcast() function is an optimization hint for the join execution plan; it does not perform data filtering.
B. broadcast() affects how data is distributed for the join operation, not the configured partition size of the DataFrames.
D. The join condition (on='id') and join type (how='inner') determine which rows match, not the broadcast() hint.

References:

1. Databricks Documentation, "Broadcast hash joins": "When you join a large table to a small table, a broadcast hash join is usually more performant than other join algorithms. A broadcast hash join broadcasts the small table to all nodes in the cluster and performs a hash join locally on each node. The main benefit of a broadcast hash join is that it avoids shuffling the large table."
2. Apache Spark 3.5.1 Documentation, "Join Optimization": "Broadcast hash join is used when one side of the join is small... Spark can broadcast the smaller join side to all executors. This avoids a shuffle of the data in both join sides." The documentation further explains that the broadcast hint can be used to explicitly force this strategy.

Question: 9

A Spark application suffers from too many small tasks due to excessive partitioning. How can this be fixed without a full shuffle? Options:

A. Use the distinct() transformation to combine similar partitions

B. Use the coalesce() transformation with a lower number of partitions

C. Use the sortBy() transformation to reorganize the data

D. Use the repartition() transformation with a lower number of partitions

Answer:

B

Explanation:

The coalesce() transformation is specifically designed to decrease the number of partitions in a DataFrame efficiently. It achieves this by combining existing partitions on the same executor, which minimizes data movement across the network and avoids a full shuffle. This makes it the ideal solution for consolidating an excessive number of small partitions into a smaller number of larger, more manageable ones, thereby reducing task overhead without incurring the high cost of a wide transformation.

CertEmpire

Why Incorrect Options are Wrong:

A. Use the distinct() transformation to combine similar partitions
The distinct() transformation is used to find unique records, which requires a full shuffle to compare all records. It is not a tool for managing partition count.
C. Use the sortBy() transformation to reorganize the data
The sortBy() transformation sorts the entire dataset, which is a wide operation that necessitates a full shuffle to move data between executors.
D. Use the repartition() transformation with a lower number of partitions
The repartition() transformation always triggers a full shuffle of the data across all executors, even when decreasing the partition count. This violates the core requirement of the question.

References:

1. Databricks Documentation, "Learning Spark, 2nd Edition": In Chapter 10, "Spark SQL and Performance," the authors state, "To reduce the number of partitions, we recommend using coalesce(), which is a more efficient version of repartition() that avoids a full shuffle of the data. It does this by having a smaller number of partitions 'swallow' a larger number of partitions, with data movement minimized to only what is necessary." (Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2020). Learning Spark, 2nd Ed. O'Reilly Media, Inc. Chapter 10, Section: "Coalesce

and Repartition").
2. Apache Spark 3.5.0 Official Documentation: The documentation for the coalesce function in the pyspark.sql.DataFrame API notes its efficiency: "This operation results in a narrow dependency, e.g. if you go from 1000 partitions to 100 partitions, there will not be a shuffle, instead each of the 100 new partitions will claim 10 of the current partitions." This confirms it avoids a full shuffle. (Source: Apache Spark Project, pyspark.sql.DataFrame.coalesce API documentation).
3. Databricks Documentation, "Repartition vs Coalesce": "Coalesce uses existing partitions to minimize the amount of data that's shuffled. Repartition creates new partitions and runs a full shuffle. Coalesce is faster, but it can lead to unevenly sized partitions. Repartition is slower, but it creates evenly sized partitions." (Source: Databricks Documentation Apache Spark Development Repartition).

CertEmpire

Question: 10

What is the relationship between jobs, stages, and tasks during execution in Apache Spark?
Options:

A. A job contains multiple stages, and each stage contains multiple tasks.

B. A job contains multiple tasks, and each task contains multiple stages.

C. A stage contains multiple jobs, and each job contains multiple tasks.

D. A stage contains multiple tasks, and each task contains multiple jobs.

Answer:

A

Explanation:

In Apache Spark, the execution hierarchy begins when an action is called on a DataFrame or RDD, which triggers a job. Spark's DAGScheduler analyzes the computation's dependency graph and divides the job into a set of stages. Stages are separated by wide transformations (shuffles), where data needs to be redistributed across the cluster. Each stage is then broken down into the smallest units of execution, called tasks. A task is a command sent to an executor to process a single data partition. Therefore, a single job is composed of one or more stages, and each stage is composed of one or more tasks.

CertEmpire

Why Incorrect Options are Wrong:

B. A job contains multiple tasks, and each task contains multiple stages.
This inverts the relationship between stages and tasks. A stage is a collection of tasks, not the other way around.
C. A stage contains multiple jobs, and each job contains multiple tasks.
This inverts the relationship between jobs and stages. A job is the highest-level computation unit, which contains stages.
D. A stage contains multiple tasks, and each task contains multiple jobs.
This incorrectly places jobs as the lowest-level unit. A job is the top-level computation triggered by an action.

References:

1. Apache Spark 3.4.1 Documentation, "Web UI": In the "Jobs Tab" section, it states, "A job is associated with a chain of RDD dependencies organized in a DAG... Each job gets divided into a set of stages, and each stage has a set of tasks." This explicitly defines the hierarchy as Job - Stage - Task.
Source: Apache Spark Official Documentation.

Section: Monitoring and Instrumentation - Web UI - Jobs Tab.
2. Databricks Documentation, "Apache Spark UI - Jobs": The documentation describes the Spark UI, explaining that clicking on a job leads to a detail page showing the stages of that job. Clicking on a stage then shows the tasks associated with it. This navigational structure mirrors the execution hierarchy.
Source: Databricks Official Documentation.
Section: Development - Apache Spark UI - Jobs.
3. Zaharia, M., et al. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. USENIX NSDI'12. This foundational paper on Spark describes the architecture. Section 3.2, "Job Scheduling," explains: "The scheduler computes a directed acyclic graph (DAG) of stages for each job... It then launches tasks to compute missing partitions from each stage until it has computed the target RDD." This confirms that jobs are broken into stages, which are executed via tasks.
Source: Peer-reviewed academic publication.
DOI: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf (Page 5, Section 3.2).

CertEmpire

Question: 11

What is the benefit of using Pandas on Spark for data transformations? Options:

A. It is available only with Python, thereby reducing the learning curve.

B. It computes results immediately using eager execution, making it simple to use.

C. It runs on a single node only, utilizing the memory with memory-bound DataFrames and hence cost-efficient.

D. It executes queries faster using all the available cores in the cluster as well as provides Pandas's
rich set of features.

Answer:

D

Explanation:

The primary benefit of the Pandas API on Spark is that it combines the familiar, productive, and feature-rich API of the pandas library with the distributed, parallel processing power of Apache Spark. This allows data scientists to scale their existing pandas code to work on large datasets that exceed the memory of a single machine. By translating pandas API calls into Spark's execution plan, it leverages all the cores across the cluster to execute transformations in a distributed manner, significantly improving performance on big data while maintaining a well-known interface.

CertEmpire

Why Incorrect Options are Wrong:

A. While the API does reduce the learning curve for Python users familiar with pandas, its main technical benefit is distributed execution, not its language exclusivity.
B. Pandas on Spark is built on Spark's core engine, which uses lazy evaluation. Computations are planned but not executed until an action is called, unlike standard pandas which is eagerly executed.
C. This describes the limitation of standard pandas, which runs on a single node. The purpose of Pandas on Spark is precisely to overcome this single-node bottleneck by distributing work across a cluster.

References:

1. Databricks Documentation, "Pandas API on Spark": "The pandas API on Spark allows you to scale your pandas workload to any size by running it distributed on a Spark cluster. If you are already familiar with pandas, you can be immediately productive with the pandas API on Spark...". This supports the combination of scalability (distributed on a cluster) and the use of a familiar, feature-rich API.

2. Apache SparkTM 3.5.1 Documentation, "pyspark.pandas": "This project makes data scientists more productive when interacting with big data, by implementing the pandas DataFrame API on top of Apache Spark." This directly states the goal is to provide the pandas API on top of Spark for big data, which aligns with executing queries across a cluster using familiar features.
3. Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). Learning Spark: Lightning-Fast Big Data Analysis. O'Reilly Media, Inc. (Conceptual basis): Chapter 1 discusses the limitations of single-machine processing (like standard pandas) and introduces Spark's model of distributed computation across a cluster as the solution for big data. The Pandas API on Spark is a direct application of this principle, providing a high-level API over the distributed engine.

CertEmpire

Question: 12

A data scientist wants each record in the DataFrame to contain: The first attempt at the code does read the text files but each record contains a single line. This code is shown below:

```
raw_txt_path = '/datasets/raw_txt/*'

corpus = spark.read.text(raw_txt_path) \
  .select('*', '_metadata.file_path')
```

The entire contents of a file The full file path The issue: reading line-by-line rather than full text per file. Code: corpus = spark.read.text("/datasets/rawtxt/*") \ .select('*', 'metadata.filepath') Which change will ensure one record per file? Options:

A. Add the option wholetext=True to the text() function

B. Add the option lineSep='\n' to the text() function

C. Add the option wholetext=False to the text() function

D. Add the option lineSep=", " to the text() function

CertEmpire

Answer:

A

Explanation:

The spark.read.text() method by default reads input files line by line, creating one DataFrame record for each line. To achieve the desired outcome of having one record per file, the wholetext option must be set to True. This option instructs Spark to treat the entire content of each file as a single value, thus creating a DataFrame where each row corresponds to a complete file. The metadata.filepath column can then be used to identify the source file for each record.

Why Incorrect Options are Wrong:

B. Add the option lineSep='\n' to the text() function: This specifies the line separator. Since \n is the default, this would not change the line-by-line reading behavior.
C. Add the option wholetext=False to the text() function: This is the default setting for the text data source. Explicitly setting it to false would ensure the file is read line-by-line, which is the opposite of the desired outcome.
D. Add the option lineSep=", " to the text() function: This changes the record delimiter from a newline to a comma and a space. It does not instruct Spark to read the entire file as a single

record.

References:

1. Apache Spark 3.5.1 Documentation, SQL Reference, Data Sources, Text:
"Text data source options: wholetext (default false): If true, reads each file as a single row." This directly supports the correct answer.
Reference: https://spark.apache.org/docs/latest/sql-data-sources-text.html
2. Databricks Documentation, Read and write text files:
"To read whole text files, set the wholeText option to true." This confirms the usage of the option within the Databricks environment.
Reference: https://docs.databricks.com/en/sql/language-manual/sql-ref-text-files.html
3. Chambers, B., & Zaharia, M. (2018). Spark: The Definitive Guide. O'Reilly Media, Inc.
Chapter 9, "Data Sources," Section on "Text Files": "If you would like to read in each file as a single row in a DataFrame, you can set the wholetext option to true." (Page 181). This provides an authoritative explanation of the option's purpose.

CertEmpire

Question: 13

Which Spark configuration controls the number of tasks that can run in parallel on the executor?
Options:

A. spark.executor.cores

B. spark.task.maxFailures

C. spark.driver.cores

D. spark.executor.memory

Answer:

A

Explanation:

The spark.executor.cores configuration property specifies the number of CPU cores allocated to each executor JVM. In Apache Spark's execution model, each core within an executor can run one task concurrently. Therefore, this setting directly controls the maximum number of tasks that can run in parallel on a single executor. For instance, if spark.executor.cores is set to 5, each executor in the cluster can execute up to five tasks simultaneously, assuming sufficient resources are available on the worker node. This is a fundamental parameter for tuning the parallelism of a Spark job.

CertEmpire

Why Incorrect Options are Wrong:

B. spark.task.maxFailures: This configuration controls fault tolerance by defining how many times a single task can be retried upon failure before the entire job is aborted. It does not affect parallelism.
C. spark.driver.cores: This setting allocates CPU cores for the driver process, which is responsible for creating the SparkContext, building the execution plan, and scheduling tasks, but not for executing them.
D. spark.executor.memory: This property sets the amount of heap memory allocated to each executor process. While crucial for task execution and data storage, it does not determine the number of concurrent tasks.

References:

1. Apache Spark 3.5.1 Documentation, Configuration Page, Application Properties:
spark.executor.cores: The documentation describes this as "The number of cores to use on each executor." This directly translates to the number of concurrent task slots available on that executor.
spark.task.maxFailures: Described as the "Number of failures of the same task before this job fails."

spark.driver.cores: Described as the "Number of cores to use for the driver process."
spark.executor.memory: Described as the "Amount of memory to use per executor process."
Source: Apache Spark Documentation. (2024). Configuration - Application Properties. Retrieved from https://spark.apache.org/docs/latest/configuration.html#application-properties
2. Databricks Documentation, Compute Configuration Reference, Spark Configuration:
The Databricks documentation explains how Spark properties are set and their impact on cluster performance. It clarifies that the total number of concurrent tasks on a cluster is a function of the number of workers and the cores per executor (spark.executor.cores), reinforcing that this property governs parallelism at the executor level.
Source: Databricks Documentation. (2024). Spark configuration. Retrieved from https://docs.databricks.com/en/clusters/spark-config.html

CertEmpire

Question: 14

How can a Spark developer ensure optimal resource utilization when running Spark jobs in Local Mode for testing? Options:

A. Configure the application to run in cluster mode instead of local mode.

B. Increase the number of local threads based on the number of CPU cores.

C. Use the spark.dynamicAllocation.enabled property to scale resources dynamically.

D. Set the spark.executor.memory property to a large value.

Answer:

B

Explanation:

In Spark's Local Mode, the entire application-driver, master, and executor-runs within a single Java Virtual Machine (JVM) on a single machine. Parallelism is achieved through multi-threading. To ensure optimal resource utilization, specifically of the CPU, the developer should configure the number of threads Spark can use for task execution. By setting the master URL to localk, where k is the number of threads, or more commonly local, Spark is instructed to create as many worker threads as there are logical CPU cores on the machine. This maximizes parallelism and ensures the machine's processing capacity is fully utilized during testing.

CertEmpire

Why Incorrect Options are Wrong:

A. Configuring the application for cluster mode is a different deployment strategy and does not optimize the existing Local Mode setup as requested.
C. Dynamic allocation is a feature for cluster managers (like YARN or Kubernetes) to add or remove executors; it is not applicable to Local Mode, which runs in a single JVM.
D. While memory is important, spark.executor.memory is less relevant in Local Mode where everything shares the driver's memory. The primary lever for CPU utilization is the thread count, not memory allocation.

References:

1. Apache Spark 3.5.1 Documentation, Submitting Applications, Master URLs: The documentation explicitly states for Local Mode: "localK: Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine)." and "local Run Spark locally with as many worker threads as logical cores on your machine." This directly supports configuring the number of threads based on CPU cores for optimal use.
2. Apache Spark 3.5.1 Documentation, Configuration, Dynamic Allocation: This section details the spark.dynamicAllocation.enabled property and specifies its use with Spark Standalone mode, YARN mode, and Kubernetes mode. It makes no mention of Local Mode, confirming the feature is

for cluster environments.

3. Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). Learning Spark: Lightning-Fast Big Data Analysis. O'Reilly Media, Inc. Chapter 4, "Deploying Spark": This foundational text explains that in Local Mode, "you can specify the number of threads to use by passing a number in brackets (for example, local4)." It further clarifies that using local is a common practice to use one thread per core, which is the standard approach for maximizing local resource utilization.

CertEmpire

Question: 15

A data analyst wants to add a column date derived from a timestamp column. Options:

A. datesdf.withColumn("date", f.unixtimestamp("timestamp")).show()

B. datesdf.withColumn("date", f.todate("timestamp")).show()

C. datesdf.withColumn("date", f.dateformat("timestamp", "yyyy-MM-dd")).show()

D. datesdf.withColumn("date", f.fromunixtime("timestamp")).show()

Answer:

B

Explanation:

The pyspark.sql.functions.todate() function is the correct and most direct method to convert a column of TimestampType to DateType. It extracts the date part from the timestamp, effectively truncating the time information, and returns a new column with the appropriate DateType. This directly addresses the analyst's requirement to derive a date column from a timestamp column, ensuring type correctness and semantic accuracy within the Spark DataFrame.

Why Incorrect Options are Wrong:

CertEmpire

A. unixtimestamp converts a timestamp into a long integer representing seconds since the Unix epoch, not a date.
C. dateformat converts a timestamp into a string formatted to look like a date, but the underlying data type is StringType, not the more appropriate DateType.
D. fromunixtime is used to convert a Unix timestamp (a long integer) into a string, not to convert a TimestampType column.

References:

1. Apache Spark 3.4.1 Documentation, pyspark.sql.functions.todate: "Converts the column into DateType by casting it." This confirms that todate is the designated function for converting to a DateType.
Source: https://spark.apache.org/docs/3.4.1/api/python/reference/pyspark.sql/api/pyspark.sql.functions.todate.html
2. Apache Spark 3.4.1 Documentation, pyspark.sql.functions.dateformat: "Converts a date/timestamp/string to a value of string in the format specified by the date format given by the second argument." This shows the function returns a string, not a date object.
Source: https://spark.apache.org/docs/3.4.1/api/python/reference/pyspark.sql/api/pyspark.sql.functions.dateformat.html
3. Databricks Documentation, SQL language reference, Date and timestamp functions: The documentation lists todate as the primary function for casting string, timestamp, or numeric values

to a DATE. It explicitly distinguishes this from functions like dateformat which produce strings. Source: https://docs.databricks.com/en/sql/language-manual/functions/todate.html
4. Learning Spark, 2nd Edition (O'Reilly), Chapter 4: Spark SQL and DataFrames, Section: "Working with Dates and Timestamps": This book, co-authored by Databricks employees, explains that todate() is used to convert a string or timestamp column to a date column, while dateformat() is used for string formatting. This distinction is fundamental to correct type handling in Spark. (ISBN: 978-1492050049)

CertEmpire

Question: 16

A data engineer is working on a real-time analytics pipeline using Apache Spark Structured Streaming. The engineer wants to process incoming data and ensure that triggers control when the query is executed. The system needs to process data in micro-batches with a fixed interval of 5 seconds. Which code snippet the data engineer could use to fulfil this requirement? A)

```
query = df.writeStream \
    .outputMode("append") \
    .trigger(continuous='5 seconds') \
    .start()
```

B)

```
query = df.writeStream \
    .outputMode("append") \
    .trigger() \
    .start()
```

C)

```
query = df.writeStream \
    .outputMode("append") \
    .trigger(processingTime='5 seconds') \
    .start()
```

D)

```
query = df.writeStream \
    .outputMode("append") \
    .trigger(processingTime=5000) \
    .start()
```

Options:

A. Uses trigger(continuous='5 seconds') - continuous processing mode.

B. Uses trigger() - default micro-batch trigger without interval.

C. Uses trigger(processingTime='5 seconds') - correct micro-batch trigger with interval.

D. Uses trigger(processingTime=5000) - invalid, as processingTime expects a string.

Answer:

C

Explanation:

The requirement is to process data in micro-batches with a fixed interval of 5 seconds. The trigger(processingTime='5 seconds') configuration is specifically designed for this purpose. It instructs the Structured Streaming engine to initiate a new micro-batch at the specified interval of processing time. This ensures that data is processed periodically every 5 seconds, which directly fulfills the engineer's requirement for a fixed-interval micro-batching system.

Why Incorrect Options are Wrong:

A: trigger(continuous='5 seconds') configures continuous processing mode, which aims for millisecond latency, not fixed-interval micro-batches. The interval here is for checkpointing, not for batch triggering.
B: trigger() is not valid syntax as the method requires an argument. The default behavior (if .trigger() is omitted) processes a new micro-batch as soon as the previous one finishes, not at a fixed interval.
D: trigger(processingTime=5000) is syntactically incorrect. The processingTime parameter expects a string representing the time interval (e.g., '5 seconds'), not an integer representing milliseconds.

References:

1. Apache Spark Official Documentation: The Structured Streaming Programming Guide explicitly defines the processingTime trigger.
Source: Apache Spark 3.5.1 Documentation, Structured Streaming Programming Guide, Section: Triggers.
Details: The documentation states, "ProcessingTime trigger: The query will be executed in micro-batch mode, where micro-batches will be kicked off at the user-specified intervals. If processingTime is 0, the query will be executed as fast as possible." It provides the syntax trigger(processingTime='5 seconds'). This directly supports option C and refutes D. It also describes the continuous trigger, confirming why A is incorrect for this scenario.
2. Databricks Official Documentation: The Databricks documentation on Structured Streaming triggers reinforces this concept.
Source: Databricks Documentation, "Configure Structured Streaming triggers".

Details: The section on "Processing time trigger" explains: "If you specify a processing time interval, Structured Streaming will trigger a micro-batch at that interval." It shows the example df.writeStream.trigger(processingTime='10 seconds'), which validates the syntax and functionality described in option C.

CertEmpire

Question: 17

A data scientist has identified that some records in the user profile table contain null values in any of the fields, and such records should be removed from the dataset before processing. The schema includes fields like userid, username, dateofbirth, createdts, etc. The schema of the user profile table looks like this:

```
user_id STRING,
username STRING,
full_name STRING,
date_of_birth DATE,
primary_email STRING,
created_ts TIMESTAMP,
updated_ts TIMESTAMP,
last_login_ts TIMESTAMP
```

Which block of Spark code can be used to achieve this requirement? Options:

A. filtereddf = usersrawdf.na.drop(thresh=0)

B. filtereddf = usersrawdf.na.drop(how='all')

C. filtereddf = usersrawdf.na.drop(how='any')

D. filtereddf = usersrawdf.na.drop(how='all', thresh=None)

Answer:

C

Explanation:

The requirement is to remove any record that contains a null value in any of its fields. The pyspark.sql.DataFrameNaFunctions.drop() method is used for this purpose. The how parameter controls the dropping logic. Setting how='any' instructs Spark to drop a row if it contains one or more null values across any of its columns. This directly fulfills the data scientist's requirement. The default behavior of df.na.drop() is also equivalent to how='any'.

Why Incorrect Options are Wrong:

A. filtereddf = usersrawdf.na.drop(thresh=0): The thresh parameter keeps rows with at least that many non-null values. thresh=0 would keep all rows, as every row has at least zero non-nulls.
B. filtereddf = usersrawdf.na.drop(how='all'): This option drops a row only if all of its values are null, which is too restrictive and does not meet the requirement to drop rows with any null value.
D. filtereddf = usersrawdf.na.drop(how='all', thresh=None): This is functionally identical to option B. Specifying thresh=None is redundant as it's the default and does not alter the incorrect how='all' logic.

References:

1. Apache Spark Official Documentation: The pyspark.sql.DataFrameNaFunctions.drop documentation explicitly defines the how parameter.
Source: Apache Spark 3.5.1 Documentation, pyspark.sql.DataFrameNaFunctions.drop.
Reference: "how - str, optional - 'any' or 'all'. If 'any', drop a row if it contains any nulls. If 'all', drop a row only if all its values are null."
2. Databricks Official Documentation: The Databricks guide on data cleaning confirms this behavior.
Source: Databricks Documentation, "Handle missing data".
Reference: In the section on dropping rows, it states: "You can drop rows with any null or NaN values using df.na.drop() or df.dropna(). This is the default." It further clarifies that df.na.drop("any") is the explicit form.
3. University Courseware: Reputable university courses on data science with Spark teach this fundamental operation.
Source: University of California, Berkeley, Data 100: Principles and Techniques of Data Science.
Reference: Course materials on data cleaning with PySpark demonstrate that df.dropna(how='any') is the standard method to remove rows containing any missing values. The dropna() alias is interchangeable with na.drop().

CertEmpire

Question: 18

A data engineer needs to persist a file-based data source to a specific location. However, by default, Spark writes to the warehouse directory (e.g., /user/hive/warehouse). To override this, the engineer must explicitly define the file path. Which line of code ensures the data is saved to a specific location? Options:

A. users.write(path="/some/path").saveAsTable("defaulttable")

B. users.write.saveAsTable("defaulttable").option("path", "/some/path")

C. users.write.option("path", "/some/path").saveAsTable("defaulttable")

D. users.write.saveAsTable("defaulttable", path="/some/path")

Answer:

C

Explanation:

To save a DataFrame as an unmanaged (external) table at a specific file path, the path option must be specified on the DataFrameWriter before the saveAsTable action is called. The correct syntax uses the builder pattern, chaining the .option("path", "/some/path") method to configure the writer. This instructs Spark to create the table's metadata in the metastore but store the actual data files in the user-defined external location, thereby overriding the default Spark warehouse directory. This approach creates an external table, meaning if the table is dropped, the data at the specified path is preserved.

CertEmpire

Why Incorrect Options are Wrong:

A. users.write(path="/some/path").saveAsTable("defaulttable")
The DataFrameWriter (.write) does not accept a path argument directly. The .path() method is a separate terminal action, not a configuration option for saveAsTable.
B. users.write.saveAsTable("defaulttable").option("path", "/some/path")
The .option() method must be called before the write action (saveAsTable). Calling it after the action is syntactically incorrect as the operation has already been executed.
D. users.write.saveAsTable("defaulttable", path="/some/path")
While this syntax may work in PySpark by passing path as a keyword option, the canonical and explicitly documented method is using .option("path", ...), making C the most standard and correct answer.

References:

1. Apache Spark 3.5.1 Documentation, SQL Guide, Data Sources, Generic Load/Save Functions: In the "Saving to Persistent Tables" section, the documentation explicitly demonstrates the creation of an external table using the .option("path", ...) syntax. It states, "You can also create external data source tables by using the path option... When the table is dropped, the files at the path will not be removed." The example provided is: df.write.option("path", "/some/path").saveAsTable("t").
Source: Apache Spark Official Documentation. Navigate to: SQL Guide - Data Sources - Generic Load/Save Functions - Saving to Persistent Tables.
2. Databricks Documentation, Apache Spark, DataFrames, Save tables: The documentation on saving DataFrames to tables shows that options are set on the DataFrameWriter before the save action. For external tables, the LOCATION (which corresponds to the path option) must be specified. The standard pattern shown is df.write.option("path", "").saveAsTable("").
Source: Databricks Official Documentation. Navigate to: Documentation - Apache Spark - DataFrames - Save tables.
3. Learning Spark, 2nd Edition (O'Reilly), Chapter 4: Spark SQL and DataFrames: Interacting with External Data Sources: This book, co-authored by Databricks employees, explains that to create an external table, you provide a path option to the DataFrameWriter. The examples consistently use the .option("path", "...") syntax before calling .saveAsTable().
Source: Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2020). Learning Spark, 2nd Edition. O'Reilly Media, Inc. (Chapter 4, Section on "Managed Versus Unmanaged Tables").

CertEmpire

Question: 19

Given this view definition: df.createOrReplaceTempView("usersvw") Which approach can be used to query the usersvw view after the session is terminated? Options:

A. Query the usersvw using Spark

B. Persist the usersvw data as a table

C. Recreate the usersvw and query the data using Spark

D. Save the usersvw definition and query using Spark

Answer:

B

Explanation:

A temporary view created with df.createOrReplaceTempView() is session-scoped, meaning its lifecycle is tied to the SparkSession that created it. Once the session terminates, the view and its definition are destroyed. To make the data accessible across different sessions, it must be materialized and stored persistently. Persisting the DataFrame as a table using a method like df.write.saveAsTable("userstable") writes the data to a durable storage system (e.g., DBFS) and registers the table in a metastore (like the Hive metastore). This allows new Spark sessions to query the data from the newly created permanent table.

CertEmpire

Why Incorrect Options are Wrong:

A. Query the usersvw using Spark
This is incorrect because the temporary view usersvw is dropped when the session ends and will not be found in a new session.
C. Recreate the usersvw and query the data using Spark
This describes a re-computation workflow, not a method for querying the state of the original view. It requires re-executing the code that created the initial DataFrame.
D. Save the usersvw definition and query using Spark
This is incorrect because a temporary view's definition is an in-memory construct within a specific Spark session and cannot be saved as a persistent artifact.

References:

1. Apache Spark 3.4.1 Documentation, pyspark.sql.DataFrame.createOrReplaceTempView:
Reference: "The lifetime of this temporary view is tied to the SparkSession that was used to create this DataFrame."
Link: https://spark.apache.org/docs/3.4.1/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.createOrReplaceTempView.html

2. Apache Spark 3.4.1 Documentation, SQL Guide, "Saving to Persistent Tables":
Reference: "DataFrames can also be saved as persistent tables into a Hive metastore using the saveAsTable command. Note that an existing Hive deployment is not necessary to use this feature. Spark will create a default local Hive metastore (using Derby) for you." This section explicitly details the method for making DataFrame data persistent across sessions.
Link: https://spark.apache.org/docs/3.4.1/sql-data-sources-load-save-functions.html#saving-to-persistent-tables
3. Databricks Documentation, "What are views?":
Reference: Under the "Temporary view" section: "Temporary views are session-scoped and are not registered to a schema or catalog. They cannot be referenced by other notebooks or jobs." This confirms that the view is inaccessible after the session terminates.
Link: https://docs.databricks.com/en/sql/user/views.html

CertEmpire

Question: 20

A developer wants to refactor some older Spark code to leverage built-in functions introduced in Spark 3.5.0. The existing code performs array manipulations manually. Which of the following code snippets utilizes new built-in functions in Spark 3.5.0 for array operations?

```
import pyspark.sql.functions as F

min_price = 110.50

result_df = prices_df \
 .filter(F.col("spot_price") >= F.lit(min_price)) \
 .agg(F.count("*"))
```

A)

```
result_df = prices_df \
 .withColumn("valid_price", F.when(F.col("spot_price") > F.lit(min_price), 1).otherwise(0))
```

B)

CertEmpire

```
result_df = prices_df \
 .agg(F.count_if(F.col("spot_price") >= F.lit(min_price)))
```

C)

```
result_df = prices_df \
 .agg(F.min("spot_price"), F.max("spot_price"))
```

D)

```
result_df = prices_df \
 .agg(F.count("spot_price").alias("spot_price")) \
 .filter(F.col("spot_price") > F.lit("min_price"))
```

A. resultdf = pricesdf \ .withColumn("validprice", F.when(F.col("spotprice") F.lit(minprice), 1).otherwise(0))

B. resultdf = pricesdf \ .agg(F.countif(F.col("spotprice") = F.lit(minprice)))

C. resultdf = pricesdf \ .agg(F.min("spotprice"), F.max("spotprice"))

D. resultdf = pricesdf \ .agg(F.count("spotprice").alias("spotprice")) \ .filter(F.col("spotprice") F.lit("minprice"))

Answer:

B

Explanation:

The question requires identifying a code snippet that uses a built-in function newly introduced in Apache Spark 3.5.0. The countif function is an aggregate function that was added in Spark 3.5.0. It counts the number of rows for which a specified boolean condition is true. The snippet in option B correctly applies this function to count records where the spotprice meets a certain condition, directly leveraging a new feature of Spark 3.5.0. The other options utilize functions that have been part of the Spark API for many versions prior to 3.5.0.

Why Incorrect Options are Wrong:

A. The when function is a fundamental conditional expression that has been available in Spark since version 1.4.0, not a new feature in 3.5.0.

C. The min and max aggregate functions are standard, core functions that have been part of Spark since version 1.3.0.

D. The count aggregate function and the filter transformation are basic, long-standing operations in Spark, available since version 1.3.0.

CertEmpire

References:

1. Apache Spark 3.5.0 Documentation for pyspark.sql.functions.countif: The official API documentation explicitly states that countif is a new function in this version.
Source: Apache SparkTM 3.5.1 documentation, pyspark.sql.functions.
Reference: In the documentation for the countif(col) function, it is noted: "New in version 3.5.0."
2. Apache Spark 3.5.0 Release Notes: The official release notes for Spark 3.5.0 list COUNTIF as a newly added SQL function.
Source: Apache SparkTM Release 3.5.0.
Reference: Section "New notable features", subsection "SQL Functions", which lists COUNTIF as a new addition.
3. Apache Spark 3.5.0 Documentation for pyspark.sql.functions.when: The documentation for older functions confirms their long-standing availability.
Source: Apache SparkTM 3.5.1 documentation, pyspark.sql.functions.
Reference: The documentation for when(condition, value) does not have a "New in version 3.5.0" tag and its existence can be traced back to very early Spark versions (e.g., 1.4.0).

Question: 21

What is the benefit of Adaptive Query Execution (AQE)?

A. It allows Spark to optimize the query plan before execution but does not adapt during runtime.

B. It enables the adjustment of the query plan during runtime, handling skewed data, optimizing join
strategies, and improving overall query performance.

C. It optimizes query execution by parallelizing tasks and does not adjust strategies based on runtime
metrics like data skew.

D. It automatically distributes tasks across nodes in the clusters and does not perform runtime adjustments to the query plan.

Answer:

B

Explanation:

Adaptive Query Execution (AQE) is a performance optimization feature in Apache Spark 3 that re-optimizes query execution plans during runtime. It uses statistics from completed stages of the query to make adjustments. Key benefits include dynamically coalescing shuffle partitions to handle small data partitions, switching join strategies (e.g., from sort-merge join to broadcast hash join) if one side of the join is smaller than anticipated, and optimizing joins to handle data skew. These runtime adaptations lead to significant improvements in overall query performance by addressing issues that are not apparent during the initial static planning phase.

CertEmpire

Why Incorrect Options are Wrong:

A. This describes the standard Catalyst optimizer, which creates a static plan before execution. AQE's core benefit is its ability to adapt during runtime.
C. While Spark parallelizes tasks, this option incorrectly claims AQE does not adjust for runtime metrics like data skew, which is one of its primary functions.
D. This describes the general task scheduling and distribution of Spark, not the specific runtime query plan re-optimization provided by AQE.

References:

1. Databricks Documentation, "Adaptive query execution": "Adaptive query execution (AQE) is query re-optimization that occurs during query execution... Databricks enables AQE by default. It includes three main features: Dynamically coalescing shuffle partitions, Dynamically switching join strategies, and Dynamically optimizing skew joins." This directly supports the runtime adjustments mentioned in option B.

2. Apache Spark 3.5.0 Documentation, SQL Guide, "Adaptive Query Execution": "Spark SQL can reoptimize the query plan in the middle of the execution based on the runtime statistics... As of Spark 3.2, AQE is enabled by default... AQE provides the following features: Coalescing Post Shuffle Partitions, Converting sort-merge join to broadcast join, and Skew Join Optimization." This confirms that AQE operates at runtime to handle skewed data and optimize joins.

CertEmpire

Question: 22

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: txtpath = "/datasets/rawtxt/*" df = spark.read.text(txtpath) # one row per line by default df = df.withColumn("filepath", inputfilename()) # add full path Which code change can be implemented in a DataFrame that meets the data scientist's requirements?

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, spark.read.text() reads files on a line-by-line basis, creating one DataFrame record for each line. To change this behavior and ingest the entire content of each file into a single record, the wholetext option must be set to true. This instructs the text data source reader to treat each file as a single, complete record. The correct implementation would be spark.read.option("wholetext", "true").text(txtpath). The inputfilename() function can then be used as in the original code to add the source file path to each record.

CertEmpire

Why Incorrect Options are Wrong:

B. The lineSep option specifies a custom line separator but still results in one record per separated segment, not one record for the entire file.
C. Setting wholetext=False explicitly enforces the default behavior, which is to read one record per line, failing to meet the requirement.
D. This is a specific use of the lineSep option. It would split the file content by ", ", which does not guarantee reading the whole file into one record.

References:

1. Apache Spark 3.5.1 Documentation, SQL Guide, Data Sources, Text Files: In the "Data Source Option" section, the wholetext option is described: "If true, reads each file as a single row. The default value is false." This directly supports the correct answer.
Source: Apache Spark Documentation - Text Files
2. Databricks Documentation, Data sources, Text files: The documentation lists available options for reading text files, including wholetext. It states: "When set to true, reads each file as a single

row. This is incompatible with lineSep." This confirms that wholetext is the correct option for the specified task.
Source: Databricks Documentation - Text files

CertEmpire

Question: 23

2 of 55. Which command overwrites an existing JSON file when writing a DataFrame?

A. df.write.json("path/to/file")

B. df.write.mode("append").json("path/to/file")

C. df.write.option("overwrite").json("path/to/file")

D. df.write.mode("overwrite").json("path/to/file")

Answer:

D

Explanation:

In Apache Spark, the DataFrameWriter API is used to save a DataFrame to an external storage system. The behavior for handling existing data at the destination is controlled by the .mode() method, which accepts a SaveMode string. To overwrite an existing file or directory, the mode must be explicitly set to "overwrite". This instructs Spark to delete the contents at the specified path before writing the new DataFrame's data. The default mode is "errorifexists", which would cause the operation to fail if the path already contains data.

Why Incorrect Options are Wrong:

CertEmpire

A. This command uses the default save mode (errorifexists), which will raise an exception if the file already exists, rather than overwriting it.
B. This command uses the append mode, which adds the DataFrame's data to the existing file, but does not overwrite the original content.
C. The .option() method is used for format-specific settings (e.g., compression, date formats), not for setting the fundamental save behavior like overwriting.

References:

1. Apache Spark 3.4.1 Documentation, pyspark.sql.DataFrameWriter.mode: "Specifies the behavior when data or table already exists. Options include: append: Append contents of this DataFrame to existing data. overwrite: Overwrite existing data. ignore: Silently ignore this operation if data already exists. error or errorifexists (default): Throw an exception if data already exists."
Source: Apache Spark Documentation - pyspark.sql.DataFrameWriter.mode
2. Databricks Documentation, "Save to JSON files": The documentation provides examples of writing DataFrames to various formats and explicitly shows the use of .mode('overwrite') to replace existing files. "You can use overwrite to replace a file."
Source: Databricks Documentation - Data I/O with JSON files, Section: "Save to JSON files".
3. Learning Spark, 2nd Edition (O'Reilly), Chapter 4: Spark SQL and DataFrames: Introduction to

Built-in Data Sources: This book, often used in academic settings, details the DataFrameReader and DataFrameWriter APIs. In the section on "Writing DataFrames to Files," it explains the different save modes, stating, "If you want to overwrite the output directory, you must explicitly state it using mode("overwrite")."
Source: Chapter 4, Page 91.

CertEmpire

Question: 24

3 of 55. A data engineer observes that the upstream streaming source feeds the event table frequently and sends duplicate records. Upon analyzing the current production table, the data engineer found that the time difference in the eventtimestamp column of the duplicate records is, at most, 30 minutes. To remove the duplicates, the engineer adds the code: df = df.withWatermark("eventtimestamp", "30 minutes") What is the result?

A. It removes all duplicates regardless of when they arrive.

B. It accepts watermarks in seconds and the code results in an error.

C. It removes duplicates that arrive within the 30-minute window specified by the watermark.

D. It is not able to handle deduplication in this scenario.

Answer:

C

Explanation:

In Apache Spark Structured Streaming, the withWatermark transformation is used to define the event-time threshold for late-arriving data. When used in conjunction with dropDuplicates(), it enables stateful deduplication. The watermark tells the Spark engine how long to keep the state (i.e., previously seen records) for deduplication purposes. In this case, by setting a 30-minute watermark on eventtimestamp, Spark will maintain the necessary state to identify and discard any duplicate records that arrive as long as their eventtimestamp is within the moving 30-minute window of the latest event time seen so far. Records arriving later than the watermark are considered too late and may not be deduplicated correctly.

CertEmpire

Why Incorrect Options are Wrong:

A. It removes all duplicates regardless of when they arrive.
This is incorrect. The watermark bounds the amount of state Spark maintains, so it can only guarantee deduplication for events arriving within the specified time threshold.
B. It accepts watermarks in seconds and the code results in an error.
This is incorrect. The withWatermark function accepts a string interval like "30 minutes", "1 hour", or "10 seconds". The provided syntax is valid.
D. It is not able to handle deduplication in this scenario.
This is incorrect. Using withWatermark in combination with dropDuplicates is the standard and correct pattern for handling event-time-based deduplication in Structured Streaming.

References:

1. Apache SparkTM 3.4.1 Documentation, Structured Streaming Programming Guide, "Deduplication" section:
"You can use dropDuplicates to deduplicate records in a data stream... However, for streams, the engine needs to know from when to track the duplicate records. You have to specify a watermark on a event time column. The engine will then use the watermark to know how late a duplicate record can be and accordingly clean up its internal state." This confirms that withWatermark is essential for defining the time window for deduplication.
2. Databricks Documentation, "Drop duplicates from a streaming DataFrame" section:
"Structured Streaming can deduplicate records using a unique identifier. To do this, it uses one or more fields to uniquely identify records and a field for the event time to set a watermark. The watermark allows the system to track records within a certain time range and remove duplicates." This directly links the watermark's time range to the deduplication capability.

CertEmpire

Question: 25

4 of 55. A developer is working on a Spark application that processes a large dataset using SQL queries. Despite having a large cluster, the developer notices that the job is underutilizing the available resources. Executors remain idle for most of the time, and logs reveal that the number of tasks per stage is very low. The developer suspects that this is causing suboptimal cluster performance. Which action should the developer take to improve cluster utilization?

A. Increase the value of spark.sql.shuffle.partitions

B. Reduce the value of spark.sql.shuffle.partitions

C. Enable dynamic resource allocation to scale resources as needed

D. Increase the size of the dataset to create more partitions

Answer:

A

Explanation:

The scenario describes classic symptoms of insufficient parallelism in a Spark application. A low number of tasks per stage, despite a large cluster, indicates that the work is not being divided into enough small units to keep all executor cores busy. In Spark SQL, the spark.sql.shuffle.partitions configuration parameter controls the number of partitions that are created after a shuffle operation (e.g., during joins or aggregations). By increasing this value, the developer forces Spark to create more partitions, which in turn generates more tasks for downstream stages. This increased number of tasks can then be distributed across the available executors, leading to better parallelism and improved cluster utilization.

CertEmpire

Why Incorrect Options are Wrong:

B. Reduce the value of spark.sql.shuffle.partitions: This would create even fewer partitions and tasks, worsening the underutilization of the cluster.
C. Enable dynamic resource allocation to scale resources as needed: Dynamic allocation would likely respond to the low workload by removing idle executors, shrinking the cluster rather than fixing the root cause of low parallelism.
D. Increase the size of the dataset to create more partitions: While a larger dataset might create more initial partitions, it does not control the number of partitions after a shuffle, which is the bottleneck here. This is not a practical tuning strategy.

References:

1. Databricks Documentation, "Optimize performance with caching, shuffle partitions, and Spark AQE": "The number of shuffle partitions is controlled by the spark.sql.shuffle.partitions configuration. The default is 200, which is great for smaller or medium-sized data, but not for terabytes of data. We recommend somewhere between 1 to 1.5 times the number of cores in your cluster." This directly supports increasing the value to improve performance on large clusters.
Source: Databricks Documentation Apache Spark Development Best practices Optimize performance.
2. Apache Spark 3.4.1 Documentation, "Configuration - SQL": Describes the spark.sql.shuffle.partitions parameter: "Configures the number of partitions to use when shuffling data for joins or aggregations." This confirms its role in controlling parallelism during key wide-transformation stages.
Source: Apache Spark Documentation 3.4.1 Configuration SQL, AQE, and Other Sections.
3. Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). Learning Spark: Lightning-Fast Big Data Analysis. O'Reilly Media, Inc.
Chapter 4, "Tuning and Debugging Spark," explains that an insufficient number of partitions can lead to poor resource utilization. It recommends ensuring that the number of partitions is greater than the number of cores to maximize parallelism. While this book predates Spark 3, the core principle remains fundamental and is directly applicable.

CertEmpire