Contents

[Question 1 3](#_Toc158052289)

[Question 2: 4](#_Toc158052290)

[Question 3: 5](#_Toc158052291)

[Question 4: 6](#_Toc158052292)

[Question 5: 7](#_Toc158052293)

[Question 6: 8](#_Toc158052294)

[Question 7: 9](#_Toc158052295)

[Question 8: 10](#_Toc158052296)

[Question 9: 11](#_Toc158052297)

[Question 10: 12](#_Toc158052298)

[Question 11: 13](#_Toc158052299)

[Question 12: 14](#_Toc158052300)

[Question 13: 15](#_Toc158052301)

[Question 14: 16](#_Toc158052302)

[Question 15: 17](#_Toc158052303)

[Question 16: 18](#_Toc158052304)

[Question 17: 19](#_Toc158052305)

[Question 18: 20](#_Toc158052306)

[Question 19: 21](#_Toc158052307)

[Question 21: 23](#_Toc158052308)

[Question 22: 24](#_Toc158052309)

[Question 23: 25](#_Toc158052310)

[Question 24: 26](#_Toc158052311)

[Question 25: 27](#_Toc158052312)

[Question 26: 28](#_Toc158052313)

[Question 27: 29](#_Toc158052314)

[Question 28: 30](#_Toc158052315)

[Question 29: 31](#_Toc158052316)

[Question 30: 32](#_Toc158052317)

[Question 31: 33](#_Toc158052318)

[Question 32: 34](#_Toc158052319)

[Question 33: 35](#_Toc158052320)

[Question 34: 37](#_Toc158052321)

[Question 35: 38](#_Toc158052322)

[Question 36: 39](#_Toc158052323)

[Question 37: 40](#_Toc158052324)

[Question 38: 41](#_Toc158052325)

[Question 39: 42](#_Toc158052326)

[Question 40: 43](#_Toc158052327)

[Question 41: 44](#_Toc158052328)

[Question 42: 45](#_Toc158052329)

[Question 43: 46](#_Toc158052330)

[Question 44: 47](#_Toc158052331)

[Question 45: 48](#_Toc158052332)

[Question 46: 49](#_Toc158052333)

[Question 47: 50](#_Toc158052334)

[Question 48: 51](#_Toc158052335)

[Question 49: 52](#_Toc158052336)

[Question 50: 53](#_Toc158052337)

[Question 51: 54](#_Toc158052338)

[Question 52: 55](#_Toc158052339)

[Question 53: 56](#_Toc158052340)

[Question 54: 57](#_Toc158052341)

[Question 55: 58](#_Toc158052342)

[Question 56: 59](#_Toc158052343)

[Question 57: 61](#_Toc158052344)

[Question 58: 62](#_Toc158052345)

[Question 59: 64](#_Toc158052346)

[Question 60: 66](#_Toc158052347)

## Question 1

What is the primary role of executors within the Apache Spark framework?

* Executors handle tasks from the driver, perform those tasks, and provide results to the driver.
* Executors manage jobs initiated by the driver, analyze those jobs, and return results to the cluster manager.
* Executors accept tasks from the driver, process those tasks, and return results to the driver.
* Executors take on tasks from the cluster manager, execute those tasks, and deliver results to the driver.
* Executors receive jobs from the driver, create job execution plans, and return results to the cluster manager.

**Explanation**

**The executors accept tasks from the driver, execute those tasks, and return results to the driver.**

Correct!

**The executors accept tasks from the cluster manager, execute those tasks, and return results to the driver.**

No, the job of the cluster manager is to manage computing resources in the cluster, not to distribute tasks among executors. This is the job of the driver.

**The executors accept tasks from the driver, execute those tasks, and return results to the cluster manager.**  No, results get returned to the driver, not the cluster manager.

**The executors accept jobs from the driver, analyze those jobs, and return results to the driver.**

Wrong, tasks are passed along to the executors, but not jobs. A job usually contains multiple tasks. Tasks are split among executors. Also, executors do not merely analyze the tasks they get passed from the driver, but execute those tasks.

**The executors accept jobs from the driver, plan those jobs, and return results to the cluster manager.**

Incorrect. Spark generates logical and physical plans on the driver, not on the executors. Results get returned to the driver. Executors accept tasks from the driver .

## Question 2:

In Apache Spark, how are tasks characterized within the execution hierarchy?

* Tasks represent the most granular level of work within the execution process.
* Each task is associated with slots, which define the work for data partitions.
* Tasks are considered the secondary level of granularity in the execution hierarchy.
* Single-stage tasks can be combined if they have narrow dependencies.
* Tasks encompassing broad dependencies are consolidated into a single stage.

**Explanation**

In the hierarchical structure of Spark's execution process, a **job**is divided into multiple stages, and each **stage**is further broken down into tasks.

The **task**is essentially the smallest unit of execution and is responsible for processing a specific data partition in a stage.

It's the core element where the computation takes place in Spark's distributed environment.

The term **slots** in Spark refers to the available resources on an executor to run tasks, not the actual work performed on data partitions.

Additionally, the organization of tasks is based on the stages they belong to, not vice versa. Stages are defined by their dependencies, which may be narrow or wide, but this pertains to how stages are structured, not the tasks themselves.

Therefore, it's accurate to say that tasks are the most granular level of work within Spark's execution framework.

## Question 3:

What function does the cluster manager serve in Apache Spark?

* It is responsible for task scheduling across the cluster in client mode.
* It handles task scheduling in the cluster while operating in local mode.
* Its primary role is to allocate resources to Spark applications and oversee executor processes when in client mode.
* It is tasked with resource allocation for Spark applications and managing executor processes in remote mode.
* It is in charge of assigning resources to the DataFrame manager.

**Explanation**

The cluster manager in Spark plays a vital role in **resource management within the cluster**. It distributes resources among various Spark applications, adhering to the demands of each application.

The cluster manager can operate in several modes including **standalone, Mesos, YARN, or Kubernetes**.

In client mode, the driver program is executed on the initiating machine, external to the cluster, yet the cluster manager continues to handle the**allocation of resources and the maintenance of executor processes**. Task scheduling is actually a responsibility of the Spark driver rather than the cluster manager, rendering options 1 and 2 incorrect.

The terminology 'remote mode' is not commonly used in Spark, making option 4 a less accurate description.

Furthermore, the cluster manager's functions are not limited to interacting with the DataFrame manager, which negates option 5.

Therefore, the correct answer is that the cluster manager's main task is to allocate resources and maintain executors in client mode, *although it's important to note that this function is also applicable in other operational modes of the cluster manager.*

## Question 4:

What is the primary purpose of dynamic partition pruning in Apache Spark?

* Its main function is to bypass unnecessary data during the processing of query results.
* It aims to merge columns of similar data types for enhanced join efficiency.
* The technique involves performing wide transformations on disk as opposed to in-memory operations.
* It focuses on adjusting physical plans by considering data types and the use of broadcast variables.
* Its goal is to refine query plans using runtime statistics gathered during the execution of the query.

**Explanation**

Dynamic partition pruning in Apache Spark represents an advanced optimization strategy to enhance query performance, particularly in scenarios involving the joining of large data tables.

In cases where one table in the join is partitioned, this technique can substantially decrease the volume of data read from the partitioned table.

It achieves this by dynamically identifying which partitions are relevant based on a filter applied to another table in the join operation.

The essential benefit of dynamic partition pruning lies in its ability to eliminate unnecessary partitions that don't match any rows in the joining table, thus reducing I/O load and accelerating query execution.

It is not related to merging columns or conducting disk-based transformations. While dynamic partition pruning can influence the physical execution plan, its primary intent is not reoptimization based on data types, broadcast variables, or runtime statistics.

Instead, its core function is to minimize reading irrelevant data that doesn't contribute to the final output of the query, affirming option 1 as the correct answer.

## Question 5:

What distinguishes Apache Spark's performance from Hadoop in terms of data processing?

* Spark enhances performance by utilizing DAG data storage, unlike Hadoop's reliance on parquet files.
* Spark boosts query resilience compared to Hadoop through its Kubernetes deployment capability.
* Spark's performance edge comes from its in-memory data storage and computation, in contrast to Hadoop's reliance on disk I/O operations for large jobs.
* Spark outperforms Hadoop by using HDFS data storage, whereas Hadoop is limited to parquet file usage.
* Spark provides performance improvements through an extended DataFrame API that is more developer-friendly compared to Hadoop.

**Explanation**

The fundamental performance advantage of Apache Spark over Hadoop lies in Spark's ability to process data in-memory.

Unlike Hadoop's MapReduce framework, which often resorts to disk I/O operations for handling large datasets,

Spark minimizes this bottleneck by performing most of its **computations in memory**. This difference is particularly crucial for large-scale data processing tasks, where disk I/O can significantly slow down operations.

The other options do not accurately capture this advantage:

- Option 1 is incorrect as DAG refers to Spark's execution plan, not its data storage method, and Hadoop does not exclusively use parquet files. - Option 2's emphasis on Kubernetes deployment does not directly relate to the inherent performance difference between Spark and Hadoop.

- Option 4 is misleading because both Spark and Hadoop can utilize HDFS, and it is not a distinguishing performance factor.

- Option 5 confuses API enhancements with performance benefits, which is not the primary aspect of Spark's performance superiority over Hadoop.

Thus, Spark's in-memory computing capability stands out as the key factor that gives it a performance edge over Hadoop's disk-based processing approach.

## Question 6:

**In the structure of Apache Spark's execution model, which element represents the most fundamental unit?**

* **The overarching job that initiates the process**
* **The individual task that performs the computation**
* **The executor that runs the tasks**
* **The slot that denotes resource allocation**
* **The stage that groups similar tasks**

**Explanation**

Apache Spark's execution model is hierarchically structured. At its core are the tasks, which constitute the most basic unit of computation within this structure. Here's a breakdown of the hierarchy:

- **Job**: A higher-level abstraction in Spark that consists of a series of stages, usually initiated by an action like `saveAsTextFile()`.

- **Stage**: Within a job, stages are formed, which are essentially collections of tasks that execute the same computation but on different data.

- **Task**: This is the lowest level in the hierarchy. A task represents the actual execution work done on the executor.

- **Executor**: These are processes on the cluster nodes, executing the tasks and managing data storage.

- **Slot**: This term refers to the resources (like CPU and memory) allocated to an executor, not a hierarchical level.

  Since tasks are the elements where the actual computation is executed, they represent the deepest level in Spark's execution structure, making 'The individual task that performs the computation' the correct answer.

## Question 7:

Identify the incorrect statement regarding garbage collection in Apache Spark:

* Information about garbage collection is available in the stage detail section of the Spark UI.
* Enhancing garbage collection in Spark could potentially restrict its caching capabilities.
* Explicit persistence of RDDs in Spark protects them from being subject to garbage collection.
* The G1 garbage collector serves as an alternative to the default Parallel garbage collector in Spark.
* Using serialization for caching is a technique employed to boost garbage collection efficiency.

**Explanation**

Garbage collection is a crucial aspect of memory management in Apache Spark.

Understanding the correctness of statements related to it is key:

* Statement 1 is correct as Spark UI does provide information about garbage collection in its stage detail view.
* Statement 2 is also correct; optimizing garbage collection can sometimes limit caching ability due to the trade-off between memory management and data persistence.
* Statement 3 is incorrect. Manually persisting RDDs does not prevent them from being garbage collected. Persisting RDDs helps in reusing them across multiple operations, but they can still be cleared from memory if the system requires freeing up space.
* Statement 4 is correct. In Spark, the G1 garbage collector can be used as an alternative to the default Parallel garbage collector.
* Statement 5 is correct as serialized caching is a strategy employed to enhance the performance of garbage collection, especially in environments with large datasets and limited memory resources.

Hence, the incorrect statement about garbage collection in Spark is that manually persisting RDDs prevents them from being garbage collected.

## Question 8:

What accurately describes the features of Apache Spark's Dataset API?

* The Dataset API is limited in handling structured data only.
* Python's Dataset API closely mirrors the DataFrame API found in Pandas.
* Schema definition in Python's Dataset API is enabled through type hints.
* Scala supports the Dataset API, while Python does not.
* Compile-time type safety is not a feature provided by the Dataset API.

**Explanation**

Apache Spark's Dataset API is known for its type-safe, object-oriented programming interface, combining the merits of RDDs with the DataFrame API's optimization.

It is specifically available in languages like Scala and Java, which have static type systems conducive to compile-time type safety.

However, Python, with its dynamic type system, **does not support the Dataset API**; Python programmers typically use the dynamically typed DataFrame API.

1. The Dataset API can indeed handle both structured and unstructured data, which makes option 1 incorrect.

2. Although Python's DataFrame API shares similarities with Pandas' DataFrame, this statement does not correctly describe the Dataset API, as it is not available in Python.

3. The concept of using type hints for schema construction is not applicable in Python's context for the Dataset API.

4. Compile-time type safety is a significant feature of the Dataset API,

contrary to what's suggested in option 5.

Thus, the statement that Scala, but not Python, supports the Dataset API correctly identifies one of its key characteristics.

## Question 9:

What distinguishes client mode from cluster mode in terms of execution in Apache Spark?

* Cluster mode involves running the driver on worker nodes, whereas client mode hosts the driver on the client's machine.
* In cluster mode, the driver operates from the edge node, as opposed to the worker node in client mode.
* Cluster mode allows each node to initiate its executor, while client mode confines executors to the client's machine.
* Client mode sees the cluster manager and driver on the same host, contrasting with their separation in cluster mode.
* The driver is positioned on the master node in cluster mode, while it operates on a cloud-based virtual machine in client mode.

**Explanation**

The fundamental difference between client and cluster modes in Apache Spark lies in the location of the driver program.

In cluster mode, the driver runs on one of the nodes within the cluster, typically on a worker node. This placement ensures that the driver can manage its executors more efficiently, as it is closer to the action within the cluster itself.

Conversely, in client mode, the driver program resides on the client machine.

This is often a machine outside the cluster, such as the developer's local machine or an edge node.

The client mode is advantageous for interactive and development workloads, as it keeps the driver closer to the user.

Options 2, 3, 4, and 5 present misleading or incorrect configurations of where the driver and executors are located or run in client and cluster modes.

Therefore, the accurate distinction is that in cluster mode, the driver runs on a node within the cluster, while in client mode, it runs on the client's machine.

<https://spark.apache.org/docs/latest/cluster-overview.html>

## Question 10:

Identify the **correct** statement regarding executors in Apache Spark, given that each executor in JVM can be viewed as a pool of task execution slots:

* **A slot is essentially synonymous with an executor.**
* **The number of executors should always be less than the number of tasks.**
* **An executor operates exclusively on a single core.**
* **The quantity of slots must surpass the total number of tasks.**
* **Slots facilitate the parallel execution of tasks.**

**Explanation**

In the context of Apache Spark, an executor is essentially a Java Virtual Machine (JVM) process located on a worker node. Each executor contains multiple slots, which are indicative of the cores allocated to it for task execution. These slots enable the executor to run tasks in parallel. Contrary to option 1, a slot is not the same as an executor but rather a part of it.

Regarding option 2, it's common for the number of tasks to exceed the number of executors, as tasks can queue for available slots.

Option 3 is incorrect because an executor can utilize multiple cores, allowing concurrent task execution.

Option 4 presents an overly rigid view; while more slots can enable more parallelism, it's not a strict necessity to have more slots than tasks. Thus, the statement that accurately reflects the functionality of executors in Spark is that tasks are executed in parallel through the slots within each executor.

## Question 11:

Which statement about Apache Spark's RDDs is not true?

* **An RDD is characterized by having only one partition.**
* **DataFrames are developed upon the foundational RDD API.**
* **Once created, RDDs cannot be altered; they are immutable.**
* **RDD is an acronym for Resilient Distributed Dataset.**
* **RDDs provide detailed control over query execution in Spark.**

**Explanation**

RDDs, or Resilient Distributed Datasets, are a core concept in Apache Spark, designed to be distributed across various nodes for parallel processing.

One of the central features of RDDs is their ability to be partitioned into multiple segments, enabling parallelism in data processing.

The notion of an RDD being limited to a single partition is incorrect; in fact, the power of RDDs largely comes from their distributed nature across multiple partitions.

- DataFrames in Spark are indeed higher-level abstractions built on the RDD API.

- The immutable nature of RDDs is a fundamental characteristic, ensuring that any transformations result in the creation of new RDDs.

- The term RDD correctly stands for Resilient Distributed Dataset, reflecting its distributed and resilient nature in Spark's architecture.

- RDDs offer granular control over the execution of queries, providing detailed instruction capabilities that are more in-depth compared to DataFrames and Datasets.

For more in-depth understanding of RDDs and their functioning in Apache Spark, the following resources can be useful:

- [Apache Spark Programming Guide](https://spark.apache.org/docs/latest/rdd-programming-guide.html)

- [Deep Dive into Apache Spark RDDs](https://www.datacamp.com/community/tutorials/apache-spark-tutorial-machine-learning)

## Question 12:

**Which of the following statements about broadcast variables in Apache Spark is incorrect?**

* **Broadcast variables are local to all worker nodes and not shared across the cluster.**
* **Broadcast variables are commonly used for tables that do not fit into memory.**
* **Broadcast variables are serialized with every single task.**
* **Broadcast variables are occasionally dynamically updated on a per-task basis.**

**Explanation**

Broadcast variables in Apache Spark are a mechanism to optimize the efficiency of distributed data processing.

The incorrect statement about them is that they are local to all worker nodes and not shared across the cluster. In reality, broadcast variables are shared across the cluster to ensure that large, read-only data is efficiently distributed to all nodes.

This reduces the network I/O and serialization costs associated with distributing this data for each task.

- Statement 2 is incorrect as broadcast variables are typically used for small, memory-fittable datasets, contrary to the idea of broadcasting large tables that do not fit into memory.

- Statement 3 is also incorrect because broadcast variables are cached on each machine in the cluster, which eliminates the need for serializing them with every task.

- Statement 4 is wrong because broadcast variables are immutable, meaning they cannot be dynamically updated.

<https://sparkbyexamples.com/spark/spark-broadcast-variables/>

## Question 13:

**Which statement about broadcast variables in Apache Spark is accurate?**

* **Broadcast variables are serialized alongside each task.**
* **Large tables that exceed memory limits are typically managed using broadcast variables.**
* **Broadcast variables, once created, cannot be modified and are immutable.**
* **Broadcast variables can be dynamically updated for each task.**
* **Broadcast variables are confined to worker nodes and are not distributed across the cluster.**

**Explanation**

Broadcast variables in Apache Spark are a crucial feature for optimizing the efficiency of data processing across nodes in a cluster.

They are read-only variables, meaning once a broadcast variable is created and distributed, it cannot be altered.

This immutability ensures consistency and efficiency in data processing across different nodes.

- Statement 1 is incorrect because broadcast variables are cached on executors and not serialized with every task. This caching reduces the need to transmit data repeatedly.

- Statement 2 is also incorrect as broadcast variables are ideally used for small to medium-sized datasets that fit into memory, not for excessively large tables.

- Statement 4 is wrong because broadcast variables are immutable and cannot be dynamically updated once created.

- Statement 5 is incorrect as broadcast variables are indeed shared across the entire cluster, not just local to worker nodes.

Therefore, the accurate description of broadcast variables in Spark is that they are immutable, providing a consistent, read-only data source for tasks across the cluster.

<https://sparkbyexamples.com/spark/spark-broadcast-variables/>

<https://blog.knoldus.com/spark-broadcast-variables-simplified-with-example/>

## Question 14:

What is an effective method to enhance Spark's performance when processing large data volumes with a single application on the cluster?

* **Raise the `spark.default.parallelism` and `spark.sql.shuffle.partitions` values**
* **Lower the `spark.default.parallelism` and `spark.sql.shuffle.partitions` values**
* **Increase the `spark.sql.parallelism` and `spark.sql.shuffle.partitions` values**
* **Increment the `spark.sql.parallelism` and `spark.sql.shuffle.partitions` properties**
* **Boost `spark.dynamicAllocation.maxExecutors`, `spark.default.parallelism`, and `spark.sql.shuffle.partitions` values**

**Explanation**

In scenarios where large data sets are being processed by a solitary application on a Spark cluster, optimizing parallelism is key. This can be achieved by adjusting specific configuration properties:

- `**spark.default.parallelism**` dictates the default partition count in RDDs after transformations like `**join**` and `**reduceByKey**`.

Increasing this value can enhance parallel processing efficiency.

- `**spark.sql.shuffle.partitions**` sets the number of partitions used for shuffling operations during joins or aggregations.

A higher value can lead to more efficient data distribution and processing.

Enhancing these properties fosters better parallelism and resource utilization, particularly when the cluster's capacity is dedicated to a single application.

* Options 2 and 4 are less effective as they may reduce parallelism.
* Option 3 is incorrect as `spark.sql.parallelism` is not a standard configuration property in Spark.
* Option 5, while considering dynamic allocation, does not focus exclusively on parallelism adjustments.

Therefore, the most effective method, given the options, is to increase `**spark.default.parallelism**` and `**spark.sql.shuffle.partitions**` for improved performance in handling large datasets.

## Question 15:

Which statement accurately describes the features of Spark's Adaptive Query Execution?

* AQE includes dynamic coalescing of shuffle partitions, scan filter injection, and skew join optimization.
* AQE is a feature that's automatically enabled in all Spark configurations.
* AQE involves reoptimizing queries at different stages during their execution.
* AQE dynamically switches join strategies and optimizes skew joins.
* AQE is applicable to every type of query in Spark.

**Explanation**

Adaptive Query Execution (AQE) in Apache Spark is a powerful optimization framework that enhances query performance by adapting to data characteristics at runtime. Key features of AQE include:

- Dynamically coalescing shuffle partitions: Adjusting the number of shuffle partitions based on the size of shuffled data, leading to less resource consumption and improved performance.

- Dynamically injecting scan filters: Applying beneficial filters during query execution to minimize the amount of data processed.

- Dynamically optimizing skew joins: Addressing data skew in join operations to balance the workload and prevent bottlenecks.

Option 1 correctly encapsulates these features of AQE.

Option 2 is incorrect as AQE is not enabled by default in all Spark versions and requires specific configuration.

Option 3 is partially correct but doesn't fully capture the essence of AQE's capabilities.

Option 4 mentions key aspects but misses other vital features like coalescing shuffle partitions and scan filter injection.

Option 5 is incorrect because AQE does not apply to all query types; it's specifically designed for certain operations like joins and aggregations.

For a comprehensive understanding of AQE in Spark, it's recommended to consult the official Apache Spark documentation and resources on query optimization techniques.

[Adaptive Query Execution: Speeding Up Spark SQL at Runtime](https://www.databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html)

[Spark 3.0 – Adaptive Query Execution with Example](https://sparkbyexamples.com/spark/spark-adaptive-query-execution/)

Bottom of Form

## Question 16:

Identify the error in the following Spark code block which intends to join DataFrame `itemsDf` with a larger DataFrame `transactionsDf` on column `itemId`:    transactionsDf.join(itemsDf, "itemId", how="broadcast")

* **The syntax is incorrect; `how=` should be omitted from the code.**
* **`join` method should be replaced with the `broadcast` method.**
* **Broadcast operation will only be performed if it's enabled on the Spark cluster.**
* **The larger DataFrame `transactionsDf` is being broadcasted, not the smaller `itemsDf`.**
* **`broadcast` is not a recognized join type in Spark.**

**Explanation**

The error in the provided code block is that `broadcast` is not a valid join type in Spark.

The join method in Spark accepts join types like `inner`, `outer`, `full\_outer`, and similar, but not `broadcast`.

To perform a broadcast join, the `broadcast` function should be used from the `**pyspark.sql.functions**` module to suggest broadcasting a DataFrame.

For instance:

1. from pyspark.sql.functions import broadcast
3. transactionsDf.join(broadcast(itemsDf), "itemId")

This use of `broadcast` hints to Spark that `itemsDf` is small enough to be broadcasted, but the final decision is made by Spark based on factors like the `spark.sql.autoBroadcastJoinThreshold` setting.

The original code block's syntax is incorrect as it incorrectly specifies `broadcast` as a join type.

## Question 17:

How can you adjust the partition count of the DataFrame `salesData` from 15 partitions to 30 partitions in Apache Spark?

* `salesData.adjustPartitions(30)` for direct increase in partition count
* `salesData.setPartitions(30)` as a method to change partitioning
* Use `salesData.alterPartitioning('productId', 30)` for adjusting partitions
* Implement `salesData.modifyPartitioning(30)` to increase partitioning
* Employ `salesData.repartition(30)` for explicit modification of partition count

**Explanation**

In Apache Spark, effectively managing the number of partitions in a DataFrame is crucial for optimizing data processing efficiency.

The `**repartition**` method is designed for redistributing data across a specified number of partitions, allowing for both an increase or decrease in the partition count.

- Options 1, 2, 3, and 4 are not actual methods in Spark's DataFrame API.

They serve as hypothetical choices to provide varied options but do not reflect actual Spark functionalities.

- Option 5, `salesData.repartition(30)`, is the correct method. It instructs Spark to redistribute the DataFrame `salesData` across 30 partitions.

This adjustment helps in optimizing the data processing by altering the DataFrames partitioning structure to the specified number of partitions.

## Question 18:

How would you eliminate rows in the DataFrame `salesData`, which has 6 columns, if they contain missing data in at least 3 columns?

* `salesData.dropna('all')` to remove rows with all missing values
* `salesData.dropna(thresh=4)` to filter out rows with insufficient non-null values
* `salesData.removeMissing(3)` as a custom method to drop rows with missing data
* `salesData.dropna(thresh=3)` for rows with less than 3 non-null values
* `salesData.omitna('min', 3)` as a hypothetical method for data omission

**Explanation**

In Apache Spark, the `**dropna**` method is utilized for removing rows with missing data in a DataFrame.

The key parameter here is `thresh`, which specifies the minimum number of non-null values needed for a row to be retained.

- For a DataFrame with 6 columns, and the requirement to drop rows with missing data in at least 3 columns, we need to set `thresh=4`.

This ensures that rows with 3 or more missing values are dropped, keeping those with at least 4 non-missing values.

- Option 1, using 'all', would remove only rows where all values are missing, which does not meet the requirement.

- Options 3 and 5 present fictional methods and are not part of Spark's DataFrame API.

- Option 4, with `thresh=3`, is not strict enough as it would retain rows with up to 3 missing values.

Therefore, the correct approach is `salesData.dropna(thresh=4)`, effectively filtering out rows with at least 3 missing values in the 6-column DataFrame `salesData`.

## Question 19:

How would you eliminate rows in the DataFrame `salesData`, which has 6 columns, if they contain missing data in at least 3 columns?

* **`salesData.dropna('all')` to remove rows with all missing values**
* **`salesData.dropna(thresh=4)` to filter out rows with insufficient non-null values**
* **`salesData.removeMissing(3)` as a custom method to drop rows with missing data**
* **`salesData.dropna(thresh=3)` for rows with less than 3 non-null values**
* **`salesData.omitna('min', 3)` as a hypothetical method for data omission**

**Explanation**

In Apache Spark, the `dropna` method is utilized for removing rows with missing data in a DataFrame.

The key parameter here is `thresh`, which specifies the minimum number of non-null values needed for a row to be retained.

- For a DataFrame with 6 columns, and the requirement to drop rows with missing data in at least 3 columns, we need to set `thresh=4`. This ensures that rows with 3 or more missing values are dropped, keeping those with at least 4 non-missing values.

- Option 1, using 'all', would remove only rows where all values are missing, which does not meet the requirement.

- Options 3 and 5 present fictional methods and are not part of Spark's DataFrame API.

- Option 4, with `thresh=3`, is not strict enough as it would retain rows with up to 3 missing values.

Therefore, the correct approach is `salesData.dropna(thresh=4)`, effectively filtering out rows with at least 3 missing values in the 6-column DataFrame `salesData`.

Bottom of Form

Top of Form

Question 20:

Which code snippet correctly saves the DataFrame `salesData` to memory, allowing for recalculation of partitions that don't fit in memory when needed?

* from pyspark import StorageLevel

salesData.cache(StorageLevel.MEMORY\_ONLY)

* **salesData.cache() as a simple caching method**
* **salesData.storage\_level('MEMORY\_ONLY') for specifying storage level**
* **salesData.persist() to persist with the default storage level**
* **salesData.clear\_persist() to clear any persistence settings**
* from pyspark import StorageLevel

salesData.persist(StorageLevel.MEMORY\_ONLY)

**Explanation**

In Apache Spark, persisting a DataFrame in memory is a common operation to optimize data processing. The `persist` method, when used with a specified storage level, controls how the DataFrame is stored.

- Option 1 is incorrect because `**cache**` method does not take parameters and `cache(StorageLevel.MEMORY\_ONLY)` is not a valid method.

- Option 2, `**salesData.cache()**`, is a valid method but it uses the default storage level, which includes both memory and disk storage.

- Option 3 uses an incorrect method syntax; `**storage\_level**` is not a method to set the storage level in Spark.

- Option 4 is valid but `**persist()**` without parameters does not specifically use `**MEMORY\_ONLY**` storage.

- Option 5, `**clear\_persist**`, is not a valid method in Spark for handling persistence.

- Option 6 is the correct approach. The `**persist(StorageLevel.MEMORY\_ONLY)**` method explicitly sets the DataFrame to be stored in memory only. If a partition does not fit in memory, it will be recomputed as needed, which aligns with the requirement.

Thus, the correct code block for saving `salesData` to memory only, allowing for recalculation of partitions when necessary, is `**salesData.persist(StorageLevel.MEMORY\_ONLY)**`.

## Question 21:

Identify the error in the following Spark code block intended to create DataFrame **productAttributesDF**from **productsDF**, listing each **productAttribute**alongside the productId from the productsDF DataFrame:

1. productAttributesDF = productsDF.explode("productAttributes").alias("productAttribute").select("productAttribute", "productId")

Given a sample of DataFrame **productsDF**:

+----------+--------------------------+--------------+

| productId|productAttributes |supplier |

+----------+--------------------------+--------------+

| P1 |[blue, casual, comfortable]|CoolWear Inc. |

| P2 |[green, formal, trendy] |FashionHub |

| P3 |[black, evening, elegant] |GlamourSuits |

+----------+--------------------------+--------------+

* **Since productId is the index, it does not need to be included in the `select()` method.**
* **The alias() method must be invoked after the select() method.**
* **The explode() function requires a Column object, not a string.**
* **The explode() function is not directly callable on DataFrame; it should be used within the select() method.**
* **Use the split() function instead of explode() within the select() method.**

**Explanation**

The error in the provided Spark code block lies in the incorrect usage of the `explode()` function.

In Apache Spark, the `explode()` function is not a method that can be directly called on a DataFrame object.

Instead, it should be used inside a `select` or `withColumn` method call. The correct way to transform `productsDF` and create `productAttributesDF` is to **use `explode()` within a `select()` method**, like so:

1. productAttributesDF = productsDF.selectExpr("productId", "explode(productAttributes) as
2. productAttribute")

- Option 4 is correct because it identifies the misuse of `explode()` as a direct method on a DataFrame, which is not allowed in Spark's DataFrame API.

- The other options are incorrect because they either misunderstand the role of `productId`, suggest incorrect method sequencing, or recommend using a function (`split`) that serves a different purpose than `explode`.

## Question 22:

Select the code block that **correctly**reads the parquet file `/DataFiles/exports.parquet` into a DataFrame in Apache Spark:

* **spark.mode('parquet').read('/DataFiles/exports.parquet')**
* **spark.read.path('/DataFiles/exports.parquet', format='parquet')**
* **spark.read().parquet('/DataFiles/exports.parquet')**
* **spark.read.parquet('/DataFiles/exports.parquet')**
* **spark.read().format('parquet').open('/DataFiles/exports.parquet')**

**Explanation**

In PySpark, the standard method for reading a parquet file into a DataFrame involves using the `**spark.read.parquet()**` function, where `spark` is typically a `SparkSession` object.

The correct syntax is spark.read.parquet('/path/to/parquet/file').

- Option 1 is incorrect because `mode('parquet')` is **not** a valid approach for reading data in Spark.

- Option 2 incorrectly uses a **non-existent** `path` method combined with a `format` argument, which does not conform to the PySpark API.

- Option 3 is syntactically incorrect due to the unnecessary parentheses.

- Option 5 uses `**open**` instead of the correct `**load**` method, and the usage of `**format('parquet')**` is not needed when the `**parquet**` method is directly available.

Thus, the correct and efficient way to read a parquet file into a DataFrame in Apache Spark is through `spark.read.parquet('/DataFiles/exports.parquet')`, as stated in option 4.

## Question 23:

Complete the following Python code block to convert up to 7 rows in DataFrame `purchaseData` that have the value 30 in column `storeID` to a Python list:

purchaseData.\_\_1\_\_(\_\_2\_\_).\_\_3\_\_(\_\_4\_\_)

* filter,

'storeID' == 30,

collect,

7

* filter,

col('storeID') == 30,

toLocalIterator,

7

* select,

storeID == 30,

head,

7

* filter,

col('storeID') == 30,

take,

7

* filter,

col('storeID') == 30,

collect,

7

**Explanation**

The task is to filter rows in the DataFrame `purchaseData` based on a condition and then convert a specific number of these rows to a Python list.

The appropriate methods to achieve this are `**filter**` for row filtering and `**take**` for retrieving a fixed number of rows.

- `**filter(col('storeID') == 30)**` is used to apply the condition to select rows where `storeID` equals 30.

- `**take(7)**` is then used to fetch the first 7rows that meet this condition and return them as a Python list.

Option 4, `filter(col('storeID') == 30).take(7)`, accurately constructs this operation.

The other options either use incorrect methods or specify the condition in a non-standard format for PySpark.

*Therefore, the correct code to achieve the desired outcome is:*

1. from pyspark.sql.functions import col
2. purchaseData.filter(col('storeID') == 30).take(7)

Bottom of Form

Top of Form

## Question 24:

Which code snippet ***correctly***reads the JSON file `data.json` into a DataFrame in Apache Spark?

* **`spark.read().mode('json').path('/DataFiles/data.json')`**
* **`spark.read.format('json').path('/DataFiles/data.json')`**
* **`spark.read('json', '/DataFiles/data.json')`**
* **`spark.read.json('/DataFiles/data.json')`**
* **`spark.read().json('/DataFiles/data.json')`**

**Explanation**

In PySpark, the **read.json** method is used for reading JSON files into a DataFrame.

This method is a part of the **SparkSession** object (usually instantiated as `spark`), and it directly takes the file path as an argument.

The correct and standard syntax for reading a JSON file is spark.read.json('/path/to/json/file').

- Options 1 and 2 are incorrect due to their syntax;

`mode()` is not used for specifying the data format, and while **format('json')** is a valid approach, it should be coupled with **load()**, not `path()`.

- Option 3 incorrectly formats the `read` method's arguments.

- Option 5 introduces unnecessary parentheses, which are not typical in PySpark syntax for this operation.

Thus, the accurate and efficient way to read a JSON file into a DataFrame in Apache Spark is through spark.read.json('/DataFiles/data.json'), as indicated in option 4.

## Question 25:

Select the code block that adds a new column **calcError**to DataFrame **salesData**, where **calcError** is the squared value of an existing column **errorValue** in **salesData**

* **`salesData.withColumn('errorValue', pow(col('calcError'), 2))`**
* **`salesData.withColumnRenamed('calcError', pow(errorValue, 2))`**
* **`salesData.withColumn('calcError', pow(col('errorValue'), lit(2)))`**
* **`salesData.withColumn('calcError', pow(errorValue, lit(2)))`**
* **`salesData.withColumn('calcError', 'errorValue' \*\* 2)`**

**Explanation**

In PySpark, the **withColumn**method is commonly used to add a new column or modify an existing one in a DataFrame.

The **pow** function, along with **col**and **lit**, are essential for performing **mathematical operations** on DataFrame columns.

- Option 3 is correct as it effectively uses `withColumn` to add a new column `calcError`. It employs the `pow` function to square the values in the `errorValue` column, using `col('errorValue')` to reference the column and `lit(2)` to specify the exponent.

- Options 1, 2, 4, and 5 are incorrect due to syntax errors or misuse of functions.

Option 1 incorrectly tries to redefine `errorValue`.

Option 2 uses `withColumnRenamed`, which is not suitable for creating new columns with calculated values. Option 4 fails to correctly reference the `errorValue` column,

and option 5 uses incorrect syntax for column operations in PySpark.

Thus, salesData.withColumn('calcError', pow(col('errorValue'), lit(2))) is the appropriate way to add a new column by squaring an existing column in a DataFrame.

## Question 26:

Identify the error in the following Python code block that aims to filter rows in DataFrame purchaseData where values in column errorMargin are at least 5:

purchaseData.where("col(errorMargin) >= 5")

* **The correct syntax for the where method is 'errorMargin >= 5'**
* **Use `filter()` method instead of `where()`.**
* **To get a new DataFrame, modify the code to `purchaseData.toNewDataFrame().where('col(errorMargin) >= 5')`.**
* **String expressions are not allowed as arguments in the `where` method.**
* **Replace `>=` with the SQL operator `GEQ`.**

**Explanation**

The error in the code block lies in the incorrect usage of the string expression within the **where**method.

In PySpark, the **where**method is used for filtering DataFrame rows based on a condition. The condition should be provided **as a string expression**that correctly references the column name.

- Option 1 is correct. The proper syntax to specify the condition is 'errorMargin >= 5' , directly referencing the column name without using `**col()**` in a string expression.

- Option 2 is technically not an error since `filter()` and `where()` are interchangeable in PySpark.

- Option 3 is incorrect; creating a new DataFrame before applying the `where` method is unnecessary. The `where` method itself returns a new DataFrame.

- Option 4 is incorrect as string expressions are valid in the `where` method.

- Option 5 is also incorrect; `>=` is the correct SQL operator to use, not `GEQ`.

Therefore, the corrected code should be:  purchaseData.where('errorMargin >= 5')

For more detailed examples and explanations of the **where** method in PySpark, you can refer to these resources:

* [Spark By Examples - PySpark Where Filter Function](https://sparkbyexamples.com/pyspark/pyspark-where-filter/)
* [SkyTowner - PySpark DataFrame | where method with Examples](https://www.skytowner.com/explore/pyspark_dataframe_where_method)

## Question 27:

Choose the code block that correctly saves DataFrame **salesData**to the location `/DataFiles/sales.csv` as a CSV file and throws an error**if a file already exists** at that location:

* **`salesData.write.save('/DataFiles/sales.csv')`**
* **`salesData.write.format('csv').mode('error').path('/DataFiles/sales.csv')`**
* **`salesData.write.format('csv').mode('ignore').path('/DataFiles/sales.csv')`**
* **`salesData.write('csv').mode('error').save('/DataFiles/sales.csv')`**
* **`salesData.write.format('csv').mode('error').save('/DataFiles/sales.csv')`**

**Explanation**

In Apache Spark, when saving a DataFrame to a file, the file format and behavior on file existence can be specified using the DataFrameWriter methods.

The correct approach to save a DataFrame as a CSV and throw an error if the file already exists is to use the write.format('csv').mode('error').save() method chain.

- Option 1 does not specify the CSV format and lacks the explicit error mode.

- Option 2 incorrectly uses `.path()`; the correct method for specifying the save path is `.save()`.

- Option 3 uses 'ignore' mode, which won't throw an error if the file exists but will ignore the write operation.

- Option 4 incorrectly uses the `write` method with a direct string argument for the format.

- Option 5, `salesData.write.format('csv').mode('error').save('/DataFiles/sales.csv')`, correctly sets the CSV format, specifies the error mode for file existence, and provides the correct path for saving the file.

This ensures that the DataFrame is saved as a CSV file and an error is thrown if the file already exists at the given path.

## Question 28:

Choose the correct sequence of operations to transform DataFrame **productDetailsDf**so that it results in a DataFrame with two columns, **productId** and **col**. The new DataFrame should list each element of the **features**column in **productDetailsDf** as a separate row and include the associated **productId**for each. The transformation should only include rows where the **features**column contains the element **comfortable**:

productDetailsDf.\_\_1\_\_(\_\_2\_\_).\_\_3\_\_(\_\_4\_\_).\_\_5\_\_(\_\_6\_\_)

Fill in the blanks with the appropriate options:

* **`filter`, `array\_contains("comfortable")`, `select`, `"productId"`, `explode`, `"features"`**
* **`where`, `"array\_contains(features, 'comfortable')"`, `select`, `productId`, `explode`, `features`**
* **`filter`, `"array\_contains(features, 'comfortable')"`, `select`, `"productId"`, `map`, `"features"`**
* **`filter`, `"array\_contains(features, comfortable)"`, `select`, `"productId"`, `explode`, `"features"`**
* **`filter`, `"array\_contains(features, 'comfortable')"`, `select`, `"productId"`, `explode`, `"features"`**

**Explanation**

The objective is to create a new DataFrame from `productDetailsDf` with two columns: `productId` and an exploded list of elements from the `features` column, specifically including only rows where 'comfortable' is an element in the `features` array.

- Option 5  is correct.

productDetailsDf.filter("array\_contains(features, 'comfortable')").select("productId", explode("features")).alias("col")

It uses the `filter` method with the `array\_contains` function to select rows where 'comfortable' is present in the `features` column. Then, it uses `select` to choose `productId` and employs `explode` on the `features` column to create individual rows for each element.

- Options 1, 2, 3 ans 4 are incorrect due to various syntax errors or misuse of functions. Particularly, A lacks the correct string expression,

2 uses `where` which is synonymous with `filter` but not required,

3 incorrectly introduces `map`,

and 4 misses the quotes around 'comfortable'.

## Question 29:

Determine the error in the following Python code block that aims to return the average of rows in column **salesValue**grouped by unique **shopId**in DataFrame **salesData**:   salesData.agg('shopId').avg('salesValue')

* **Use `avg('salesValue')` as `avg(col('salesValue'))` instead.**
* **The `avg('salesValue')` should be a parameter inside `agg()`.**
* **Wrap all column names with `col()` operators.**
* **Replace `agg` with `groupBy`.**
* **Swap 'shopId' and 'salesValue'.**

**Explanation**

The correct approach to calculate the average of a column grouped by another column in PySpark involves using the `groupBy` method, followed by an aggregate function like `avg`. In the provided code block, the use of `agg` alone is incorrect as it should be preceded by `groupBy` for such operations.

The corrected code should be:

1. from pyspark.sql.functions import avg
2. salesData.groupBy('shopId').avg('salesValue')

This code correctly groups the data by `shopId` and then computes the average of `salesValue` for each group.

Using the `**col()**` function is optional here, as `**avg()**` can accept a string literal for the column name.

The other options do not address the primary issue of incorrectly using `**agg**` instead of `**groupBy**`.

Therefore, the error in the code block is the use of `**agg**` where `**groupBy**` should be used, as pointed out in option 4.

## Question 30:

Select the code block that correctly renames the column **vendor**to **producer** in the DataFrame **productDetailsDf**:

* **`productDetailsDf.withColumn('vendor', 'producer')`**
* **`productDetailsDf.withColumn('vendor').alias('producer')`**
* **`productDetailsDf.withColumnRenamed('vendor', 'producer')`**
* **`productDetailsDf.withColumnRenamed(col('producer'), col('vendor'))`**
* **`productDetailsDf.withColumnsRenamed('vendor', 'producer')`**

**Explanation**

In PySpark, the method `withColumnRenamed` is specifically designed for renaming an existing column in a DataFrame.

This method requires two string arguments: the first is the name of the existing column to be renamed, and the second is the new name for the column.

- Option 3 is correct as it uses `withColumnRenamed('vendor', 'producer')`, effectively changing the column name from `vendor` to `producer`.

- Option 1 incorrectly uses `withColumn`, which is intended for adding a new column or replacing an existing column with a new value or expression, not for renaming.

- Option 2 incorrectly chains `withColumn` with `alias`, which is not the appropriate method for renaming a column.

- Option 4 misuses `col` within `withColumnRenamed`, which expects simple string arguments, not `col` expressions.

- Option 5 mentions `withColumnsRenamed`, which is not a valid method in the PySpark DataFrame API.

Therefore, the correct approach for renaming the column `vendor` to `producer` in `productDetailsDf` is productDetailsDf.withColumnRenamed('vendor', 'producer').

## Question 31:

Select the code block that correctly sorts DataFrame **salesData**in descending order by column **errorMargin**, with missing values appearing at the end:

* **`salesData.sort(asc\_nulls\_last('errorMargin'))`**
* **`salesData.orderBy('errorMargin').desc\_nulls\_last()`**
* **`salesData.sort('errorMargin', ascending=False)`**
* **`salesData.desc\_nulls\_last('errorMargin')`**
* **`salesData.orderBy('errorMargin').asc\_nulls\_last()`**

**Explanation**

In Apache Spark, sorting a DataFrame based on a specific column can be achieved using the `sort` or `orderBy` method.

However, placing missing values at the end of the sorted DataFrame requires the correct combination of sorting direction and null handling.

- Option 3 is correct as it uses

sort('errorMargin', ascending=False) to sort `salesData` in descending order by `errorMargin`.

While this method sorts the DataFrame in descending order, it does not explicitly specify the handling of null values.

In Spark, by default, null values are sorted last when sorting in descending order.

- Options 1 and 5 incorrectly use `**asc\_nulls\_last**`, which sorts in ascending order, not descending.

- Option 2's `**desc\_nulls\_last**()` is not a valid method in PySpark.

- Option 4's `**desc\_nulls\_last**` is not a method that can be directly applied to a DataFrame.

Therefore, the correct approach to sort `salesData` in descending order by `errorMargin` while placing missing values last is : salesData.sort('errorMargin', ascending=False)

## Question 32:

Determine the ***error***in the following Python code block that aims to perform an **outer join** between DataFrames `purchaseDataDf` and `productDataDf` on columns `productID` in `purchaseDataDf` and `itemID` in `productDataDf`:

purchaseDataDf.join(productDataDf, [productDataDf.itemID, purchaseDataDf.productID], 'outer')

* **Remove the 'outer' argument as it is the default join type.**
* **The join type should be appended like `join().outer()` instead of as the last argument.**
* **Replace `[productDataDf.itemID, purchaseDataDf.productID]` with `productDataDf.itemID == purchaseDataDf.productID`.**
* **Use `productDataDf.col('itemID') == purchaseDataDf.col('productID')` instead of `[productDataDf.itemID, purchaseDataDf.productID]`.**
* **Eliminate 'outer' from the call and replace `join` with `joinOuter`.**

**Explanation**

The error in the provided code block lies in the way the join condition is specified.

In PySpark, when performing a join between two DataFrames, the join condition should be specified as an **equality comparison** between the columns on which the join is based, **not as a list of columns**.

- Option 3 correctly identifies the issue.

The join condition should be expressed as `productDataDf.itemID == purchaseDataDf.productID`, which correctly specifies the columns from each DataFrame to be used for the join.

- The other options either suggest incorrect syntax changes or misunderstand the default behavior of the `join` method in PySpark.

The '**outer**' argument is correctly placed, and the syntax of the join method call in the code block is valid, except for the join condition expression.

Therefore, the corrected code should be:

purchaseDataDf.join(productDataDf, productDataDf.itemID == purchaseDataDf.productID, 'outer')

## Question 33:

Select the code block that performs a join where the small DataFrame `ordersDf` is **broadcasted**to all executors and then joined with DataFrame `productsDf` on columns `shopId` in `ordersDf` and `productId` in `productsDf`:

* **`productsDf.join(ordersDf, productsDf.productId == ordersDf.shopId, 'right\_outer')`**
* **`productsDf.join(ordersDf, productsDf.productId == ordersDf.shopId, 'broadcast')`**
* **`productsDf.merge(ordersDf, 'productsDf.productId == ordersDf.shopId', 'broadcast')`**
* **`productsDf.join(broadcast(ordersDf), productsDf.productId == ordersDf.shopId)`**
* **`productsDf.join(ordersDf, broadcast(productsDf.productId == ordersDf.shopId))`**

**Explanation**

In PySpark, the `broadcast` function is used to hint that a DataFrame should be broadcasted to all executors for a join operation. This is especially useful when joining a small DataFrame with a larger one, as it can significantly reduce the amount of data that needs to be shuffled across the cluster.

- Option 4 is correct.

It applies the `broadcast` function from `pyspark.sql.functions` to the smaller DataFrame `ordersDf`, and then performs the join on the specified columns.

- The other options have various errors, such as incorrect join types, misuse of the `broadcast` function, or using methods not available in PySpark's DataFrame API (`merge`).

Thus, the correct approach for broadcasting `ordersDf` and joining it with `productsDf` is

productsDf.join(broadcast(ordersDf), productsDf.productId == ordersDf.shopId)

**How the broadcast Function Works:**

* **Data Broadcasting**: When you broadcast a DataFrame, Spark sends it to each executor only once, and the executors hold a copy of this DataFrame in memory. This eliminates the need for shuffling this DataFrame across the executors during the join operation.
* **Efficiency in Join Operations**: Broadcasting is particularly effective when joining a small DataFrame with a larger one. By broadcasting the smaller DataFrame, Spark can perform the join locally on each executor with the larger DataFrame, significantly reducing network traffic and improving the overall efficiency of the join operation.

**Additional References:**

[PySpark Broadcast Variables](https://sparkbyexamples.com/pyspark/pyspark-broadcast-variables/)

[Spark Broadcast Joins: What They Are and When to Use Them](https://www.sparkcodehub.com/spark-broadcast-joins)

## Question 34:

Select the code block that decreases the number of partitions in DataFrame `dataFrame` from 12 to 6 and involves a complete data shuffle:

* **`dataFrame.repartition(12)`**
* **`dataFrame.coalesce(6).shuffle()`**
* **`dataFrame.coalesce(6)`**
* **`dataFrame.coalesce(6, shuffle=True)`**
* **`dataFrame.repartition(6)`**

**Explanation**

In Apache Spark, the `**repartition**` method is utilized for changing the number of partitions in a DataFrame and involves a full shuffle of the data.

This approach is particularly useful when a complete redistribution of the data is needed, often to optimize the layout of the data for subsequent operations that benefit from specific partitioning.

- Option 5, `dataFrame.repartition(6)`, **correctly** states the method for reducing the number of partitions to 6 while performing a full shuffle.

- Option 1 mistakenly suggests repartitioning to 12 partitions instead of reducing to 6.

- Option 2's `coalesce(6).shuffle()` is incorrect as `shuffle` is not a method that follows `coalesce` in PySpark, and `coalesce` itself reduces partitions without a full shuffle.

- Option 3, while reducing the number of partitions, does not involve a full shuffle.

- Option 4 incorrectly includes a non-existent `shuffle=True` parameter for `coalesce`.

Thus, to decrease the partition count to 6 with a full shuffle, the appropriate code is `dataFrame.repartition(6)`.

## Question 35:

Identify the error in the following Python code block intended to write DataFrame `salesDataDf` to disk as a parquet file at the location `/DataStore/sales\_data\_split`,

partitioning the data using the column `shopId`:  salesDataDf.write.format('parquet').partitionOn('shopId').save('/DataStore/sales\_data\_split')

* **Using `format('parquet')` is incorrect, it should be the first argument in `save()` along with the path.**
* **Replace `partitionOn` with `partitionBy` for correct DataFrame partitioning.**
* **Substitute `partitionOn` with `bucketBy` for proper partitioning by `shopId`.**
* **`partitionOn('shopId')` should be placed before the `write` method.**
* **Remove `format('parquet')` and use `write('parquet')` instead.**

**Explanation**

The error in the provided code block is the usage of `partitionOn` instead of `**partitionBy**`.

In PySpark, the `partitionBy` method is used for partitioning data when writing a DataFrame to disk. It allows data to be partitioned by one or more columns, which can be beneficial for storage and querying efficiency. The corrected code block should be: salesDataDf.write.format('parquet').partitionBy('shopId').save('/DataStore/sales\_data\_split')

This code correctly writes `salesDataDf` as a parquet file, partitioned by the `shopId` column.

The use of `bucketBy` is not appropriate here as it is meant for a different kind of partitioning, typically used in table joins.

The `format` method is correctly used to specify the output data format, and there is no `partitionOn` method in PySpark.

The `save` method's arguments are also correctly utilized.

## Question 36:

Identify the error in the following Python code block that aims to **return all columns** of DataFrame `salesDataDf` except for columns `errorMargin`, `productCode`, and `saleValue`:

salesDataDf.select(col('errorMargin'), col('productCode'), col('saleValue'))

* **Replace `select` with `drop` and use `col` operator like `drop(col('errorMargin'), col('productCode'), col('saleValue'))`.**
* **Substitute `select` with `deselect`.**
* **Column names in `select` should not be strings; use `select(col(errorMargin), col(productCode), col(saleValue))`.**
* **Replace `select` with `drop`.**
* **Change `select` to `drop` and pass column names as strings like `drop('errorMargin', 'productCode', 'saleValue')`.**

**Explanation**

The error in the provided code block is the misuse of the `select` method to exclude columns from a DataFrame. In PySpark, `select` is used to include specific columns, not to exclude them.

- The correct approach is to use the `drop` method, which is explicitly designed for **removing specified columns from a DataFrame**. The `drop` method can take column names as string arguments directly.

- Option 4, salesDataDf.drop('errorMargin', 'productCode', 'saleValue'), correctly identifies this, specifying the column names as strings.

- The other options either suggest incorrect method replacements (`deselect`) or misuse of the `select` method. Using `col` operator with `drop` is unnecessary as column names can be passed as strings directly.  Therefore, the corrected code should be: salesDataDf.drop('errorMargin', 'productCode', 'saleValue')

## Question 37:

Find the error in the following Python code block that intends to rename the column `shopId` to `shopNumber` in DataFrame `purchaseDataDf`:  purchaseDataDf.withColumn('shopNumber', 'shopId')

* **Replace `withColumn` with `withColumnRenamed`.**
* **Wrap 'shopNumber' and 'shopId' in `col()`.**
* **Switch the order of arguments to `withColumn`, making 'shopId' first and 'shopNumber' second.**
* **Change `withColumn` to `copyDataFrame`.**
* **Use `withColumnRenamed` instead, with 'shopId' as the first argument and 'shopNumber' as the second.**

**Explanation**

The error in the code block is the misuse of the `**withColumn**` method, which is intended for adding new columns or replacing existing columns with new expressions.

For simply renaming a column, the appropriate method is `**withColumnRenamed**`.

- The corrected code should be:

purchaseDataDf.withColumnRenamed('shopId', 'shopNumber') This code correctly renames the column `shopId` to `shopNumber` in `purchaseDataDf`.

The `copyDataFrame` operator is not a valid method in PySpark, making that option 4 incorrect.

The use of `col()` is unnecessary here as the operation is just a simple renaming of the column, not an expression-based transformation.

Thus, the correct approach is to use `**withColumnRenamed**` with the current and new column names as string arguments.

## Question 38:

Select the code block that correctly **adds a new column** `purchaseDateFormatted` to DataFrame `salesDataDf`, **converting unix epoch timestamps** in column `purchaseDate` **to strings in the format MM/dd/yyyy**:

* `salesDataDf.withColumn('purchaseDateFormatted', from\_unixtime('purchaseDate', 'format=MM/dd/yyyy'))`
* `salesDataDf.withColumnRenamed('purchaseDate', 'purchaseDateFormatted', from\_unixtime('purchaseDateFormatted', 'format=MM/dd/yyyy'))`
* `salesDataDf.apply(from\_unixtime(format='MM/dd/yyyy')).asColumn('purchaseDateFormatted')`
* `salesDataDf.withColumn('purchaseDateFormatted', from\_unixtime('purchaseDate', 'MM/dd/yyyy'))`
* `salesDataDf.withColumn('purchaseDateFormatted', from\_unixtime('purchaseDate'))`

**Explanation**

In PySpark, the `from\_unixtime` function is used to convert Unix epoch timestamps to human-readable date strings.

To add a new column with formatted dates, the `withColumn` method is used, along with `from\_unixtime`, specifying the desired date format.

- Option 4 is correct as it correctly uses `salesDataDf.withColumn` to add a new column `purchaseDateFormatted`.

It utilizes `from\_unixtime` to convert the Unix timestamp in `purchaseDate` to a date string in the format 'MM/dd/yyyy'.

- The other options are incorrect due to syntax errors or misuse of functions.

Option 1 incorrectly specifies the format string, option 2 misuses `withColumnRenamed` which is for renaming columns,

option 3 uses a non-existent `apply` method for DataFrame transformation,

and option 5 does not specify the date format, resulting in a default format that may not match the required 'MM/dd/yyyy'.

Therefore, to convert the Unix epoch timestamp in `purchaseDate` to a formatted string and add it as a new column `purchaseDateFormatted`,

the appropriate code is salesDataDf.withColumn('purchaseDateFormatted, from\_unixtime('purchaseDate', 'MM/dd/yyyy'))`.

## Question 39:

Find the error in the following Python code block that is intended to divide DataFrame `salesDataDf` into 14 parts based on columns `shopId` and `saleDate` (in this order):

salesDataDf.coalesce(14, ('shopId', 'saleDate'))

* **Remove the parentheses around the column names and append `.select()`.**
* **Replace `coalesce` with `repartition`, remove parentheses around column names, and append `.count()`.**
* **Replace `coalesce` with `repartition`, remove parentheses around column names, and append `.select()`.**
* **Change `coalesce` to `repartition` and switch parentheses to square brackets around the column names.**
* **Switch `coalesce` to `repartition`.**

**Explanation**

The error in the provided code block is the misuse of the `coalesce` method for a task that requires partitioning the DataFrame by specific columns.

The `coalesce` method is used to reduce the number of partitions, typically without a shuffle, and does not support column-based partitioning.

- The correct method for this task is `repartition`. Additionally, the column names should be provided as a list, enclosed in square brackets, not in parentheses.  - The corrected code should be either:  salesDataDf.repartition(14, 'shopId', 'saleDate') or salesDataDf.repartition(14, ['shopId', 'saleDate'])

This code will repartition `salesDataDf` into 14 parts based on the values in the `shopId` and `saleDate` columns. The use of `.count()` or `.select()` is not necessary for the repartitioning operation.

Therefore, option 4 correctly identifies the changes needed to partition the DataFrame as intended.

## Question 40:

Choose the code block that correctly creates a DataFrame with two columns `weather` and `humidity` **where**`weather` is a string and `humidity` is a double:

spark.DataFrame({'weather': ['rainy', 'sunny'], 'humidity': [75.0, 40.0]})

spark.createDataFrame([('sunny', 40.0), ('rainy', 75.0)], ['weather', 'humidity'])

from pyspark.sql import types as T

spark.createDataFrame([('sunny', 40.0), ('rainy', 75.0)],

T.StructType([T.StructField('weather', T.StringType()),

T.StructField('humidity', T.DoubleType())]))

spark.newDataFrame([('sunny', 40.0), ('rainy', 75.0)],

['weather', 'humidity'])

spark.createDataFrame({'weather': ['rainy', 'sunny'],

'humidity': [75.0, 40.0]})

**Explanation**

Option 3 is correct because it explicitly defines the schema of the DataFrame, setting the `weather` column as a `StringType` and `humidity` as a `DoubleType`.

This explicit schema definition is accomplished using a `**StructType**` schema with `**StructField**` entries for each column.

- Options 1 and 5 attempt to use a dictionary to create a DataFrame, but the correct method in PySpark for this is `**createDataFrame**`.

Moreover, when using a dictionary, `createDataFrame` **expects an RDD**, a **list**of tuples, or a list of dictionaries, not a single dictionary.

- Option 2 does not explicitly define the schema, and while Spark might infer the schema correctly, it's not assured by this syntax that `humidity` will be a `DoubleType`.

- Option 4 incorrectly uses `newDataFrame`, which is not a valid method in PySpark's **DataFrame API**. Thus, the assured way to create a DataFrame with the specified data types for `weather` and `humidity` is option 3, using the `createDataFrame` method with a defined `StructType` schema.

## Question 41:

Choose the code block that correctly uses the schema `dataSchema` to read a Parquet file from the location `dataFilePath` into a DataFrame in PySpark:

* **`spark.read.schema(dataSchema).format('parquet').load(dataFilePath)`**
* **`spark.read.schema('dataSchema').format('parquet').load(dataFilePath)`**
* **`spark.read().schema(dataSchema).parquet(dataFilePath)`**
* **`spark.read().schema(dataSchema).format(parquet).load(dataFilePath)`**
* **`spark.read.schema(dataSchema).open(dataFilePath)`**

**Explanation**

The task is to read a Parquet file into a DataFrame using a predefined schema. The correct method involves specifying the schema, data source format, and file path. - Option 1 is correct.

Ituses spark.read.schema(dataSchema).format('parquet').load(dataFilePath) to specify the schema (`dataSchema`), set the format to 'parquet', and load the data from the given file path (`dataFilePath`).

- Option 2 is incorrect because it passes the schema name as a string, whereas the `schema` method requires a schema object.

- Option 3 is almost correct but uses unnecessary parentheses after `read`. The shorthand method `parquet` correctly loads the Parquet file but should be preceded by `spark.read.schema(dataSchema)`.

- Option 4 is incorrect due to syntax errors: `format(parquet)` should be `format('parquet')`, and there should not be parentheses after `read`.

- Option 5 incorrectly uses a non-existent `open` method.

The correct method to load the file is `load`.  Thus, the correct approach to read a Parquet file with a predefined schema is using option 1.

## Question 42:

Determine the error in the following Python code block intended to add a column `productKeywords` to DataFrame `productsDf` that contains an array of all words in the `productName` column:

productsDf.withColumnRenamed('productKeywords', split('productName'))

* **Wrap all column names with `col()` operator.**
* **Change `withColumnRenamed` to `withColumn` and pass a space `' '` as the second argument to `split`.**
* **Replace `withColumnRenamed` with `withColumn` and use `splitString` instead of `split`.**
* **Switch `withColumnRenamed` to `withColumn` and provide a space `' '` as the second argument to `split`.**
* **Swap 'productKeywords' and `split('productName')`.**

**Explanation**

The error in the code block is the use of the `**withColumnRenamed**` method, which is for renaming columns, not for adding a new column or transforming an existing one.

The `**split**` function in PySpark is used to split a string into an array based on a delimiter.

The correct approach is to use the `withColumn` method to add a new column based on a column expression. - The corrected code should be:

1. from pyspark.sql.functions import split, col
3. productsDf.withColumn('productKeywords', split(col('productName'), ' '))

This code correctly adds a new column `productKeywords` to `productsDf` by splitting the `productName` column into an array of words using a space as the delimiter.

The `**withColumn**` method is appropriate for adding a new column or replacing an existing one based on a column expression.

The use of `**splitString**` is **incorrect** as it's not a standard function in PySpark.

The option that correctly addresses the error is option 4.

## Question 43:

Choose the correct option to complete the following PySpark code block, which should return all rows from DataFrame `productDetailsDf` where the array column `productFeatures` contains at least three elements:

productDetailsDf.\_1\_(\_2\_('productFeatures')).\_3\_(\_4\_)

* **`select`, `count`, `col('productFeatures')`, `>3`**
* **`filter`, `count`, `productFeatures`, `>=3`**
* **`select`, `count`, `'productFeatures'`, `>3`**
* **`filter`, `size`, `'productFeatures'`, `>=3`**
* **`select`, `size`, `'productFeatures'`, `>3`**

**Explanation**

The task is to filter rows in a DataFrame based on the size of an array column.

The correct approach involves using the `**filter**` method, which is used for filtering rows, along with the `**size**` function, which returns the size of an array column.

- Option 4 is correct. It uses `filter` to apply the row filtering and `size` to determine the size of the array in the `productFeatures` column.

The condition `>=3` ensures that only rows with at least three elements in the `productFeatures` array are retained.

- The other options are incorrect due to either using the wrong method (`select` instead of `filter`), the wrong function (`count` instead of `size`), or the wrong condition (using `>` instead of `>=`).

Thus, the correct code to filter rows based on the size of the `productFeatures` array is `productDetailsDf.filter(size(col('productFeatures')) >= 3)`.

## Question 44:

Determine the correct code block that samples around 200 rows **randomly** from a DataFrame `customerDataDf` consisting of 1500 rows. Each row in the DataFrame can appear multiple times in the sample.

Choose the appropriate option:

* **`customerDataDf.sampleWithReplacement(0.2, 5678)`**
* **`customerDataDf.randomSample(0.2, 5678)`**
* **`customerDataDf.sample(0.13)`**
* **`customerDataDf.sample(0.7, 1234)`**
* **`customerDataDf.sample(True, 0.13, 5678)`**

**Explanation**

In PySpark, the `**sample**` method is used to**randomly sample rows** from a DataFrame. The method takes three parameters:

* `withReplacement` (whether rows can be sampled more than once),
* `fraction` (the fraction of rows to sample),
* and `seed` (for reproducibility). - Since the DataFrame has 1500 rows and the goal is to sample approximately 200 rows, a fraction close to 0.13 is suitable.

- `withReplacement` should be set to `True` to allow the possibility of duplicating rows in the sample.

- `seed` (e.g., 5678) ensures that the random sampling is reproducible.

- Option 5, `customerDataDf.sample(True, 0.13, 5678)`, correctly configures the sampling method for the described scenario.

It samples about 13% of the DataFrame, allowing for duplicate rows, thus meeting the requirement of approximately 200 rows from a 1500-row DataFrame.

## Question 45:

Identify the code block that correctly removes columns `errorMargin` and `productCode` from DataFrame `salesDataDf`

* **`salesDataDf.removeColumns('errorMargin', 'productCode')`**
* **`salesDataDf.exclude(['errorMargin', 'productCode'])`**
* **`salesDataDf.drop('errorMargin', 'productCode')`**
* **`salesDataDf.omit('errorMargin', 'productCode')`**
* **`salesDataDf.dropColumns('errorMargin', 'productCode')`**

**Explanation**

In PySpark, the `**drop**` method is used to remove one or more columns from a DataFrame.

The correct syntax to drop multiple columns is DataFrame.drop(\*cols),

*where `\*cols` can be multiple column names as separate arguments.*

- Option 3, salesDataDf.drop('errorMargin', 'productCode'), correctly uses the `drop` method to remove the columns `errorMargin` and `productCode` from `salesDataDf`.

- The other options use methods that **do not exist**in the PySpark DataFrame API, such as `removeColumns`, `exclude`, `omit`, and `dropColumns`.

These methods are not standard and would result in an error if used.

Therefore, the correct way to remove the specified columns from `salesDataDf` is using the `drop` method as shown in option 3.

## Question 46:

Identify the **error**in the following Python code block intended to return a DataFrame with rows where the `manufacturer` column **contains** the letter combination 'co' in this order:   productDataDf.filter(Column('manufacturer').isIn('co'))

* **Replace `Column` with `col` and change `isIn` to `contains`.**
* **Correct the expression to `isIn('co', 'manufacturer')`.**
* **Switch `isIn` to `contains` and use `col('manufacturer')` to access the column.**
* **Change `filter` to `select` as the expression returns a single column.**

**Explanation**

The error in the code block lies in the use of the `**isIn**` method, which checks if a column's value is within a given list of values.

This is not suitable for finding a substring within string values of a column.

The correct method for this task is `**contains**`, which checks for the presence of a substring.

- The corrected code block should be:  productDataDf.filter(col('manufacturer').contains('co'))

This uses the `**col**` function to refer to the `manufacturer` column and then applies the `contains` method to filter rows where the `manufacturer` column contains the substring 'co'.

The other options are incorrect because they either suggest using `isIn`, which is not the right method for substring checking, or they propose a change to `select`, which is not relevant to the task of filtering rows.

Bottom of Form

Top of Form

## Question 47:

What is the **correct** sequence of methods and options in PySpark to write DataFrame `salesDataDf` to **disk**at path `dataFilePath` as a single CSV file, using tabs as separators, expressing missing values as '**n/a**', and without including a header row?

salesDataDf.coalesce(1).write.option('sep', '\t').option('header', 'true').csv(dataFilePath)

salesDataDf.coalesce(1).write.option('sep','\t').option('nullValue', 'n/a').csv(dataFilePath)

salesDataDf.repartition(1).write.option(‘sep’, ‘\t’).option(‘nullValue’, ‘n/a’).csv(dataFilePath)

salesDataDf.csv().write.option('sep', '\t').option('emptyValue', 'n/a').csv(dataFilePath)

salesDataDf.repartition(1).write.mode('sep', '\t').mode('nullValue', 'n/a').csv(dataFilePath)

**Explanation**

To write a DataFrame to a CSV file with specific formatting in PySpark, the correct sequence involves using the `**coalesce(1)**` method to write the data into a single file.

The `**write**` method is chained with `option` to specify the separator as a tab character with `'**sep**', '\t'` and to express missing values as 'n/a' with `'nullValue', 'n/a'`.

The `csv` method is then called with the file path to perform the write operation. Notably, the header should be omitted, which is done by setting the option `'header', 'false'`.

However, since none of the provided options include this, the closest correct option based on the available choices is the second one, which correctly sets the separator and null value representation but does not explicitly set the header option to 'false'.

By default, the header is not included unless explicitly set to 'true', so the second option will still produce the desired output.

## Question 48:

**Which of the following code blocks returns a new DataFrame containing only the columns `errorIndicator` and `transactionAmount` from every odd-numbered row of DataFrame `purchaseRecordsDf`?**

* **`purchaseRecordsDf.filter(col('recordId').isin([2,4,6])).select(['errorIndicator', 'transactionAmount'])`**
* **`purchaseRecordsDf.select(col('recordId').isin([2,4,6]), 'errorIndicator', 'transactionAmount')`**
* **`purchaseRecordsDf.filter('recordId' % 2 != 0).select('errorIndicator', 'transactionAmount')`**
* **`purchaseRecordsDf.filter(col('recordId') % 2 != 0).select('errorIndicator', 'transactionAmount')`**
* **`purchaseRecordsDf.filter(col('recordId').isin([2,4,6]))`**
* **`purchaseRecordsDf.createOrReplaceTempView('purchaseRecords'); spark.sql('SELECT errorIndicator, transactionAmount FROM purchaseRecords WHERE recordId % 2 != 0')`**

**Explanation**

To select every odd-numbered row based on the `recordId` column, we can use the modulo operation with the condition `recordId % 2 != 0`, which checks for a remainder of 1 when dividing by 2 (odd numbers).

After filtering for every odd-numbered row, we want to select only the columns `errorIndicator` and `transactionAmount`. The `select` method allows us to specify the columns to include.

The correct code block that performs this operation is represented by option 4: purchaseRecordsDf.filter(col('recordId') % 2 != 0).select('errorIndicator', 'transactionAmount')

The other options do not correctly use the modulo operation to filter every odd-numbered row, do not select the specified columns, or they use incorrect methods.

## Question 49:

The code block shown below contains an **error**. It is intended to display the schema of DataFrame `customerOrdersDf`.

Identify the error in the code block:   customerOrdersDf.rdd.printSchema

* **There is no direct way to print a schema in Spark; the schema should be printed using `print (customerOrdersDf.columns)`.**
* **The code block should be encapsulated within a `print()` function.**
* **`printSchema` is accessible only through the Spark session object, so the code should be `spark.printSchema(customerOrdersDf)`.**
* **`printSchema` is a DataFrame method and should be invoked directly on the DataFrame object, not on its RDD.**
* **`printSchema` is not a method of the RDD API; the schema should be printed using `customerOrdersDf.printSchema()`.**

**Explanation**

In PySpark, `**printSchema**` is a method available on DataFrame objects that prints out the schema in a tree format, including column names and data types.

The method `printSchema` does not exist for **RDD**s, as RDDs do not have a defined schema like DataFrames do.

Therefore, calling `printSchema` on an RDD, as with `customerOrdersDf.rdd.printSchema`, is incorrect and will result in an AttributeError.

The correct way to print the schema of a DataFrame is to call `printSchema` directly on the DataFrame object, as in customerOrdersDf.printSchema().

## Question 50:

Which of the following code blocks returns a one-column DataFrame consisting of **unique values** from the `vendor` column in the DataFrame `productDetailsDf`, **excluding** any values that contain the letter 'Z'?

* **`productDetailsDf.filter(col('vendor').not\_contains('Z')).select('vendor').distinct()`**
* **`productDetailsDf.select(~col('vendor').contains('Z')).distinct()`**
* **`productDetailsDf.filter(~col('vendor').contains('Z')).select('vendor').distinct()`**
* **`productDetailsDf.filter(col('vendor').contains('Z')).select('vendor').distinct()`**
* **`productDetailsDf.filter(col('vendor').contains('Z')).select(col('vendor')).distinct()`**

**Explanation**

The correct approach to filter out values that contain a specific letter from a DataFrame column, and to return distinct values only, is to use the `**filter**` method with a condition that negates the `contains` method.

After filtering, the `select` method is used to specify the column of interest, and `**distinct**` is called to ensure each value is only listed once.

- Option 3 is correct because it applies a **logical NOT (`~`)**to the condition that checks for the presence of 'Z' in the `vendor` column.

This ensures that only rows where the `vendor` column does not contain 'Z' are returned. It then selects the `vendor` column and uses `distinct` to return unique values only.

- The other options either do not apply the negation correctly, use non-existent methods like `not\_contains`, or incorrectly use the `contains` method without negation,

which would result in selecting values that contain 'Z', opposite of the task's requirement.

## Question 51:

In what sequence should the following code snippets be executed to**generate a new DataFrame** that combines all **unique** values in the `distributor` column with the **exploded**values of the `features` array column from the `productCatalogDf` DataFrame?

* **productCatalogDf.createOrReplaceTempView('productCatalog') followed by spark.sql('SELECT distributor, explode(features) FROM productCatalog')**
* **productCatalogDf.createOrReplaceTempView('productCatalog') followed by spark.sql('SELECT distributor, explode(features) AS feature FROM productCatalog')**
* **Just run spark.sql('SELECT distributor, explode(features) FROM productCatalogDF') without creating a view**
* **spark.sql('SELECT distributor, explode(features) AS feature FROM productCatalog') without prior view creation**
* **productCatalogDf.createOrReplaceTempView('productCatalog') without subsequent SQL query**

**Explanation**

To create a new DataFrame that lists all unique `distributor` values alongside the exploded `features` from the `productCatalogDf`, **you must first create a temporary view of the DataFrame**.

This allows you to use **Spark SQL** to query the DataFrame.

The correct sequence is to first execute `productCatalogDf.createOrReplaceTempView('productCatalog')` and then run the SQL query

spark.sql('SELECT distributor, explode(features) AS feature FROM productCatalog').

This sequence ensures that the DataFrame is available as a temporary view (`productCatalog`) for the SQL query to operate on, which is correctly represented by the second option.

## Question 52:

How to transform the DataFrame `salesData` to remove the columns `revenue` and `productId` and add a new column `categoryId` with all values set to 5?

* `salesData.withColumn('categoryId', lit(5)).drop('revenue', 'productId')`
* `salesData.withNewColumn('categoryId', lit(5)).remove('revenue', 'productId')`
* `salesData.withColumnRenamed('categoryId', lit(5)).dropColumns('revenue', 'productId')`
* `salesData.addColumn('categoryId', lit(5)).without('revenue', 'productId')`

**Explanation**

The `**withColumn**` method in PySpark is used to add a new column to a DataFrame or to replace an existing column with a new one.

The `**lit**` function is used to create a new Column of literal value 5.

The `**drop**` method is then used to remove multiple columns from a DataFrame.

The correct sequence is to first add the new column using `**withColumn**` and `**lit**`, then to remove the unwanted columns with `**drop**`.

Therefore, the correct code is salesData.withColumn('categoryId', lit(5)).drop('revenue', 'productId'), which corresponds to the first option.

Bottom of Form

Top of Form

## Question 53:

To save the DataFrame `salesSummaryDf` to a specified directory `outputLocation` as a **Parquet** file with efficient **compression**, **overwriting** any existing data,

which sequence of operations should be used?

* `salesSummaryDf.write.format('overwrite').option('compression', 'brotli').path(outputLocation)`
* `salesSummaryDf.saveAsParquetFile('overwrite', 'brotli', outputLocation)`
* `salesSummaryDf.write.save('parquet', 'overwrite', 'brotli', outputLocation)`
* `salesSummaryDf.write.mode('overwrite').option('compression', 'brotli').parquet(outputLocation)`
* `salesSummaryDf.write.parquet('overwrite', 'brotli', outputLocation)`

**Explanation**

To write a DataFrame to Parquet format while applying **Brotli compression** and overwriting existing files at the destination, you need to chain the appropriate methods from the DataFrame writer API.

The sequence begins with `**write**` to set the DataFrame in write mode, followed by `mode('overwrite')` to ensure existing data at the path is replaced.

The compression type is specified using `option('compression', 'brotli')`. Finally, `parquet(outputLocation)` is called to write the DataFrame as a Parquet file to the given directory.

Therefore, the sequence that accomplishes this task is encapsulated in option 4: salesSummaryDf.write.mode('overwrite').option('compression', 'brotli').parquet(outputLocation).

## Question 54:

How can the DataFrame `productDetailsDf` be written to the path `dataOutputLocation` in Avro format without overwriting any existing files?

* `productDetailsDf.saveAsAvro(dataOutputLocation)`
* `productDetailsDf.write.configuration('avro').mode('doNotOverwrite').save(dataOutputLocation)`
* `productDetailsDf.write.format('avro').option('saveMode', 'ignore').toPath(dataOutputLocation)`
* `productDetailsDf.write.format('avro').mode('ignore').save(dataOutputLocation)`
* `sparkSession.writeDataFrame(productDetailsDf).inFormat('avro').withSaveMode('ignore').atLocation(dataOutputLocation)`

**Explanation**

To write a DataFrame in Avro format without overwriting existing data, the `write` method should be used with the `format` set to 'avro'.

The `mode` should be set to 'ignore' to skip the write operation if the file already exists at the output location. This prevents any existing data from being overwritten.

The `save` method is then used to specify the output path. Therefore, the correct sequence is productDetailsDf.write.format('avro').mode('ignore').save(dataOutputLocation), which is represented by the fourth option in the list.

Other options either use non-existent methods or incorrect parameters that do not align with PySpark's API for DataFrame I/O operations.

## Question 55:

Select the code snippet that will export the DataFrame `productCatalog` to a specified directory `destinationPath` in Avro format without overwriting any existing data at the destination.

* `productCatalog.write.avro(destinationPath)`
* `productCatalog.write.format('avro').mode('skipIfExists').save(destinationPath)`
* `productCatalog.write.format('avro').mode('ignore').save(destinationPath)`
* `productCatalog.save.format('avro').mode('ignore').write(destinationPath)`
* `sparkSession.getDataFrameWriter(productCatalog).format('avro').mode('ignore').write(destinationPath)`

**Explanation**

When saving a DataFrame to a file, the `.write` method is used to specify the output format and configuration. The `.format('avro')` method indicates that the output should be in Avro format.

The `.mode('ignore')` option is used to ensure that if the file already exists at the destination path, the operation will not overwrite it but will silently ignore the write request.

The `.save(destinationPath)` method is then called to specify the output directory.

Therefore, the correct code snippet that performs this operation without overwriting existing files is productCatalog.write.format('avro').mode('ignore').save(destinationPath), corresponding to option 3.

## Question 56:

Identify the issue(s) in the code segment designed to generate a DataFrame with a single column `result`.

This column should contain the **fifth power of all non-null numerical**values from the `salesAmount` column in the `salesData` DataFrame, and **null** for all rows where `salesAmount` is null.

# Assuming there is already a DataFrame named 'salesData' with a column 'salesAmount'

def raise\_to\_power(value):

# This will raise an error if value is None

return value \*\* 5

power\_udf = udf(raise\_to\_power, DoubleType())

salesData = salesData.withColumn('result', power\_udf(salesData['salesAmount']))

**The `raise\_to\_power` function cannot handle null values in the `salesAmount` column and the resulting DataFrame's column should be named `result`.**

**The resulting DataFrame incorrectly includes multiple columns instead of the expected single `result` column.**

**The `raise\_to\_power` function cannot process null values in the `salesAmount` column, the DataFrame column should be named `result`, and the SparkSession cannot access the `salesData` DataFrame for the operation.**

**The `raise\_to\_power` function does not handle null values in the `salesAmount` column, the resulting DataFrame's column should be `result`, and the Spark driver does not invoke the UDF function correctly.**

**The `raise\_to\_power` function is not registered with the Spark driver correctly, and the resulting DataFrame's column is not named `result`.**

**Explanation**

The provided code segment is expected to use a UDF to compute the fifth power of the `salesAmount` column values.

However, the function named `raise\_to\_power` is not handling null values correctly, which can cause errors when the function encounters null entries.

Additionally, the output column of the resultant DataFrame must be named `result` to meet the requirements specified.

The other options include incorrect statements regarding the code's issues or mention elements not present in the provided code snippet, such as incorrect DataFrame naming or Spark driver's interaction with the UDF.

## Question 57:

**How does the DataFrame API apply a transformation to filter rows with a specific condition?**

* **`DataFrame.filter(condition)`**
* **`DataFrame.where(condition)`**
* **`DataFrame.select(condition)`**
* **`DataFrame.withFilter(condition)`**

**Explanation**

The DataFrame API in Spark allows for row filtering using the `filter` method, where `condition` is a boolean expression that specifies which rows to retain.

The `where` method is an alias for `filter` and can also be used interchangeably.

The `select` method is used for selecting specific columns, not for filtering rows based on a condition.

There is no `withFilter` method in the DataFrame API.

**Here's an deep explanation:**

In the context of DataFrame operations in Apache Spark and similar data processing libraries, filtering is a common operation used to select rows from a DataFrame that match a particular condition. The **filter** method takes a condition and returns a new DataFrame containing only the rows that match the condition.

Here's what you need to know about the correct answer and the other options:

* **DataFrame.filter(condition)** is indeed a method in Apache Spark's DataFrame API that applies a boolean condition to filter rows. The **filter** method is synonymous with **DataFrame.where(condition)**; both methods are interchangeable and will produce the same results.
* **DataFrame.where(condition)** is functionally the same as **filter** and can also be used to filter rows in a DataFrame based on a specified condition. This means that while **DataFrame.filter(condition)** is marked as the correct answer, **DataFrame.where(condition)** is also correct and could have been marked as such.
* **DataFrame.select(condition)** is not correct in this context because the **select** method is used to select specific columns from a DataFrame, not to filter rows based on a condition.
* **DataFrame.withFilter(condition)** is not a method in the DataFrame API; thus, it is incorrect.

Both **filter** and **where** are used to refine the data by selecting rows that satisfy the condition provided, which is typically expressed using DataFrame column expressions. These methods are part of the transformation operations in Spark, meaning they are lazily evaluated and will only be executed when an action (like **show**, **collect**, or **count**) is called on the DataFrame.

## Question 58:

What is the correct method to aggregate data after grouping by a column in a DataFrame?

* **`DataFrame.groupBy(column).agg(aggregateFunctions)`**
* **`DataFrame.aggregateBy(column).group(aggregateFunctions)`**
* **`DataFrame.mapGroups(column, aggregateFunctions)`**
* **`DataFrame.reduceByKey(column, aggregateFunctions)`**

**Explanation**

In Spark's DataFrame API, the `**groupBy**` method is used to collect rows into groups based on the values of a specified column, and the `agg` method is then used to apply aggregate functions to each group.

The `**aggregateBy**` method **does not exist**, `**mapGroups**` is an operation on Datasets, and `**reduceByKey**` is an RDD transformation, not applicable directly to DataFrames.

**Here's a deeper explanation:**

In Apache Spark, a DataFrame is a distributed collection of data organized into named columns, conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood.

When you want to perform an aggregation over a grouped dataset, you typically follow a two-step process:

1. **Grouping the Data**: First, you need to group the data by one or more columns. This is done using the **groupBy** method. This method takes one or more columns on which you want to group the data and returns a **RelationalGroupedDataset**, which is a special type of dataset that has grouping semantics but does not have the data associated with the groups yet.
   1. groupedData = DataFrame.groupBy('column')

In this step, no data is actually moved around or aggregated yet—it's just a transformation that defines how the data should be grouped.

1. **Aggregating the Data**: After you have defined the grouping, you then define the aggregation. This is done using the **agg** method on the result of the **groupBy** method. The **agg** method takes as its argument a set of aggregation functions that specify the calculations to perform on the grouped data.
   1. aggregatedData = groupedData.agg({'column': 'sum'})

The **agg** method can take either a single aggregation function or a dictionary of column names to aggregation functions. This allows for very flexible and powerful aggregations, and the computation happens when an action is called on the aggregated data.

The other options provided in the practice test question are incorrect for this context:

* **DataFrame.aggregateBy(column).group(aggregateFunctions)** is not a correct method because **aggregateBy** is not a method available on a DataFrame in Spark.
* **DataFrame.mapGroups(column, aggregateFunctions)** is incorrect because **mapGroups** is a method used with **GroupedData** in Spark, but it's used for iterating over each group of data rather than aggregating.
* **DataFrame.reduceByKey(column, aggregateFunctions)** is also incorrect. This method is not available on DataFrames; it's part of the RDD API, and it's used for reducing values by key in a pair RDD, not in a DataFrame.

So, in summary, **DataFrame.groupBy(column).agg(aggregateFunctions)** is the standard way to group and then aggregate data in a Spark DataFrame. This syntax is part of the higher-level DataFrame API, which is built for complex aggregations and operations on structured data, leveraging Spark's Catalyst optimizer for efficient execution.

## Question 59:

Determine how to complete the following PySpark code block so that it returns a column indicating whether rows in DataFrame `salesDataDf` have `shopId` values between 20 and 30 (inclusive) and `productCode` equal to 2. Fill in the blanks with the appropriate options:

  salesDataDf.\_\_1\_\_(\_\_2\_\_.\_\_3\_\_() \_\_4\_\_ \_\_5\_\_)

Options for filling in the blanks:

`select`

`col('shopId')`

`between(20, 30)`

`and`

`col('productCode')==2`

`filter`

`col('shopId')`

`geq(20) & leq(30)`

`and`

`col('productCode')==2`

`where`

`col('shopId')`

`>=20 & <=30`

`and`

`col('productCode')==2`

`filter`

`col('shopId')`

`between(20, 30)`

`&`

`col('productCode')==2`

`select`

`col('shopId')`

`geq(20) & leq(30)`

`&`

`col('productCode')==2`

**Explanation**

The task requires constructing a query that checks whether `shopId` values fall within a specific range and `productCode` equals a certain value.

In PySpark, the `filter` or `where` function is typically used for such conditions, with logical operators to combine them. However, the given options seem to suggest using `select`, which is not the standard way to filter data.

- The closest option that represents the logic, despite being syntactically incorrect, is the first one.

It uses `select`, `col('shopId')`, `between(20, 30)`, `and`, `col('productCode')==2`.

But this code will not execute correctly in PySpark because `select` is not used for filtering, `between` is not directly applicable, and `and` should be `&`.

- The correct approach in PySpark, using `filter` or `where`, would be:

1. from pyspark.sql.functions import col
3. salesDataDf.filter((col('shopId') >= 20) & (col('shopId') <= 30) & (col('productCode') == 2))

However, based on the provided structure and options, the first option is the closest to the intended logic.

## Question 60:

Select the code block that correctly displays the 10 rows with the **smallest** values in column `purchaseAmount` in DataFrame `salesDataDf` in a formatted manner:

* **`salesDataDf.sort(asc('purchaseAmount')).show(10)`**
* **`salesDataDf.sort(col('purchaseAmount')).show(10)`**
* **`salesDataDf.sort(col('purchaseAmount').desc()).head()`**
* **`salesDataDf.sort(col('purchaseAmount').asc()).print(10)`**
* **`salesDataDf.orderBy('purchaseAmount').asc().show(10)`**

**Explanation**

The task is to display rows with the smallest values in a specified column. The `**sort**` function in PySpark sorts the DataFrame, and by default, it sorts in**ascending order**, which is required for finding the smallest values.

- Option 2 is correct. It uses salesDataDf.sort(col('purchaseAmount')).show(10) to sort the DataFrame in ascending order by the `purchaseAmount` column and then displays the first 10 rows in a nicely formatted way.

- Option 1 is incorrect because `**asc**` is not used correctly within the `sort` function.

It should be col('purchaseAmount').asc().

- Option 3 sorts in descending order, which is not what the question asks for, and `**head()**` returns a list rather than displaying the DataFrame.

- Option 4 uses a **non-existent `print` method**. The correct method for displaying a DataFrame is `**show**`.

- Option 5 is incorrect due to the syntactical error in chaining `**asc()**` after `**orderBy()**`.

Therefore, to display the smallest `purchaseAmount` values, the appropriate code is : salesDataDf.sort(col('purchaseAmount')).show(10)

Bottom of Form