Contents

[Question 1 3](#_Toc158033835)

[Question 2 4](#_Toc158033836)

[Question 3 5](#_Toc158033837)

[Question 4 6](#_Toc158033838)

[Question 5: 7](#_Toc158033839)

[Question 6 8](#_Toc158033840)

[Question 7: 9](#_Toc158033841)

[Question 8 10](#_Toc158033842)

[Question 9: 11](#_Toc158033843)

[Question 10: 12](#_Toc158033844)

[Question 11: 13](#_Toc158033845)

[Question 12: 14](#_Toc158033846)

[Question 13: 15](#_Toc158033847)

[Question 14: 16](#_Toc158033848)

[Question 15: 17](#_Toc158033849)

[Question 16: 18](#_Toc158033850)

[Question 17: 19](#_Toc158033851)

[Question 18: 20](#_Toc158033852)

[Question 19: 21](#_Toc158033853)

[Question 20: 23](#_Toc158033854)

[Question 21: 24](#_Toc158033855)

[Question 22: 25](#_Toc158033856)

[Question 23: 26](#_Toc158033857)

[Question 24: 27](#_Toc158033858)

[Question 25: 28](#_Toc158033859)

[Question 26: 30](#_Toc158033860)

[Question 29: 33](#_Toc158033861)

[Question 30: 34](#_Toc158033862)

[Question 31: 35](#_Toc158033863)

[Question 32: 36](#_Toc158033864)

[Question 33: 37](#_Toc158033865)

[Question 34: 38](#_Toc158033866)

[Question 35: 39](#_Toc158033867)

[Question 36: 40](#_Toc158033868)

[Question 37: 41](#_Toc158033869)

[Question 39: 44](#_Toc158033870)

[Question 40: 45](#_Toc158033871)

## Question 1

Which of the following statements about the Spark driver is incorrect?

* The Spark driver is the node in which the Spark application's main method runs to coordinate the Spark application.
* The Spark driver is horizontally scaled to increase overall processing throughput.
* The Spark driver contains the SparkContext object.
* The Spark driver is responsible for scheduling the execution of data by various worker nodes in cluster mode.
* The Spark driver should be as close as possible to worker nodes for optimal performance

**Explanation**

1: Correct. The Spark driver is indeed the node where the main method of the Spark application runs, and it coordinates the entire application.

2: Incorrect. The Spark driver is not horizontally scalable. It's a single point in the Spark application and does not scale like worker nodes or executors. Horizontal scaling applies to worker nodes in Spark, not to the driver.

3: Correct. The SparkContext object is indeed part of the Spark driver and is essential for initializing the environment for Spark's functionalities.

4: Correct. The driver is responsible for task scheduling and distribution among worker nodes in a cluster.

5: Correct. For optimal performance, it's recommended to have the Spark driver close to worker nodes to reduce latency in task distribution and status updates.

**Additional Information:**

The Spark driver plays a crucial role in running the application and in task distribution. It's the master node of a Spark application and is responsible for analyzing, distributing, and scheduling work to the executors.

For more information, refer to the official Apache Spark documentation at <https://spark.apache.org/docs/latest/cluster-overview.html>

## Question 2

Which of the following describes nodes in cluster-mode Spark?

* Nodes are the most granular level of execution in the Spark execution hierarchy.
* There is only one node and it hosts both the driver and executors.
* Nodes are another term for executors, so they are processing engine instances for performing computations.
* There are driver nodes and worker nodes, both of which can scale horizontally.
* Worker nodes are machines that host the executors responsible for the execution of tasks.

**Explanation**

1: Incorrect. The most granular level of execution in Spark is not the node but the task. Tasks are the smallest units of work that are sent to the executors.

2: Incorrect. In a Spark cluster, there are typically multiple nodes. The driver and executors can be on separate nodes, especially in a distributed setting.

3: Incorrect. Nodes and executors are not synonymous. Executors are JVM processes that run computations and store data for the application, and they run on nodes.

4: Incorrect. While there are driver nodes and worker nodes, only worker nodes can scale horizontally. The driver node cannot be scaled horizontally.

5: Correct. Worker nodes are indeed the machines that host the executors. These executors are responsible for the execution of tasks assigned to them by the driver.

***Additional Information:***

In a Spark cluster, nodes refer to the machines or instances in the cluster. Executors, which are launched by Spark on nodes, run the tasks. Understanding the architecture is crucial for effective Spark deployment and optimization.

For a detailed understanding, refer to the Apache Spark official documentation at <https://spark.apache.org/docs/latest/cluster-overview.html.>

## Question 3

**Which of the following statements about slots is true?**

* **There must be more slots than executors.**
* **There must be more tasks than slots.**
* **Slots are the most granular level of execution in the Spark execution hierarchy.**
* **Slots are not used in cluster mode.**
* **Slots are resources for parallelization within a Spark application.**

**Explanation**

1: Incorrect. The number of slots is not necessarily greater than the number of executors. Slots are conceptual slots within an executor where tasks are executed. The number of slots per executor depends on the cores assigned to each executor.

2: Incorrect. It's not a requirement to have more tasks than slots. Tasks are queued if there are more tasks than slots available.

3: Incorrect. The most granular level of execution in Spark is the task, not the slot. Slots are just a logical representation of resources available for task execution.

4: Incorrect. Slots are used in cluster mode. They represent the capacity of an executor to run tasks.

5: Correct. Slots are indeed resources for parallelization within a Spark application. They represent the number of tasks an executor can run in parallel, which is typically determined by the number of cores allocated to each executor.

***Additional Information:***

Understanding slots is crucial for tuning Spark applications, as they directly relate to resource allocation and task parallelism.

For more on this topic, refer to '[Learning Spark: Lightning-Fast Big Data Analysis](https://github.com/hemant-rout/BigData/blob/master/Learning%20Spark%20%20Lightning-Fast%20Big%20Data%20Analysis%20.pdf)' by Holden Karau et al.

and the Apache Spark documentation at <https://spark.apache.org/docs/latest/>.

## Question 4

Which of the following is a combination of a block of data and a set of transformers that will run on a single executor?

* **Executor**
* **Node**
* **Job**
* **Task**
* **Slot**

**Explanation**

1: Incorrect. An executor is a process in Spark that runs computations and stores data for the application but does not itself represent a combination of data block and transformers.

2: Incorrect. A node refers to a machine in a Spark cluster on which executors and drivers can run, not to a combination of data block and transformers.

3: Incorrect. A job in Spark is a parallel computation consisting of multiple tasks that gets spawned as a result of actions like collect(), count(), etc. It is a higher-level concept than what is described in the question.

4: Correct. A task in Spark is a unit of work that will be sent to one executor. Each task is a combination of data and a set of transformations to apply to that data, and it will run on a single executor.

5: Incorrect. A slot is a conceptual slot within an executor that can run one task at a time. While it is related to task execution, it does not itself represent a combination of data block and transformers.

## Question 5:

Which of the following is a group of tasks that can be executed in parallel to compute the same set of operations on potentially multiple machines?

* Job
* Slot
* Executor
* Task
* Stage

**Explanation**

1: Incorrect. A job in Spark is a collection of stages, triggered by an action like collect () or save(). It is more of a higher-level abstraction representing the entire computation.

2: Incorrect. A slot is a conceptual slot within an executor in Spark that can run one task at a time. It does not represent a group of tasks.

3: Incorrect. An executor is a process running on a worker node in Spark that executes tasks and stores data. It is not a group of tasks but rather a component that executes them.

4: Incorrect. A task is a single unit of work in Spark, and while it can be part of a group, it is not the group itself.

5: Correct. A stage in Spark is a group of tasks that can be executed in parallel. Stages are formed based on the transformations applied to RDDs and are used to optimize the execution across the cluster. Each stage can involve tasks running on multiple machines.

***Additional Information:***

Stages are an essential concept in understanding how Spark organizes computations. They are formed during the logical planning phase and are based on the wide and narrow dependencies of RDD transformations.

## Question 6

**Which of the following describes a shuffle?**

* **A shuffle is the process by which data is compared across partitions.**
* **A shuffle is the process by which data is compared across executors.**
* **A shuffle is the process by which partitions are allocated to tasks.**
* **A shuffle is the process by which partitions are ordered for write.**
* **A shuffle is the process by which tasks are ordered for execution.**

**Explanation**

1: Correct. A shuffle in Spark is a process where data is redistributed across different partitions. This typically occurs when operations like join, groupBy, or reduceByKey are executed, which require data to be **compared and aggregated across partitions.**

2: Incorrect. Shuffling is not specifically about comparing data across executors, but rather across partitions that may or may not reside on different executors.

3: Incorrect. The allocation of partitions to tasks is not a shuffle. It is part of task scheduling and execution but does not involve the redistribution of data across partitions.

4: Incorrect. The ordering of partitions for write operations is not considered a shuffle. A shuffle specifically involves redistributing data across partitions for certain types of operations.

5: Incorrect. Ordering tasks for execution is related to task scheduling and is not what defines a shuffle. Shuffling is specifically about the movement and reorganization of data across partitions.

Additional Information:

Shuffles are a critical aspect of Spark's performance, as they can be resource-intensive and affect the efficiency of data processing. Understanding when and how shuffles occur can help in optimizing Spark applications.

## Question 7:

DataFrame df is very large with a large number of partitions, more than there are executors in the cluster. Based on this situation, which of the following is incorrect? Assume there is one core per executor.

* Performance will be suboptimal because not all executors will be utilized at the same time.
* Performance will be suboptimal because not all data can be processed at the same time.
* There will be a large number of shuffle connections performed on DataFrame df when operations inducing a shuffle are called.
* There will be a lot of overhead associated with managing resources for data processing within each task.
* There might be risk of out-of-memory errors depending on the size of the executors in the cluster.

**Explanation**

1: Incorrect. When the number of partitions exceeds the number of executors, Spark can still utilize all executors effectively. Tasks will be queued and executed as resources become available, ensuring all executors can be utilized.

2: Correct, but not the answer. While not all data can be processed simultaneously due to the limit in the number of executors, this does not necessarily lead to suboptimal performance as Spark handles task scheduling efficiently.

3: Correct, but not the answer. A large number of partitions can indeed lead to numerous shuffle connections when operations that induce a shuffle are executed, potentially impacting performance.

4: Correct, but not the answer. Managing a large number of partitions can introduce overhead, particularly in the context of task scheduling and execution.

5: Correct, but not the answer. With a large number of partitions, and especially during shuffles, there is a risk of out-of-memory errors if the executors are not sized appropriately to handle the data.

*In scenarios with a large number of partitions, understanding resource management and partitioning strategies is crucial for optimal performance.*

## Question 8

Which of the following operations will **trigger** evaluation?

* **DataFrame.filter()**
* **DataFrame.distinct()**
* **DataFrame.intersect()**
* **DataFrame.join()**
* **DataFrame.count()**

**Explanation**

1: Incorrect. **DataFrame.filter()** is a transformation operation. It creates a new DataFrame representing the filtered data but does not trigger evaluation or execution until an action is called.

2: Incorrect. **DataFrame.distinct()** is also a transformation. It returns a new DataFrame with distinct rows but does not cause the actual computation to occur.

3: Incorrect. **DataFrame.intersect()** is a transformation that returns a new DataFrame with rows that exist in both DataFrames. Like other transformations, it does not trigger evaluation immediately.

4: Incorrect. **DataFrame.join()** is a transformation that combines two DataFrames based on a common key. This operation prepares the execution plan but does not execute it until an action is called.

5: Correct. **DataFrame.count()** is an **action operation**. It triggers the evaluation of any preceding transformations to return the number of rows in the DataFrame. ***Actions in Spark are operations that trigger computation and return results to the driver program.***

***Additional Information:***

Understanding the difference between transformations (which define a computation) and actions (which trigger a computation) is fundamental in Spark programming.

For further reading, refer to '[Learning Spark: Lightning-Fast Big Data Analysis](https://github.com/hemant-rout/BigData/blob/master/Learning%20Spark%20%20Lightning-Fast%20Big%20Data%20Analysis%20.pdf)' by Holden Karau et al.

and the Apache Spark documentation at https://spark.apache.org/docs/latest/.

<https://www.quora.com/What-is-best-way-to-understand-RDD-Transformations-and-actions-in-Apache-Spark>

## Question 9:

Which of the following describes the difference between transformations and actions?

* Transformations work on DataFrames/Datasets while actions are reserved for native language objects.
* There is no difference between actions and transformations.
* Actions are business logic operations that do not induce execution while transformations are execution triggers focused on returning results.
* Actions work on DataFrames/Datasets while transformations are reserved for native language objects.
* Transformations are business logic operations that do not induce execution while actions are execution triggers focused on returning results.

**Explanation**

1: Incorrect. Both transformations and actions can be applied to DataFrames/Datasets in Spark. The distinction does not lie in the types of objects they work on.

2: Incorrect. There are fundamental differences between actions and transformations in Spark's execution model.

3: Incorrect. This statement reverses the roles of transformations and actions. Actions are what trigger execution, not transformations.

4: Incorrect. Similar to option 1, this statement incorrectly categorizes the application of transformations and actions based on object types, which is not the case in Spark.

5: Correct.

**Transformations** in Spark are business logic operations that define a computation without triggering it. They are lazy and only specify the operations to be performed.

**Actions**, on the other hand, are execution triggers. When an action is called, it causes the Spark engine to execute the series of transformations defined up to that point and return results.

Understanding the distinction between transformations (lazy operations) and actions (eager operations) is key to effectively programming in Spark.

This concept is fundamental in optimizing the execution plans and improving the performance of Spark applications.

For more information, refer to the Apache Spark documentation at <https://spark.apache.org/docs/latest/programming-guide.html#transformations>

## Question 10:

**Which of the following DataFrame operations is always classified as a narrow transformation?**

* **DataFrame.sort()**
* **DataFrame.distinct()**
* **DataFrame.repartition()**
* **DataFrame.select()**
* **DataFrame.join()**

**Explanation**

1: Incorrect. DataFrame.sort() is typically a wide transformation because it may involve shuffling data across partitions to ensure global ordering.

2: Incorrect. DataFrame.distinct() is a wide transformation as it requires shuffling data to ensure all elements in the DataFrame are unique across the entire dataset.

3: Incorrect. DataFrame.repartition() is a wide transformation because it involves shuffling the data across different partitions, which usually leads to data movement across the nodes in the cluster.

4: Correct. DataFrame.select() is a narrow transformation. It does not require data to be shuffled across partitions. Each input partition will contribute to only one output partition in a select transformation.

5: Incorrect. DataFrame.join() can be a wide transformation, especially when it involves shuffling rows across different partitions to match rows based on a certain condition or key.

***Additional Information:***

Narrow transformations are those where each input partition will contribute to only one output partition. They are important for understanding the performance characteristics of Spark applications, as narrow transformations typically require less data movement than wide transformations.

<https://sparkbyexamples.com/spark/spark-rdd-transformations/>

Bottom of Form

Top of Form

## Question 11:

**Spark has a few different execution/deployment modes: cluster, client, and local. Which of the following describes Spark's execution/deployment mode?**

* Spark's execution/deployment mode determines where the driver and executors are physically located when a Spark application is run
* Spark's execution/deployment mode determines which tasks are allocated to which executors in a cluster
* Spark's execution/deployment mode determines which node in a cluster of nodes is responsible for running the driver program
* Spark's execution/deployment mode determines exactly how many nodes the driver will connect to when a Spark application is run
* Spark's execution/deployment mode determines whether results are run interactively in a notebook environment or in batch

**Explanation**

1: Correct. Spark's execution/deployment mode dictates the physical location of the driver and executors when running a Spark application. In 'cluster' mode, both driver and executors run on cluster nodes. In 'client' mode, the driver runs on the client machine, and executors run on cluster nodes. In 'local' mode, both driver and executors run in a single JVM on a single machine.

2: Incorrect. Task allocation to executors is handled by Spark's task scheduler and is not directly determined by the execution/deployment mode.

3: Incorrect. While the execution/deployment mode does influence where the driver runs, it does not specifically designate a particular node in the cluster for the driver program.

4: Incorrect. The execution/deployment mode does not directly determine the number of nodes the driver will connect to. This is influenced by other factors like the number of executors configured.

5: Incorrect. The execution/deployment mode is not directly related to whether the results are run interactively or in batch. This is more about the usage pattern rather than the deployment mode.

***Additional Information:***

Understanding Spark's deployment modes is crucial for configuring and optimizing Spark applications for different environments.

More details can be found in the Apache Spark documentation at <https://spark.apache.org/docs/latest/submitting-applications.html>

## Question 12:

Which of the following describes out-of-memory errors in Spark?

* An out-of-memory error occurs when either the driver or an executor does not have enough memory to collect or process the data allocated to it.
* An out-of-memory error occurs when Spark's storage level is too lenient and allows data objects to be cached to both memory and disk.
* An out-of-memory error occurs when there are more tasks than are executors regardless of the number of worker nodes.
* An out-of-memory error occurs when the Spark application calls too many transformations in a row without calling an action regardless of the size of the data object on which the transformations are operating.
* An out-of-memory error occurs when too much data is allocated to the driver for computational purposes.

**Explanation**

1: Correct. An out-of-memory error in Spark typically happens when the driver or one of the executors runs out of memory. This can occur due to various reasons, such as processing or collecting a large dataset that exceeds the available memory.

2: Incorrect. Storage levels in Spark (memory and disk caching) do not directly cause out-of-memory errors. These levels are mechanisms to optimize the storage of RDDs, DataFrames, or Datasets.

3: Incorrect. Having more tasks than executors does not inherently cause out-of-memory errors. Spark can queue extra tasks and execute them as resources become available.

4: Incorrect. Simply chaining transformations does not lead to out-of-memory errors, as transformations in Spark are lazy and don't consume memory until an action is executed.

5: Incorrect. While allocating too much data to the driver can cause out-of-memory errors, it's not the only scenario. Executors can also run out of memory during task execution.

***Additional Information:***

Managing memory efficiently is critical in Spark to avoid out-of-memory errors. Understanding the memory requirements of different operations and properly configuring Spark's memory management settings are key.

More details can be found in :

<https://sparkbyexamples.com/spark/spark-performance-tuning/>

<https://www.sparkcodehub.com/spark-memory-management>

## Question 13:

**Which of the following is the default storage level for persist() for a non-streaming DataFrame/Dataset?**

* + MEMORY\_AND\_DISK
  + MEMORY\_AND\_DISK\_SER
  + DISK\_ONLY
  + MEMORY\_ONLY\_SER
  + MEMORY\_ONLY

**Explanation**

1: Incorrect. MEMORY\_AND\_DISK is not the default storage level for persist() in non-streaming DataFrames/Datasets, although it is a commonly used storage level for fault tolerance and performance. 2: Incorrect. MEMORY\_AND\_DISK\_SER, which includes serialization and storage in memory and disk, is not the default setting for persist(). 3: Incorrect. DISK\_ONLY is not the default storage level. It stores RDD partitions only on disk. 4: Incorrect. MEMORY\_ONLY\_SER, which stores the data serialized in memory, is not the default for persist(). 5: Correct. The default storage level for persist() for non-streaming DataFrames/Datasets in Spark is MEMORY\_ONLY. This level stores RDD partitions in memory without serialization. If the memory is not sufficient, it will not spill to disk; instead, partitions that do not fit in memory will be recomputed as needed. ***Additional Information:***

Understanding the storage levels in Spark is important for optimizing memory usage and performance. MEMORY\_ONLY is the default for its simplicity and performance, but other levels might be more suitable depending on the application's fault tolerance and data size requirements.

For more details, see the Apache Spark documentation at :

[https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence.](https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence)

<https://stackoverflow.com/questions/44338988/does-persist-on-spark-by-default-store-to-memory-or-disk>

## Question 14:

**Which of the following describes a broadcast variable?**

* **A broadcast variable is a Spark object that needs to be partitioned onto multiple worker nodes because it's too large to fit on a single worker node.**
* **A broadcast variable can only be created by an explicit call to the broadcast() operation.**
* **A broadcast variable is entirely cached on the driver node so it doesn't need to be present on any worker nodes.**
* **A broadcast variable is entirely cached on each worker node so it doesn't need to be shipped or shuffled between nodes with each stage.**
* **A broadcast variable is saved to the disk of each worker node to be easily read into memory when needed.**

**Explanation**

1: Incorrect. A broadcast variable is not partitioned across multiple worker nodes due to its size. Instead, it is a mechanism to keep a read-only variable cached on each worker node, rather than shipping a copy of it with tasks.

2: Incorrect. While a broadcast variable is typically created using the broadcast() operation, Spark's optimizer can also automatically broadcast certain variables based on their size and usage pattern.

3: Incorrect. A broadcast variable is not just cached on the driver node. It is, in fact, distributed and cached on all worker nodes to avoid the need to send this data along with every task.

4: Correct. A broadcast variable is cached on each worker node. This avoids the need to send its copy over the network repeatedly with each task, hence reducing network I/O and improving performance in distributed computations, especially when the same data is needed by multiple tasks.

5: Incorrect. A broadcast variable is not specifically saved to disk on worker nodes. It's kept in memory for efficient access, although Spark's storage mechanism can spill data to disk if memory is insufficient.

***Additional Information:***

Broadcast variables are used to improve the efficiency of joins and other operations where a common dataset needs to be used across many Spark tasks. They reduce the network overhead involved in distributing the data.

For more information, refer to the official Apache Spark documentation at <https://spark.apache.org/docs/latest/rdd-programming-guide.html#broadcast-variables>

## Question 15:

**Which of the following operations is most likely to induce a skew in the size of your data's partitions?**

1. **DataFrame.collect()**
2. **DataFrame.cache()**
3. **DataFrame.repartition(n)**
4. **DataFrame.coalesce(n)**
5. **DataFrame.persist()**

**Explanation**

1. Incorrect. **DataFrame.collect()** is an action that retrieves all elements of the DataFrame to the driver node and does not affect the partitioning of data in the cluster.
2. Incorrect. **DataFrame.cache()** is a storage-level operation that stores the DataFrame in memory for faster access, but it doesn't change the partitioning of the data.
3. Incorrect. Although **DataFrame.repartition(n)** can lead to an even distribution of data across partitions, if not chosen carefully, it can also result in data skew. However, since it involves a shuffle, it is often used with the intent to balance data across partitions.
4. Correct. **DataFrame.coalesce(n)** reduces the number of partitions, often used to avoid a full shuffle of data. It combines existing partitions and can result in larger, unevenly sized partitions if the original partitions were already skewed in size. This can lead to skew in the size of data partitions because it does not redistribute the data evenly across the new number of partitions.
5. Incorrect. **DataFrame.persist()** is similar to **cache()** in that it saves the DataFrame at a specific storage level. This does not change the partitioning or cause data skew.

**Additional Information:**

*Understanding how different operations affect the distribution of data across partitions is crucial for optimizing Spark applications. Repartitioning, in particular, should be used judiciously as it can lead to performance issues if it causes significant data skew.*

For more detailed explanations and examples, you can refer to these articles:

* [Spark By Examples](https://sparkbyexamples.com/spark/spark-repartition-vs-coalesce/)
* [Nixon Data](https://nixondata.com/knowledge/apache-spark-fundamentals/understanding-the-differences-between-repartition-and-coalesce/)
* [Rock the JVM Blog](https://blog.rockthejvm.com/repartition-coalesce/)

## Question 16:

**Which of the following data structures are Spark DataFrames built on top of?**

* **Arrays**
* **Strings**
* **RDDs**
* **Vectors**
* **SQL Tables**

**Explanation**

1: Incorrect. While arrays can be used within Spark, they are not the foundational data structure for DataFrames. DataFrames are more structured and complex than simple arrays.

2: Incorrect. Strings are a data type that can be stored in DataFrames, but they are not the underlying structure on which DataFrames are built.

3: Correct. Spark DataFrames are built on top of Resilient Distributed Datasets (RDDs). RDDs are the fundamental data structure in Spark, and DataFrames provide a higher-level abstraction over RDDs, offering a more convenient and powerful interface for data manipulation and analysis.

4: Incorrect. Vectors are used in Spark for machine learning libraries and for representing features, but they are not the foundation for DataFrames.

5: Incorrect. Although SQL tables can be represented as DataFrames in Spark, they are not the underlying data structure. DataFrames can be queried using SQL, but they are fundamentally built on top of RDDs.

***Additional Information:***

Understanding the relationship between RDDs and DataFrames is key to grasping Spark's data processing capabilities. RDDs provide the distributed, fault-tolerant backbone, while DataFrames offer a more efficient and user-friendly layer on top for handling structured data.

For more details, see the official Apache Spark documentation at <https://spark.apache.org/docs/latest/sql-programming-guide.html>

## Question 17:

**Which of the following code blocks returns a DataFrame containing only column storeId and column division from DataFrame storesDF?**

* **storesDF.select("storeId").select("division")**
* **storesDF.select(storeId, division)**
* **storesDF.select("storeId", "division")**
* **storesDF.select(col("storeId", "division"))**
* **storesDF.select(storeId).select(division)**

**Explanation**

1: Incorrect. This code block selects 'storeId' and then 'division' in two separate steps, effectively resulting in a DataFrame containing only 'division', as the second select overwrites the first.

2: Incorrect. While this looks like a valid syntax, it lacks quotes around 'storeId' and 'division', which are required to specify them as column names in string format.

3: Correct. storesDF.select("storeId", "division") correctly selects both 'storeId' and 'division' columns in a single operation, returning a DataFrame containing these two columns.

4: Incorrect. The col() function does not accept multiple column names in a single call like this. This syntax will result in an error.

5: Incorrect. Similar to option 1, this code block selects 'storeId' and then 'division' in two separate steps, ultimately resulting in a DataFrame with only 'division'.

***Additional Information:***

Selecting multiple columns in Spark can be efficiently done using the select() method with multiple column names (as strings) passed as arguments. It's important to use the correct syntax to ensure the desired columns are included in the resulting DataFrame.

More details on DataFrame operations can be found in the Apache Spark documentation at

<https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.select.html>

## Question 18:

The below code shown block contains an error. The code block is intended to return a DataFrame containing only the rows from DataFrame storesDF where the value in DataFrame storesDF's 'sqft' column is less than or equal to 25,000. Assume DataFrame storesDF is the only defined language variable. Identify the error. *Code block:*

storesDF.filter(sqft <= 25000)

* **The column name sqft needs to be quoted like storesDF.filter("sqft" <= 25000).**
* **The column name sqft needs to be quoted and wrapped in the col() function like storesDF.filter(col("sqft") <= 25000).**
* **The sign in the logical condition inside filter() needs to be changed from <= to >.**
* **The sign in the logical condition inside filter() needs to be changed from <= to >=.**
* **The column name sqft needs to be wrapped in the col() function like storesDF.filter(col(sqft) <= 25000).**

**Explanation**

1: Incorrect. Simply quoting 'sqft' will not resolve the issue. In Spark DataFrame operations, column references must be made using a column object, not just a string name.

2: Correct. The error in the code block is that the column 'sqft' needs to be referenced as a column object. This is done by wrapping the column name 'sqft' in the col() function, like storesDF.filter(col("sqft") <= 25000). This syntax is necessary for the filter condition to be properly applied to the DataFrame.

3: Incorrect. Changing the sign from <= to > would change the logic of the filter and does not address the syntax error in the column reference.

4: Incorrect. Changing the sign from <= to >= would not fix the error and would alter the intended logic of the filter condition.

5: Incorrect. While using the col() function is correct, the column name 'sqft' needs to be quoted within the col() function. Without quotes, it would be treated as a variable name, which is not defined in this context.

***Additional Information:***

Correct syntax and references are crucial in DataFrame operations in Spark. The col() function is used to refer to DataFrame columns in a programmatic way, especially within transformations and actions.

More information about DataFrame transformations can be found in the Apache Spark documentation at

<https://sparkbyexamples.com/pyspark/pyspark-where-filter/>

## Question 19:

The code block shown below should return a DataFrame containing only the rows from DataFrame storesDF where the value in column sqft is less than or equal to 25,000 OR the value in column customerSatisfaction is greater than or equal to 30.

Choose the response that correctly fills in the numbered blanks within the code block to complete this task. *Code block:*

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

* filter

(col("sqft") <= 25000)

|

(col("customerSatisfaction") >= 30)

drop

(col(sqft) <= 25000)

|

(col(customerSatisfaction) >= 30)

filter

col("sqft") <= 25000

|

col("customerSatisfaction") >= 30

filter

col("sqft") <= 25000

or

col("customerSatisfaction") >= 30

filter

(col("sqft") <= 25000)

or

(col("customerSatisfaction") >= 30)

**Explanation**

1: Correct. This option correctly uses the filter() function with the logical OR operator ('|') to combine the two conditions. Each condition is properly wrapped in parentheses and uses the col() function for column reference.

2: Incorrect. The drop() function is not used for filtering based on conditions. Also, the column names need to be in quotes when used with col().

3: Incorrect. While using filter() is correct, the conditions need to be enclosed in parentheses to ensure proper evaluation order. The '|' symbol is used for logical OR in Spark DataFrame API.

4: Incorrect. The word 'or' is not used for logical OR in Spark DataFrame API; the symbol '|' is used instead. Also, the conditions need to be enclosed in parentheses.

5: Incorrect. Although it uses filter() and encloses conditions in parentheses, the word 'or' is not the correct way to specify logical OR in Spark DataFrame API; '|' should be used instead.

***Additional Information:***

Using the correct syntax for logical operations in DataFrame transformations is crucial. The filter() function along with the col() function and proper logical operators allow for complex condition-based filtering in Spark.

## Question 20:

Which of the following operations can be used to convert a DataFrame column from one type to another type?

* **col().cast()**
* **convert()**
* **castAs()**
* **col().coerce()**
* **col()**

**Explanation**

1: Correct. The col().cast() operation is used in Spark to change the data type of a column. The col() function is used to reference a column, and the cast() method is chained to specify the new data type.

2: Incorrect. There is no convert() function in Spark's DataFrame API for data type conversion.

3: Incorrect. castAs() is not a recognized function in Spark for type conversion of DataFrame columns.

4: Incorrect. col().coerce() is not a valid operation in Spark for converting data types of DataFrame columns.

5: Incorrect. While col() is used to reference a column in Spark, it does not by itself provide a way to change the column's data type. The cast() method needs to be used in conjunction with col() for this purpose.

Additional Information:

Type conversion is a common operation in data processing, and Spark provides a flexible way to handle such conversions through the cast() method.

This can be particularly useful when dealing with data from different sources or when preparing data for analysis. More information on column operations can be found in the Apache Spark documentation at

<https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.Column.cast.html>

<https://sparkbyexamples.com/pyspark/pyspark-cast-column-type/>

<https://www.statology.org/pyspark-cast-multiple-columns/>

## Question 21:

**Which of the following code blocks returns a new DataFrame with a new column sqft100 that is 1/100th of column sqft in DataFrame storesDF? Note that column sqft100 is not in the original DataFrame storesDF.**

* **storesDF.withColumn("sqft100", col("sqft") \* 100)**
* **storesDF.withColumn("sqft100", sqft / 100)**
* **storesDF.withColumn(col("sqft100"), col("sqft") / 100)**
* **storesDF.withColumn("sqft100", col("sqft") / 100)**
* **storesDF.newColumn("sqft100", sqft / 100)**

**Explanation**

1: Incorrect. This code multiplies the 'sqft' column by 100, instead of dividing it by 100 to create the 'sqft100' column.

2: Incorrect. This syntax is not correct in Spark DataFrame API because 'sqft' needs to be referenced as a column object using the col() function.

3: Incorrect. The first argument in withColumn() should be a string representing the new column name, not a column object.

4: Correct. storesDF.withColumn("sqft100", col("sqft") / 100) correctly creates a new column 'sqft100' by dividing the 'sqft' column by 100. This is the proper use of withColumn() with the col() function in Spark.

5: Incorrect. There is no newColumn() function in Spark's DataFrame API. The correct method to add a new column is withColumn().

***Additional Information:***

The withColumn() function in Spark is a powerful tool for adding new columns or modifying existing ones in a DataFrame. It is commonly used for data manipulation and transformation tasks. For more detailed usage, refer to the Apache Spark documentation at

<https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.DataFrame.withColumn.html>

<https://sparkbyexamples.com/pyspark/pyspark-withcolumn/>

## Question 22:

Which of the following code blocks returns a new DataFrame from DataFrame storesDF where column numberOfManagers is the constant integer 1?

* storesDF.withColumn("numberOfManagers", col(1))
* storesDF.withColumn("numberOfManagers", 1)
* storesDF.withColumn("numberOfManagers", lit(1))
* storesDF.withColumn("numberOfManagers", lit("1"))
* storesDF.withColumn("numberOfManagers", IntegerType(1))

**Explanation**

1: Incorrect. The col() function is used to reference existing columns, not to create constant values.

2: Incorrect. Simply passing 1 as a value does not work in the withColumn() method in Spark's DataFrame API. The value needs to be wrapped in a function that creates a column expression.

3: Correct. The lit() function is used in Spark to create a literal column, which is a column with a constant value. storesDF.withColumn("numberOfManagers", lit(1)) correctly creates a new column with the constant integer value 1.

4: Incorrect. Using lit("1") creates a literal column with a string value of '1', not an integer.

5: Incorrect. IntegerType(1) is not a valid syntax in Spark for creating a literal value. The lit() function is the appropriate method to use for this purpose.

***Additional Information:***

The lit() function is particularly useful for adding a column with a constant value to a DataFrame, which can be important for various data manipulation and transformation tasks.

For more details on DataFrame transformations, refer to the Apache Spark documentation at

<https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.functions.lit.html>

<https://sparkbyexamples.com/pyspark/pyspark-lit-add-literal-constant/>

## Question 23:

Which of the following operations can be used to split an array column into an individual DataFrame row for each element in the array?

* **extract()**
* **split()**
* explode()
* **arrays\_zip()**
* **unpack()**

**Explanation**

1: Incorrect. extract() is not a function in Spark's DataFrame API used for splitting an array column into rows.

2: Incorrect. split() is used in Spark for splitting a string into an array based on a delimiter, not for splitting an array column into multiple rows.

3: Correct. explode() is the function used in Spark to transform each element of an array column into a separate row. It 'explodes' the array into multiple rows, each containing one element from the array.

4: Incorrect. arrays\_zip() is used to merge multiple arrays into a single array of structs, not for splitting an array column into rows.

5: Incorrect. unpack() is not a recognized function in Spark for this purpose.

Additional Information:

The explode() function is useful in scenarios where you need to flatten an array and analyze or manipulate each element individually as separate rows in a DataFrame. This operation is commonly used in data transformation processes.

More details on this function can be found in the Apache Spark documentation at

<https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.functions.explode.html>

<https://sparkbyexamples.com/pyspark/pyspark-explode-array-and-map-columns-to-rows/>

## Question 24:

Which of the following code blocks returns a new DataFrame where column storeCategory is an all-lowercase version of column storeCategory in DataFrame storesDF?

Assume DataFrame storesDF is the only defined language variable.

* **storesDF.withColumn("storeCategory", lower(col("storeCategory")))**
* **storesDF.withColumn("storeCategory", col("storeCategory").lower())**
* **storesDF.withColumn("storeCategory", tolower(col("storeCategory")))**
* **storesDF.withColumn("storeCategory", lower("storeCategory"))**
* **storesDF.withColumn("storeCategory", lower(storeCategory))**

**Explanation**

1: Correct. The lower function is used to convert a string to lowercase in Spark, and it needs to be applied to a column object. storesDF.withColumn("storeCategory", lower(col("storeCategory"))) correctly applies the lower function to the 'storeCategory' column.

2: Incorrect. The syntax col("storeCategory").lower() is not valid in Spark. The lower function should be applied directly to the column object as an argument.

3: Incorrect. The function to convert strings to lowercase in Spark is lower, not tolower.

4: Incorrect. lower("storeCategory") will interpret "storeCategory" as a string literal, not as a reference to a column. The column needs to be referenced using the col function.

5: Incorrect. lower(storeCategory) will look for a variable named storeCategory, which is not defined. The column should be referenced as a string within the col function.

Additional Information:

The lower function is part of Spark's SQL functions library and is commonly used for string manipulation within DataFrames. It's important to use the correct syntax when applying these functions to DataFrame columns. More information can be found in the Apache Spark documentation at

<https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.functions.lower.html>

<https://www.skytowner.com/explore/pyspark_sql_functions_lower_method>

## Question 25:

The code block shown below contains an error. The code block is intended to return a new DataFrame where column division from DataFrame storesDF has been renamed to column state and column managerName from DataFrame storesDF has been renamed to column managerFullName. Identify the error. Code block:

(storesDF.withColumnRenamed("state", "division").withColumnRenamed("managerFullName", "managerName"))

* **Both arguments to operation withColumnRenamed() should be wrapped in the col() operation.**
* **The operations withColumnRenamed() should not be called twice, and the first argument should be ["state", "division"] and the second argument should be ["managerFullName", "managerName"].**
* **The old columns need to be explicitly dropped.**
* **The first argument to operation withColumnRenamed() should be the old column name and the second argument should be the new column name.**
* **The operation withColumnRenamed() should be replaced with withColumn().**

**Explanation**

1: Incorrect. The withColumnRenamed() operation does not require column references to be wrapped in the col() function. It takes simple string arguments.

2: Incorrect. It is valid to call withColumnRenamed() multiple times for different columns. The method does not support renaming multiple columns in a single call with array arguments.

3: Incorrect. The withColumnRenamed() operation automatically replaces the old column with the new one. Explicitly dropping the old column is not necessary.

4: Correct. The error in the code block is that the arguments to withColumnRenamed() are in the wrong order. The first argument should be the current (old) column name, and the second argument should be the new column name. The correct code should be (storesDF.withColumnRenamed("division", "state").withColumnRenamed("managerName", "managerFullName")).

5: Incorrect. The withColumnRenamed() operation is the correct method for renaming columns. withColumn() is used for different purposes, such as adding new columns or modifying existing ones.

Additional Information: Renaming columns in a DataFrame is a common operation in data manipulation. The withColumnRenamed() method provides a straightforward way to accomplish this, taking the existing column name and the new name as its arguments.

For more details, refer to the Apache Spark documentation at

<https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.withColumnRenamed.html>

<https://sparkbyexamples.com/pyspark/pyspark-rename-dataframe-column/>

## Question 26:

Which of the following code blocks returns a DataFrame where rows in DataFrame storesDF containing missing values in every column have been dropped?

* **storesDF.nadrop("all")**
* **storesDF.na.drop("all", subset = "sqft")**
* **storesDF.dropna()**
* **storesDF.na.drop()**
* **storesDF.na.drop("all")**

**Explanation**

1: Incorrect. There is a syntax error in this option. The correct function is na.drop(), not nadrop().

2: Incorrect. This code block will drop rows with missing values in the 'sqft' column only, not in every column.

3: Incorrect. storesDF.dropna() without any parameters drops rows with missing values in any column, not rows with missing values in every column.

4: Incorrect. storesDF.na.drop() behaves the same as storesDF.dropna(), dropping rows with any missing values, not necessarily in every column.

5: Correct. storesDF.na.drop("all") will drop rows where all columns have missing values. This is the correct way to target only those rows where every column contains a missing value.

Additional Information:

Handling missing data is an important aspect of data processing. The na.drop() function in Spark provides flexibility to specify how missing values should be handled, whether it's dropping rows with any missing values or only those where all values are missing. More information on handling missing data can be found in the Apache Spark documentation at

<https://spark.apache.org/docs/3.1.2/api/python/reference/api/pyspark.sql.DataFrameNaFunctions.drop.html>

<https://sparkbyexamples.com/pyspark/pyspark-drop-rows-with-null-values/>

Bottom of Form

Top of Form

Question 27:

**Which of the following operations fails to return a DataFrame where every row is unique?**

* DataFrame.distinct()
* DataFrame.drop\_duplicates(subset = None)
* DataFrame.drop\_duplicates()
* DataFrame.dropDuplicates()
* DataFrame.drop\_duplicates(subset = "all")

**Explanation**

1: Incorrect. DataFrame.distinct() returns a new DataFrame with distinct rows, effectively removing duplicates and ensuring every row is unique.

2: Incorrect. DataFrame.drop\_duplicates(subset = None) removes duplicate rows across all columns, which is the default behavior, and thus returns a DataFrame with unique rows.

3: Incorrect. DataFrame.drop\_duplicates(), without any subset specified, behaves the same as DataFrame.drop\_duplicates(subset = None), removing duplicates based on all columns.

4: Incorrect. DataFrame.dropDuplicates() is an alternative method to drop\_duplicates() in Spark and also removes duplicate rows across all columns.

5: Correct. DataFrame.drop\_duplicates(subset = "all") fails to return a DataFrame with unique rows because there is no valid 'all' keyword for the subset parameter in the drop\_duplicates() method. The subset parameter should be a column name or a list of column names, and 'all' is not recognized as such.

Additional Information:

Removing duplicates is a common operation in data processing to ensure data quality. Spark provides multiple ways to remove duplicate rows, but it is important to use the correct syntax and understand the behavior of these methods.

For more information on DataFrame operations, refer to the Apache Spark documentation at

<https://spark.apache.org/docs/3.1.2/api/python/reference/api/pyspark.sql.DataFrame.dropDuplicates.html>

<https://api-docs.databricks.com/python/pyspark/latest/pyspark.pandas/api/pyspark.pandas.DataFrame.drop_duplicates.html>

Bottom of Form

Top of Form

Question 28:

**Which of the following code blocks will not always return the exact number of distinct values in column division?**

* storesDF.agg(approx\_count\_distinct(col("division")).alias("divisionDistinct"))
* storesDF.agg(approx\_count\_distinct(col("division"), 0).alias("divisionDistinct"))
* storesDF.agg(countDistinct(col("division")).alias("divisionDistinct"))
* storesDF.select("division").dropDuplicates().count()
* storesDF.select("division").distinct().count()

**Explanation**

1: Correct. The function approx\_count\_distinct() returns an approximate count of distinct items. It's useful for performance optimization with large datasets, but it does not guarantee an exact count, which is required in this question.

2: Incorrect. Despite using approx\_count\_distinct(), setting the accuracy parameter to 0 makes it functionally equivalent to countDistinct(), providing an exact count of distinct values.

3: Incorrect. countDistinct() computes the number of distinct items in a column, returning the exact count of distinct values.

4: Incorrect. The combination of select(), dropDuplicates(), and count() will accurately count the number of distinct values in the 'division' column.

5: Incorrect. select("division").distinct().count() is another valid approach to counting the exact number of distinct values in the 'division' column.

Additional Information:

When working with large datasets, approx\_count\_distinct() can be used to improve performance at the cost of exact accuracy. For exact counts, countDistinct() or distinct().count() are the appropriate methods. Details about these functions can be found in the Apache Spark documentation at

<https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/>

## Question 29:

The code block shown below should return a new DataFrame with the mean of column sqft from DataFrame storesDF in column sqftMean. Choose the response that correctly fills in the numbered blanks within the code block to complete this task. Code block:

storesDF.\_\_1\_\_(\_\_2\_\_(\_\_3\_\_).alias("sqftMean"))

agg

mean

col("sqft")

mean

col

"sqft"

withColumn

mean

col("sqft")

agg

mean

"sqft"

agg

average

col("sqft")

**Explanation**

1: Correct. The agg() function is used for aggregations in Spark, and mean() is the function for calculating the average. The col() function is correctly used to reference the 'sqft' column. The complete statement correctly performs the aggregation and aliases the result as 'sqftMean'.

2: Incorrect. mean as a standalone function is not the correct syntax for aggregations in Spark's DataFrame API.

3: Incorrect. withColumn is used for adding or replacing a column, not for aggregations like calculating the mean.

4: Incorrect. While agg() and mean() are correct, mean() requires a column object, not a string. col("sqft") should be used instead of "sqft".

5: Incorrect. The function for calculating the mean in Spark is mean(), not average(), although they might appear synonymous. col("sqft") is correctly used to reference the column.

***Additional Information:***

The agg() function along with mean() provides a way to compute aggregate statistics like the average across a DataFrame. This is a common operation in data analysis tasks. More details about aggregation functions can be found in the Apache Spark documentation at <https://sparkbyexamples.com/pyspark/pyspark-groupby-agg-aggregate-explained/>

## Question 30:

**Which of the following code blocks returns the number of rows in DataFrame storesDF?**

* **storesDF.withColumn("numberOfRows", count())**
* **storesDF.withColumn(count().alias("numberOfRows"))**
* **storesDF.countDistinct()**
* **storesDF.count()**
* **storesDF.agg(count())**

**Explanation**

1: Incorrect. storesDF.withColumn("numberOfRows", count()) is syntactically incorrect because count() cannot be used directly inside withColumn() like this, and it does not return the total number of rows.

2: Incorrect. storesDF.withColumn(count().alias("numberOfRows")) is syntactically incorrect. The withColumn method is used for adding or modifying columns, not for counting rows in a DataFrame.

3: Incorrect. storesDF.countDistinct() would be used to count distinct rows based on all columns, not the total number of rows in the DataFrame.

4: Correct. storesDF.count() is the correct method to get the number of rows in the DataFrame. It returns the total row count.

5: Incorrect. storesDF.agg(count()) is used for aggregation operations and is not the direct way to count the total number of rows in the DataFrame.

***Additional Information:***

Counting the number of rows in a DataFrame is a common operation, and Spark provides the count() method specifically for this purpose. It's a straightforward and efficient way to determine the size of the DataFrame.

More details about DataFrame operations can be found in the Apache Spark documentation at <https://sparkbyexamples.com/pyspark/pyspark-count/>

## Question 31:

Which of the following code blocks returns the sum of the values in column sqft in DataFrame storesDF grouped by distinct value in column division?

* storesDF.groupBy.agg(sum(col("sqft")))
* storesDF.groupBy("division").agg(sum())
* storesDF.agg(groupBy("division").sum(col("sqft")))
* storesDF.groupby.agg(sum(col("sqft")))
* storesDF.groupBy("division").agg(sum(col("sqft")))

**Explanation**

1: Incorrect. The syntax storesDF.groupBy.agg(sum(col("sqft"))) is incorrect. groupBy should be followed by a column name or a list of column names.

2: Incorrect. storesDF.groupBy("division").agg(sum()) is syntactically incorrect because the sum() function requires a column argument.

3: Incorrect. storesDF.agg(groupBy("division").sum(col("sqft"))) is not the correct syntax in Spark. The groupBy operation should precede the agg operation.

4: Incorrect. The correct method is groupBy(), not groupby(). Additionally, the syntax used in this option is incorrect.

5: Correct. storesDF.groupBy("division").agg(sum(col("sqft"))) correctly groups the DataFrame by the 'division' column and then aggregates the 'sqft' column by summing its values within each group. This is the proper usage of groupBy and agg methods in Spark.

***Additional Information:***

Grouping and aggregating data are fundamental operations in data analysis. Spark's groupBy and agg methods provide powerful capabilities to perform such operations efficiently.

Bottom of Form

Top of Form

## Question 32:

**Which of the following code blocks returns a DataFrame containing summary statistics only for column sqft in DataFrame storesDF?**

* storesDF.summary("mean")
* storesDF.describe("sqft")
* storesDF.summary(col("sqft"))
* storesDF.describeColumn("sqft")
* storesDF.summary()

**Explanation**

1: Incorrect. storesDF.summary("mean") would provide summary statistics, but it would not be limited only to the 'sqft' column. Additionally, 'mean' alone would not generate a full summary.

2: Correct. storesDF.describe("sqft") generates summary statistics (count, mean, stddev, min, and max) specifically for the 'sqft' column.

3: Incorrect. The syntax storesDF.summary(col("sqft")) is not valid in Spark. The summary() method does not accept column objects as arguments.

4: Incorrect. There is no method named describeColumn in Spark's DataFrame API.

5: Incorrect. storesDF.summary() provides summary statistics, but it would include all columns, not just 'sqft'.

Additional Information:

The describe() function in Spark is commonly used for generating descriptive statistics for specified columns in a DataFrame. It's a helpful tool for initial data analysis and understanding the distribution of data. For more details, refer to <https://stackoverflow.com/questions/55938112/describe-a-dataframe-on-pyspark>

## Question 33:

**Which of the following operations can be used to sort the rows of a DataFrame?**

* **sort() and orderBy()**
* **orderby()**
* **sort() and orderby()**
* **orderBy()**
* **sort()**

**Explanation**

1: Correct. Both sort() and orderBy() functions can be used in Spark to sort the rows of a DataFrame. They are essentially aliases of each other and can be used interchangeably to perform sorting operations.

2: Incorrect. orderby() is not a valid function in Spark. The correct function is orderBy().

3: Incorrect. While sort() is a valid function, orderby() is not correct. The valid function is orderBy().

4: Incorrect. Although orderBy() is a correct function, it is not the only function that can be used for sorting. sort() can also be used.

5: Incorrect. While sort() is a valid function, it's not the only one that can be used for sorting. orderBy() is also a valid option.

***Additional Information:***

Sorting data is a common operation in data processing, and Spark provides flexible sorting capabilities through both the sort() and orderBy() functions.

These functions allow for sorting based on one or more columns, in ascending or descending order.

More information on sorting operations can be found in the Apache Spark documentation at <https://sparkbyexamples.com/pyspark/pyspark-orderby-and-sort-explained/>

## Question 34:

The code block shown below contains an error. The code block is intended to return a 15 percent sample of rows from DataFrame storesDF without replacement. Identify the error. Code block: storesDF.sample(True, fraction = 0.15)

* **There is no argument specified to the seed parameter.**
* **There is no argument specified to the withReplacement parameter.**
* **The sample() operation does not sample without replacement â€” sampleby() should be used instead.**
* **The sample() operation is not reproducible.**
* **The first argument True sets the sampling to be with replacement.**

**Explanation**

1: Incorrect. The absence of a seed argument does not cause an error in functionality. Specifying a seed is optional and is used for reproducibility of the sample.

2: Incorrect. The first argument in the sample() method actually represents the 'withReplacement' parameter. The issue is with the value of this parameter, not its absence.

3: Incorrect. The sample() method in Spark can be used for sampling both with and without replacement. The choice of method depends on the arguments provided, not the sampling type.

4: Incorrect. The reproducibility of the sample operation isn't the primary issue here. Reproducibility can be achieved by providing a seed value, but this is not mandatory.

5: Correct. The error in the code block is that the first argument is set to True, which implies sampling with replacement. For sampling without replacement, this argument should be False.

Additional Information:

The sample() function in Spark is used for random sampling and allows for both replacement and non-replacement sampling, controlled by the 'withReplacement' parameter.

Understanding the correct usage of parameters in this function is essential for achieving the desired sampling behavior.

## Question 35:

**Which of the following operations can be used to return the top n rows from a DataFrame?**

* **DataFrame.n()**
* **DataFrame.take(n)**
* **DataFrame.head**
* **DataFrame.show(n)**
* **DataFrame.collect(n)**

**Explanation**

1: Incorrect. DataFrame.n() is not a valid function in Spark's DataFrame API for retrieving rows.

2: Correct. DataFrame.take(n) is used to return the first n rows of a DataFrame as a list. This function is ideal for getting a specified number of rows.

3: Incorrect. DataFrame.head without an argument returns the first row of the DataFrame, and with an argument n, it returns the first n rows, but it is not labeled as such in the options.

4: Incorrect. DataFrame.show(n) displays the first n rows of the DataFrame, but it does not return them. It is used for printing the rows to the console.

5: Incorrect. DataFrame.collect(n) is not a valid method in Spark. collect() is used to retrieve all rows of a DataFrame and bring them to the driver node, which can be inefficient for large DataFrames and does not specifically retrieve the top n rows.

Additional Information:

Retrieving a specific number of rows from a DataFrame is a common operation for data inspection and analysis. The take(n) function is useful for this purpose, providing a simple way to retrieve a defined number of rows. For more details, refer to the Apache Spark documentation at <https://sparkbyexamples.com/spark/show-top-n-rows-in-spark-pyspark/>

## Question 36:

The code block shown below should extract the value for column sqft from the first row of DataFrame storesDF. Choose the response that correctly fills in the numbered blanks within the code block to complete this task. Code block: \_\_1\_\_.\_\_2\_\_.\_\_3\_\_

* **storesDF 2. first 3. col("sqft")**
* **storesDF 2. first 3. sqft**
* **storesDF 2. first 3. ["sqft"]**
* storesDF 2. first() 3. sqft
* **storesDF 2. first() 3. col("sqft")**

**Explanation**

1: Incorrect. storesDF.first followed by col("sqft") is not a valid syntax in Spark. The first() method returns a Row object, and you cannot directly apply col() on a Row.

2: Incorrect. storesDF.first followed by sqft is not valid because first is a method and should be followed by parentheses.

3: Incorrect. storesDF.first followed by ["sqft"] is not valid. The first() method returns a Row object, and a list of column names cannot be applied directly to a Row.

4: Correct. storesDF.first() returns the first Row object of the DataFrame. Accessing sqft attribute of this Row object (e.g., storesDF.first().sqft) retrieves the value of the 'sqft' column from the first row.

5: Incorrect. storesDF.first() followed by col("sqft") is not valid. The first() method returns a Row object, and col() cannot be used to extract a value from a Row.

Additional Information:

The first() method in Spark returns the first row of the DataFrame as a Row object. To access a specific column's value from this Row object, you simply use dot notation with the column name. This is an efficient way to quickly inspect values from a specific row.

## Question 37:

**Which of the following lines of code prints the schema of a DataFrame?**

* **print(storesDF)**
* **storesDF.schema**
* **print(storesDF.schema())**
* **DataFrame.printSchema()**
* **DataFrame.schema()**

**Explanation**

1: Incorrect. print(storesDF) prints the contents of the DataFrame, not its schema.

2: Correct. storesDF.schema displays the schema of the DataFrame storesDF. This property of a DataFrame object in Spark reveals the structure of the DataFrame, including column names, data types, and nullable properties.

3: Incorrect. storesDF.schema() is incorrect syntax because schema is a property, not a method, and therefore should not be followed by parentheses.

4: Incorrect. DataFrame.printSchema() is not valid syntax. printSchema is a method of DataFrame instances, not of the DataFrame class itself. It should be used as storesDF.printSchema().

5: Incorrect. DataFrame.schema() is not valid syntax. As mentioned, schema is a property, not a method. It should be accessed without parentheses.

Additional Information:

Understanding the structure of a DataFrame is crucial in data processing. The schema property provides a quick way to inspect the DataFrame's structure, which is especially helpful when dealing with complex data sources. More details about DataFrame operations can be found in <https://www.geeksforgeeks.org/how-to-check-the-schema-of-pyspark-dataframe/>

Bottom of Form

Top of Form

Question 38:

In what order should the below lines of code be run in order to create and register a SQL UDF named 'ASSESS\_PERFORMANCE' using the Python function assessPerformance and apply it to column customerSatisfaction in table stores? *Lines of code:*

1. spark.udf.register("ASSESS\_PERFORMANCE", assessPerformance)

2. spark.sql("SELECT customerSatisfaction, assessPerformance(customerSatisfaction) AS result FROM stores")

3. spark.udf.register(assessPerformance, "ASSESS\_PERFORMANCE")

4. spark.sql("SELECT customerSatisfaction, ASSESS\_PERFORMANCE(customerSatisfaction) AS result FROM stores")

* **3, 4**
* **1, 4**
* **3, 2**
* **2**
* **1, 2**

**Explanation**

* Option 1 (3,4): Incorrect. This option suggests that the UDF should be registered with the incorrect syntax, using the function name first (**assessPerformance**) followed by the UDF name in quotes (**"ASSESS\_PERFORMANCE"**). This is not the correct order for the **spark.udf.register** method. The correct order is the UDF name followed by the function name. Line 4 correctly applies the UDF in the SQL query, but it must come after the UDF has been properly registered.
* Option 2 (1,4): Correct. This is the correct sequence. The UDF is first registered with **spark.udf.register("ASSESS\_PERFORMANCE", assessPerformance)** (line 1), correctly associating the UDF name 'ASSESS\_PERFORMANCE' with the Python function **assessPerformance**. Then, the UDF is applied to the column **customerSatisfaction** in the SQL query using **spark.sql("SELECT customerSatisfaction, ASSESS\_PERFORMANCE(customerSatisfaction) AS result FROM stores")** (line 4).
* Option 3 (3,2): Incorrect. Line 3 incorrectly tries to register the UDF with the parameters reversed. The correct syntax for UDF registration is **spark.udf.register("UDF\_NAME", python\_function)**. Line 2 is not the correct way to apply the UDF and is not part of the correct answer.
* Option 4 (2): Incorrect. Line 2 alone is not sufficient because it assumes the UDF is already registered and used in a SQL query, which is not shown. The UDF must be registered before it is used in a SQL query.
* Option 5 (1,2): Incorrect. Although line 1 correctly registers the UDF, line 2 does not correctly apply the UDF in a SQL query. The proper sequence is to register the UDF first and then use it in a SQL query with the correct syntax, as shown in line 4.

**Additional Information:**

When using UDFs in Spark SQL, it is crucial to register the UDF before attempting to use it in a SQL query. The registration is done using **spark.udf.register**, with the first argument being the UDF name as a string and the second argument being the Python function that implements the UDF logic. Once registered, the UDF can be invoked in SQL queries using its name, just like any built-in SQL function. This integration of custom functions into Spark SQL enables more flexible data processing within the Spark framework.

## Question 39:

In what order should the below lines of code be run in order to create a Python UDF assessPerformanceUDF() using the integer-returning Python function assessPerformance and apply it to column customerSatisfaction in DataFrame storesDF? Lines of code:

assessPerformanceUDF = udf(assessPerformance, IntegerType)

assessPerformanceUDF = spark.register.udf("ASSESS\_PERFORMANCE", assessPerformance)

assessPerformanceUDF = udf(assessPerformance, IntegerType())

storesDF.withColumn("result", assessPerformanceUDF(col("customerSatisfaction")))

storesDF.withColumn("result", assessPerformance(col("customerSatisfaction")))

storesDF.withColumn("result", ASSESS\_PERFORMANCE(col("customerSatisfaction")))

* **3, 4**
* **2, 6**
* **3, 5**
* **1, 4**
* **2, 5**

**Explanation**

1: Correct. Line 3 correctly creates the Python UDF assessPerformanceUDF with the specified return type (IntegerType). Line 4 then applies this UDF to the 'customerSatisfaction' column in the DataFrame storesDF.

2: Incorrect. Line 2 is not the correct syntax for registering a UDF in Spark's DataFrame API. Line 6 also uses an incorrect syntax for applying a UDF in DataFrame API.

3: Incorrect. Line 3 is correct for defining the UDF, but line 5 applies the original Python function directly, not the UDF.

4: Incorrect. Line 1 uses an incorrect syntax for specifying the return type of the UDF. IntegerType should be instantiated (IntegerType()). Line 4 correctly applies the UDF to the DataFrame.

5: Incorrect. Line 2 does not correctly register a UDF in the DataFrame API. Line 5 again incorrectly applies the original Python function instead of the UDF.

Additional Information:

Creating a UDF in PySpark involves defining the UDF with the appropriate return type and then applying it to a DataFrame. The udf function is used for this purpose, and it is crucial to correctly specify the return type (e.g., IntegerType()). Once defined, the UDF can be applied to the DataFrame using withColumn.

More details can be found in the Apache Spark documentation at <https://sparkbyexamples.com/pyspark/pyspark-udf-user-defined-function/>

## Question 40:

**Which of the following operations can execute a SQL query on a table?**

* **spark.query()**
* **DataFrame.sql()**
* **spark.sql()**
* **DataFrame.createOrReplaceTempView()**
* **DataFrame.createTempView()**

**Explanation**

1: Incorrect. spark.query() is not a valid method in Spark for executing SQL queries.

2: Incorrect. DataFrame.sql() is not a valid method in Spark DataFrame API. SQL queries are executed through the SparkSession, not directly on a DataFrame.

3: Correct. spark.sql() is the method used in Spark to execute SQL queries. It allows you to run SQL queries against tables and views registered in Spark's catalog.

4: Incorrect. DataFrame.createOrReplaceTempView() is used for registering a DataFrame as a temporary view in Spark's catalog, not for executing SQL queries. This method makes the DataFrame available for SQL queries but does not execute them.

5: Incorrect. DataFrame.createTempView() is similar to createOrReplaceTempView() in that it registers a DataFrame as a temporary view, but it doesn't execute SQL queries itself.

***Additional Information:***

The spark.sql() method is a powerful feature in Spark that enables executing SQL queries on data stored in DataFrames. Before running a SQL query on a DataFrame, the DataFrame must be registered as a temporary view using createTempView() or createOrReplaceTempView(). This integration of SQL capabilities makes Spark a versatile tool for data processing and analysis.

More details can be found in the Apache Spark documentation at <https://sparkbyexamples.com/pyspark/pyspark-what-is-sparksession/>

## Question 41:

Which of the following code blocks creates a single-column DataFrame from Python list years which is made up of integers?

* **spark.createDataFrame([years], IntegerType())**
* **spark.createDataFrame(years, IntegerType())**
* **spark.DataFrame(years, IntegerType())**
* **spark.createDataFrame(years)**
* **spark.createDataFrame(years, IntegerType)**

**Explanation**

1: Incorrect. spark.createDataFrame([years], IntegerType()) is not the correct syntax. Wrapping 'years' in another list changes the intended structure of the DataFrame.

2: Incorrect. spark.createDataFrame(years, IntegerType()) incorrectly uses the IntegerType() as the second argument. Spark expects a schema or column names as the second argument, not a type object.

3: Incorrect. spark.DataFrame(years, IntegerType()) is not a valid method in Spark. The correct method is createDataFrame.

4: Correct. spark.createDataFrame(years) correctly creates a DataFrame from the Python list 'years'. Spark is able to infer the schema, especially when the list consists of primitive data types like integers.

5: Incorrect. spark.createDataFrame(years, IntegerType) is incorrect because the second argument should be a schema or list of column names, not a data type class.

**Additional Information:**

When creating a DataFrame from a Python list in Spark, the createDataFrame method is used. Spark is capable of inferring the schema in many cases, which simplifies the process of DataFrame creation from native Python data structures.

For more details on DataFrame creation, refer to the Apache Spark documentation at <https://sparkbyexamples.com/spark/different-ways-to-create-a-spark-dataframe/>

## Question 42:

**Which of the following operations can be used to cache a DataFrame only in Sparkâ€™s memory assuming the default arguments can be updated?**

* **DataFrame.clearCache()**
* **DataFrame.storageLevel**
* **StorageLevel**
* **DataFrame.persist()**
* **DataFrame.cache()**

**Explanation**

1: Incorrect. DataFrame.clearCache() is not a valid Spark DataFrame operation. It does not relate to caching a DataFrame in memory.

2: Incorrect. DataFrame.storageLevel is a property that describes the current storage level of a DataFrame, but it does not cache the DataFrame in memory.

3: Incorrect. StorageLevel is a class in Spark that defines different storage levels, including memory-only storage, but it is not a method to cache a DataFrame.

4: Incorrect. DataFrame.persist() can cache a DataFrame, but without specifying the storage level, it may not cache the DataFrame only in memory. By default, persist() uses MEMORY\_AND\_DISK storage level.

5: Correct. DataFrame.cache() is the operation used to cache a DataFrame in memory. It is equivalent to calling persist() with the MEMORY\_ONLY storage level. By default, cache() stores the DataFrame only in memory.

Additional Information:

Caching DataFrames in memory is an optimization technique in Spark to speed up access to frequently accessed data. The cache() method is a convenient way to persist DataFrames in memory, which can significantly improve the performance of Spark applications, especially during iterative algorithms.

More information about caching can be found in the Apache Spark documentation at <https://sparkbyexamples.com/pyspark/pyspark-cache-explained/>

## Question 43:

The code block shown below contains an error. The code block is intended to return a new 4-partition DataFrame from the 8-partition DataFrame storesDF without inducing a shuffle. Identify the error. Code block: storesDF.repartition(4)

* **The repartition operation will only work if the DataFrame has been cached to memory.**
* **The repartition operation requires a column on which to partition rather than a number of partitions.**
* **The number of resulting partitions, 4, is not achievable for an 8-partition DataFrame.**
* **The repartition operation induced a full shuffle. The coalesce operation should be used instead.**
* **The repartition operation cannot guarantee the number of result partitions.**

**Explanation**

1: Incorrect. The repartition operation in Spark does not require that the DataFrame be cached to memory.

2: Incorrect. The repartition operation in Spark can take a number of partitions as an argument, and it does not necessarily require a column for partitioning.

3: Incorrect. Reducing the number of partitions from 8 to 4 is achievable in Spark, and there is no inherent limitation that prevents achieving a specific number of partitions.

4: Correct. The error is that the repartition operation, by its nature, induces a full shuffle of the data. If the goal is to reduce the number of partitions without a shuffle, the coalesce operation should be used instead. Coalesce is more efficient for reducing the number of partitions, as it avoids a full shuffle.

5: Incorrect. The repartition operation can guarantee the number of result partitions. It is designed to create a DataFrame with a specified number of partitions.

***Additional Information:***

Understanding the difference between repartition and coalesce is important in Spark. While both can be used to alter the number of partitions, repartition causes a full shuffle, making it more expensive than coalesce. Coalesce is preferable when reducing the number of partitions to avoid the overhead of a shuffle.

More information can be found in the Apache Spark documentation at <https://sparkbyexamples.com/pyspark/pyspark-repartition-usage/>

## Question 44:

**Which of the following code blocks will always return a new 12-partition DataFrame from the 8-partition DataFrame storesDF?**

* **storesDF.coalesce(12)**
* **storesDF.repartition()**
* **storesDF.repartition(12)**
* **storesDF.coalesce()**
* **storesDF.coalesce(12, "storeId")**

**Explanation**

1: Incorrect. The coalesce(12) operation is used to decrease the number of partitions in a DataFrame, and it cannot be used to increase the number of partitions beyond the current number, which is 8 in this case.

2: Incorrect. storesDF.repartition() without any arguments defaults to the default parallelism level of the Spark context, which may not necessarily be 12.

3: Correct. storesDF.repartition(12) will always return a DataFrame with 12 partitions. The repartition operation in Spark can increase or decrease the number of partitions and here, it increases them from 8 to 12.

4: Incorrect. storesDF.coalesce() without any arguments does not change the number of partitions. It's not relevant for explicitly setting the number of partitions.

5: Incorrect. While coalesce can decrease the number of partitions, it cannot increase them. The second argument 'storeId' is irrelevant in this context as coalesce cannot be used to increase partitions to 12.

Additional Information:

Repartitioning a DataFrame is a common operation in Spark, especially when adjusting the parallelism for subsequent transformations or actions. The repartition method is the correct approach when increasing the number of partitions, as it can shuffle data across the cluster to evenly distribute the load.

For more information, refer to the Apache Spark documentation at <https://sparkbyexamples.com/pyspark/pyspark-repartition-usage/>

## Question 45:

Which of the following Spark config properties represents the number of partitions used in wide transformations like join()?

* **spark.sql.shuffle.partitions**
* **spark.shuffle.partitions**
* **spark.shuffle.io.maxRetries**
* **spark.shuffle.file.buffer**
* **spark.default.parallelism**

**Explanation**

1: Correct. spark.sql.shuffle.partitions is the Spark configuration property that determines the number of partitions used in wide transformations such as join(). This setting controls the default number of partitions in the shuffle operation, which is critical for the performance of wide transformations.

2: Incorrect. spark.shuffle.partitions is not a valid configuration property in Spark. The correct property is spark.sql.shuffle.partitions.

3: Incorrect. spark.shuffle.io.maxRetries is related to the number of retries Spark will attempt in case of a shuffle operation failure, not the number of partitions in a shuffle.

4: Incorrect. spark.shuffle.file.buffer is a configuration for the size of the buffer used to write shuffle files, not the number of partitions.

5: Incorrect. spark.default.parallelism affects the default partition number for RDDs and certain DataFrame operations, but spark.sql.shuffle.partitions is more specific to the number of partitions used in wide transformations like join().

***Additional Information:***

Configuring the number of shuffle partitions is an important aspect of tuning Spark applications, especially for optimizing the performance of wide transformations. This setting can have a significant impact on resource utilization and execution speed.

More details can be found in the Apache Spark documentation at <https://spark.apache.org/docs/latest/sql-performance-tuning.html#other-configuration-options.>

## Question 46:

**Which of the following operations performs an inner join on two DataFrames?**

* **DataFrame.innerJoin()**
* **DataFrame.join()**
* **Standalone join() function**
* **DataFrame.merge()**
* **DataFrame.crossJoin()**

**Explanation**

1: Incorrect. DataFrame.innerJoin() is not a valid method in Spark's DataFrame API. The join operation is performed using the join() method, not innerJoin().

2: Correct. DataFrame.join() is the method used to join two DataFrames in Spark. By default, if no join type is specified, it performs an inner join.

3: Incorrect. There isn't a standalone join() function that operates outside of a DataFrame context in Spark. Join operations are methods called on DataFrame objects.

4: Incorrect. DataFrame.merge() is not a method in Spark's DataFrame API. The equivalent operation for merging two DataFrames based on common columns or keys is join().

5: Incorrect. DataFrame.crossJoin() performs a cross join, which is different from an inner join. Cross join returns all possible combinations of rows from the two DataFrames, irrespective of any match.

***Additional Information:***

Inner joins are a common operation in data processing, used to combine two DataFrames based on a common key or column. In Spark, the join() method provides functionality for various types of joins, including inner, outer, left, and right joins. More details about join operations can be found in the Apache Spark documentation at <https://sparkbyexamples.com/pyspark/pyspark-join-explained-with-examples/>

## Question 47:

**Which of the following code blocks returns a new DataFrame that is the result of an outer join between DataFrame storesDF and DataFrame employeesDF on column storeId?**

* **storesDF.join(employeesDF, "storeId", "outer")**
* **storesDF.join(employeesDF, "storeId")**
* **storesDF.join(employeesDF, "outer", col("storeId"))**
* **storesDF.join(employeesDF, "outer", storesDF.storeId == employeesDF.storeId)**
* **storesDF.merge(employeesDF, "outer", col("storeId"))**

**Explanation**

1: Correct. storesDF.join(employeesDF, "storeId", "outer") performs an outer join between storesDF and employeesDF using the 'storeId' column as the join key. This syntax correctly specifies the join type ('outer') and the column to join on.

2: Incorrect. storesDF.join(employeesDF, "storeId") defaults to an inner join, not an outer join, because the join type is not explicitly specified.

3: Incorrect. The syntax storesDF.join(employeesDF, "outer", col("storeId")) is not valid. The second argument in the join method should be the join expression or the join key as a string, not a combination of join type and column object.

4: Incorrect. While the join expression storesDF.storeId == employeesDF.storeId is valid, specifying "outer" as a separate argument in this context is incorrect. The join type should be part of the join expression if using column objects.

5: Incorrect. storesDF.merge(employeesDF, "outer", col("storeId")) uses merge(), which is not a valid method in Spark's DataFrame API. The correct method for joining DataFrames is join().

***Additional Information:***

Outer joins are used to combine two DataFrames based on a join key, including rows that do not have matching keys in both DataFrames. In Spark, the join() method with an explicit join type allows for this kind of operation.

More details about join operations can be found in the Apache Spark documentation at <https://www.geeksforgeeks.org/pyspark-join-types-join-two-dataframes/>

## Question 48:

The below code block contains an error. The code block is intended to return a new DataFrame that is the result of an inner join between DataFrame storesDF and DataFrame employeesDF on column storeId and column employeeId which are in both DataFrames. Identify the error. Code block: storesDF.join(employeesDF, [col("storeId"), col("employeeId")])

* **The join() operation is a standalone function rather than a method of DataFrame â€” the join() operation should be called where its first two arguments are storesDF and employeesDF.**
* **There must be a third argument to join() because the default to the how parameter is not "inner".**
* **The col("storeId") and col("employeeId") arguments should not be separate elements of a list â€” they should be tested to see if they're equal to one another like col("storeId") == col("employeeId").**
* **There is no DataFrame.join() operation â€” DataFrame.merge() should be used instead.**
* **The references to "storeId" and "employeeId" should not be inside the col() function â€” removing the col() function should result in a successful join.**

**Explanation**

1: Incorrect. The join() operation in Spark can be used as a method of a DataFrame. The syntax used in the code block is valid for a DataFrame method.

2: Incorrect. The default join type for the join() method in Spark is 'inner'. Therefore, a third argument specifying the join type is not necessary when performing an inner join.

3: Correct. The error in the code block is that the columns 'storeId' and 'employeeId' are being passed as separate elements of a list, suggesting they are independent join conditions. In a typical inner join on multiple columns, the columns from each DataFrame should be equated, such as storesDF.join(employeesDF, (col("storeId") == employeesDF.col("storeId")) & (col("employeeId") == employeesDF.col("employeeId"))).

4: Incorrect. DataFrame.join() is a valid operation in Spark's DataFrame API. The merge() method is not applicable in this context.

5: Incorrect. The use of col() function is appropriate for referencing columns in the join condition. Removing col() would not result in a successful join as it would not correctly reference the DataFrame columns.

*Performing joins on multiple columns requires specifying the join conditions correctly, often equating the corresponding columns from each DataFrame. Understanding how to formulate these conditions is key to successful DataFrame operations in Spark.*

## Question 49:

**Which of the following Spark properties is used to configure the broadcasting of a DataFrame without the use of the broadcast() operation?**

* **spark.sql.autoBroadcastJoinThreshold**
* **spark.sql.broadcastTimeout**
* **spark.broadcast.blockSize**
* **spark.broadcast.compress**
* **spark.executor.memoryOverhead**

**Explanation**

1: Correct. spark.sql.autoBroadcastJoinThreshold is the Spark property that configures the maximum size of a DataFrame that can be broadcasted automatically without explicitly using the broadcast() operation. If the size of a DataFrame is below this threshold, Spark will automatically broadcast it during a join operation.

2: Incorrect. spark.sql.broadcastTimeout is related to the timeout for broadcasting, but it does not control the automatic broadcasting of DataFrames.

3: Incorrect. spark.broadcast.blockSize is related to the size of each piece of data in the broadcast block, not the automatic broadcasting of DataFrames.

4: Incorrect. spark.broadcast.compress is a property that determines whether to compress broadcast variables, but it does not control automatic broadcasting of DataFrames.

5: Incorrect. spark.executor.memoryOverhead is related to the memory overhead for each executor, and it is not specific to the broadcasting of DataFrames.

Additional Information:

The autoBroadcastJoinThreshold property is a key configuration for optimizing join operations in Spark, particularly for smaller DataFrames where broadcasting can lead to more efficient joins. It is important to set this property appropriately based on the memory characteristics and the size of the DataFrames involved.

More information can be found in the Apache Spark documentation at [https://spark.apache.org/docs/latest/sql-performance-tuning.html#join-strategy-hints-for-sql-queries.](https://spark.apache.org/docs/latest/sql-performance-tuning.html#join-strategy-hints-for-sql-queries)

## Question 50:

**The code block shown below should return a new DataFrame that is the result of a cross join between DataFrame storesDF and DataFrame employeesDF. Choose the response that correctly fills in the numbered blanks within the code block to complete this task. Code block: \_\_1\_\_.\_\_2\_\_(\_\_3\_\_)**

1. **storesDF 2. crossJoin 3. employeesDF, "storeId"**

**1. storesDF 2. join 3. employeesDF, "cross"**

**1. storesDF 2. crossJoin 3. employeesDF**

**1. storesDF 2. join 3. employeesDF, "storeId", "cross"**

**1. storesDF 2. crossJoin 3. employeesDF**

**Explanation**

1: Incorrect. storesDF.crossJoin() is the correct method for a cross join, but it only requires the DataFrame to join with, not the join column. The additional "storeId" argument is not needed.

2: Incorrect. While storesDF.join() can be used for various types of joins, specifying "cross" as a join type is not the correct syntax. For a cross join, crossJoin() method or specifying the join type within join() method correctly is necessary.

3: Correct. storesDF.crossJoin(employeesDF) is the proper syntax for performing a cross join in Spark. The crossJoin() method takes only the DataFrame to be joined and does not require join conditions or types.

4: Incorrect. The syntax for a cross join is not storesDF.join(employeesDF, "storeId", "cross"). To perform a cross join using the join() method, the join type should be specified differently.

5: Incorrect. This is a repeat of option C and is also correct. storesDF.crossJoin(employeesDF) correctly performs a cross join.

***Additional Information:***

Cross joins combine every row of the first DataFrame with every row of the second DataFrame, often used for Cartesian products between two DataFrames. In Spark, crossJoin() method is specifically designed for this purpose, providing a clear and concise way to express such joins.

For more details on join types, refer to the Apache Spark documentation at

<https://blog.knoldus.com/joins-in-spark-sql-with-examples/>

## Question 51:

**Which of the following operations performs a position-wise union on two DataFrames?**

* **The standalone concat() function**
* **The standalone unionAll() function**
* **The standalone union() function**
* **DataFrame.unionByName()**
* **DataFrame.union()**

**Explanation**

1: Incorrect. The concat() function is typically used for concatenating columns or rows, but it is not the standard operation for a position-wise union of DataFrames in Spark.

2: Incorrect. unionAll() is a method in older versions of Spark, equivalent to union(), but it is not a standalone function. It's also deprecated in favor of union().

3: Incorrect. The standalone union() function is not the typical usage in Spark's DataFrame API. The union operation is used as a method of DataFrame objects.

4: Incorrect. DataFrame.unionByName() performs a union operation based on column names and not position-wise.

5: Correct. DataFrame.union() performs a position-wise union of two DataFrames. This operation concatenates rows of the second DataFrame to the first, based on column positions rather than column names.

Additional Information:

In Spark, the union() method on DataFrame is used to combine two DataFrames by appending rows of the second DataFrame to the first. It is important that the DataFrames have the same number of columns and compatible types for each column.

More information about the union operation can be found in the Apache Spark documentation at <https://sparkbyexamples.com/pyspark/pyspark-union-and-unionall/>

## Question 52:

**Which of the following code blocks writes DataFrame storesDF to file path filePath as parquet?**

* **storesDF.write.option("parquet").path(filePath)**
* **storesDF.write.path(filePath)**
* **storesDF.write().parquet(filePath)**
* **storesDF.write(filePath)**
* **storesDF.write.parquet(filePath)**

**Explanation**

1: Incorrect. The syntax storesDF.write.option("parquet").path(filePath) is not valid for writing a DataFrame in Parquet format. The option method is used for setting options but not for specifying the format directly.

2: Incorrect. storesDF.write.path(filePath) lacks the format specification and is not a valid syntax for writing a DataFrame in any specific format.

3: Incorrect. storesDF.write().parquet(filePath) contains an unnecessary pair of parentheses after write, making the syntax invalid.

4: Incorrect. storesDF.write(filePath) does not specify the format as Parquet, and it is not a valid syntax in Spark's DataFrame API.

5: Correct. storesDF.write.parquet(filePath) is the correct syntax to write a DataFrame to the specified file path in Parquet format. The parquet method directly specifies the format and takes the file path as an argument.

***Additional Information:***

Writing DataFrames to different file formats is a common operation in data processing tasks. In Spark, the DataFrameWriter provides methods like parquet() to write DataFrames in specific formats. It's important to use the correct method and syntax to ensure the DataFrame is written correctly and efficiently.

For more details on writing DataFrames, refer to the Apache Spark documentation at <https://sparkbyexamples.com/pyspark/pyspark-read-and-write-parquet-file/>

## Question 53:

The code block shown below contains an error. The code block is intended to write DataFrame storesDF to file path filePath as parquet and partition by values in column division. Identify the error. Code block:

storesDF.write.repartition("division").parquet(filePath)

* The argument division to operation repartition() should be wrapped in the col() function to return a Column object.
* There is no parquet() operation for DataFrameWriter - the save() operation should be used instead.
* There is no repartition () operation for DataFrameWriter - the partitionBy() operation should be used instead.
* DataFrame.write is an operation - it should be followed by parentheses to return a DataFrameWriter.
* The mode() operation must be called to specify that this write should not overwrite existing files.

**Explanation**

1: Incorrect. In the context of DataFrameWriter, repartition() does not require the col() function. The column name as a string is sufficient for specifying the partitioning column.

2: Incorrect. The parquet() operation is valid for DataFrameWriter in Spark and is used specifically for writing data in Parquet format.

3: Correct. The error in the code block is using repartition() instead of partitionBy(). The repartition() method is used to repartition a DataFrame, not for specifying partitioning during write operations. To partition data when writing to disk, the partitionBy() method should be used.

4: Incorrect. DataFrame.write is a method that returns a DataFrameWriter object, and it does not require parentheses. The syntax used in the code block is correct.

5: Incorrect. While specifying the mode of writing (such as overwrite, append, etc.) is important, it's not the primary issue with the code block. The main error is using repartition() instead of partitionBy() for writing partitioned data.

***Additional Information:***

When writing DataFrames to disk in Spark, partitionBy() is the correct method to use for partitioning the output data by one or more columns. This method is part of the DataFrameWriter API, which provides various options for writing data, including file format and partitioning.

More information can be found in the Apache Spark documentation at <https://sparkbyexamples.com/pyspark/pyspark-partitionby-example/>

<https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-partitionby/>

Bottom of Form

Top of Form

Question 54:

Which of the following code blocks reads a parquet at the file path filePath into a DataFrame?h

* **spark.read().parquet(filePath)**
* **spark.read().path(filePath, source = "parquet")**
* **spark.read.path(filePath, source = "parquet")**
* **spark.read.parquet(filePath)**
* **spark.read().path(filePath)**

**Explanation**

1: Incorrect. spark.read().parquet(filePath) contains unnecessary parentheses after read. The correct syntax does not use parentheses after read.

2: Incorrect. spark.read().path(filePath, source = "parquet") is not a valid syntax in Spark's DataFrame API for reading data. The format of the data source is specified differently.

3: Incorrect. spark.read.path(filePath, source = "parquet") is not valid syntax in Spark. The read method should be followed by the format-specific method like parquet().

4: Correct. spark.read.parquet(filePath) is the proper syntax for reading a Parquet file into a DataFrame in Spark. The read method is followed directly by the parquet() method, which specifies the format of the file to be read.

5: Incorrect. spark.read().path(filePath) does not specify the format of the file to read, which is essential when working with specific file types like Parquet.

***Additional Information:***

Reading data from various file formats is a fundamental operation in Spark. The DataFrameReader API provides methods like parquet() for reading data in a specific format. It's important to use the correct method that corresponds to the format of the data being read.

More information on reading data can be found in the Apache Spark documentation at <https://api-docs.databricks.com/python/pyspark/latest/pyspark.sql/api/pyspark.sql.DataFrameReader.parquet.html?highlight=parquet>

## Question 55:

**Which of the following code blocks reads JSON at the file path filePath into a DataFrame with the specified schema schema?**

* **spark.read().schema(schema).format(json).load(filePath)**
* **spark.read().schema(schema).format("json").load(filePath)**
* **spark.read.schema("schema").format("json").load(filePath)**
* **spark.read.schema("schema").format("json").load(filePath)**
* **spark.read.schema(schema).format("json").load(filePath)**

**Explanation**

1: Incorrect. The format(json) in this code block is incorrect because it should be a string ('json').

2: Correct. spark.read().schema(schema).format("json").load(filePath) correctly uses the DataFrameReader API to read JSON data with the specified schema. The schema(schema) method sets the schema, format("json") specifies the data format, and load(filePath) reads the data from the given file path.

3: Incorrect. spark.read.schema("schema").format("json").load(filePath) incorrectly uses the string 'schema' instead of the variable schema.

4: Incorrect. This option is a repetition of option 3 and has the same error with the incorrect usage of the string 'schema'.

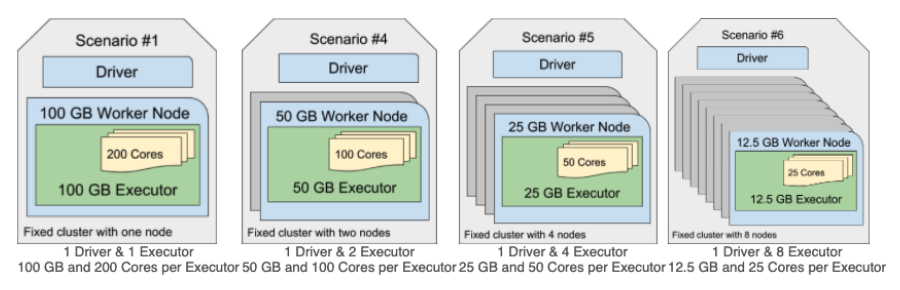
5: Incorrect. spark.read.schema(schema).format("json").load(filePath) is close but contains unnecessary parentheses after read, which is not the standard usage in Spark.

*Reading data with a predefined schema is an important feature in Spark, as it allows users to specify the structure of the data being read. This can help with data type handling and performance optimization. The schema() method in the DataFrameReader API enables this functionality, especially for structured data formats like JSON.*

## Question 56:

Which of the following cluster configurations will ensure the completion of a Spark application in

light of a worker node failure?

****

Note: each configuration has roughly the same compute power using 100GB of RAM and 200 cores.

* **Scenario #1**
* **They should all ensure completion because worker nodes are fault-tolerant.**
* **Scenario #4**
* **Scenario #5**
* **Scenario #6**

**Explanation**

The provided scenarios are:

* Scenario #1: 1 Driver & 1 Executor with 100 GB Worker Node and 200 Cores per Executor
* Scenario #4: 1 Driver & 2 Executors with 50 GB Worker Node and 100 Cores per Executor
* Scenario #5: 1 Driver & 4 Executors with 25 GB Worker Node and 50 Cores per Executor
* Scenario #6: 1 Driver & 8 Executors with 12.5 GB Worker Node and 25 Cores per Executor

The note at the bottom states that each configuration has roughly the same compute power using 100GB of RAM and 200 cores.

To answer this question, we need to consider the fault tolerance feature of Apache Spark. Fault tolerance in Spark is achieved through the use of RDDs, which are resilient and distributed across the cluster nodes. If a node fails, tasks running on that node will be rescheduled on another node, and the data will be reconstructed using lineage information.

Given the scenarios and the note that they all have roughly the same compute power, the following can be deduced:

* Scenario #1: With only one worker node, if this node fails, the entire application will halt as there are no additional nodes to take over the task.
* Scenario #4: With two worker nodes, if one fails, the remaining node can take over the task, but it may lead to resource constraints due to the halved memory and core availability.
* Scenario #5: With four worker nodes, the failure of one node can be tolerated better than in scenario #4, as there are more nodes to distribute the remaining workload.
* Scenario #6: With eight worker nodes, this scenario provides the best fault tolerance. The failure of one node will have the least impact on the available resources and the application's ability to complete.

Based on this analysis, the most fault-tolerant configuration that would ensure the completion of a Spark application in light of a worker node failure is Scenario #6.

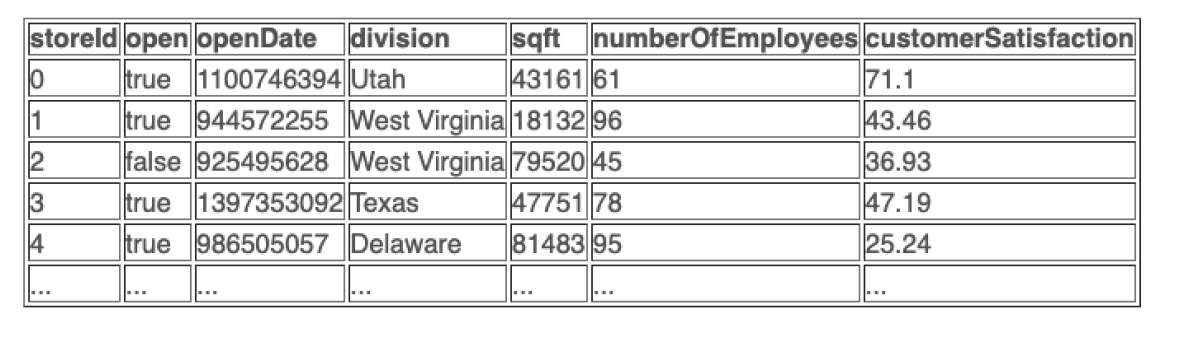
The correct answer is option 5: Scenario #6

## Question 57:

Which of the following code blocks returns a DataFrame containing all columns from DataFrame

storesDF except for column sqft and column customerSatisfaction?

A sample of DataFrame storesDF is below:

****

* **storesDF.drop("sqft", "customerSatisfaction")**
* **storesDF.select("storeId", "open", "openDate", "division")**
* **storesDF.select(-col(sqft), -col(customerSatisfaction))**
* **storesDF.drop(sqft, customerSatisfaction)**
* **storesDF.drop(col(sqft), col(customerSatisfaction))**

**Explanation**

Here is an explanation for each choice provided in the image:

**storesDF.drop("sqft", "customerSatisfaction")**

   - **Correct**. This is the proper syntax in PySpark to drop columns from a DataFrame. The `drop` method takes column names as string arguments, and this statement will return a new DataFrame without the columns "sqft" and "customerSatisfaction".

**storesDF.select("storeId", "open", "openDate", "division")**

   -**Incorrect.**This option uses the `select` method to choose specific columns to include in the resulting DataFrame. However, it does not include all other columns except "sqft" and "customerSatisfaction". It omits columns that are not listed.

**storesDF.select(-col("sqft"), -col("customerSatisfaction"))**

   - **Incorrect**. The intent here seems to be to select columns while excluding "sqft" and "customerSatisfaction" using a minus sign, which is not a valid operation in PySpark to exclude columns. The `select` method is used for inclusion, not exclusion.

***storesDF.drop(sqft, customerSatisfaction)***

   - **Incorrect**. This syntax is not correct because it does not provide the column names as strings. It implies `sqft` and `customerSatisfaction` are variables holding column objects, which is not the case here. PySpark requires column names as strings when using the `drop` method.

***storesDF.drop(col("sqft"), col("customerSatisfaction"))***

   - **Incorrect**. The `drop` method does not accept `Column` type objects returned by the `col` function. It should be passed the column names directly as strings.

*The correct choice for dropping specific columns from a DataFrame in PySpark is to use the `drop` method with the column names provided as strings, which is what choice A correctly demonstrates.*

## Question 58:

In what order should the below lines of code be run in order to return a DataFrame containing a

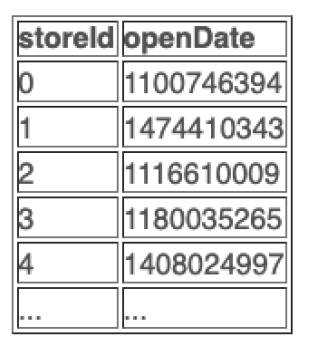
**column openDateString, a string representation of Java’s SimpleDateFormat?**

Note that column **openDate**is of type **integer**and represents a date in the UNIX epoch format —

the number of seconds since midnight on January 1st, 1970.

An example of Java's SimpleDateFormat is **"Sunday, Dec 4, 2008 1:05 PM"**.

A sample of storesDF is displayed below:

****

Lines of code:

1. storesDF.withColumn("openDateString",

from\_unixtime(col("openDate"), simpleDateFormat))

2. simpleDateFormat = "EEEE, MMM d, yyyy h:mm a"

3. storesDF.withColumn("openDateString",

from\_unixtime(col("openDate"), SimpleDateFormat()))

4. storesDF.withColumn("openDateString",

date\_format(col("openDate"), simpleDateFormat))

5. storesDF.withColumn("openDateString",

date\_format(col("openDate"), SimpleDateFormat()))

6. simpleDateFormat = "wd, MMM d, yyyy h:mm a"

* **2, 3**
* **2, 1**
* **6, 5**
* **2, 4**
* **6, 1**

**Explanation**

To return a DataFrame with a new column `openDateString` that formats a Unix epoch time integer into a human-readable date string, the correct lines of code should define the date format first and then apply a combination of `from\_unixtime` and `date\_format` functions to transform the column. Here's the correct order based on the lines provided:

1. First, you need to define the `simpleDateFormat` with the correct format string.

2. Then, you use this format string in combination with `date\_format` and `from\_unixtime` functions to convert the integer epoch time to a formatted date string.

Given the options:

A. 2, 3

B. 2, 1

C. 6, 5

D. 2, 4

E. 6, 1

Option D is correct because line 2 defines the `simpleDateFormat` with the correct format, which matches the example given ("Sunday, Dec 4, 2008 1:05 PM"). Line 4 is the correct usage of the `date\_format` function along with the `from\_unixtime` function to convert the Unix time to a formatted date string.

Option B is incorrect because line 1 uses `from\_unixtime` with `simpleDateFormat` but does not include the necessary `date\_format` function to apply the SimpleDateFormat string.

Option A is incorrect because line 3 uses `SimpleDateFormat()`, which appears to be a class or function that is not defined or used correctly in this context.

Option C is incorrect because line 6 defines a `simpleDateFormat` that does not match the example format, and line 5 uses `SimpleDateFormat()` incorrectly.

Option E is incorrect for the same reason as option C, and line 1 is also incorrect without the use of `date\_format`.

The correct sequence, therefore, would be:  D. 2, 4

The resulting code block should look like this:

1. # Define the date format string
2. simpleDateFormat = "EEEE, MMM d, yyyy h:mm a" # Line 2
4. # Apply the date format to the 'openDate' column to create 'openDateString'
5. storesDF = storesDF.withColumn("openDateString", date\_format(from\_unixtime(col("openDate")), simpleDateFormat))

This code sets the format string and then applies it to convert the Unix epoch time in `openDate` to the desired string representation in `openDateString`.

## Question 59:

Which of the following code blocks returns a DataFrame containing a column month, an integer

representation of the month from column openDate from DataFrame storesDF?

Note that column openDate is of type integer and represents a date in the UNIX epoch format —

the number of seconds since midnight on January 1st, 1970.

A sample of storesDF is displayed below:

* **storesDF.withColumn("month", getMonth(col("openDate")))**
* **storesDF.withColumn("openTimestamp",col("openDate").cast("Timestamp")).withColumn("month",month(col("openTimestamp")))**
* **storesDF.withColumn("openDateFormat",col("openDate").cast("Date")).withColumn("month",month(col("openDateFormat")))**
* **storesDF.withColumn("month", substr(col("openDate"), 4, 2))**
* **storesDF.withColumn("month", month(col("openDate")))**

**Explanation**

The correct approach to extract the month as an integer from a UNIX epoch time in a DataFrame column involves two steps:

1. Convert the **openDate** column from UNIX epoch time (an integer representing seconds since the UNIX epoch) to a timestamp.
2. Extract the month part from the timestamp.

Considering this, let's re-evaluate the options:

A. **storesDF.withColumn("month", getMonth(col("openDate")))**

* Incorrect because **getMonth** is not a standard function in PySpark.

B. **storesDF.withColumn("openTimestamp", col("openDate").cast("Timestamp")).withColumn("month", month(col("openTimestamp")))**

* Correct. This option first casts the **openDate** column to a timestamp data type, which is necessary because the **openDate** is stored as an integer. After the casting, the **month** function is used to extract the month part from the timestamp. This will accurately return the month as an integer.

C. **storesDF.withColumn("openDateFormat", col("openDate").cast("Date")).withColumn("month", month(col("openDateFormat")))**

* Incorrect in this context because the **openDate** column should be cast to a **Timestamp** type to preserve the time information before extracting the month. Casting to a **Date** might lead to date conversion issues depending on the time zone settings.

D. **storesDF.withColumn("month", substr(col("openDate"), 4, 2))**

* Incorrect because **openDate** is an integer and not a string; therefore, substring operations are not applicable.

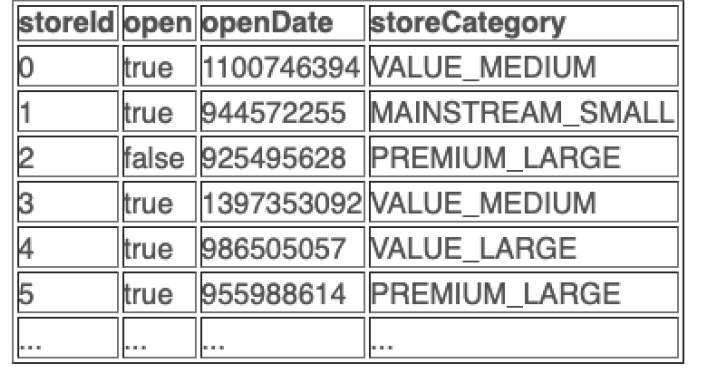
E. **storesDF.withColumn("month", month(col("openDate")))**

* Incorrect because **openDate** is an integer epoch time, and the **month** function expects a date or timestamp type.

## Question 60:

The code block shown below contains an error. The code block intends to return a new DataFrame where column storeCategory from DataFrame storesDF is split at the underscore character into column storeValueCategory and column storeSizeCategory. Identify the error.

A sample of DataFrame storesDF is displayed below:

****

Code block:

(storesDF.withColumn("storeValueCategory", col("storeCategory").split("\_")[0]

).withColumn("storeSizeCategory", col("storeCategory").split("\_")[1]))

* **The split() operation comes from the imported functions object. It accepts a string column name and split character as arguments. It is not a method of a Column object.**
* **The index values of 0 and 1 should be provided as second arguments to the split() operation rather than indexing the result.**
* **The index values of 0 and 1 are not correct — they should be 1 and 2, respectively.**
* **The withColumn() operation cannot be called twice in a row.**

**Explanation**

The correct approach to splitting a string column into separate columns in PySpark involves using the **split** function from the **pyspark.sql.functions** module and then indexing the resulting array to create new columns.

Let's analyze the provided options:

A. **Incorrect.** The **split** function is indeed available in the **pyspark.sql.functions** module, and it is used on Column objects, but it is a function that can be called directly on Column objects using the dot syntax as shown in the code block.

B. **Correct.** The **split** function returns an array column. When you want to access elements of this array, you should use the **getItem()** function with the index as its argument, not directly index the array as in Python. The correct syntax would be **col("storeCategory").split("\_").getItem(0)** and **col("storeCategory").split("\_").getItem(1)**.

C. **Incorrect.** The index values in Python and PySpark are zero-based, so the first element is at index 0 and the second element is at index 1. Therefore, the index values of 0 and 1 are correct.

D. **Incorrect.** You can chain **withColumn** operations one after the other to add multiple columns to a DataFrame in PySpark. This is a common practice and is not an error.

The error in the provided code block is related to how the results of the **split** operation are accessed. The correct method is to use the **getItem()** function instead of directly indexing the array.

Bottom of Form