Contents

[Question 1: 4](#_Toc158135667)

[Question 2: 5](#_Toc158135668)

[Question 3: 6](#_Toc158135669)

[Question 4: 7](#_Toc158135670)

[Question 5: 8](#_Toc158135671)

[Question 6: 9](#_Toc158135672)

[Question 7: 10](#_Toc158135673)

[Question 8: 11](#_Toc158135674)

[Question 9: 12](#_Toc158135675)

[Question 10 13](#_Toc158135676)

[Question 11 14](#_Toc158135677)

[Question 12: 15](#_Toc158135678)

[Question 13: 16](#_Toc158135679)

[Question 14: 17](#_Toc158135680)

[Question 15: 18](#_Toc158135681)

[Question 16: 19](#_Toc158135682)

[Question 17: 20](#_Toc158135683)

[Question 18: 21](#_Toc158135684)

[Question 19: 22](#_Toc158135685)

[Question 20: 23](#_Toc158135686)

[Question 21: 24](#_Toc158135687)

[Question 22: 25](#_Toc158135688)

[Question 23: 26](#_Toc158135689)

[Question 24: 27](#_Toc158135690)

[Question 25: 28](#_Toc158135691)

[Question 26: 29](#_Toc158135692)

[Question 28: 31](#_Toc158135693)

[Question 29: 32](#_Toc158135694)

[Question 30: 33](#_Toc158135695)

[Question 31: 34](#_Toc158135696)

[Question 32: 35](#_Toc158135697)

[Question 33: 36](#_Toc158135698)

[Question 34: 37](#_Toc158135699)

[Question 35: 39](#_Toc158135700)

[Question 36: 40](#_Toc158135701)

[Question 37: 41](#_Toc158135702)

[Question 38: 42](#_Toc158135703)

[Question 39: 43](#_Toc158135704)

[Question 41: 45](#_Toc158135705)

[Question 42: 46](#_Toc158135706)

[Question 43: 47](#_Toc158135707)

[Question 44: 48](#_Toc158135708)

[Question 45: 49](#_Toc158135709)

[Question 46: 50](#_Toc158135710)

[Question 47: 51](#_Toc158135711)

[Question 48: 52](#_Toc158135712)

[Question 49: 53](#_Toc158135713)

[Question 50: 54](#_Toc158135714)

[Question 51: 55](#_Toc158135715)

[Question 52: 56](#_Toc158135716)

[Question 53: 57](#_Toc158135717)

[Question 55: 59](#_Toc158135718)

[Question 56: 60](#_Toc158135719)

[Question 57: 61](#_Toc158135720)

[Question 58: 62](#_Toc158135721)

[Question 59: 63](#_Toc158135722)

[Question 60: 64](#_Toc158135723)

## Question 1:

When considering the various storage levels available in Spark, which of the following assertions is false?

* The `persist` method on Datasets behaves as an action when invoked.
* In cluster mode, Datasets cached at the `MEMORY\_ONLY\_SER` level will use serialized storage on executor nodes.
* To remove a Dataset from persistent storage, the `unpersist` method is applicable.
* The storage level `MEMORY\_AND\_DISK\_SER` serializes data and writes to disk if memory is insufficient.
* The `MEMORY\_ONLY` storage level exclusively utilizes the memory of executor nodes, bypassing disk storage.

**Explanation**

The statement that the `persist` method behaves as an action is false. In Apache Spark, the `persist` method is a lazy transformation that schedules the Dataset to be cached the next time an action is executed on it. The caching operation itself does not trigger the computation of the Dataset. All other statements correctly represent the behavior of Spark's storage levels.

## Question 2:

Assuming two DataFrames named `firstQuarterSales` and `secondQuarterSales`, each with five different columns, how would you generate a single DataFrame by uniting their rows?

* By applying the `stack` function with column realignment.
* By invoking the `attach` function with specified column references.
* By executing the `joinRows` function with a schema-matching condition.
* By employing the `unionAll` function despite differing column identifiers.
* By using the `mergeRows` method with a structure-alignment parameter.

**Explanation**

In Spark, the `unionAll` function can concatenate the rows of two DataFrames, even if their column names do not match. It aligns the rows based on the column order, not the names. The other options are either incorrect or refer to non-existent Spark methods. The `mergeRows` method is not standard in Spark, and `stack`, `attach`, and `joinRows` do not correspond to DataFrame operations in Spark.

## Question 3:

Given a DataFrame `productDetails` and another DataFrame `dailyDeals`, each with a unique set of column labels, how would one concatenate their rows into a single DataFrame?

* Utilize the `stackTogether` method with a map for column labels.
* Apply the `extendRows` function with explicit column identifiers.
* Invoke the `mergeVertical` function with schema alignment.
* Implement the `unify` method with disregard to column names.
* Execute the `combineFrames` function with column correspondence.

**Explanation**

The `unify` method can be used to stack DataFrames vertically without considering the discrepancy in column labels. It assumes that the DataFrame columns are aligned based on their order. The other alternatives are not actual methods for DataFrame manipulation in Spark. Specifically, `stackTogether`, `extendRows`, `mergeVertical`, and `combineFrames` are not standard operations for DataFrames in Spark.

## Question 4:

Imagine you have two DataFrames: `recordsA` and `recordsB`, each containing 5 columns. How would you merge their rows into a new DataFrame, disregarding any differences in column names between the two DataFrames?

* Use the `merge` function with a parameter to ignore column names.
* Apply the `combineRows` method with a column alignment feature.
* Invoke the `unionAll` function, which does not require column name identity.
* Employ the `addRows` method from one DataFrame to the other, aligning data by position.
* Implement the `join` method with a non-strict column matching policy.

**Explanation**

In Spark, the `unionAll` function (or `union` in later versions) is utilized to concatenate rows from two DataFrames, and it operates on the principle of position, not column names. Therefore, it can merge DataFrames with different column names, as long as the number and order of columns are the same.

## Question 5:

In a DataFrame named `salesDataDf` with columns 'amount' and 'itemCode', how can you create a new DataFrame listing all unique values from both columns?

* Using `salesDataDf.select('amount').join(salesDataDf.select('itemCode'), on='amount', how='outer')`.
* Applying `salesDataDf.select('amount', 'itemCode').groupby().count()`.
* Selecting `salesDataDf.select('amount', 'itemCode').dropDuplicates()`.
* Employing `salesDataDf.select('amount').union(salesDataDf.select('itemCode')).dropDuplicates()`.
* Implementing `salesDataDf.agg({'amount': 'collect\_set', 'itemCode': 'collect\_set'})`.

**Explanation**

The correct approach is to first select each column separately, then use the `union` function to stack these columns into one, and finally apply `dropDuplicates` to get all unique values across both columns. This is efficiently done in option 4, which combines the 'amount' and 'itemCode' columns into one and removes any duplicates, resulting in a DataFrame with all unique values.

## Question 6:

In a DataFrame `vendorDataDf`, how can you calculate the number of rows where the string 'Ltd.' appears in the column `vendorName`?

* Using a loop to iterate through `vendorDataDf` and count 'Ltd.' appearances.
* Defining a function to increment a counter when 'Ltd.' is found, applied with `foreach`.
* Utilizing a lambda function in `foreach` to check for 'Ltd.' in each row.
* Applying a lambda function with `foreach` and accumulating the count of 'Ltd.' appearances.
* Creating an accumulator for a count, incrementing inside a function that checks for 'Ltd.' in `vendorName`, and applying this with `foreach`.

**Explanation**

Option 5 correctly uses a Spark accumulator with a `foreach` function to count occurrences of 'Ltd.' in the `vendorName` column. Accumulators are special variables in Spark that are used for summing or aggregating data across all nodes in a distributed manner. The function checks each row for 'Ltd.' in `vendorName` and increments the accumulator, which can be safely updated across multiple nodes.

## Question 7:

You have two DataFrames: `yesterdayData` and `todayData`, both with 6 columns. How can you create a new DataFrame by combining their rows, even if the column names in the two DataFrames are different?

* Using the `combine` method with column name mapping.
* Using the `append` method with explicit column selection.
* Using the `concatenate` method with matching column names.
* Using the `union` method with compatible schemas.
* Using the `merge` method with column name alignment.

**Explanation**

To create a new DataFrame by combining rows from DataFrames with different column names, you can use the `concatenate` method. This method appends the rows of one DataFrame to another, assuming that the columns are in the same order. It doesn't require the column names to be the same. The other options either do not exist in Apache Spark DataFrame operations or do not provide the desired functionality. Therefore, the correct answer is to use the `concatenate` method.

## Question 8:

Given two DataFrames, `archiveSales` with columns A-E and `recentSales` with columns V-Z, which Spark method correctly merges their rows into one DataFrame while disregarding the disparity in column labels?

* Using `archiveSales.mergeWith(recentSales)` method for row integration.
* Applying `archiveSales.joinRows(recentSales)` method for consolidation.
* Utilizing `archiveSales.addRowFrame(recentSales)` for data amalgamation.
* Executing `archiveSales.union(recentSales)` despite column label mismatch.
* Implementing `archiveSales.fuse(recentSales)` for DataFrame unification.

**Explanation**

The `union` method in Spark DataFrame allows for the merging of two DataFrames by appending the rows from one to another. It necessitates the DataFrames to have the same schema structure in terms of column order but does not enforce matching column names. Hence, `archiveSales.union(recentSales)` will combine the rows while the columns' differing names are overlooked.

## Question 9:

In Apache Spark, how can you redistribute a DataFrame named `orderDataDf`, which is currently split into 6 partitions, to have 8 partitions instead?

* Using `orderDataDf.repartition(orderDataDf.partitionCount()+2)` for redistribution.
* Applying `orderDataDf.repartition(orderDataDf.rdd.partitionCount()+2)` to increase partitions.
* Invoking `orderDataDf.coalesce(8)` to adjust the partition count.
* Executing `orderDataDf.coalesce(orderDataDf.partitionCount()+2)` for repartitioning.
* Employing `orderDataDf.repartition(orderDataDf.\_partitionCount+2) ` for reallocation.

**Explanation**

To increase the number of partitions of a DataFrame in Spark, the `repartition` method is used. This method redistributes the data across the specified number of partitions, involving a shuffle. Option 2 correctly calculates the new number of partitions by accessing the current count via the RDD representation of the DataFrame with `orderDataDf.rdd.partitionCount()`. Adding 2 to the current 6 partitions results in 8 partitions. The `coalesce` method is typically used for reducing the number of partitions without a shuffle, which is not the objective here.

## Question 10

How can you create a new DataFrame from `salesDataDf` where each row is an array of integers from 0 up to but not including the value in the `errorCount` column, with `null` for null `errorCount` values?

* Defining a function to generate ranges up to `errorCount` and applying it as a UDF to `errorCount`.
* Creating a range function and using it in a UDF to transform the `errorCount` column.
* Utilizing a lambda function to create ranges up to each value in `errorCount` and applying it.
* Implementing a UDF that generates a list of integers up to each `errorCount` value, handling nulls.

**Explanation**

The correct answer involves defining a user-defined function (UDF) that checks for nulls and returns a list of integers from 0 up to the target number, excluding the target if it's not null. This UDF is then applied to the `errorCount` column, creating a new DataFrame that meets the specified criteria. The UDF needs to handle null values appropriately to avoid errors in processing.

## Question 11

How can you execute an inner join between `purchaseDf` and `productDf` on the matching of `purchaseDf.orderId` and `productDf.productId`, while excluding `totalAmount` and `shopId` from `purchaseDf` and `details` from `productDf` in the final DataFrame?

* By dropping the unwanted columns first and then joining `purchaseDf` with `productDf`.
* Using `purchaseDf` to join with selected columns from `productDf` excluding `details`.
* Joining `purchaseDf` and `productDf` and then selecting only the required columns.
* Creating temporary views for both DataFrames and using Spark SQL to join and exclude specified columns.

**Explanation**

Option 4 correctly addresses the requirement by using Spark SQL. It first creates temporary views for both `purchaseDf` and `productDf`, then performs an inner join using an SQL query, and finally drops the unwanted columns (`totalAmount`, `shopId`, and `details`). This method is efficient for joining and selectively excluding columns in the output DataFrame.

## Question 12:

Identify the error in the code snippet intended to set the Spark configuration to divide data into 15 segments during shuffle operations like joins or aggregations.

* Incorrect command used to set a configuration option.
* Inappropriate configuration option specified for data segmentation.
* Configuration key is incorrectly formatted in the code.
* Number of data segments set is not suitable for optimal performance.
* Missing a required argument in the configuration setting command.

**Explanation**

The issue in the code snippet lies in the format of the configuration key. In Spark, configuration keys are strings and should be enclosed in quotes. The correct syntax for setting the number of partitions for shuffle operations is `spark.conf.set("spark.sql.shuffle.partitions", 15)`, with the key expressed as a string. The number of segments, as set to 15, is not inherently incorrect, but the way the option key is presented in the code is erroneous.

## Question 13:

Identify the error in the code snippet designed to sort `salesRecordsDf` first by `amount` in ascending order and then by `errorFrequency` in the reverse order of `amount`.

* Chain two separate `orderBy` functions for each column instead of combining them.
* Wrap `amount` in the `col()` function for proper column reference.
* Sort `errorFrequency` in descending order, placing nulls at the end.
* Use `desc\_nulls\_first()` for sorting `errorFrequency`.
* Replace `orderBy` with `sort` for the sorting operation.

**Explanation**

The error in the code is related to the sorting order of the `errorFrequency` column. As `amount` is sorted in ascending order, `errorFrequency` should be sorted in descending order to achieve the inverse order. Moreover, to align with the question's requirement, null values in `errorFrequency` should be placed last, not first. The correct code should use `desc\_nulls\_last()` to sort `errorFrequency` in descending order while handling nulls appropriately.

## Question 14:

Identify the errors in a Spark DataFrame operation intended to transform dates in `orderDate` column of `orderDf` to Unix timestamps in a new column `orderTimestamp`, and then remove the original `orderDate` column.

* Drop `orderDate` before creating `orderTimestamp`. Adjust the date format string. Replace column assignment with `withColumn`. Use `to\_unixtime()` instead of `unix\_timestamp()`.
* Drop `orderDate` post-creation of `orderTimestamp`. Use `withColumn` instead of direct assignment. Wrap `orderDate` in `col()` function.
* Wrap `orderDate` in `col()` function in the transformation.
* Adjust the date format string. Use `withColumnReplaced` instead of `drop` and assign pattern to replace `orderDate` with `orderTimestamp`.
* Drop `orderDate` after adding `orderTimestamp`. Adjust the date format string. Use `withColumn` for column transformation.

**Explanation**

The correct approach involves using the `withColumn` method to create `orderTimestamp` from `orderDate` and then dropping the `orderDate` column. The `unix\_timestamp` function is correctly used for converting to Unix timestamps, but the code should adhere to Spark DataFrame API syntax, not Python's. The date format string should match the actual data format in `orderDate`. First, the transformation is applied, and then the original column is removed.

## Question 15:

How can you generate a modified DataFrame from `productDataDf` where the column `features` is renamed to `spec0` and `manufacturer` to `spec1`?

* By chaining `withColumnRenamed` method to rename `features` to `spec0` and `manufacturer` to `spec1`.
* Through separate executions of `withColumnRenamed` for each column renaming without chaining.
* Utilizing `withColumnRenamed` with `col` function for both columns' renaming.
* Applying `withColumn` method to change the names of `features` and `manufacturer`.

**Explanation**

The correct approach is to use the `withColumnRenamed` method in a chain to rename the columns. This method is directly called on the DataFrame and accepts the old and new column names as string arguments. Chaining these methods together applies both renamings to create a new DataFrame with the updated column names. The other options either incorrectly apply the method or use the wrong functions for renaming columns.

Bottom of Form

Top of Form

## Question 16:

Identify the errors in the following code snippet, which aims to reshape `orderDataDf` by keeping only `orderNumber`, `errorMargin`, `amount`, and `shopId`, but incorrectly targets `productCode` and `flag` - code : orderDataDf.select([col('productCode'), col('flag')])

* Directly list desired column names in `select`, not as a list.
* Use `drop` instead of `select`, list `productCode` and `flag` as arguments without `col()` operators.
* Switch from `select` to `drop` for removing specific columns.
* Directly list required columns in `select`, replacing `productCode` and `flag`.
* Change to `drop`, listing `productCode` and `flag` as arguments without `col()` operators.

**Explanation**

The correct method is to use `drop` instead of `select` for excluding columns. The `drop` method should be directly called with the column names `productCode` and `flag` as arguments, without wrapping them in `col()`. The corrected code would be `orderDataDf.drop('productCode', 'flag')`, effectively removing these columns while retaining the desired ones.

## Question 17:

How to obtain a consistent subset of about 1,000 rows from a DataFrame `productDataDf` containing 10,000 rows, ensuring no duplicates and reproducibility on multiple executions?

* By using `productDataDf.sampleBy('row', fractions={0: 0.1}, seed=82371)` for stratified sampling.
* Through `productDataDf.sample(fraction=0.1, seed=87238)` to get a reproducible sample.
* Using `productDataDf.sample(fraction=1000, seed=98263)` for an exact number of rows.
* Applying `productDataDf.sample(withReplacement=True, fraction=0.1, seed=23536)` for duplicate allowance.
* Executing `productDataDf.sample(fraction=0.1)` without a seed for varied results.

**Explanation**

The correct method for obtaining a reproducible sample without duplicates is to use the `sample` method with a fraction and a seed. Setting `fraction` to 0.1 samples approximately 10% of the rows, and the seed ensures the same subset is returned each time. This approach ensures that about 1,000 rows are consistently returned from the 10,000-row DataFrame.

## Question 18:

How to create a new DataFrame from `orderDataDf` that includes columns `itemCode`, `maximum`, and `minimum`, displaying the highest and lowest values in the `amount` column for each `itemCode`?

* Using `orderDataDf.max('amount').min('amount')`.
* Applying `orderDataDf.agg(max('amount').alias('maximum'), min('amount').alias('minimum'))`.
* Executing `orderDataDf.groupby(col('itemCode')).agg(max(col('amount')).alias('maximum'), min(col('amount')).alias('minimum'))`.
* Implementing `orderDataDf.groupby('itemCode').agg(max('amount').alias('maximum'), min('amount').alias('minimum'))`.
* Employing `orderDataDf.groupby('itemCode').agg({'maximum': max('amount'), 'minimum': min('amount')})`.

**Explanation**

The correct approach involves using `groupby` to group `orderDataDf` by `itemCode`, followed by an aggregation (`agg`) with `max` and `min` functions applied to the `amount` column. This results in a new DataFrame with the highest (`maximum`) and lowest (`minimum`) values of `amount` for each `itemCode`. The use of `alias` renames the output columns appropriately.

## Question 19:

Identify the mistake in the code intended to save `salesDataDf` as a parquet file at `filePath`, partitioned by the `shopId` column. code : salesDataDf.write.partitionOn('shopId').parquet(filePath)

* Both partitioning column and file path should be directly passed to `write()` method.
* Invoke `partitionOn` before the `write` method.
* Include `mode()` option in DataFrameWriter to overwrite existing files at `filePath`.
* Wrap `shopId` with `col()` in the partitioning method.
* Replace non-existing `partitionOn()` with the correct `partitionBy()` method.

**Explanation**

The error lies in using a non-existent method `partitionOn()`. The correct method for partitioning data when writing a DataFrame to a file in PySpark is `partitionBy()`. This method is a part of DataFrameWriter API, used for partitioning data by specified columns when saving to disk. The corrected code should be `salesDataDf.write.partitionBy('shopId').parquet(filePath)`, which correctly partitions the data by `shopId` before writing it as a Parquet file.

## Question 20:

What is the correct PySpark code to load a JSON file located at `dataPath` into a DataFrame?

* Use `spark.read.json(dataPath)` for direct JSON file reading.
* Apply `spark.read.path(dataPath, source='json')` for specified source reading.
* Invoke `spark.read().path(dataPath)` for generic file reading.
* Execute `spark.read().json(dataPath)` using empty read call initially.
* Implement `spark.read.path(dataPath)` without specifying file format.

**Explanation**

Option 1 correctly demonstrates the PySpark syntax for reading a JSON file into a DataFrame. The `spark.read.json` method directly loads the JSON file from the specified path into a DataFrame. This method is part of PySpark's DataFrameReader API, specifically designed to handle JSON file formats efficiently.

## Question 21:

How to modify `purchaseDataDf` by adding a new column `purchaseDateFormatted`, transforming Unix timestamp values in `purchaseDate` into a string format like 'Jan 20 (Thursday)'? code : purchaseDataDf.\_1\_(\_2\_, from\_unixtime(\_3\_, \_4\_))

* Use `withColumn`, 'purchaseDateFormatted', 'purchaseDate', 'MMM d (EEEE)' for correct transformation.
* Apply `select`, 'purchaseDate', 'purchaseDateFormatted', 'MMM d (EEEE)' for data selection.
* Invoke `withColumn`, 'purchaseDateFormatted', 'purchaseDate', 'MMM d (EEEE)' for desired output.
* Employ `withColumn`, 'purchaseDateFormatted', 'purchaseDate', 'M d (EEE)' for a different date format.
* Implement `withColumnRenamed`, 'purchaseDate', 'purchaseDateFormatted', 'M d (EEE)' for renaming.

**Explanation**

Option 3 correctly uses the `withColumn` method to add a new column `purchaseDateFormatted` to `purchaseDataDf`. It transforms the `purchaseDate` Unix timestamps into a human-readable string format specified by 'MMM d (EEEE)', which corresponds to the format like 'Jan 20 (Thursday)'.

## Question 22:

Which command correctly displays the structure, including column names and data types, of a DataFrame named `productDetailsDf` in a hierarchical format?

* Execute `print(productDetailsDf.columns); print(productDetailsDf.types)` for column details.
* Use `productDetailsDf.printSchema()` to show DataFrame structure.
* Invoke `spark.schema(productDetailsDf)` for schema details.
* Apply `productDetailsDf.rdd.printSchema()` to view the RDD schema.
* Run `productDetailsDf.print.schema()` for schema information.

**Explanation**

The `printSchema()` method in PySpark is the correct way to display a DataFrame's schema, including its column names and data types, in a structured, tree-like format. It provides a clear, readable representation of the DataFrame's structure, making it ideal for schema inspection.

## Question 23:

**How to read a Parquet file at `dataPath` with multiple partitions into a DataFrame, ensuring a unified schema if each partition has a different schema?**

* **Use `spark.read.parquet(dataPath, mergeSchema='y')` for schema merging.**
* **Apply `spark.read.option('mergeSchema', 'true').parquet(dataPath)` to unify different schemas.**
* **Invoke `spark.read.parquet(dataPath)` without additional options.**
* **Implement a loop to read each file separately and use `union` for merging.**
* **Employ a loop with file reading and `join` on each iteration for merging.**

**Explanation**

Option 2 correctly uses the `mergeSchema` option set to `true` with `spark.read.option`. This approach ensures that when reading a multi-partition Parquet file where each partition might have a different schema, Spark will merge these schemas into a single unified schema in the resulting DataFrame. This method is efficient for handling schema discrepancies in Parquet files with multiple partitions.

## Question 24:

How to modify `productInfoDf` by adding a column `productComponents`, containing arrays of up to four strings split from `productName` using `-` or spaces as separators? code : productInfoDf.\_1\_(\_2\_, \_3\_(\_4\_, \_5\_))

* Utilize `withColumn`, 'productComponents', `split`, 'productName', 4 for array creation.
* Apply `withColumnRenamed`, 'productName', `split`, 'productComponents', 4 for renaming.
* Invoke `withColumn`, 'productComponents', `split`, 'productName', 5 for extended splitting.
* Use `withColumn`, 'productComponents', `str\_split`, 'productName', 5 for different splitting.

**Explanation**

Option 1 is correct as it leverages `withColumn` to add a new column `productComponents` in `productInfoDf`. The `split` function is used for splitting the `productName` into an array of strings, based on `-` or whitespace characters, limited to a maximum of four elements. The precise splitting pattern to match both hyphens and whitespaces is not detailed, and a more complex logic might be needed for an exact limit of four elements.

## Question 25:

You have two DataFrames: `yesterdayData` and `todayData`, both with 6 columns. How can you create a new DataFrame by combining their rows, even if the column names in the two DataFrames are different?

* Using the `combine` method with column name mapping.
* Using the `append` method with explicit column selection.
* Using the `concatenate` method with matching column names.
* Using the `union` method with compatible schemas.
* Using the `merge` method with column name alignment.

**Explanation**

To create a new DataFrame by combining rows from DataFrames with different column names, you can use the `concatenate` method. This method appends the rows of one DataFrame to another, assuming that the columns are in the same order. It doesn't require the column names to be the same. The other options either do not exist in Apache Spark DataFrame operations or do not provide the desired functionality. Therefore, the correct answer is to use the `concatenate` method.

## Question 26:

In the script below, the goal is to append a new column `eventDateFormatted` to the DataFrame `eventsDf`. This column should convert epoch timestamps from `eventDate` into a string format like `Jun 15 (Tuesday)`. Select the correct answer to fill in the blanks in this code to achieve the desired outcome.

* Use `transformColumn`, `eventDateFormatted`, `eventDate`, `MMM dd (EEEE)`
* Apply `addColumn`, `eventDateFormatted`, `eventDate`, `MMM dd (EEEE)`
* Invoke `withColumn`, `eventDateFormatted`, `eventDate`, `MMM dd (EEEE)`
* Employ `withColumn`, `eventDateFormatted`, `eventDate`, `MMMM d (EEE)`
* Implement `changeColumnName`, `eventDate`, `eventDateFormatted`, `MM d (EEEE)`

**Explanation**

Option 3 correctly employs the `withColumn` method to add the new column `eventDateFormatted` to `eventsDf`. The new column is named `eventDateFormatted`, and it's based on the `eventDate` column. The format `MMM dd (EEEE)` matches the desired output, similar to `Jun 15 (Tuesday)`. Other options are incorrect due to either the misuse of non-existent methods or incorrect format specifications.

Bottom of Form

Top of Form

Question 27:

Identify the code snippet that correctly displays the schema of a DataFrame, `productDetailsDf`, in a structured format showing column names and their respective data types.

print(productDetailsDf.columnNames) print(productDetailsDf.dataTypes)

`productDetailsDf.displaySchema()`

`spark.describeSchema(productDetailsDf)`

`productDetailsDf.dataframeSchema.print()`

`productDetailsDf.showSchema()`

**Explanation**

The correct method to display a DataFrame's schema in PySpark, including its column names and data types in a structured format, is `showSchema()`. This function presents the schema in an easy-to-read tree-like structure, aiding in understanding the DataFrame's layout. The other options either use incorrect syntax or refer to non-existent functions in PySpark's DataFrame API.

## Question 28:

Select the code snippet that correctly imports a multi-partition Parquet dataset located at 'dataLocation', ensuring a unified structure even if each partition varies in schema.

**`spark.read.load(dataLocation, mergeSchema='yes')`**

**`spark.read.format('parquet').option('mergeSchema', 'true').load(dataLocation)`**

**`spark.read.format('parquet').load(dataLocation)`**

partitionIndex = 0

for file in dbutils.fs.ls(dataLocation):

if not file.name.endswith('.parquet'):

continue

partDf = spark.read.parquet(file.path)

if partitionIndex == 0:

unifiedDf = partDf

else:

unifiedDf = unifiedDf.unionByName(partDf)

partitionIndex += 1

unifiedDf

partitionIndex = 0

for file in dbutils.fs.ls(dataLocation):

if not file.name.endswith('.parquet'):

continue

partDf = spark.read.parquet(file.path)

if partitionIndex == 0:

unifiedDf = partDf

else:

unifiedDf = unifiedDf.merge(partDf)

partitionIndex += 1

unifiedDf

**Explanation**

Option 2 is accurate as it uses the '**mergeSchema**' option set to 'true' to unify schemas when reading a multi-partition Parquet dataset.

This is essential when dealing with datasets partitioned across multiple files with potentially varying schemas.

The 'mergeSchema' option ensures that all columns from each partition are included once, providing a consistent schema in the final DataFrame.

Other options either use incorrect syntax or do not properly address the issue of merging disparate schemas.

## Question 29:

**Identify the code snippet that produces a DataFrame with two columns, displaying each unique 'orderID' from the DataFrame 'salesDataDf' and the corresponding count of rows for each 'orderID'.**

* **Apply `salesDataDf.count('orderID').distinct()`**
* **Invoke `salesDataDf.groupBy('orderID').agg(col('quantity')).count()`**
* **Use `salesDataDf.count('orderID')`**
* **Employ `salesDataDf.groupBy('orderID').count()`**
* **Select `salesDataDf.groupBy('orderID').select(count('quantity'))`**

**Explanation**

The correct answer is option 4. In this code block, `groupBy('orderID')` organizes the DataFrame `salesDataDf` into groups based on each unique 'orderID'. The subsequent `count()` function then tallies the number of rows in each of these groups. The result is a new DataFrame with two columns: the first being 'orderID' and the second being the count of rows corresponding to each 'orderID'. This method is a standard approach in PySpark for aggregating data based on a specific column and counting the occurrences within each group. Options 1, 2, 3, and 5 either use incorrect methods or do not achieve the desired result of grouping and counting the occurrences of each unique 'orderID'.

## Question 30:

Select the code block that effectively transforms the DataFrame `productsDf`, converting the integer column `productCode` into a string column, while keeping the rest of the DataFrame unchanged.

* Invoke `productsDf.withColumn('productCode', convert('productCode', 'string'))`
* Use `productsDf.withColumn('productCode', col('productCode').cast('string'))`
* Apply `productsDf.select(cast('productCode', 'string'))`
* Employ `productsDf.withColumn('productCode', col('productCode').convert('string'))`
* Select `spark.cast(productsDf, 'productCode', 'string')`

**Explanation**

Option 2 is the correct choice. It uses the `withColumn` method to overwrite the 'productCode' column in `productsDf`. The `cast` function within this method is then used to change the data type of 'productCode' from an integer to a string. This approach is ideal for altering the data type of a specific column in a DataFrame in PySpark without affecting the rest of the DataFrame's structure. Other options either use incorrect syntax or functions not available in PySpark's DataFrame API, or they don't achieve the required task of type conversion while preserving the DataFrame structure.

## Question 31:

You are working with a DataFrame named 'inventoryData' in Apache Spark.

Your goal is to calculate the average 'inventoryError' for each 'storeCode' and 'productCategory', considering only 'productCategory' values of 10 or 20.

The resulting DataFrame should be sorted by 'storeCode' in ascending order, excluding rows where 'storeCode' is null.

Arrange the following code blocks in the correct sequence to accomplish this task:

**avg("inventoryError")**

**.groupBy("storeCode")**

**.orderBy("storeCode")**

**inventoryData.filter(inventoryData.storeCode.isNotNull())**

**.pivot("productCategory", [10, 20])**

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

**Explanation**

The correct order of operations to process 'inventoryData' as required is: 4, 2, 5, 1, 3.

Begin by filtering out rows where 'storeCode' is null (4) to ensure data integrity.

Then, group the data by 'storeCode' (2) to prepare for aggregation.

Pivot the data on 'productCategory' for categories 10 and 20 (5) to focus the analysis on these categories. Calculate the average of 'inventoryError' (1) after grouping and pivoting.

Finally, sort the results by 'storeCode' (3) to organize the data in ascending order.

This sequence ensures proper data preparation, grouping, pivoting, aggregation, and sorting.

## Question 32:

There is an error in the code block below, which aims to merge data from two DataFrames: `productsDf` and `salesDf`. The goal is to display all rows from `productsDf` where the `productId` column matches a value in the `salesId` column of `salesDf`.

Identify the error in the code:

productsDf.join(productsDf.productId == salesDf.salesId)

* **The 'join' statement lacks necessary details.**
* **The 'union' function is the appropriate method, not 'join'.**
* **Usage of the join method is incorrect in this context.**
* **Replace 'join' with the 'merge' function for correct results.**
* **The syntax of the join condition is incorrect.**

**Explanation**

The error lies in the incompleteness of the 'join' statement. It doesn't mention the DataFrame `salesDf` that needs to be joined, and the join type is not specified.

In PySpark, a join operation should include the other DataFrame to join with and can optionally specify the join type for non-inner joins.

The corrected syntax would be:

1. productsDf.join(salesDf, productsDf.productId == salesDf.salesId)

This syntax defaults to an inner join, which is typically used when matching rows based on a condition. The provided code only specifies the condition without referencing the second DataFrame, hence the error.

## Question 33:

**You need to merge the rows of two DataFrames, `salesDataMonday` and `salesDataTuesday`, into a new DataFrame, matching column names and inserting `null` values where column names do not appear in both DataFrames. Find the error in the following code block: `sc.union([salesDataMonday, salesDataTuesday])`**

* **The DataFrames' RDDs need to be passed into the `sc.union` method instead of the DataFrame variable names.**
* **Instead of `union`, the `concat` method should be used, making sure not to use its default arguments.**
* **Instead of the Spark context, `salesDataMonday` should be called with the `join` method instead of the `union` method, making sure to use its default arguments.**
* **Instead of the Spark context, `salesDataMonday` should be called with the `union` method.**
* **Instead of the Spark context, `salesDataMonday` should be called with the `unionByName` method instead of the `union` method, making sure not to use its default arguments.**

**Explanation**

The correct way to merge two DataFrames in Spark while matching column names and handling non-matching columns by inserting `null` values is to use the `unionByName` method, which performs a union based on column names rather than column positions. The `sc.union` method is not appropriate because `sc` (Spark context) is used for operations on RDDs, not DataFrames. The `union` method without 'ByName' does not match columns by name, but rather by position, which is not the requirement here. The correct code block should look like this: ```python salesDataMonday.unionByName(salesDataTuesday) ```

## Question 34:

The code block shown below should return a two-column DataFrame with columns `transactionId` and `supplier`, with combined information from DataFrames `itemsDf` and `transactionsDf`. The code block should merge rows in which column `productId` of DataFrame `transactionsDf` matches the value of column `itemId` in DataFrame `itemsDf`, but only where column `storeId` of DataFrame `transactionsDf` does not match column `itemId` of DataFrame `itemsDf`. Choose the answer that correctly fills the blanks in the code block to accomplish this: Code block: ```python transactionsDf.\_1\_(\_2\_, \_3\_).select(\_4\_) ```

`join` `[transactionsDf.productId==itemsDf.itemId, transactionsDf.storeId!=itemsDf.itemId]` `select` `transactionId, supplier`

**2. `merge` `[transactionsDf.productId==itemsDf.itemId, transactionsDf.storeId!=itemsDf.itemId]` `select` `transactionId**

* 

**supplier`**

* 

**3. `combine` `[transactionsDf.itemId==itemsDf.productId, transactionsDf.storeId!=itemsDf.itemId]` `select` `transactionId**

* 

**supplier`**

* 

**4. `concat` `[transactionsDf.itemId==itemsDf.productId, transactionsDf.storeId!=itemsDf.itemId]` `select` `transactionId**

* 

**supplier`**

* 

**5. `join` `[transactionsDf.itemId==itemsDf.productId, transactionsDf.storeId==itemsDf.itemId]` `select` `transactionId**

* 

**supplier`**

**Explanation**

The correct answer uses the `join` operation to combine the two DataFrames based on the conditions provided. It specifies that `productId` from `transactionsDf` should match `itemId` from `itemsDf`, and that `storeId` from `transactionsDf` should not match `itemId` from `itemsDf`. After joining based on these conditions, the `select` method is used to project only the `transactionId` and `supplier` columns into the resulting DataFrame. The given conditions ensure that the join is performed correctly according to the specified requirements.

## Question 35:

**The code block shown below should return an exact copy of DataFrame `transactionsDf` that does not include rows in which values in column `storeId` have the value `25`. Choose the answer that correctly fills the blanks in the code block to accomplish this: Code block: ```python transactionsDf.\_1\_(\_2\_!=25) ```**

**1. `transactionsDf.remove(transactionsDf.storeId==25)`**

**2. `transactionsDf.where(transactionsDf.storeId!=25)`**

**3. `transactionsDf.filter(transactionsDf.storeId==25)`**

**4. `transactionsDf.drop(transactionsDf.storeId==25)`**

**5. `transactionsDf.select(transactionsDf.storeId!=25)`**

**Explanation**

Option 2 is correct because the `where` function in PySpark is used to filter rows based on a condition. In this case, `transactionsDf.storeId!=25` filters out the rows where `storeId` is equal to `25`. This leaves only the rows where `storeId` is not `25`, which is the requested operation. The `where` and `filter` functions in PySpark are interchangeable, and both can be used to accomplish this task. However, the correct logical condition should use '!=' to exclude the rows with `storeId` equal to `25`. The other options either do not correctly apply the filter or use methods that do not apply to filtering rows based on a condition.

## Question 36:

**The code block shown below should return a copy of DataFrame `transactionsDf` with an added column `cos`. This column should have the values in column `value` converted to degrees and having the cosine of those converted values taken, rounded to two decimals. Choose the answer that correctly fills the blanks in the code block to accomplish this: Code block: ```python transactionsDf.\_1\_(\_2\_, round(\_3\_(\_4\_(\_5\_)),2)) ```**

* **`withColumn` 2. `cos` 3. `cos` 4. `degrees` 5. `transactionsDf.value`**
* **`cos` 3. `cos` 4. `degrees` 5. `transactionsDf.value` 1. `withColumn`**
* **`cos` 2. `cos` 4. `degrees` 5. `transactionsDf.value` 1. `withColumn`**
* **`transactionsDf.value` 4. `degrees` 3. `cos` 2. `cos` 1. `withColumn`**
* **`withColumn` 2. `cos` 4. `degrees` 3. `cos` 5. `transactionsDf.value`**

**Explanation**

The correct answer is to use the `withColumn` method to add a new column to the DataFrame. The new column is named "cos". The `cos` function should be applied to the "value" column, which first needs to be converted from radians to degrees using the `degrees` function. The `round` function is then applied to round the result to two decimal places. In Spark, you might need to import the `cos` and `degrees` functions from `pyspark.sql.functions` and use them as `cos(degrees("value"))`. The provided options represent the correct sequence of operations for this task.

## Question 37:

The code block shown below should return the number of columns in the CSV file stored at location `filePath`. From the CSV file, only lines should be read that do not start with a `#` character.

Choose the answer that correctly fills the blanks in the code block to accomplish this:   \_1\_(\_2\_.\_3\_.\_csv(filePath, \_4\_).\_\_5\_\_)

* **`size` 2. `spark` 3. `read()` 4. `comment='#'` 5. `columns`**
* **`spark` 1. `len` 3. `read` 4. `comment='#'` 5. `columns`**
* **`read` 2. `spark` 1. `len` 4. `comment='#'` 5. `columns`**
* **`comment='#'` 2. `spark` 3. `read` 1. `len` 5. `columns`**
* **`columns` 4. `comment='#'` 3. `read` 2. `spark` 1. `len`**

**Explanation**

The correct answer is using the `**len**` function to get the number of columns in a DataFrame, which can be determined by the length of the columns list.

The `**spark**` session is used to read in data with the `**read**` method.

The `**csv**` method is called to specify the CSV file format.

The option `**comment='#'**` is used to ignore lines starting with `#`.

Finally, `**columns**` will give us the list of column names in the DataFrame.

The completed code block would be: len(spark.read.csv(filePath, comment='#').columns) This line of code reads the CSV file into a DataFrame while ignoring lines starting with `#`, then counts the number of columns in the DataFrame by taking the length of the list of columns.

## Question 38:

**Which of the following code blocks reads in the JSON file stored at `filePath`, enforcing the schema expressed in JSON format in variable `json\_schema`, shown in the code block below?**

* **`spark.read.json(filePath, schema=json\_schema)`**
* **`spark.read.schema(json\_schema).json(filePath)`**
* **`schema = StructType.fromJson(json.loads(json\_schema)) spark.read.json(filePath, schema=schema)`**
* **`spark.read.json(filePath, schema=schema\_of\_json(json\_schema))`**
* **`spark.read.json(filePath, schema=spark.read.json(json\_schema))`**

**Explanation**

The correct way to read a JSON file with a predefined schema in PySpark is to parse the JSON schema string into a `StructType` object and then use this object as the schema when reading the JSON file. The `fromJson` method is used to create a `StructType` from a JSON string. The `json.loads` function from Python's standard library is used to parse the JSON string into a Python dictionary which is then converted into a `StructType`. This schema is then passed to the `spark.read.json` method to read the JSON file with the specified schema. This ensures that the DataFrame conforms to the schema defined by `json\_schema`. The other options do not provide the correct method for parsing the schema and applying it during the read operation.

## Question 39:

Which of the following code blocks applies the Python function `to\_limit` on column `predError` in table `transactionsDf`, returning a DataFrame with columns `transactionId` and `result`?

* **`spark.udf.register("LIMIT\_FCN", to\_limit) spark.sql("SELECT transactionId, LIMIT\_FCN(predError) AS result FROM transactionsDf")`**
* **`spark.udf.register("LIMIT\_FCN", to\_limit) spark.sql("SELECT transactionId, LIMIT\_FCN(predError) FROM transactionsDf AS result")`**
* **`spark.udf.register("LIMIT\_FCN", to\_limit) spark.sql("SELECT transactionId, to\_limit(predError) AS result FROM transactionsDf")`**
* **`spark.sql("SELECT transactionId, udf(to\_limit(predError)) AS result FROM transactionsDf")`**
* **`spark.udf.register(to\_limit, "LIMIT\_FCN") spark.sql("SELECT transactionId, LIMIT\_FCN(predError) AS result FROM transactionsDf")`**

**Explanation**

The correct answer involves two steps. First, the Python function `to\_limit` is registered as a UDF (User-Defined Function) with Spark SQL, using the name "LIMIT\_FCN". Next, a Spark SQL query is used to select the `transactionId` and apply the "LIMIT\_FCN" UDF to the `predError` column, aliasing the output as `result`. This will create a new DataFrame with the specified columns, where `result` is the output of the `to\_limit` function applied to the `predError` column. The correct syntax for registering a UDF and using it in a Spark SQL query is shown in the first option.

Bottom of Form

Top of Form

Question 40:

Which of the following code blocks returns a single-row DataFrame that only has a column `corr` which shows the Pearson correlation coefficient between columns `predError` and `value` in DataFrame `transactionsDf`?

* `transactionsDf.select(corr(["predError", "value"])).alias("corr")).first()`
* `transactionsDf.select(corr(col("predError"), col("value"))).alias("corr")).first()`
* `transactionsDf.select(corr(predError, value).alias("corr"))`
* `transactionsDf.select(corr(col("predError"), col("value"))).alias("corr"))`
* `transactionsDf.select(corr("predError", "value"))`

**Explanation**

The correct answer is the code block that correctly uses the `corr` function from Spark SQL functions, which computes the Pearson correlation coefficient for two columns. The `corr` function is called with `col("predError")` and `col("value")` to specify the columns on which the correlation is to be calculated. The result of the correlation calculation is then aliased to "corr" to provide a meaningful column name in the resulting DataFrame. The `.first()` method is not required as it would return the first row of the DataFrame as a Row object, not a DataFrame. The correct code block does not use `.first()` and instead returns a DataFrame with a single column named `corr`.

## Question 41:

Given the code block template: `transactionsDf.\_\_1\_\_(\_\_2\_\_).\_\_3\_\_`, choose the correct combination to filter and limit the DataFrame.

* **`where` - `"productId" > 2` - `max(2)`**
* **`where` - `transactionsDf[productId] >= 2` - `limit(2)`**
* **`filter` - `productId > 2` - `max(2)`**
* **`filter` - `col("productId") >= 2` - `limit(2)` (Correct Answer)**
* **`where` - `productId >= 2` - `limit(2)`**

**Explanation**

The correct sequence is the fourth option. It uses the `filter` method to select rows where the `productId` is greater than or equal to 2. The `col` function is called to refer to the column within the DataFrame. Finally, `limit(2)` is used to restrict the output to at most two rows. This follows the typical syntax in PySpark for filtering and limiting the results in a DataFrame.

## Question 42:

Which of the following code blocks returns a single-column DataFrame of all entries in Python list `throughputRates` which contains only float-type values?

* `spark.createDataFrame((throughputRates), FloatType)`
* `spark.createDataFrame(throughputRates, FloatType)`
* `spark.DataFrame(throughputRates, FloatType)`
* `spark.createDataFrame(throughputRates)`
* `spark.createDataFrame(throughputRates, FloatType())`

**Explanation**

The correct answer is the fifth option which uses `spark.createDataFrame()` method to create a DataFrame from the `throughputRates` list, specifying the data type of the column as `FloatType` by invoking `FloatType()` constructor. This ensures that all the values in the resulting DataFrame column are treated as floats. Other options are incorrect due to either syntax errors or missing the type specification required to enforce all values as floats.

## Question 43:

**Which of the following describes a narrow transformation?**

* **A narrow transformation is an operation in which data is exchanged across partitions.**
* **A narrow transformation is a process in which data from multiple RDDs is used.**
* **A narrow transformation is a process in which 32-bit float variables are cast to smaller float variables, like 16-bit or 8-bit float variables.**
* **A narrow transformation is an operation in which no data is exchanged across the cluster.**

**Explanation**

Narrow transformations in Spark involve operations where the data required to compute the records in a single partition reside in at most one partition of the parent RDD. Examples include `map()` and `filter()`. There's no shuffle or data exchange across the cluster required to perform a narrow transformation, which makes them efficient as they can be done in-memory without incurring the cost of network communication.

## Question 44:

**Which of the following statements about stages is correct?**

**1. Different stages in a job may be executed in parallel.**

**2. Stages consist of one or more jobs.**

**3. Stages ephemerally store transactions, before they are committed through actions.**

**4. Tasks in a stage may be executed by multiple machines at the same time.**

**Explanation**

In Spark, stages are the result of breaking down a job into smaller sets of tasks that can be executed in parallel. Each stage contains tasks based on narrow or wide transformations that depend on the data from a single or multiple partitions. Tasks within the same stage can be distributed across multiple machines in the cluster to run in parallel, which enhances the processing speed and efficiency. The other options describe incorrect relationships or functionalities regarding stages in Spark.

## Question 45:

What is the role of tasks in Spark's distributed computing model?

* **A task is a command sent from the driver to the executors in response to a transformation.**
* **Tasks transform jobs into DAGs.**
* **A task is a collection of slots.**
* **A task is a collection of rows.**
* **Tasks get assigned to the executors by the driver.**

**Explanation**

In Spark's distributed computing model, tasks represent the smallest unit of work that is sent to the executors. Each task corresponds to a combination of data and computation that runs on a part of the data in an executor. The driver schedules the tasks and assigns them to executors where they are executed. Tasks are based on the stages of the job, which are derived from the DAG (Directed Acyclic Graph) of the data transformation workflow. The option correctly reflects that tasks are indeed assigned to executors by the Spark driver.

## Question 46:

**What is a key difference between Spark's cluster and client execution modes?**

* **In cluster mode, the cluster manager resides on a worker node, while it resides on an edge node in client mode.**
* **In cluster mode, executor processes run on worker nodes, while they run on gateway nodes in client mode.**
* **In cluster mode, the driver resides on a worker node, while it resides on an edge node in client mode.**
* **In cluster mode, a gateway machine hosts the driver, while it is co-located with the executor in client mode.**
* **In cluster mode, the Spark driver is not co-located with the cluster manager, while it is co-located in client mode.**

**Explanation**

In Apache Spark, the execution mode determines where the driver process runs. In cluster mode, the driver runs on one of the nodes within the cluster, often managed by a cluster manager such as YARN, Mesos, or Kubernetes. In client mode, the driver runs on the node where the job is submitted, which is outside of the cluster and is usually called an edge node. This allows the user to directly interact with the job submission process. The correct option clearly states this distinction between the placement of the driver in cluster and client modes.

## Question 47:

**What is a characteristic of Spark's standalone deployment mode?**

* **Standalone mode uses a single JVM to run Spark driver and executor processes.**
* **Standalone mode means that the cluster does not contain the driver.**
* **Standalone mode is how Spark runs on YARN and Mesos clusters.**
* **Standalone mode uses only a single executor per worker per application.**
* **Standalone mode is a viable solution for clusters that run multiple frameworks, not only Spark.**

**Explanation**

In Spark's standalone deployment mode, each worker node will run an executor for each application, and each executor is a separate JVM process. This allows for multiple applications to be run on the same cluster by allocating separate executors to each application, ensuring that each application has its own set of executors. The correct option reflects the fact that in standalone mode, each worker can run executors for multiple applications, with typically one executor per application.

## Question 48:

What is a key property of a shuffle operation in Spark?

* **Operations involving shuffles are never evaluated lazily.**
* **Shuffles involve only single partitions.**
* **Shuffles belong to a class known as 'full transformations'.**
* **A shuffle is one of many actions in Spark.**
* **In a shuffle, Spark writes data to disk.**

**Explanation**

A shuffle in Spark occurs when data needs to be redistributed across different executors or even different machines. This usually happens when operations like `groupBy` or `reduceByKey` are called, which require data corresponding to the same key to be on the same executor. During a shuffle, intermediate data is generally written to disk, making shuffle operations expensive due to the I/O involved. The correct answer highlights this property of shuffles, where Spark indeed writes shuffled data to disk.

## Question 49:

**Which statement accurately distinguishes between actions and transformations in Apache Spark?**

**1. Transformations are executed immediately, whereas actions are evaluated lazily.**

**2. Transformations result in the creation of new RDDs, unlike actions.**

**3. Transformations, not actions, transmit results back to the driver.**

**4. Transformations are executed on demand, whereas actions are set for immediate processing.**

**5. Unlike transformations, actions facilitate Adaptive Query Execution.**

**Explanation**

In Apache Spark, transformations represent lazy operations that create a new RDD or DataFrame by applying a function to data, but they don't compute immediately. Spark retains a transformation lineage on a dataset. Actions, conversely, are the operations that actualize those transformations, yielding results to the driver or saving them. They are crucial for Spark computations. Actions, unlike transformations, can engage Adaptive Query Execution, a dynamic query optimization feature. Transformations alone don't initiate execution or optimization.

## Question 50:

**Identify a true statement about the role of a cluster manager in Apache Spark.**

* **A cluster manager operates on individual data partitions exclusively.**
* **Cluster manager obtains directives from the driver via SparkContext.**
* **Cluster managers are absent in Spark's standalone deployment mode.**
* **Cluster managers are responsible for converting jobs into Directed Acyclic Graphs (DAGs).**
* **The cluster manager operates on the edge node in client deployment mode.**

**Explanation**

In Apache Spark, the cluster manager plays a pivotal role in resource allocation and cluster administration. It indeed gets instructions from the driver through the SparkContext, the primary interface for cluster interaction and application configuration. The cluster manager doesn't directly handle data, convert jobs into DAGs, nor is it limited to specific deployment modes. It is a critical component in all deployment modes of Spark, including standalone, YARN, Mesos, and Kubernetes, where it manages resource orchestration.

## Question 51:

**What accurately describes the function of Spark actions?**

* **Actions mainly focus on persisting data to the filesystem.**
* **Actions facilitate data transfer among Spark's executors.**
* **Data is relayed to the driver upon the execution of actions.**
* **Actions are instrumental in defining the boundaries of stages.**
* **RDD modification is primarily achieved through Spark actions.**

**Explanation**

Actions in Apache Spark are critical for instigating computations and delivering results back to the driver or for data persistence. They serve as the mechanism to initiate computation in Spark's lazy evaluation framework. By executing actions, Spark evaluates the transformations applied to RDDs or DataFrames/Datasets, yielding results to the driver or writing them to storage. Typical actions include collecting data with `collect()`, counting elements with `count()`, or aggregating data with `reduce()`. The statement about actions facilitating the driver's receipt of data encapsulates their role in triggering computations and gathering results.

## Question 52:

**Identify the correct statement regarding Spark executors.**

* **The driver is responsible for initiating executors.**
* **Executors typically terminate following the conclusion of the application.**
* **A single executor is allocated per node.**
* **In-memory storage is the exclusive data storage method for executors.**
* **One executor can manage tasks for several applications simultaneously.**

**Explanation**

Executors in Apache Spark are the entities responsible for executing tasks and storing data relevant to the application. They are initialized at the start of a Spark application and, under usual circumstances, continue to operate for the application's entire lifespan, a process referred to as 'static allocation' of executors. Upon the applicationâ€™s completion, these executors are generally terminated. Each Spark application has its distinct set of executors, and they are capable of storing data in both memory and disk. Furthermore, it's feasible for multiple executors to operate on a single node, depending on the configuration settings.

## Question 53:

**Select the statement that accurately reflects a consideration regarding Spark partitioning.**

* **By default, a shuffle operation generates 200 partitions unless a different setting is specified.**
* **Narrow transformations run faster with fewer partitions when more executors than partitions are available.**
* **Executor-to-executor data transfer is absent when executing the `coalesce()` method.**
* **Rapid processing of partitions usually suggests minimal data skew.**
* **`coalesce()` is typically used for increasing partition counts.**

**Explanation**

In Apache Spark, a shuffle operation such as `reduceByKey` or `groupBy` defaults to creating 200 partitions if no specific value is set through `spark.default.parallelism` or `spark.sql.shuffle.partitions`. This default setting could be problematic if it doesn't match the dataset size or the available resources. Excessively many partitions can lead to high scheduling overheads and small task sizes, while too few partitions might result in not fully utilizing all available executors. Contrary to the fifth option, the `coalesce()` function is generally used to reduce the number of partitions, often to avoid a complete shuffle, rather than to increase them.

Bottom of Form

Top of Form

Question 54:

**Identify a true statement about accumulators in Apache Spark.**

* **Accumulators are only visible in the Spark UI if they are unnamed.**
* **Accumulators are restricted to handling numeric data types.**
* Only the driver can read accumulator values, not the executors.
* **Accumulators are immediately evaluated and do not follow lazy evaluation.**
* **Using accumulators for debugging is challenging as they update only once, regardless of task re-execution.**

**Explanation**

Accumulators in Apache Spark are special variables designed for parallel operations, where they are 'added' to via associative and commutative operations. They are typically used for counters or sum calculations. While Spark supports numeric type accumulators natively, custom types can also be implemented. A key aspect of accumulators is their accessibility; they are readable by the driver after task completion, but executors (or tasks) cannot read them during accumulation. This design choice prevents consistency issues, particularly when actions are rerun, ensuring accurate and reliable operation in parallel processing environments.

## Question 55:

**Which statement is false regarding strategies to mitigate out-of-memory errors in Spark?**

* **Merging several string columns into one column is thought to reduce out-of-memory errors.**
* **Smaller partition sizes can aid in preventing out-of-memory errors.**
* **Controlling the volume of data auto-broadcasted during joins may minimize out-of-memory errors.**
* **Imposing a cap on the maximum size of data serialized back to the driver can be effective against out-of-memory errors.**
* **Reducing the number of cores per executor might assist in managing out-of-memory errors.**

**Explanation**

The notion that combining multiple string columns into a single column can prevent out-of-memory errors is inaccurate. In reality, such concatenation often leads to increased memory usage, as it results in longer strings, thus consuming more memory. In contrast, the other strategies listed can indeed contribute to better memory management and help avoid out-of-memory errors. Specifically, downsizing partition size ensures each partition is manageable in memory; limiting data broadcast in joins stops extensive datasets from being distributed across the cluster; setting a limit on serialized data size aids in controlling the driver's data load; and reducing the number of cores for each executor can lower concurrent task execution, thereby decreasing overall memory demand.

## Question 56:

**Fill in the blanks in the following code snippet to display the data type of the `storeId` column in the DataFrame `transactionsDf`.**

* **`select` / `'storeId'` / `printSchema()`**
* **`filter` / `1` / `schema`**
* **`show` / `'storeId'` / `dataType()`**
* **`select` / `1` / `showSchema()`**
* **`select` / `storeId` / `displayType()`**

**Explanation**

To display the data type of a specific column in a PySpark DataFrame, one should use the `printSchema()` method, which outputs the DataFrame's schema, including the data types of all columns. The `select` method helps to isolate the column of interest. Therefore, the syntax `select('storeId').printSchema()` is the appropriate choice. It selects the `storeId` column and then utilizes `printSchema()` to show its schema information. This syntax correctly employs the method names without quotes and uses parentheses for method invocation, aligning with PySpark's syntax conventions.

## Question 57:

**Complete the code snippet to efficiently retrieve a DataFrame from `transactionsDf` that matches `transactionId` in `itemsDf`, given `itemsDf` is significantly smaller.**

* `transactionsDf` / `merge` / `broadcast(itemsDf)` / `transactionsDf.transactionId == itemsDf.transactionId` / `'full\_outer'`
* `transactionsDf` / `join` / `itemsDf` / `transactionsDf.transactionId == itemsDf.transactionId` / `'left\_anti'`
* `transactionsDf` / `join` / `broadcast(itemsDf)` / `'transactionId'` / `'left\_semi'`
* `itemsDf` / `broadcastJoin` / `transactionsDf` / `'transactionId'` / `'left\_semi'`
* `itemsDf` / `join` / `broadcast(transactionsDf)` / `'transactionId'` / `'left\_semi'`

**Explanation**

The optimal solution for this scenario is to use the `join` method on `transactionsDf` with a `broadcast` hint for `itemsDf`, leveraging the fact that `itemsDf` is much smaller. The `broadcast` hint advises Spark to distribute `itemsDf` across all nodes in the cluster, thus optimizing the join process. The join condition is established by matching `transactionId` columns from both DataFrames. The `left\_semi` join type is crucial as it ensures only the columns from `transactionsDf` are returned, while including only those rows with a corresponding `transactionId` in `itemsDf`. This approach effectively filters `transactionsDf` to include only rows that have a match in `itemsDf`, aligning with the requirement for an efficient query execution.

Bottom of Form

Top of Form

## Question 58:

Identify the error in the following code snippet that aims to load Parquet files ending with `\_\_723.parquet` at '/DataStore/' and enforce a specific schema in Spark:

schema = StructType([

StructType("itemId", IntegerType(), True),

StructType("attributes", ArrayType(StringType(), True), True),

StructType("supplier", StringType(), True)

])

spark.read.options("pathGlobFilter", "\*\_\_723.parquet").schema(schema).load("/DataStore/")

* The attributes array is specified incorrectly and the syntax of the call to Spark's `DataFrameReader` is incorrect.
* Columns in the schema definition use the wrong object type and the types defined for the columns do not match the schema.
* The data type of the `schema` is incompatible with the `schema()` operator and the file name filter is specified incorrectly.
* Columns in the schema definition use the wrong object type and the file name filter is specified incorrectly.
* Columns in the schema are unable to handle empty values and the file name filter is specified incorrectly.

**Explanation**

The error is in the definition of the schema. The code incorrectly uses `StructType` for each column definition inside the `StructType` of the schema.

Instead, it should use `StructField` for each column. The corrected schema definition should look like this:

1. schema = StructType([
2. StructField("itemId", IntegerType(), True),
3. StructField("attributes", ArrayType(StringType(), True), True),
4. StructField("supplier", StringType(), True)
5. ])

The rest of the code appears to be correctly using the `options` method to filter the files ending with `\_\_723.parquet` and then applying the schema with `.schema(schema)` before loading the data from `/DataStore/`.

## Question 59:

**In a Spark application, what is the impact of setting the `spark.sql.shuffle.partitions` configuration to a higher value than the default, especially for large-scale data processing?**

* **It leads to a smaller number of larger partitions, increasing the risk of out-of-memory errors.**
* **It results in more partitions, potentially improving parallelism but increasing shuffle overhead.**
* **It has no significant impact on the execution of Spark SQL queries.**
* **This setting is ignored in Spark SQL and is only relevant for RDD operations.**

**Explanation**

Increasing the `spark.sql.shuffle.partitions` setting results in a higher number of partitions. This can improve parallelism as more tasks can run concurrently. However, it also increases shuffle overhead during tasks like joins or aggregations. While this setting can optimize performance for large-scale data, it requires careful tuning based on the cluster's resources and the nature of the data being processed.

## Question 60:

In a distributed computing environment, what is the primary challenge when implementing a distributed sort algorithm on a large dataset across multiple nodes, and how does Spark address it?

* Data skew, addressed by the Catalyst optimizer through cost-based optimization.
* Network bandwidth limitations, mitigated by Spark's use of broadcast variables.
* Load balancing, managed by Spark through adaptive query execution (AQE).
* Synchronization overhead, reduced by Spark's lazy evaluation model.

**Explanation**

The main challenge in a distributed sort operation is ensuring load balancing across nodes to prevent bottlenecks. Data skew can lead to some nodes having significantly more data to sort than others. Spark addresses this challenge through adaptive query execution (AQE), which dynamically adjusts the execution plan based on runtime statistics. This includes repartitioning data to balance the load across nodes, ensuring efficient sorting in a distributed environment.

Bottom of Form