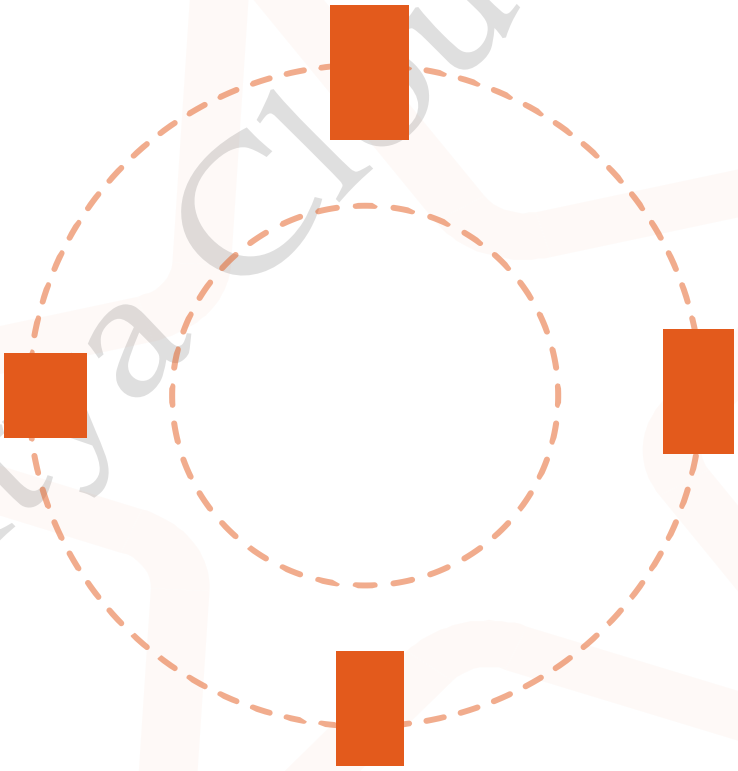
Nitya CloudTech Pvt Ltd.

PySpark Scenario-Based Interview Questions & Answers



1. How would you handle null values in a large DataFrame?

Answer: To handle null values, PySpark provides functions like `fillna()`, `dropna()`, and `replace()`.

- **Remove rows with nulls**: Use `dropna()` to remove rows containing null values. You can specify `how='any'` to remove rows if any column has a null value or `how='all'` to remove rows only if all columns are null.

```
df = df.dropna(how='any')
```

- **Fill nulls**: Use `fillna()` to fill null values with a specified value for either the entire DataFrame or specific columns.

```
df = df.fillna({'column_name': 'value'})
```

- **Replace nulls conditionally**: You may also use `when()` with `isNull()` for more complex replacements, depending on column values.

```
from pyspark.sql.functions import when, col
df = df.withColumn("column_name",
when(col("column_name").isNull(),
"default_value").otherwise(col("column_name")))
```

---

2. How do you improve the performance of a PySpark job that involves multiple DataFrames and joins?

Answer: To optimize PySpark jobs with joins:

- **Broadcast Joins**: Use `broadcast()` for joining a small DataFrame to a large one. Broadcasting replicates the small DataFrame to all nodes, reducing shuffle.

```
from pyspark.sql.functions import broadcast
result_df = large_df.join(broadcast(small_df),
"join_column")
```

- **Partitioning**: Repartition DataFrames based on join keys to minimize shuffling. You can use `repartition()` or `coalesce()` to set the number of partitions based on cluster resources.

```
df = df.repartition("join_column")
```

- **Cache or Persist Intermediate Results**: If you reuse a DataFrame after a complex transformation, persist it in memory with `.cache()` or `.persist()` to avoid recalculating.

```
df = df.cache()
```

---

3. How would you remove duplicate rows from a DataFrame based on a specific column?

4. How would you handle skewed data in a join operation?

Answer: For skewed data, where one key has significantly more data, try these methods:

- **Salting**: Add a random prefix to skewed keys and repartition by this new salted key, which spreads the skewed data more evenly.

```
from pyspark.sql.functions import rand, concat_ws
df1 = df1.withColumn("salt", (rand() *
10).cast("int"))
df2 = df2.withColumn("salt", (rand() *
10).cast("int"))
```

```
df1 = df1.withColumn("skewed_key_salted",
concat_ws("_", col("skewed_key"), col("salt")))
df2 = df2.withColumn("skewed_key_salted",
concat_ws("_", col("skewed_key"), col("salt")))
result = df1.join(df2, "skewed_key_salted")
```

- **Broadcast smaller DataFrame**: If one DataFrame is small and the other is skewed, broadcast the smaller DataFrame to optimize the join.
- **Increase shuffle partitions**: Adjust the `spark.sql.shuffle.partitions` setting to increase partitions, which can help manage load.

---

5. Explain how you would perform aggregation on grouped data and calculate custom metrics.

Answer: To aggregate grouped data and calculate custom metrics, use the `groupBy()` and aggregate functions like `sum()`, `avg()`, or `count()`.

```
from pyspark.sql.functions import avg, sum, count

result = df.groupBy("group_column").agg(
    sum("column1").alias("total"),
    avg("column2").alias("average"),
    count("column3").alias("count")
)
```

- For custom metrics, you can use `expr()` or define UDFs if necessary.

```
from pyspark.sql.functions import expr

result = df.groupBy("group_column").agg(
    expr("sum(column1) / count(column2) as
custom_metric")
)
```

---

6. How would you perform an outer join and fill missing values from another column?

Answer: Perform an outer join and then use `coalesce()` to fill missing values from another column.

```
joined_df = df1.join(df2, "join_column", "outer")
result = joined_df.withColumn("column_name",
coalesce(df1["column_name"], df2["column_name"]))
```

This approach fills nulls in `df1.column_name` with values from `df2.column_name`.

---

7. How can you add a running total or cumulative sum in a DataFrame?

Answer: Use the `Window` function with `rowsBetween()` to calculate a running total.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import sum

window_spec =
Window.orderBy("date_column").rowsBetween(Window.unbo
undedPreceding, 0)
df = df.withColumn("running_total",
sum("column_name").over(window_spec))
```

---

8. How would you handle real-time streaming data with PySpark?

Answer: Use `spark.readStream` for ingesting streaming data and `writeStream` for continuous output to a storage location. For example, for reading from Kafka and writing to a console:

```
df =
spark.readStream.format("kafka").option("kafka.bootst
rap.servers", "localhost:9092").option("subscribe",
"topic").load()
query =
df.writeStream.outputMode("append").format("console")
.start()
query.awaitTermination()
```

You can also write to other sinks like Kafka, HDFS, or databases.

---

9. How do you convert a JSON column to individual columns in a PySpark DataFrame?

Answer: Use `from_json()` with a defined schema and `selectExpr()` to parse JSON and create separate columns.

```
from pyspark.sql.functions import from_json
from pyspark.sql.types import StructType,
StructField, StringType

schema = StructType([StructField("column1",
StringType()), StructField("column2", StringType())])
df = df.withColumn("json_column",
from_json("json_column", schema))
df = df.selectExpr("json_column.column1 as column1",
"json_column.column2 as column2")
```

---

10. How do you optimize PySpark code that performs multiple actions on the same DataFrame?

Answer: Since actions trigger Spark jobs, applying multiple actions on the same DataFrame can be inefficient. Use `.cache()` or `.persist()` to store the DataFrame in memory if it's being reused in multiple actions.

```
df = df.cache()  # or df.persist()
action1_result = df.count()
action2_result = df.collect()
```

This approach avoids re-evaluating the entire DataFrame lineage each time an action is called.