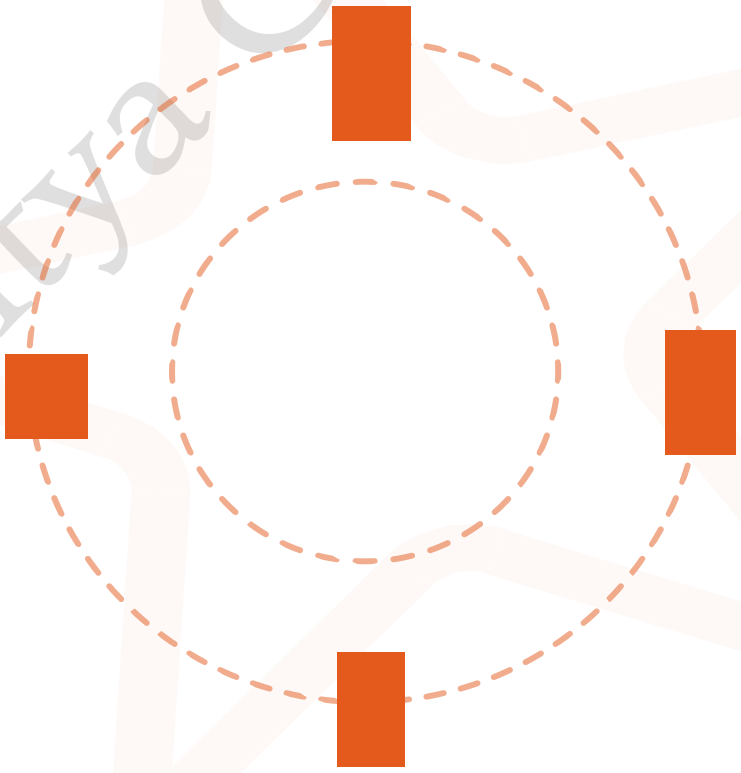
Nitya CloudTech Pvt Ltd.

PySpark Scenario-Based Interview Questions & Answers



1. Question: You have a PySpark DataFrame of daily sales records, but there are missing dates. How would you fill in these missing dates with zero sales?

- **Answer:** Generate a complete date range using `sequence()`, perform an outer join, and fill in missing values:

```
from pyspark.sql.functions import sequence, to_date, col, lit
from pyspark.sql import functions as F

# Generate date range
min_date, max_date = df.select(F.min("date"), F.max("date")).first()
date_range = spark.createDataFrame([(d,) for d in range(min_date, max_date)], ["date"])

# Join and fill missing values
result = date_range.join(df, "date", "left").fillna({"sales": 0})
```

2. Question: How would you handle a large, skewed dataset in PySpark where certain keys have much higher data volume than others?

- **Answer:** Use **salting** to distribute the skewed keys across partitions:

```
from pyspark.sql.functions import col, monotonically_increasing_id

salted_df = df.withColumn("salt", (col("skewed_key") % 10)).repartition("salt")
result = salted_df.groupBy("skewed_key", "salt").sum("value").drop("salt")
```

3. Question: You need to enrich a PySpark DataFrame with data from another DataFrame based on a unique key, but you want to avoid duplicates. How would you approach this?

- **Answer:** Use a **left semi join** to filter out duplicates, ensuring only matching keys from the first DataFrame are included:

```
df =
df.join(df_enrichment.dropDuplicates(["unique_key"]),
["unique_key"], "left_semi")
```

4. Question: How would you build an ETL pipeline in PySpark to transform, aggregate, and store data on an hourly basis?

- **Answer:** Structure the pipeline as follows:
  - **Extraction:** Load data with schema enforcement.
  - **Transformation:** Apply cleaning, filtering, and enrichment.
  - **Aggregation:** Group by necessary columns.
  - **Storage:** Write output in hourly partition format.

```
df.write.partitionBy("year", "month", "day",
"hour").parquet("output_path")
```

5. Question: How would you handle real-time streaming data in PySpark from Kafka, transforming and storing it?

- **Answer:** Use Spark Structured Streaming to read from Kafka, transform data, and write the output to storage:

```
df = spark.readStream.format("kafka").option("subscribe",
"topic").load()
transformed_df = df.selectExpr("CAST(value AS STRING)")
transformed_df.writeStream.format("parquet").option("chec
kpointLocation",
"/path/checkpoint").start("/output/path")
```

6. Question: Given a DataFrame with customer transactions, how would you identify and remove customers with fraudulent patterns (e.g., more than 5 transactions per minute)?

- **Answer:** Use a `Window` function with `count()` to identify the suspicious patterns and filter them out:

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

window =
Window.partitionBy("customer_id").orderBy("timestamp").ra
ngeBetween(-60, 0)
flagged = df.withColumn("txn_count",
count("transaction_id").over(window))
```

```
result = flagged.filter(flagged["txn_count"] <= 5)
```

7. Question: How would you aggregate a large dataset by hourly intervals, while handling time zone adjustments?

- **Answer:** Convert timestamps to UTC, apply time zone adjustments as needed, and use `window()` function for aggregations:

```
from pyspark.sql.functions import window,
to_utc_timestamp

df = df.withColumn("utc_time",
to_utc_timestamp("timestamp", "your_timezone"))
result = df.groupBy(window("utc_time", "1 hour")).count()
```

8. Question: You need to merge customer profile changes from a staging DataFrame into a master DataFrame. How would you approach this?

- **Answer:** Use a **merge operation** with Delta Lake:

```
from delta.tables import DeltaTable

delta_table = DeltaTable.forPath(spark,
"/path/to/master")
delta_table.alias("master").merge(
    staging.alias("staging"),
    "master.customer_id = staging.customer_id"
).whenMatchedUpdateAll().whenNotMatchedInsertAll().execut
e()
```

9. Question: How would you implement incremental data extraction from a source table in PySpark?

- **Answer:** Track the latest timestamp or ID processed, and filter the next extraction by this value:

```
last_processed_timestamp = get_last_processed_timestamp()
new_data = source_df.filter(col("timestamp") >
last_processed_timestamp)
```

10. Question: How can you optimize a join operation between two large DataFrames where one DataFrame is small enough to fit in memory?

- **Answer:** Use a **broadcast join**:

```
from pyspark.sql.functions import broadcast

result = df_large.join(broadcast(df_small), "key_column")
```

11. Question: How would you add a new column to a PySpark DataFrame that contains the cumulative sum of another column, grouped by a specific column?

- **Answer:** Use `Window` with `sum()`:

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

window =
Window.partitionBy("group_column").orderBy("date_column")
.rowsBetween(Window.unboundedPreceding, 0)
df = df.withColumn("cumulative_sum",
sum("value_column").over(window))
```

12. Question: How would you detect anomalies in time series data in PySpark?

- **Answer:** Use statistical methods such as Z-score or a rolling average, applying `Window` functions for calculations:

```
from pyspark.sql.functions import avg, stddev, col

window =
Window.partitionBy("id").orderBy("timestamp").rowsBetween
(-5, 5)
df = df.withColumn("mean",
avg("value").over(window)).withColumn("stddev",
stddev("value").over(window))
df = df.withColumn("z_score", (col("value") -
col("mean")) / col("stddev"))
anomalies = df.filter(col("z_score").abs() > 3)
```

13. Question: How would you enforce data quality checks before loading data into a DataFrame?

- **Answer:** Use `filter()` or `where()` to apply validation rules, and add counts to track errors:

```
valid_df = df.filter(col("age") >=
0).filter(col("salary").isNotNull())
invalid_count = df.count() - valid_df.count()
```

14. Question: You have a PySpark DataFrame with duplicate rows based on multiple columns. How would you keep only the latest record based on a timestamp column?

- **Answer:** Use `Window` to rank records and filter:

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

window = Window.partitionBy("id",
"name").orderBy(col("timestamp").desc())
df = df.withColumn("row_num",
row_number().over(window)).filter(col("row_num") == 1)
```

15. Question: How would you add metadata (e.g., data load time) to each row in a PySpark DataFrame?

- **Answer:** Add a constant column using `lit()`:

```
from pyspark.sql.functions import lit, current_timestamp

df = df.withColumn("load_time", current_timestamp())
```

16. Question: How would you parse a column with nested JSON strings and create separate columns for each field?

- **Answer:** Use `from_json()` with a defined schema:

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

schema = StructType([StructField("field1", StringType()),
StructField("field2", StringType())])
df = df.withColumn("parsed_column",
from_json(col("json_column"),
schema)).select("parsed_column.*")
```

17. Question: You have a CSV file with millions of records. What would you do to speed up the read process in PySpark?

- **Answer:** Optimize by setting appropriate options:

```
df = spark.read.option("inferSchema",
"false").option("header",
"true").csv("path/to/large.csv")
```

18. Question: How would you transform data in a PySpark DataFrame and write it to S3 in Parquet format, partitioned by a date column?

- **Answer:** Apply transformations, then write with partitioning:

```
df.write.partitionBy("date_column").parquet("s3://bucket/
output_path")
```

19. Question: How can you monitor the execution time and resource usage of your PySpark job?

- **Answer:** Use Spark UI or logs. You can also measure time in the code:

```
import time

start = time.time()
# your transformation code
end = time.time()
print("Execution Time:", end - start)
```

20. Question: How do you load a JSON file with nested fields and convert it to a DataFrame with flattened columns?

- **Answer:** Use `selectExpr()` and dot notation to flatten nested fields:

```
df = spark.read.json("path/to/nested.json")
flattened_df = df.selectExpr("nested.field1 as field1",
"nested.field2 as field2")
```