Data Engineering 101 PySpark Interview Q&A

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How can you efficiently handle null values in a PySpark DataFrame during a join operation?

Handle nulls by using the na.fill() or na.replace() methods before performing the join to ensure that nulls do not affect the join logic. Additionally, consider using .coalesce() to replace nulls with a default value within the join expression.

```
df1 = df1.na.fill({"column_name": 0});
df2 = df2.na.fill({"column_name": 0});
joined_df = df1.join(df2, on=["key1", "key2"], how="left_outer");
```

```
df = df.withColumn("column_name", coalesce("column_name", lit(0)))
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How do you implement sessionization in PySpark, where events are grouped by user with session breaks after a certain inactivity period?

Implement sessionization by using window functions with a time-based lag() function to detect gaps between events. Use when() and sum() functions to assign session IDs based on gaps exceeding a threshold.

```
windowSpec = Window.partitionBy("user_id").orderBy("event_time");
df = df.withColumn("prev_time", lag("event_time") \
.over(windowSpec));
df = df.withColumn("session_gap",
    when((col("event_time") - col("prev_time")) > threshold, 1) \
    .otherwise(0));
df = df.withColumn("session_id", sum("session_gap") \
    .over(windowSpec))
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

Explain how to implement a rolling average in PySpark over a time series column.

Implement a rolling average by using the Window function with a time-based range or rows between a specified interval. Use .agg() with functions like avg() over the window.

```
windowSpec =
Window.partitionBy("id").orderBy("date") \
 .rowsBetween(-2, 0);

df = df.withColumn("rolling_avg", avg("value") \
.over(windowSpec))
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How would you implement incremental data processing in PySpark for a real-time data pipeline?

Implement incremental processing by using watermarking on event time, coupled with window-based aggregations or joins. Use .writeStream() with outputMode("append") and proper checkpointing to ensure only new data is processed.

```
df = df.withWatermark("event_time", "10 minutes");

agg_df = df.groupBy(window("event_time", "5 minutes")).agg(...);

query = agg_df.writeStream \
 .outputMode("append").format("parquet")
.option("checkpointLocation",
"/path/to/checkpoint").start()
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

What strategies can you use to reduce memory usage in PySpark when working with large datasets?

Reduce memory usage by
1) Selecting only necessary columns,
2) Filtering data early,
3) Repartitioning to smaller partitions,
4) Using efficient file formats (e.g., Parquet, ORC),
5) Persisting data with a lower memory footprint storage level,
6) Avoiding wide transformations, and
7) Using the Kryo serializer for more efficient object serialization.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How do you debug PySpark applications when working with large-scale data in a distributed environment?

Debug by enabling logging, using the Spark UI to track job execution, analyzing stages and tasks for bottlenecks, using accumulators for tracking progress, enabling adaptive query execution (AQE), and using sample data with .sample() or .limit() to test parts of the pipeline in a local environment.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

Explain how you can manage and reduce data shuffling in PySpark.

Manage and reduce shuffling by using

1) Partitioning data correctly on keys used in joins and aggregations,

2) Using map-side reductions with functions like reduceByKey(),

3) Using .coalesce() to reduce the number of partitions after shuffling,

4) Avoiding operations that cause unnecessary shuffles like groupByKey(), and

5) Broadcasting small tables to avoid shuffles.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

What is the role of Catalyst Optimizer in PySpark, and how can you leverage it to optimize queries?

The Catalyst Optimizer is the core component in Spark SQL that optimizes logical query plans to improve query execution.

Leverage it by writing queries that allow Catalyst to apply optimizations like predicate pushdown, column pruning, and efficient join strategies.

Use DataFrame API and SQL for operations, as they benefit most from Catalyst's optimizations.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How can you implement a custom aggregation function in PySpark?

Implement a custom aggregation function by extending UserDefinedAggregateFunction (UDAF) or using the pandas_udf() feature in PySpark.

The UDAF approach is more complex but provides more control over the aggregation process, while pandas_udf() is simpler and integrates with the DataFrame API.

Using pandas_udf():
from pyspark.sql.functions import pandas_udf;
@pandas_udf("double");

def custom_agg(values): return values.mean();

df_grouped = df.groupby("key").agg(custom_agg(df["value"]))

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How would you handle streaming data with late-arriving events in PySpark Structured Streaming?

Handle late-arriving events by using watermarking with event time and specifying a late allowance with the window() function.

Configure the watermark to discard state for older events beyond a certain threshold and define a window duration that allows for processing late data.

```
df = df.withWatermark("event_time", "5 minutes");
windowedCounts =
df.groupBy(window("event_time", "10 minutes")) \
.count()
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

Explain the process of partition pruning in PySpark and how it improves query performance.

Partition pruning improves query performance by filtering out unnecessary partitions based on the query predicate, reducing the amount of data read.

Spark automatically applies partition pruning when using the DataFrame API with queries on partitioned columns.

It can be explicitly triggered by using .filter() or .where() on partition columns.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

What is the difference between a cache and a checkpoint in PySpark, and when would you use each?

Cache stores RDD/DataFrame in memory for quick access during iterative processing, suitable for reusing data within a job.

Checkpoint saves the data to disk and truncates the lineage, used for fault tolerance and breaking long lineages in iterative algorithms or when persisting across jobs. Checkpointing is slower but more reliable.

df.cache(); df.checkpoint(); spark.sparkContext.setCheckpointDir("/path/to/checkpoint")

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How can you perform a cross join in PySpark, and what are the potential pitfalls to avoid?

Perform a cross join using the .crossJoin() method or by joining on a condition that always evaluates to true.

Pitfalls include generating a large number of rows (Cartesian product) which can lead to memory issues and slow performance.

Always ensure that the datasets involved are small enough to handle the cross join efficiently.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

Describe how you would implement a custom partitioner in PySpark, and in what scenarios it might be useful.

Implement a custom partitioner by extending the Partitioner class and overriding the getPartition() method. This is useful when you need to control the distribution of data across partitions based on custom logic, such as optimizing for specific keys or balancing the load in cases where the default partitioner is inefficient.

Custom Partitioner in Scala (concept similar in PySpark):
class CustomPartitioner(numParts: Int) extends Partitioner { def numPartitions: Int = numParts def getPartition(key: Any): Int = { / custom logic */ } }*

In PySpark, you would generally use repartition() with a custom key column to achieve a similar effect.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How do you implement fault tolerance in a PySpark streaming application?

Implement fault tolerance by using checkpointing to persist the state of the stream to a reliable storage system (e.g., HDFS or S3).

Enable checkpointing in the stream by setting the checkpoint directory and handling potential state recovery when the application restarts.

```
streamingDF.writeStream.format("parquet") \
.option("checkpointLocation", "/path/to/checkpoint") \
.start("/path/to/output")
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How can you handle backpressure in a PySpark Structured Streaming application?

Handle backpressure by adjusting the maxOffsetsPerTrigger option to control the number of records processed per trigger interval.

Also, consider using trigger() with a fixed processing time to allow the system to catch up during periods of high load. Monitor and optimize resource allocation (e.g., executor memory) to maintain stable processing rates.

```
df.writeStream.option("maxOffsetsPerTrigger", 1000) \
.trigger(processingTime="10 seconds") \
.start("/path/to/output")
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

What is the difference between a groupBy() and reduceByKey() operation in PySpark, and when would you use each?

groupBy() is used with DataFrames to group data and aggregate it, which triggers a shuffle operation.

reduceByKey() is an RDD operation that combines values for each key on the map side before the shuffle, making it more efficient for large datasets.

Use reduceByKey() for better performance when working with RDDs, and groupBy() when using DataFrames for more complex aggregations.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

Explain the concept of mapPartitions() in PySpark and provide a scenario where it is more efficient than using map().

mapPartitions() applies a function to each partition of an RDD, which can be more efficient than map() when the function is expensive to initialize, as the initialization happens once per partition rather than per element.

It is useful for operations that require external resources like database connections or for minimizing function call overhead on large datasets.

def process_partition(iterator):
db_conn = create_db_connection() return [db_conn.query(x) for x in iterator];
rdd.mapPartitions(process_partition)
This reduces the overhead of repeatedly opening and closing the database connection for each element in the RDD.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How would you implement stateful transformations in PySpark Structured Streaming, and why are they important?

Implement stateful transformations using operations like mapGroupsWithState() or flatMapGroupsWithState() in PySpark Structured Streaming.

These allow you to maintain and update state across streaming data, which is crucial for applications like session tracking, real-time analytics, and complex event processing where the current state depends on past events.

```
def updateState(key, values, state):
new_state = state.getOption().getOrElse(0) + sum(values)
state.update(new_state) return (key, new_state);
streaming_df.groupByKey(lambda x:
x.key).mapGroupsWithState(updateState)
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How would you optimize the performance of a large PySpark job that involves multiple transformations and actions?

Optimize by using the following:

1) Avoid shuffling by using appropriate partitioning and bucketing,
2) Utilize DataFrame API over RDDs,
3) Cache intermediate results when reused,
4) Use broadcast joins for smaller tables,
5) Avoid wide transformations,
6) Use .coalesce() for reducing the number of partitions,
7) Enable adaptive query execution (AQE),
8) Use efficient file formats like Parquet.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How do you handle skewed data in a PySpark job to avoid performance bottlenecks?

Handle skewed data by using techniques like

1) Salting to distribute data more evenly,

2) Using broadcast joins for small datasets,

3) Skewed join optimizations in Spark,

4) Aggregating or reducing data before join,

5) Filtering out unnecessary data early,

6) Increasing the number of partitions for skewed data.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

Explain the concept of window functions in PySpark and when you would use them.

Window functions allow you to perform calculations across a set of table rows that are related to the current row. Use them when you need to perform operations like ranking, cumulative sum, moving average, etc., within specific partitions of your data. Window functions don't reduce the number of rows returned.

```
from pyspark.sql.window import Window;

windowSpec =
Window.partitionBy("category").orderBy("sales");
df.withColumn("rank", rank().over(windowSpec));
df.withColumn("cumulative_sum", sum("sales") \
.over(windowSpec))
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How would you implement a complex aggregation in PySpark, such as calculating a weighted average by group?

Implement a weighted average by first calculating the product of weights and values, then summing these products and dividing by the sum of weights. Use .groupBy() for aggregation and .agg() with expressions.

```
df_weighted = df.withColumn("weighted_value",
col("value") * col("weight"));
```

```
agg_df = df_weighted.groupBy("group_col") \
.agg((sum("weighted_value") /
sum("weight")).alias("weighted_avg"))
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

Describe the use of the explode() function in PySpark and provide a scenario where it is useful.

The explode() function is used to transform an array or map column into multiple rows, one for each element in the array or each key-value pair in the map. It is useful when you have a column with nested structures like arrays or maps and you need to flatten these structures into individual rows for further processing.

```
df = spark.createDataFrame([(1, ["a", "b", "c"]), (2, ["d", "e"])], ["id", "array_col"]);
```

```
df_exploded = df.select("id", explode("array_col").alias("value"))
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How can you handle large-scale data joins in PySpark, especially when the data is too large to fit in memory?

Handle large-scale data joins by:

1) Using broadcast joins for small datasets,

2) Repartitioning data on the join key to optimize shuffling,

3) Using .join() with an appropriate join type (e.g., broadcast hint),

4) Skipping unnecessary columns before join,

5) Utilizing incremental processing or bucketing for repetitive joins.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

What are the key differences between DataFrames and Datasets in PySpark, and when should you prefer one over the other?

DataFrames are untyped and provide optimized execution via Catalyst optimizer.

Datasets are strongly-typed (in Scala/Java) and provide compile-time type safety.

DataFrames are preferred for most use cases due to their ease of use and performance, while Datasets are used when type safety is important, or when working with complex data transformations in Scala or Java.

No specific PySpark example, as Datasets are primarily used in Scala/Java, but you can think of DataFrames in PySpark as untyped Datasets.

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

How do you implement data deduplication in PySpark, especially when there are multiple criteria for identifying duplicates?

Implement deduplication using .dropDuplicates() or .distinct() with a combination of columns as criteria. For more complex cases, you might need to use window functions to rank rows within partitions and filter out all but the highest-ranking (e.g., based on a timestamp) rows.

```
windowSpec = Window.partitionBy("id").orderBy(col("timestamp").desc());
deduped_df = df.withColumn("rank", row_number() \
.over(windowSpec)).filter(col("rank") == 1).drop("rank")
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Data Engineering 101: PySpark

Explain the concept of accumulators in PySpark and when you would use them. How do they differ from broadcast variables?

Accumulators are variables that are only "added" to through an associative and commutative operation and are used to perform counters or sums across executors.

They are primarily used for debugging or tracking global states. Broadcast variables are read-only variables that are cached on each worker node to avoid the need to send copies for each task.

```
accum = spark.sparkContext.accumulator(0);
df.foreach(lambda x: accum.add(1) if x["value"] > 10 else None)
Broadcast example:
broadcast_var = spark.sparkContext.broadcast([1, 2, 3]);
df_filtered = df.filter(col("id").isin(broadcast_var.value))
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Thank you