Here are detailed answers to each Walmart PySpark interview question for a Data Engineer role in 2025:

---

1. How would you design a real-time recommendation system using PySpark and Kafka for Walmart's e-commerce platform?

Answer:

A real-time recommendation system for Walmart's e-commerce platform can be designed using **Apache Kafka**, **PySpark (Structured Streaming)**, and **MLlib**.

Architecture:

1. **Data Ingestion:**
    1. Collect real-time user interactions (clicks, searches, purchases) from Walmart's website or mobile app.
    2. Stream the data to **Kafka topics** (e.g., `user_activity`, `transactions`).
2. **Stream Processing (PySpark + Kafka Integration):**
    1. Use **Spark Structured Streaming** to consume data from Kafka.
    2. Transform and enrich data (e.g., join with product catalog, user profiles).
    3. Store preprocessed data in a **Delta Lake** for historical analysis.
3. **Real-time Recommendation Generation (MLlib / Collaborative Filtering):**
    1. Use **ALS (Alternating Least Squares)** from Spark MLlib for **collaborative filtering**.
    2. Train models using batch data and update recommendations in **real-time** based on new interactions.
4. **Serving Recommendations:**
    1. Store recommendations in a **NoSQL database (e.g., Cassandra, Redis)**.
    2. A REST API fetches recommendations based on user queries.

Code Example:

```
from pyspark.sql import SparkSession

from pyspark.sql.functions import col

from pyspark.ml.recommendation import ALS



# Spark Session

spark = SparkSession.builder.appName("WalmartRecommendations").getOrCreate()



# Read Kafka Stream

df = spark.readStream \

    .format("kafka") \

    .option("kafka.bootstrap.servers", "kafka-broker:9092") \

    .option("subscribe", "user_activity") \

    .load()



# Transform Data

user_df = df.selectExpr("CAST(value AS STRING) as json_str")

user_df = user_df.select(from_json(col("json_str"),
schema).alias("data")).select("data.*")



# Train ALS Model (Offline)

als = ALS(userCol="user_id", itemCol="product_id", ratingCol="rating",
coldStartStrategy="drop")

model = als.fit(training_data)



# Generate Recommendations (Streaming)

predictions = model.transform(user_df)



# Write Output to NoSQL

predictions.writeStream \
```

```
    .format("org.apache.spark.sql.cassandra") \
    .option("keyspace", "walmart") \
    .option("table", "recommendations") \
    .start()
```

---

2. Design a large-scale ETL pipeline to process Walmart's daily transaction data using PySpark.

Answer:

A large-scale ETL pipeline for Walmart's transaction data should be **scalable, fault-tolerant, and efficient**.

Pipeline Design:

1. **Extract (E):**
    1. Read raw transactions from a source (e.g., S3, HDFS, Azure Blob, GCS).
    2. Handle multiple formats (CSV, JSON, Avro, Parquet).
2. **Transform (T):**
    1. Clean, normalize, and validate data (e.g., missing values, duplicates).
    2. Aggregate data (e.g., daily sales, category-wise revenue).
    3. Apply business logic (e.g., discounts, tax calculation).
3. **Load (L):**
    1. Store transformed data in **Data Warehouse (Snowflake, Redshift, BigQuery)**.
    2. Save historical data in **Delta Lake for analytics**.

Code Implementation:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum, count
```

```
# Spark Session

spark = SparkSession.builder.appName("WalmartETL").getOrCreate()



# Extract: Read raw transactions from S3

transactions_df = spark.read.format("parquet").load("s3://walmart-
data/transactions")



# Transform: Clean & Aggregate

cleaned_df = transactions_df.dropna().dropDuplicates()

aggregated_df = cleaned_df.groupBy("store_id", "product_category").agg(

    sum("sales_amount").alias("total_sales"),

    count("transaction_id").alias("total_transactions")

)



# Load: Write to Snowflake

aggregated_df.write \

    .format("snowflake") \

    .option("dbtable", "daily_sales_summary") \

    .option("sfURL", "https://walmart.snowflakecomputing.com") \

    .option("user", "admin") \

    .option("password", "password") \

    .save()
```

---

3. How would you implement a data lakehouse architecture using PySpark, Delta Lake, and cloud storage?

Answer:

A **Data Lakehouse** combines **the flexibility of a data lake** with **the performance of a data warehouse**.

Architecture Components:

1. **Data Ingestion:** Use Kafka, AWS Kinesis, or Azure Event Hubs for real-time streaming data.
2. **Storage Layer:** Store data in **Delta Lake on S3, Azure Data Lake, or GCS**.
3. **Processing Layer:** Use **PySpark for ETL & Delta Lake for ACID transactions**.
4. **Serving Layer:** Expose data using **Presto, Databricks SQL, or Snowflake**.

Implementation Steps:

```
from delta.tables import DeltaTable



# Load Data into Delta Lake

df = spark.read.format("parquet").load("s3://walmart-raw-data/")

df.write.format("delta").mode("overwrite").save("s3://walmart-
lakehouse/transactions")



# Perform Upserts (Merge New Data)

delta_table = DeltaTable.forPath(spark, "s3://walmart-lakehouse/transactions")

new_data = spark.read.format("parquet").load("s3://walmart-new-transactions/")



delta_table.alias("old").merge(

    new_data.alias("new"),

    "old.transaction_id = new.transaction_id"

).whenMatchedUpdate(set={"sales_amount": "new.sales_amount"}).execute()
```

---

4. Design a fault-tolerant streaming pipeline to track inventory levels in real-time across Walmart stores.

Answer:

A fault-tolerant streaming pipeline ensures **exactly-once processing** and **real-time insights** into Walmart's inventory.

Pipeline Design:

1. **Data Ingestion:** Capture real-time inventory updates from store POS systems using **Kafka or Kinesis**.
2. **Processing:** Use **Spark Structured Streaming** to aggregate inventory data.
3. **Fault Tolerance:**
    1. **Checkpointing & WAL (Write-Ahead Logs)** for recovery.
    2. **Idempotent Writes** to prevent duplicates.
4. **Storage & Dashboarding:**
    1. Store processed data in **Delta Lake**.
    2. Use **Power BI / Grafana for real-time monitoring**.

Code Example:

```
from pyspark.sql import SparkSession

from pyspark.sql.functions import sum


spark = SparkSession.builder.appName("InventoryTracking").getOrCreate()


# Read from Kafka

inventory_stream = spark.readStream \

    .format("kafka") \

    .option("subscribe", "inventory_updates") \

    .load()


# Aggregate Inventory Changes

inventory_df = inventory_stream.groupBy("store_id", "product_id").agg(

    sum("stock_change").alias("current_stock")

)
```

```
# Write to Delta Table with Checkpointing

inventory_df.writeStream \

    .format("delta") \

    .option("checkpointLocation", "s3://walmart-checkpoints/") \

    .start("s3://walmart-lakehouse/inventory")
```

---

5. How would you implement incremental data processing for a billion-row dataset using PySpark?

Answer:

Incremental processing reduces overhead by processing **only new or changed records**.

Approach:

1. **Use Delta Lake's Merge for Upserts.**
2. **Use Partitioning (e.g., date-based) to minimize scans.**
3. **Use Watermarking for late-arriving data in streaming.**

Code Example:

```
from delta.tables import DeltaTable


# Read New Data

new_data = spark.read.format("parquet").load("s3://walmart-new-data/")


# Merge Incrementally

delta_table = DeltaTable.forPath(spark, "s3://walmart-lakehouse/transactions")

delta_table.alias("old").merge(

    new_data.alias("new"), "old.transaction_id = new.transaction_id"

).whenMatchedUpdate(set={"sales_amount": "new.sales_amount"}).execute()
```

This approach ensures **scalability, fault tolerance, and real-time insights** for Walmart's data infrastructure. 🚀

---

6. Write a PySpark job to find the top 5 best-selling products per category from a sales dataset.

Solution:

To find the top 5 best-selling products per category, we will:

- Read sales data.
- Group by `category` and `product_id`.
- Compute total sales for each product.
- Use **window functions** to rank products per category.

Implementation:

```
from pyspark.sql import SparkSession

from pyspark.sql.window import Window

from pyspark.sql.functions import sum, rank, col


# Initialize Spark session

spark = SparkSession.builder.appName("TopProducts").getOrCreate()


# Load dataset (Assume it's stored in Parquet format)

sales_df = spark.read.parquet("s3://walmart-data/sales/")


# Aggregate total sales per product per category

aggregated_df = sales_df.groupBy("category", "product_id") \

    .agg(sum("sales_amount").alias("total_sales"))
```

```
# Define a window function to rank products within each category
window_spec = Window.partitionBy("category").orderBy(col("total_sales").desc())

# Assign ranks and filter top 5
top_products_df = aggregated_df.withColumn("rank", rank().over(window_spec)) \
    .filter(col("rank") <= 5)

# Show results
top_products_df.show()
```

---

7. Given a dataset of customer transactions, write a PySpark job to detect fraudulent transactions.

Solution:

We define fraudulent transactions as:

- Transactions that exceed a certain threshold.
- Transactions happening too frequently from the same user.
- Transactions from different locations in a short time.

Implementation:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, when, lag, unix_timestamp
from pyspark.sql.window import Window

spark = SparkSession.builder.appName("FraudDetection").getOrCreate()
```

```
# Load customer transactions

transactions_df = spark.read.parquet("s3://walmart-data/transactions/")



# Define fraud conditions

threshold = 10000  # Assume fraud if amount > 10K

time_window = Window.partitionBy("customer_id").orderBy("transaction_time")



fraud_df = transactions_df.withColumn("fraud_flag",

    when(col("transaction_amount") > threshold, 1)  # High-value transactions

    .when(count("transaction_id").over(time_window) > 5, 1)  # Too many
transactions

    .otherwise(0))



# Detect location-based fraud (same user making transactions from different
locations in < 10 min)

fraud_df = fraud_df.withColumn("prev_location",
lag("location").over(time_window)) \

    .withColumn("prev_time", lag("transaction_time").over(time_window)) \

    .withColumn("time_diff", unix_timestamp(col("transaction_time")) -
unix_timestamp(col("prev_time")))



fraud_df = fraud_df.withColumn("fraud_flag",

    when((col("location") != col("prev_location")) & (col("time_diff") < 600),
1).otherwise(col("fraud_flag")))



# Show fraudulent transactions

fraud_df.filter(col("fraud_flag") == 1).show()
```

---

8. Implement a custom UDF in PySpark to clean and standardize product descriptions.

Solution:

We need to:

- Remove special characters.
- Convert to lowercase.
- Trim extra spaces.

Implementation:

```
from pyspark.sql import SparkSession

from pyspark.sql.functions import udf

from pyspark.sql.types import StringType

import re


spark = SparkSession.builder.appName("CleanProductDescriptions").getOrCreate()


# Define UDF to clean product descriptions

def clean_description(text):

    text = text.lower()  # Convert to lowercase

    text = re.sub(r"[^a-z0-9\s]", "", text)  # Remove special characters

    text = re.sub(r"\s+", " ", text).strip()  # Remove extra spaces

    return text


# Register UDF

clean_description_udf = udf(clean_description, StringType())


# Load dataset

products_df = spark.read.parquet("s3://walmart-data/products/")


# Apply UDF
```

```
cleaned_df = products_df.withColumn("cleaned_description",
clean_description_udf(col("description")))



# Show results

cleaned_df.select("description", "cleaned_description").show()
```

---

9. Write a PySpark function to perform windowed aggregations on time-series sales data.

Solution:

We will:

- Group sales data into 1-hour windows.
- Compute total revenue per store.

Implementation:

```
from pyspark.sql import SparkSession

from pyspark.sql.functions import window, sum



spark = SparkSession.builder.appName("WindowedSalesAggregation").getOrCreate()



# Load time-series sales data

sales_df = spark.readStream \

    .format("parquet") \

    .load("s3://walmart-data/sales/")



# Aggregate sales per hour

windowed_sales = sales_df \
```

```
    .groupBy(window(col("transaction_time"), "1 hour"), "store_id") \
    .agg(sum("sales_amount").alias("total_sales"))


# Write to output sink
query = windowed_sales.writeStream \
    .format("console") \
    .outputMode("update") \
    .start()


query.awaitTermination()
```

---

10. Implement a PySpark job that joins two large datasets efficiently, considering skewed data.

Solution:

When joining large datasets, consider:

1. **Broadcasting smaller dataset** if it's significantly smaller.
2. **Salting technique** to handle skewed joins.

Approach 1: Broadcast Join (If One Dataset is Small)

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import broadcast


spark = SparkSession.builder.appName("OptimizedJoins").getOrCreate()


# Load datasets
```

```
orders_df = spark.read.parquet("s3://walmart-data/orders/")

customers_df = spark.read.parquet("s3://walmart-data/customers/")



# Optimize using broadcast join

joined_df = orders_df.join(broadcast(customers_df), "customer_id")



# Show results

joined_df.show()
```

Approach 2: Salting for Skewed Data

If one key is highly skewed (e.g., a popular product ID appearing too frequently), we **add random "salt" keys**.

```
from pyspark.sql.functions import col, concat, lit, expr, rand



# Add salt column to avoid skewed joins

salted_orders = orders_df.withColumn("salt", (rand() * 10).cast("int"))

salted_customers = customers_df.withColumn("salt",
expr("explode(array(0,1,2,3,4,5,6,7,8,9))"))



# Perform salted join

joined_df = salted_orders.join(salted_customers,

                               (salted_orders.customer_id ==
salted_customers.customer_id) &

                               (salted_orders.salt == salted_customers.salt),
"inner")



# Remove salt column after join

final_df = joined_df.drop("salt")



# Show results
```

```
final_df.show()
```

Summary

Question	Concepts Used
Top 5 Best-Selling Products	Window Functions (`rank()`)
Fraud Detection	Aggregation, Lag Functions, Filtering
UDF for Cleaning	Custom UDF, Regex
Windowed Aggregations	Time-Series, Streaming
Optimized Joins	Broadcast Joins, Salting for Skew

These implementations ensure **scalability, efficiency, and fault tolerance** in real-world Walmart data scenarios. 🚀

---

11. Your PySpark job is taking too long due to shuffle operations. How do you debug and optimize it?

Solution:

Shuffle operations (e.g., `groupBy()`, `join()`, `distinct()`, and `orderBy()`) cause expensive data movement across partitions. To optimize:

1. **Reduce shuffle operations:**
    1. Use `map-side aggregations` instead of full `groupBy()`.
    2. Minimize `distinct()` and unnecessary `orderBy()`.
2. **Optimize partitions:**
    1. Increase `spark.sql.shuffle.partitions` to avoid large partitions.

    2. Use `repartition(n)` for expensive operations and `coalesce(n)` for reducing partition size.
3. **Use broadcast joins for small tables:**
    1. Use `broadcast(df)` to avoid shuffle joins.
4. **Use efficient file formats:**
    1. Prefer **Parquet** over CSV as it supports columnar storage.
5. **Monitor and debug using UI:**
    1. Use `spark.conf.set("spark.sql.adaptive.enabled", "true")`
    2. Monitor **Spark UI** (port 4040) → Check **Shuffle Read Size & Shuffle Spill**.

Implementation Example (Broadcast Join & Partition Optimization):

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import broadcast

spark = SparkSession.builder \
    .appName("ShuffleOptimization") \
    .config("spark.sql.shuffle.partitions", "200") \
    .config("spark.sql.adaptive.enabled", "true") \
    .getOrCreate()

# Load datasets
large_df = spark.read.parquet("s3://walmart-data/large_dataset/")
small_df = spark.read.parquet("s3://walmart-data/small_dataset/")

# Optimize Join using Broadcast
optimized_df = large_df.join(broadcast(small_df), "common_key")
```

```
# Reduce shuffle partitions

optimized_df = optimized_df.repartition(100)



optimized_df.show()
```

---

12. A critical PySpark pipeline is failing due to data skew issues. How do you diagnose and fix it?

Solution:

Data skew occurs when certain keys have significantly more data than others, causing performance issues.

Diagnosis:

1. **Check Skewed Keys:**

```
2. from pyspark.sql.functions import count
```

```
df.groupBy("skewed_column").agg(count("*")).orderBy("count(1)",
ascending=False).show()
```

3. **Check Stage Duration in Spark UI:**
    1. Look for **tasks taking too long** in **DAG visualization**.

Fixing Data Skew:

1. **Salting Technique (Add Random Key):**

```
2. from pyspark.sql.functions import rand

3.
```

```
# Add salt column to skewed dataset
skewed_df = skewed_df.withColumn("salt", (rand() * 10).cast("int"))

# Duplicate keys in smaller dataset
normal_df = normal_df.withColumn("salt",
expr("explode(array(0,1,2,3,4,5,6,7,8,9))"))

# Perform salted join
```

```
joined_df = skewed_df.join(normal_df, ["key", "salt"]).drop("salt")
```

11. **Increase shuffle partitions for better parallelism:**

```
spark.conf.set("spark.sql.shuffle.partitions", "400")
```

12. **Use Broadcast Join if One Table is Small:**

```
from pyspark.sql.functions import broadcast
```

```
optimized_df = skewed_df.join(broadcast(normal_df), "common_key")
```

---

13. A batch PySpark job is reading duplicate records from a data source. How would you remove duplicates efficiently?

Solution:

1. **Use `dropDuplicates()` (Preferred for Structured Data):**

```
cleaned_df = df.dropDuplicates(["id", "timestamp"])
```

2. **Use `distinct()` for full-row duplicates:**

```
cleaned_df = df.distinct()
```

3. **Use `row_number()` to keep the latest record:**

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

window_spec = Window.partitionBy("id").orderBy(df.timestamp.desc())

deduplicated_df = df.withColumn("row_num", row_number().over(window_spec)) \
    .filter("row_num = 1") \
    .drop("row_num")
```

11. **Optimize File Format:**

    1. Use **Delta Lake** for **merge** operations and upserts:

```
from delta.tables import DeltaTable

delta_table = DeltaTable.forPath(spark, "s3://walmart-data/delta/sales/")
delta_table.alias("target").merge(
    df.alias("source"),
    "target.id = source.id"
).whenMatchedUpdate(set={"target.sales": "source.sales"}) \
 .whenNotMatchedInsertAll() \
 .execute()
```

---

14. You need to implement a real-time analytics dashboard using PySpark. What tools and techniques would you use?

Solution:

1. **Real-time Streaming Pipeline:**
    1. **Apache Kafka** → Stream raw data.
    2. **PySpark Structured Streaming** → Process streaming data.
    3. **Delta Lake / Redis** → Store pre-aggregated data.
    4. **Power BI / Tableau** → Visualize.
2. **Implementation:**

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import window, sum

spark = SparkSession.builder.appName("RealTimeDashboard").getOrCreate()

# Read from Kafka
sales_stream = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka-server:9092") \
    .option("subscribe", "sales_topic") \
    .load()

# Process sales data
sales_df = sales_stream.selectExpr("CAST(value AS STRING)") \
    .groupBy(window("timestamp", "5 minutes"), "store_id") \
    .agg(sum("sales_amount").alias("total_sales"))

```

```
# Write results to a Delta Lake table
query = sales_df.writeStream \
    .format("delta") \
    .outputMode("append") \
    .option("checkpointLocation", "/tmp/checkpoints/") \
    .start("s3://walmart-data/delta/realtime_sales/")

```

```
query.awaitTermination()
```

---

15. Your PySpark job runs out of memory when processing large datasets. How do you fix this?

Solution:

1. **Increase Executor Memory (If Cluster Allows):**

```
spark.conf.set("spark.executor.memory", "8g")
```

```
spark.conf.set("spark.driver.memory", "4g")
```

3. **Use Efficient File Formats:**
    1. Prefer **Parquet** over CSV.
    2. Use **Delta Lake** for upserts.
4. **Optimize Partitions:**

```
df = df.repartition(200)  # Increase partitions
```

```
df = df.coalesce(50)  # Reduce partitions if too many
```

6. **Use Column Pruning and Projection:**

```
df = df.select("id", "timestamp", "sales_amount")  # Avoid SELECT *
```

7. **Use `persist()` or `cache()` selectively:**

```
df.persist()
```

8. **Optimize Joins using Broadcast or Salting:**

```
9. from pyspark.sql.functions import broadcast
```

```
df = df.join(broadcast(small_df), "key")
```

Summary

Question	Concepts Used
Debugging Shuffle Issues	Broadcast, Partitioning, Spark UI
Fixing Data Skew	Salting, Partitioning, Broadcast Joins
Removing Duplicates	`dropDuplicates()`, `row_number()`, Delta Lake
Real-time Analytics	Kafka, Structured Streaming, Delta Lake
Memory Optimization	Partitioning, Efficient Formats, Caching

These optimizations ensure **high performance and scalability** for Walmart's data pipelines. 🚀