INTERVIEW QUESTIONS & WITH ANSWER

INTERVIEW QUESTIONS & WITH ANSWER

*****These questions cover real-world scenarios and key concepts to help you ace your next interview! *****

PySpark Basics and RDDs

INTERVIEW QUESTIONS & WITH ANSWER

Q1. What is the difference between RDD, DataFrame, and Dataset?

RDDs:-
A distributed collection of data elements without a schema. RDDs are slower than DataFrames and Datasets for simple operations.
DataFrames:-
A distributed collection organized into named columns.
DataFrames are similar to relational database tables or Python Pandas DataFrames.
DataFrames are faster than RDDs for exploratory analysis and creating aggregated statistics.
Datasets:-
An extension of DataFrames with additional features like type-safety and object-oriented interface. Datasets are faster than RDDs but slower than DataFrames. Datasets combine the performance optimization of DataFrames and the convenience of RDDs.

Q2. How does PySpark achieve parallel processing?

PySpark achieves parallel processing by leveraging Apache Spark's distributed computing architecture.

1. RDD/DataFrame Abstractions
2. Driver and Executors
3. Task Parallelism
4. Cluster Manager Integration
5. Lazy Evaluation & DAG
6. In-Memory Computation

PySpark achieves parallel processing by:

- Distributing data across partitions
- Executing tasks concurrently on worker nodes
- Managing resources via cluster managers
- Optimizing execution through DAG and lazy evaluation

Q3. Explain lazy evaluation in PySpark with a real-world analogy.

Lazy evaluation in PySpark means that transformations are not executed immediately when you define them. Instead, Spark waits until an action (like collect() or count()) is called to actually execute the transformations. This allows Spark to optimize the execution plan for better performance

Q4. What is SparkContext, and why is it important?

SparkContext is the entry point to using Spark functionality in PySpark (or Scala Spark). It represents the connection between your application and the Spark cluster.

Role Description

1. Initializes Spark Application: It sets up the environment and allows your app to use Spark's capabilities.

2. Connects to the Cluster: Manages communication with the Cluster Manager (e.g., YARN, Standalone).

3. Resource Allocation: Requests resources (executors, cores, memory) for your Spark jobs.

4. Job Submission: Submits jobs and coordinates RDD or DataFrame transformations/actions.

5. Fault Tolerance & Lineage: Keeps track of RDD lineage for fault recovery.

Q5. How do you handle large file processing in PySpark?

TechniquePurpose

1. Use Parquet/ORC Faster, more efficient reads
2. Partitioning Process only necessary data
3. Repartition/Coalesce Control parallelism and file count
4. Caching Save repeated computations
5. Filter Early Reduce input size
6. Avoid .collect () Prevent memory issues on driver
7. Broadcast small datasets Optimize joins

Q6. What is the difference between actions and transformations in PySpark?

Feature	Transformations	Actions
Execution	Lazy	Trigger execution
Return Type	New RDD/DataFrame	Result to driver or storage
Examples	map(), filter(), select()	count(), collect(), show()
Purpose	Define a computation plan	Execute and get results

Q7. How does Spark handle data partitioning in distributed environments?

Apache Spark uses data partitioning to divide large datasets into smaller chunks (called partitions) that can be processed in parallel across multiple nodes in a cluster.

A partition is a logical chunk of data stored in memory or disk. Each partition is processed by a single task in a single executor thread.

When you create an RDD from a file or collection, Spark partitions it automatically.

Spark also partitions DataFrames internally (default number: spark.sql.shuffle.partitions = 200).

Hash Partitioning: - Spark uses a hash function on a key to distribute rows evenly. Common in joins and aggregations.

Range Partitioning: - Data is divided into ordered ranges. Useful for ordered or skewed data.

Custom Partitioning: - You can define your own partition logic using a custom Partitioner (RDD only).

reduceByKey() ✅ Keys are grouped across partitions

join() ✅ Matching keys may be on different partitions

coalesce() ✅ (sometimes) Reduces number of partitions

repartition() ✅ Redistributes data evenly

Q8. Explain the concept of fault tolerance in PySpark?

Mechanism	Description
Lineage DAG	Rebuilds lost data by reapplying transformations
Task Retries	Failed tasks are retried automatically
Data Replication	Relies on storage layer (e.g., HDFS) for fault-tolerant reads
Checkpointing	Persist intermediate RDDs to reduce recomputation cost

Q9. How do you broadcast variables in Spark, and when should you use them?

In Spark, broadcast variables are used to efficiently share small read-only data (like lookup tables or configuration settings) with all worker nodes, without sending a copy for each task.

Q10. What are accumulators in PySpark, and how do they differ from broadcast variables?

Feature	Accumulators	Broadcast Variables
Purpose	Aggregation (e.g., counters, sums)	Share read-only data with executors
Mutable?	Tasks can **only add** values	Completely **read-only**
Access	**Driver only** can read value	All tasks can read it
Usage in Tasks	Write-only in workers	Read-only in workers
Common Use Cases	Metrics, debugging, counting conditions	Lookup tables, configs, small datasets

DataFrame and Dataset Operations

Q11. How do you perform data filtering using PySpark DataFrames?

SQL String df.filter("age > 25")
Column Functions df.filter(col("age") > 25)
Complex Logic df.filter((col("age") > 30) & (col("country") == "US"))
Pattern Match df.filter(col("name").like("A%"))

Q12. What is the difference between repartition () and coalesce (), and when would you use each?

Feature	repartition()	coalesce()
Operation	Full shuffle	Narrow dependency (no shuffle)

Change partitions	Increase or decrease	Only decrease
Cost	Expensive due to shuffle	Cheap, avoids shuffle
Use case	Improve parallelism, repartition by key	Reduce partitions, optimize output files

Q13. How do you handle missing or null values in PySpark?

Task	Function / Method	Description
Detect nulls	.filter(col.isNull())	Find rows with nulls
Drop rows with nulls	.dropna()	Remove rows with nulls
Fill nulls	.fillna()	Replace nulls with specified values
Impute values	Imputer (MLlib)	Replace nulls with mean/median
Replace in expressions	coalesce()	Use first non-null value

Q14. How can you add a new column to a DataFrame using withColumn()?

df_new = df.withColumn("country", lit("USA"))

Q15. How do you perform a left join between two DataFrames in PySpark?

spark.sql("select a.id,a.name,b.product from cust a **left join** prod b on a.id=b.id").show()

Q16. What are temporary views in PySpark, and how do they differ from global temporary views?

df.createOrReplaceTempView("temp_view")
spark.sql("SELECT * FROM temp_view WHERE age > 30").show()

Q17. How do you use window functions in PySpark for advanced analytics?

```
windowSpec =
Window.partitionBy("department").orderBy(col("salary").desc())
```

Q18. How can you register a UDF (User-Defined Function) in PySpark?

```
def to_uppercase(s):
    return s.upper()

from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

# Register the UDF with a return type
to_upper_udf = udf(to_uppercase, StringType())

spark.udf.register("to_uppercase_sql", to_uppercase, StringType())
# Dataframe
df.withColumn("name_upper", to_upper_udf(col("name"))).show()

df.createOrReplaceTempView("people")
spark.sql("SELECT name, to_uppercase_sql(name) AS name_upper FROM
people").show()
```

Q19. What is the difference between persist() and cache()?

Feature	cache()	persist()
Shortcut for	.persist(MEMORY_AND_DISK)	Not a shortcut, you specify StorageLevel
Storage control	No	Yes
Custom levels	No	✅ (e.g., MEMORY_ONLY, DISK_ONLY)
Use case	Default caching needs	Advanced control over storage behavior

Q20. How do you read and write data in Parquet, CSV, and JSON formats in PySpark?

Read:

```
df_parquet = spark.read.parquet("path/to/file.parquet")
df_csv = spark.read.option("header", "true").csv("path/to/file.csv")
df_json = spark.read.json("path/to/file.json")
```

Write:

```
df.write.mode("overwrite").parquet("path/to/output_parquet")
df.write.option("header",
"true").mode("overwrite").csv("path/to/output_csv")
df.write.mode("overwrite").json("path/to/output_json")
```

Spark SQL and Query Optimization

Q21. How do you run SQL queries on a DataFrame in PySpark?

```
df.createOrReplaceTempView("temp_view")
spark.sql("SELECT * FROM temp_view WHERE age > 30").show()
```

Register temp view createOrReplaceTempView("name")
Register global view createGlobalTempView("name")
Run SQL query spark.sql("SQL QUERY")

Q22. What is the purpose of Catalyst Optimizer in Spark SQL?

The Catalyst Optimizer is the query optimization engine used by Spark SQL. Its main goal is to automatically optimize queries to improve performance and efficiency

1. Logical Optimization
2. Physical Plan Optimization
3. Rule-Based and Cost-Based Optimization

Uses rule-based techniques (static transformations) and optionally cost-based optimization (CBO) to make smarter choices.
4. Extensibility
Automatic optimization (you don't need to tune manually)
Improved performance for complex SQL/DataFrame queries
Extensible for custom logic in enterprise environments

Q23. How do you handle schema inference when reading data from external sources?

PySpark tries to infer the schema by scanning the data when you use .option("inferSchema", "true") (mainly for CSV and JSON).

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

df = spark.read \
   .option("inferSchema", "true") \
   .json("path/to/file.json")
```

Manual Schema Definition (Recommended for Large Data)
You can define a StructType schema to explicitly specify data types and improve performance.

```
schema = StructType([
   StructField("name", StringType(), True),
   StructField("age", IntegerType(), True)
])
```

Parquet & ORC – Schema is Embedded
For Parquet and ORC files, schema is already embedded in the file format:

```
df = spark.read.parquet("path/to/file.parquet")
```

Q24. What are the different join types in Spark SQL, and when would you use each?

```
spark.sql("select a.id,a.name,b.product from cust a join prod b on a.id=b.id").show()
spark.sql("select a.id,a.name,b.product from cust a left join prod b on a.id=b.id").show()
spark.sql("select a.id,a.name,b.product from cust a right join prod b on a.id=b.id").show()
spark.sql("select a.id,a.name,b.product from cust a full join prod b on a.id=b.id").show()
spark.sql("select a.id,a.name from cust a LEFT ANTI JOIN  prod b on a.id=b.id").show()
spark.sql("select a.id,a.name from cust a LEFT SEMI JOIN  prod b on a.id=b.id").show()
spark.sql("select a.id,a.name from cust a CROSS JOIN  prod b").show()
```

Q25. How do you create a persistent table in Spark SQL?

This stores both the **data and schema** persistently in the metastore.

Q26. How does dynamic partition pruning improve query performance?

Static partition pruning: Prunes partitions before query starts (e.g., WHERE region = 'US').
DPP (Dynamic Partition pruning): Prunes partitions during execution, based on values coming from another table.

```
spark.conf.set("spark.sql.optimizer.dynamicPartitionPruning.enabled", "true")
```

Q27. Explain how to use broadcast joins to optimize query performance?

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

```
# 'small_df' is the smaller table, 'large_df' is the big one
joined_df = large_df.join(broadcast(small_df), "join_key")
```

Q28. What is data skew, and how do you handle it in Spark SQL?

1. Salting Keys

Add a random prefix/suffix ("salt") to skewed keys to spread the data across multiple partitions, then join on this salted key

```
from pyspark.sql.functions import concat, lit, floor, rand

# Add salt column to both DataFrames
df1_salted = df1.withColumn("salt", floor(rand() * 10))
df1_salted = df1_salted.withColumn("salted_key",
concat(df1_salted["join_key"], lit("_"), df1_salted["salt"]))

df2_salted = df2.withColumn("salt", floor(rand() * 10))
df2_salted = df2_salted.withColumn("salted_key",
concat(df2_salted["join_key"], lit("_"), df2_salted["salt"]))

# Join on salted_key instead of join_key
result = df1_salted.join(df2_salted, "salted_key")
```

2. Broadcast Join

If one table is small, broadcast it to avoid shuffling and reduce skew impact.

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

3. Increase Shuffle Partitions

Increase spark.sql.shuffle.partitions to spread skewed keys over more partitions.

```
spark.conf.set("spark.sql.shuffle.partitions", 500)
```

4. Skew Join Optimization (Spark 3.0+)

Spark 3+ supports adaptive query execution (AQE) with built-in skew join optimization that detects skew and splits large partitions automatically.
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")

5. Filter or Aggregate Early
Reduce skew by filtering or aggregating data before the join to minimize skewed keys.

Technique	When to Use
Salting	Manual control when you know skewed keys
Broadcast Join	When one table is small
Increase Shuffle Partitions	To increase parallelism
Adaptive Query Execution	Spark 3+ automatic skew handling
Early Filtering/Aggregation	Reduce skew data volume before join

Q29. How can you perform aggregations using SQL queries on large datasets?

When working with large datasets, aggregations are common operations like SUM, COUNT, AVG, MIN, MAX, GROUP BY, etc. Spark SQL is designed to handle these efficiently even at scale

Q30. How do you enable query caching in Spark SQL?

Method	How to Enable	When to Use
SQL CACHE TABLE	CACHE TABLE tableName;	Cache tables for repeated SQL queries
PySpark .cache()	df.cache()	Cache DataFrames in Spark applications
Persist with StorageLevel	df.persist(StorageLevel)	Customize cache storage behavior
AQE Cache	Set configs for adaptive query caching	Automatic optimization in Spark 3.2+

Data Pipeline Scenarios and Real World Use Cases

INTERVIEW QUESTIONS & WITH ANSWER

Q31. How would you build an ETL pipeline using PySpark?
Key ETL Steps with PySpark

1. Extract – Read data from external sources
2. Transform – Clean, filter, join, aggregate, enrich data
3. Load – Write the final dataset to storage (Parquet, S3, Hive, etc.)

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, to_date

# Step 1: Initialize Spark Session
spark = SparkSession.builder \
   .appName("ETL Pipeline Example") \
   .getOrCreate()

# Step 2: Extract - Load raw data from CSV
raw_df = spark.read.option("header", True).csv("s3://your-
bucket/raw/sales.csv")

# Step 3: Transform - Clean and prepare the data
transformed_df = raw_df \
   .withColumn("sales_amount", col("sales_amount").cast("double")) \
   .withColumn("date", to_date(col("date"), "yyyy-MM-dd")) \
   .filter(col("sales_amount") > 0)

# Optional: Join with product/dimension data
products_df = spark.read.parquet("s3://your-bucket/dim/products/")
final_df = transformed_df.join(products_df, on="product_id", how="left")

# Step 4: Load - Write cleaned data to S3 in Parquet format
final_df.write.mode("overwrite").partitionBy("date").parquet("s3://your-
bucket/processed/sales/")
```

```
# Stop Spark session
spark.stop()
```

Q32. How do you handle real-time data processing with Structured Streaming in PySpark?

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

# 1. Create Spark session
spark = SparkSession.builder \
   .appName("RealTimeETL") \
   .getOrCreate()

# 2. Read streaming data from Kafka
df = spark.readStream \
   .format("kafka") \
   .option("kafka.bootstrap.servers", "localhost:9092") \
   .option("subscribe", "sales_topic") \
   .load()

# 3. Transform - Parse Kafka value and extract fields
sales_df = df.selectExpr("CAST(value AS STRING) as json_data") \
   .selectExpr("from_json(json_data, 'product_id INT, sales_amount DOUBLE,
ts STRING') as data") \
   .select("data.*")

# 4. Optional aggregation
agg_df = sales_df.groupBy("product_id").sum("sales_amount")

# 5. Write to output sink (console or storage)
query = agg_df.writeStream \
   .outputMode("complete") \
```

```
    .format("console") \
    .option("truncate", "false") \
    .trigger(processingTime="10 seconds") \
    .start()

query.awaitTermination()
```

Triggers.

```
trigger(processingTime="10 seconds")  # every 10 seconds
.trigger(once=True)  # one-time batch for debugging
```

Fault Tolerance

```
.writeStream.option("checkpointLocation", "/tmp/checkpoints/")
```

Q33. What are the best practices for partitioning data in large datasets?

1. Partition by Frequently Queried Columns

```
df.write.partitionBy("country", "year").parquet("s3://your-bucket/sales/")
```

2. Avoid Over-Partitioning and Small Files

```
df.coalesce(10).write.parquet("path/")
```

3. Use Repartitioning for Data Shuffle Efficiency

```
df = df.repartition(100, "customer_id")
```

4. Use .coalesce() to Reduce Partition Count Before Writing

```
df.coalesce(1).write.parquet("path/")
```

5. Choose Cardinality Wisely

Avoid partitioning by high-cardinality columns like user_id or transaction_id — leads to too many tiny partitions.

Prefer low to medium cardinality columns like:
country,year,region,event_type

6. Use Bucketing for Efficient Joins
CREATE TABLE sales_bucketed
USING parquet
CLUSTERED BY (customer_id) INTO 100 BUCKETS;

7. Leverage Partition Pruning
SELECT * FROM sales WHERE year = 2024 AND country = 'US';

8. Monitor and Tune with Spark UI
Tune spark.sql.shuffle.partitions (default: 200)

Technique	When to Use	Benefit
partitionBy()	Writing data to storage	Enables pruning, efficient reads
repartition()	Before joins, increase parallelism	Improves shuffle-based ops
coalesce()	Before writing output	Combines partitions, fewer files
Bucketing	For repetitive joins on a key	Faster joins without reshuffling
Partition Pruning	Filtering on partition columns	Reads only required data

Q34. How would you debug and optimize a slow-running Spark job?

1. Check Spark UI
URL: Usually at http://<driver-node>:4040

2. Identify Expensive Operations
Common causes of slowness:
Wide transformations (e.g., join, groupBy, distinct)

Large shuffles (data moved between nodes)
Skewed data (some tasks take much longer)
df.explain(True)
df.queryExecution.debug.codegen()

3. Check for Data Skew
Use .groupBy("key").count().orderBy("count", ascending=False) to detect skewed

4. Broadcast Joins for Small Tables
df1.join(broadcast(df2), "id")

5. Optimize Shuffles
spark.conf.set("spark.sql.shuffle.partitions", 200) # default, increase or decrease based on data size

6. Cache/Persist Intermediate Results
df.cache() # or df.persist(StorageLevel.MEMORY_AND_DISK)
df.count() # trigger caching

7. Avoid Unnecessary Collect/Show
Use .limit(n).show() instead for sampling.

8. Tune Resource Allocation
--executor-memory 4G
--executor-cores 4
--num-executors 50

spark.conf.set("spark.dynamicAllocation.enabled", "true")

9. Enable Adaptive Query Execution (AQE)
spark.conf.set("spark.sql.adaptive.enabled", "true")

10. Profile with Spark History Server

http://<your-cluster>/history

Q35. How do you handle schema evolution in PySpark pipelines?

1. Enable Schema Merging (for Parquet/ORC)

```
df = spark.read.option("mergeSchema",
"true").parquet("s3://path/to/parquet/")
```

2. Use Delta Lake for Robust Schema Evolution

Delta Lake (on Databricks or open source) supports automatic schema evolution.

```
new_data.write \
  .format("delta") \
  .option("mergeSchema", "true") \
  .mode("append") \
  .save("/mnt/delta/sales/")
```

3. Infer Schema Dynamically (for Semi-Structured Data)

```
df = spark.read \
  .option("inferSchema", "true") \
  .json("s3://path/json/")
```

4. Define and Update Explicit Schemas

```
from pyspark.sql.types import StructType, StructField, StringType,
IntegerType

schema_v2 = StructType([
  StructField("id", IntegerType(), True),
  StructField("name", StringType(), True),
  StructField("email", StringType(), True)  # new column
])
df = spark.read.schema(schema_v2).json("path")
```

5. Handle Nulls and Defaults for Missing Fields

```
df = df.withColumn("email", coalesce(col("email"),
lit("unknown@example.com")))
```

6. Monitor and Validate Schema Changes

.schema.json() to save schema versions.

7. Backfill Historical Data (Optional)

If schema changes are breaking (e.g., renaming a column):
Consider backfilling historical data to the new schema.
Or maintain versioned data models (v1, v2 folders or tables).

Q36. What is the role of checkpointing in Spark Streaming?

Checkpointing in Spark Streaming (including Structured Streaming) is a critical mechanism that enables fault tolerance, state recovery, and state management during stream processing.

Types of Checkpointing in Spark

Metadata Checkpointing:-

Saves streaming job progress (e.g., offsets, batch IDs).Required for Structured Streaming.

Data Checkpointing:-

Saves the RDD lineage and data to avoid recomputation.
Mainly used in DStream-based Spark Streaming (less common now).

How to Enable Checkpointing in Structured Streaming

```
query = df.writeStream \
   .format("parquet") \
   .outputMode("append") \
   .option("checkpointLocation", "s3://my-bucket/checkpoints/job1/") \
```

```
    .start("s3://my-bucket/output/")
```

Checkpoint directory must be reliable and durable (e.g., HDFS, S3).

Purpose	Description
Fault Recovery	Recovers the stream from failures by storing metadata and data state
Stateful Operations	Required for operations like updateStateByKey, mapGroupsWithState
Progress Tracking	Tracks offsets (Kafka, file source), watermarks, and batch info

Q37. How can you implement incremental data processing in PySpark?

Common Strategies for Incremental Processing

1. Using Timestamps or Date Columns

Assumption: Your source data has a column like last_updated, created_at, or ingestion_date.

```
# Last processed timestamp, from metadata store
last_timestamp = "2025-05-20 00:00:00"

# Filter new/updated records
new_data = df.filter(df["last_updated"] > lit(last_timestamp))
```

2. Using Watermarking in Structured Streaming

```
df = spark.readStream \
    .format("kafka") \
    .option("subscribe", "orders") \
    .load()
```

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

3. Delta Lake's Change Data Feed (CDF) 🔁

If using Delta Lake, enable CDF to get only updated/new/deleted rows:

```
df = spark.read.format("delta") \
   .option("readChangeData", "true") \
   .option("startingVersion", 23) \
   .load("/delta/orders/")
```

4. Using Surrogate Keys or Auto-Increment IDs

```
last_processed_id = 10250
new_data = df.filter(df["id"] > last_processed_id)
```

Store the last processed ID externally.

Useful when data is strictly append-only.

5. Compare Against Existing Target Table (Merge)

Use merge (upsert) to load only new/changed rows into a target:

```
from delta.tables import DeltaTable
target = DeltaTable.forPath(spark, "/delta/customers/")

target.alias("t").merge(
   source=new_data.alias("s"),
   condition="t.id = s.id"
).whenMatchedUpdateAll() \
 .whenNotMatchedInsertAll() \
 .execute()
```

Real-World Use Case Example

Scenario: You want to process only new orders added daily to a Parquet file.

```
last_processed_date = "2024-05-19"

df = spark.read.parquet("s3://bucket/orders/")
incremental_df = df.filter(df["order_date"] > last_processed_date)

# Process and write
incremental_df.write.parquet("s3://bucket/processed_orders/",
mode="append")

# Update last_processed_date in metadata store
```

Q38. How do you handle large joins between multiple DataFrames?

1. Broadcast Joins (for Small Tables)

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

2. Repartition Before Join

Ensure both DataFrames are partitioned on the join key to reduce shuffle skew.

```
df1 = df1.repartition("join_key")
df2 = df2.repartition("join_key")
joined_df = df1.join(df2, "join_key")
```

3. Use Bucketing (For Hive Tables)

```
CREATE TABLE t1 (...) CLUSTERED BY (key) INTO 50 BUCKETS;
```

4. Avoid Cross Joins Unless Necessary

```
df1.crossJoin(df2).filter("df1.col = df2.col")
```

5. Skew Join Handling (When Keys Are Uneven)

Add a salt key (e.g., key + rand()) to spread out skewed data.
Use salting or enable AQE skew join handling (in Spark 3.0+):

```
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")
```

6. Join in Stages (Multi-Table Join Strategy)

```
temp = df1.join(df2, "key1")
result = temp.join(df3, "key2")
```

7. Use SQL for Complex Joins

```
df1.createOrReplaceTempView("orders")
df2.createOrReplaceTempView("customers")
df3.createOrReplaceTempView("products")

spark.sql("""
   SELECT o.*, c.name, p.name
   FROM orders o
   JOIN customers c ON o.cust_id = c.id
   JOIN products p ON o.prod_id = p.id
""")
```

8. Tune Configurations

Setting	Purpose
spark.sql.shuffle.partitions	Controls # of shuffle partitions
spark.sql.autoBroadcastJoinThreshold	Max size (bytes) for broadcast
spark.sql.adaptive.enabled	Enables Adaptive Query Execution
spark.sql.adaptive.skewJoin.enabled	Handles skewed joins automatically

Q39. What is the difference between batch processing and stream processing in Spark?

Feature	Batch Processing	Stream Processing
Nature of Data	Processes **finite/static** data	Processes **continuous/infinite** data
Input Source	Files (Parquet, CSV, etc.), DBs	Kafka, socket, files, etc.
Execution Mode	Runs as a **job**, ends when data is processed	Runs **continuously**, processing data in real time

Q40. How would you secure sensitive data in a PySpark pipeline?

1. Data Encryption

At Rest

Enable encryption on storage systems like:

Amazon S3 (SSE-S3, SSE-KMS)
HDFS transparent encryption
Azure Data Lake encryption

Use encrypted file formats like Parquet + GZIP/Snappy.

In Transit

Enable SSL/TLS when transferring data:
Between Spark and Kafka, S3, JDBC, etc.
Use spark.ssl.enabled for Spark encryption.

2. Masking and Tokenization

Use data masking to obscure sensitive fields (e.g., SSNs, emails):

```
from pyspark.sql.functions import sha2, col

df = df.withColumn("email_hash", sha2(col("email"), 256))
```

3. Column-Level Encryption (Custom)

Encrypt sensitive columns before writing:

```
from cryptography.fernet import Fernet


key = Fernet.generate_key()
cipher = Fernet(key)
@udf("string")
def encrypt(value):
    return cipher.encrypt(value.encode()).decode()
df = df.withColumn("ssn_encrypted", encrypt(col("ssn")))
```

Store encryption keys in a secure vault (e.g., AWS KMS, Azure Key Vault, HashiCorp Vault).

4. Access Control

Use Role-Based Access Control (RBAC):

On data storage (S3, ADLS, Hive, etc.)
On Databricks / Spark clusters

Apply fine-grained access control via:

Apache Ranger (for HDFS, Hive, etc.)
Unity Catalog (Databricks)

5. Auditing and Logging

Log:

Who accessed data
What data was read or written
When and where the access occurred

Use audit logs from:

Spark history server

Cloud providers (AWS CloudTrail, Azure Monitor)

Access gateways (e.g., Lake Formation)

6. Data Governance and Classification

Tag sensitive columns in metadata catalogs like:

AWS Glue Data Catalog
Apache Atlas
Unity Catalog (Databricks)

Define policies based on sensitivity level (e.g., PII, HIPAA).

7. DevSecOps Practices

Don't hardcode credentials in scripts.

Use secrets managers:

spark.conf.set("spark.hadoop.fs.s3a.access.key", ...) via environment vars or secret scopes.

Encrypt logs and control log verbosity.

Security Measure	Technique / Tool
Encryption (at rest)	S3/KMS, HDFS encryption
Encryption (in transit)	TLS/SSL in Spark, Kafka, JDBC
Data masking/tokenizing	sha2(), custom UDFs
Access control	RBAC, Apache Ranger, Unity Catalog
Auditing	Cloud logs, Spark audit logs
Secrets management	AWS Secrets Manager, Databricks secrets

Advanced PySpark Features

Q41. How do you handle large datasets in PySpark to optimize performance and reduce memory usage?

1. Use Efficient Data Formats

Parquet or ORC are columnar storage formats optimized for Spark.
They provide better compression and faster I/O compared to formats like CSV or JSON.
df.write.parquet("path/to/output.parquet")

2. Partitioning
Use repartition(n) to increase partitions (e.g., after a wide transformation).
Use coalesce(n) to reduce the number of partitions (e.g., before writing).

df = df.repartition(100, "col1") # Better parallelism
df = df.coalesce(10) # Reduce shuffles before write

3. Cache and Persist
Cache intermediate DataFrames if reused multiple times to avoid recomputation.
Use .cache() or .persist(storage_level) only when needed.
df.persist(StorageLevel.MEMORY_AND_DISK)

4. Avoid Wide Transformations
Wide transformations (like groupByKey, join, distinct, repartition) trigger shuffling.
Prefer reduceByKey or aggregateByKey instead of groupByKey.

rdd.reduceByKey(lambda x, y: x + y) # More efficient than groupByKey

5. Use Broadcast Join
If one dataset is small, broadcast it to all nodes to avoid shuffle-heavy joins.
from pyspark.sql.functions import broadcast
df = large_df.join(broadcast(small_df), "key")

# Create a small DataFrame to broadcast
small_df = spark.read.csv("small_dataset.csv", header=True, inferSchema=True)
broadcast_small_df = spark.sparkContext.broadcast(small_df.collect())
# Use broadcast variable in a join
large_df = spark.read.csv("large_dataset.csv", header=True, inferSchema=True)
joined_df = large_df.join(small_df, "key_column")

6. Column Pruning & Filter Pushdown
Read only required columns and apply filters early using predicate pushdown.
spark.read.parquet("path").select("col1", "col2").filter("col1 > 100")

7. Avoid Collecting Large Data to Driver
Avoid using .collect() or .toPandas() on large datasets as it can crash the driver.
Use .show(), .take(n) or .limit(n) for previewing.
df.limit(10).toPandas()

8. Optimize Joins
Ensure the join keys are distributed and avoid skewed joins.
Use salting or skew join hints when facing data skew.
df1.join(df2.hint("skew"), "key")

9. Use UDFs Wisely
Avoid Python UDFs due to serialization and performance overhead.
Prefer Spark built-in functions (pyspark.sql.functions) or Pandas UDFs.

```
from pyspark.sql.functions import col, upper
df = df.withColumn("name_upper", upper(col("name")))
```

10. Resource Tuning
Tune Spark configuration:

```
--executor-memory 4G
--executor-cores 4
--num-executors 10

# Example of configuring Spark settings in the SparkSession
spark = SparkSession.builder \
 .appName("OptimizationExample") \
 .config("spark.executor.memory", "4g") \
 .config("spark.executor.cores", "4") \
 .config("spark.driver.memory", "4g") \
 .getOrCreate()


# Example of caching and partitioning
df = spark.read.csv("data.csv", header=True, inferSchema=True)  # Read data
df.cache()  # Cache the DataFrame
df_partitioned = df.repartition(numPartitions=100,
partitioningColumn="key_column")  # Repartition
```

Q42. What is the purpose of Delta Lake, and how does it improve reliability?

INTERVIEW QUESTIONS & WITH ANSWER

Delta Lake is an open-source storage layer that brings ACID transactions, schema enforcement, and time travel to big data workloads on Apache Spark and data lakes (like S3, ADLS, etc.).

Purpose of Delta Lake

- Reliable, scalable big data pipelines
- Transactional consistency on top of distributed storage
- Unified batch and streaming data processing

1. ACID Transactions
Ensures atomicity, consistency, isolation, and durability even across multiple writers

```
df.write.format("delta").mode("append").save("/path/to/delta-table")
```

2. Schema Enforcement & Evolution
Prevents bad data from corrupting tables with strict schema checks.
Supports schema evolution (e.g., adding new columns).

```
spark.read.format("delta").load("/path").printSchema()
```

3. Time Travel
Access previous versions of data using versioning or timestamps.
Useful for debugging, rollback, and reproducibility.

```
delta_table = DeltaTable.forPath(spark, "/path")
delta_table.history()  # Show all versions
spark.read.format("delta").option("versionAsOf", 3).load("/path")
```

4. Unified Batch + Streaming
Enables a single table to support both streaming reads and batch writes, improving consistency across pipelines.

```
spark.readStream.format("delta").load("/path")
```

5. Data Quality with Constraints
You can define constraints like NOT NULL, CHECK, etc.
Ensures data correctness at the write level.

6. Efficient Upserts and Deletes (MERGE)
Simplifies slow-changing dimension updates and deduplication.

```
from delta.tables import DeltaTable

deltaTable = DeltaTable.forPath(spark, "/path")
deltaTable.alias("target").merge(
    source_df.alias("source"),
    "target.id = source.id"
).whenMatchedUpdateAll().whenNotMatchedInsertAll().execute()
```

7. Scalable Metadata Handling
Delta Lake uses transaction logs (stored as _delta_log) rather than relying on file listings, making it scalable for tables with millions of files

Q43. How do you enable time travel queries using Delta Lake?

Delta Lake allows you to query past versions of a table using:

versionAsOf — specify a version number
timestampAsOf — specify a timestamp

1. Using versionAsOf

```
df = spark.read.format("delta") \
    .option("versionAsOf", 5) \
    .load("/path/to/delta-table")
```

2. Using timestampAsOf

```
df = spark.read.format("delta") \
    .option("timestampAsOf", "2024-05-20T10:00:00") \
    .load("/path/to/delta-table")
```

3. View Table History

from delta.tables import DeltaTable

delta_table = DeltaTable.forPath(spark, "/path/to/delta-table")
delta_table.history().show(truncate=False)

4. Notes
Delta Lake stores all changes as incremental commits in the _delta_log/ directory.
Older data is retained by default for 30 days, but this is configurable with the data retention period
spark.databricks.delta.retentionDurationCheck.enabled = false

5. Optional: Clean Up Old Versions
# Remove files no longer needed for time travel (older than 7 days)
spark.sql("VACUUM delta.`/path/to/delta-table` RETAIN 168 HOURS")

Q44. How do you handle complex aggregations using window functions?

1. Running Totals / Cumulative Sum
from pyspark.sql.window import Window
from pyspark.sql.functions import sum

window_spec =
Window.partitionBy("customer_id").orderBy("transaction_date") \
 .rowsBetween(Window.unboundedPreceding,
Window.currentRow)
df = df.withColumn("running_total", sum("amount").over(window_spec))

2. Moving Average

```
from pyspark.sql.functions import avg

window_spec =
Window.partitionBy("customer_id").orderBy("transaction_date") \
            .rowsBetween(-2, 0)  # 3-day moving average
df = df.withColumn("moving_avg", avg("amount").over(window_spec))
```

3. Row Number / Ranking / Dense Ranking

```
from pyspark.sql.functions import row_number, rank, dense_rank

window_spec = Window.partitionBy("category").orderBy("sales")

df = df.withColumn("row_num", row_number().over(window_spec)) \
    .withColumn("rank", rank().over(window_spec)) \
    .withColumn("dense_rank", dense_rank().over(window_spec))
```

4. Lag/Lead for Value Comparison

```
from pyspark.sql.functions import lag, lead

window_spec = Window.partitionBy("user_id").orderBy("event_time")

df = df.withColumn("prev_val", lag("score", 1).over(window_spec)) \
    .withColumn("next_val", lead("score", 1).over(window_spec))
```

5. Detecting Change Points or Gaps

```
from pyspark.sql.functions import col, lag, when

window_spec = Window.partitionBy("user_id").orderBy("event_time")
df = df.withColumn("prev_status", lag("status").over(window_spec)) \
    .withColumn("status_changed", when(col("status") != col("prev_status"),
1).otherwise(0))
```

6. First and Last Value

```
from pyspark.sql.functions import first, last
```

```
window_spec = Window.partitionBy("department").orderBy("date")

df = df.withColumn("first_sale", first("sale").over(window_spec)) \
    .withColumn("last_sale", last("sale").over(window_spec))
```

Function	Description
`row_number()`	Unique row number per partition
`rank()`	Ranking with gaps
`dense_rank()`	Ranking without gaps
`lag()`	Value from a previous row
`lead()`	Value from a following row
`sum()`	Cumulative or windowed sum
`avg()`	Moving or group average
`first()`	First value in the window
`last()`	Last value in the window

Q45. What are stateful operations in Spark Structured Streaming?

Key Characteristics of Stateful Operations
State is maintained in memory and periodically checkpointed to ensure fault tolerance.
Requires watermarks and timeout configurations to prevent unbounded state growth.
Involves grouping, windowing, or matching events over time.

Examples of Stateful Operations
1. Group-based Aggregations (with Time Window)

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

df.groupBy(
```

```
    window("event_time", "10 minutes"),
    "user_id"
).agg(sum("amount"))
```

2. Streaming Joins (between two streams)

```
stream1.join(stream2, "id")  # Requires watermarking
```

3. FlatMapGroupsWithState

```
from pyspark.sql.functions import expr
from pyspark.sql.streaming import GroupState, GroupStateTimeout

def update_state(user_id, inputs, state: GroupState):
    # custom logic here
    return ...

df.groupByKey(lambda row: row.user_id).flatMapGroupsWithState(
    update_state,
    outputMode="update",
    stateTimeoutDuration="10 minutes"
)
```

4. Deduplication

```
df.dropDuplicates(["user_id", "event_time"])
```

5. Role of Watermarking

Watermarking helps limit state size by specifying the maximum expected lateness of data.

```
df.withWatermark("event_time", "15 minutes")
```

Q46. How do you implement error handling and retries in PySpark jobs?

INTERVIEW QUESTIONS & WITH ANSWER

Implementing robust error handling and retry logic in PySpark jobs is essential for production-grade data pipelines. Here's how you can structure it across different components of a PySpark job:

1. Use Try-Except Blocks in Driver Code

```
try:
   df = spark.read.parquet("/input/path")
   result = df.groupBy("category").count()
   result.write.mode("overwrite").parquet("/output/path")
except Exception as e:
   print(f"Job failed: {e}")
   # Optionally send alert or write error to log
```

2. Implement Retries with Exponential Backoff

```
import time
import random

def retry_operation(func, retries=3):
   for i in range(retries):
      try:
         return func()
      except Exception as e:
         print(f"Retry {i + 1} failed: {e}")
         time.sleep(2 ** i + random.random())  # exponential backoff
   raise Exception("All retries failed.")
```

3. Handle Errors in UDFs Carefully

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

def safe_parse(value):
   try:
      return complex_parsing_logic(value)
   except:
      return None  # or "error"
```

```
safe_parse_udf = udf(safe_parse, StringType())
df = df.withColumn("parsed", safe_parse_udf(df["raw_col"]))
```

4. Use Accumulators or Logs for Error Tracking

```
from pyspark.accumulators import AccumulatorParam

error_count = spark.sparkContext.accumulator(0)

def parse_and_count(value):
    try:
        return int(value)
    except:
        error_count.add(1)
        return None

udf_parse = udf(parse_and_count)
df = df.withColumn("parsed", udf_parse(df["col"]))
```

5. Validate Data Early

```
expected_schema = StructType([...])
df = spark.read.schema(expected_schema).json("/data/path")

if df.filter("col IS NULL").count() > 0:
    raise ValueError("Null values found in critical column.")
```

6. Checkpoints and Recovery for Streaming

```
query.writeStream \
    .format("delta") \
    .option("checkpointLocation", "/checkpoints/stream1") \
    .start("/output/path")
```

7. Leverage Workflow Orchestration for Retries

```
PythonOperator(
    task_id='spark_job',
    python_callable=run_spark_job,
    retries=3,
    retry_delay=timedelta(minutes=2),
)
```

Area	Strategy
Driver code	Try-except with logging
External systems	Retry with exponential backoff
UDFs	Safe exception handling inside logic
Streaming	Use check pointing and watermarking
Data quality	Validate schema and critical fields early
Workflow orchestration	Handle retries and notifications externally

Q47. How do you monitor and manage Spark clusters using Spark UI?

The Spark UI is a web-based tool that provides detailed insights into:

Job and stage execution
Task-level metrics
Memory usage
Storage and caching
Executors
SQL query plans

1. Local Mode or Standalone Cluster
Default URL: http://localhost:4040

2. YARN
In YARN mode, the Spark UI is linked from the YARN ResourceManager UI under the "Tracking URL".

3. Databricks
Available as part of the "Spark UI" tab inside each job/run.

Key Spark UI Tabs
1. Jobs
2. Stages
3. Tasks
4. Storage
5. Environment
6. Executors
7. SQL (if using Spark SQL)

Q48. What is the difference between SparkSession and SparkContext?

Use SparkSession in modern Spark applications (especially with DataFrames, SQL, Delta Lake, etc.).
Use SparkContext only when working directly with RDDs or for low-level operations.

Feature	`SparkContext`	`SparkSession`
Introduced in	Spark 1.x	Spark 2.0
Purpose	Entry point for low-level RDD APIs	Unified entry point for all Spark APIs
Supports RDD	YES	YES (via `spark.sparkContext`)
Supports DataFrames	NO	YES
Supports SQL	NO	YES
Encapsulates	N/A	`SparkContext`, `SQLContext`, `HiveContext`
Recommended in	Legacy RDD-based code	Modern Spark apps (especially DataFrame-based)

Q49. How do you handle late-arriving data in Spark Structured Streaming?

INTERVIEW QUESTIONS & WITH ANSWER

Handling Late Data (Event - Time Processing)
Causes of Late - Arriving Data in Kafka
1. Network and Producer Delays: high network latency. Resource connections or retries in the producer can delay message delivery.
2. Broker overload: overloaded kafka brokers or slow replication can introduce processing delays.
3. Upstream Delays "latency in upstream systems or IoT devices can make events arrive late in kafka.

Event Time vs Processing Time in Structured Streaming
Time Ranges:
Start Time Getdate()
End Time Getdate()

Example Kafka Message:

```
{
"timestamp": "2025-11-22T08:52:10.000+00:00",
"userid" : "123",
"item" : "headphones",
"quantity" : 1
}
```

Understanding the State Store in Spark Structured Streaming
1. What is aState Dtore?
The State Store is a key-value store used by Spark to persist and manage the state for each micro-batch in a streaming query. This state is updated with each batch and saved for future use.

Example Use Case:
>> in a windowed aggregation the state store keeps track of partial results for each window until the window closes.
2. How it Works

Each streaming query operates in micro-batches , following these steps:

1) Input Data: Data is read and processed.
2) Query Existing State : the state store is queried for existing state.
3) State Update: The state is updated based on the new data.
4) Output Results : Results are written to the output sink.
5) Persist State : The updated state is saved for use in subsequent micro-batches.

>>Automatic State Cleanup:
* Spark automatically removes Old state based on the watermark, which defines when data is considered late and no longer affects the state.

provider_class=spark.conf.get("spark.sql.streaming.statestore.providerclass")

>> Background : What is RockDB in Spark Structured Streaming??
RockDB is an embedded key-value store designed for high-performance reads and writes

In the context of Spark Structured Streaming , it serves as a powerful alternative to the default file-based state store.By leveraging RocksDB, Spark can significantly boost the performance of stateful computations like aggregations and joins , particularly under heavy workloads.

this is achieved by minimizing disk I/O overhead,making RockDB an excellent choice for handling large states or high-throughput streaming queries.

Q50. What is the difference between Spark's Catalyst Optimizer and Tungsten Execution Engine?

Catalyst Optimizer — Logical & Query Optimization Layer

INTERVIEW QUESTIONS & WITH ANSWER

Purpose: Optimizes the logical and physical execution plans of Spark SQL queries.

Layer: Query Optimization (part of the planning phase).

Written in: Scala, using functional programming concepts and pattern matching.

Key Features:

Rule-based and cost-based optimization: Applies transformations like predicate pushdown, constant folding, projection pruning, etc.

Logical Plan → Optimized Logical Plan → Physical Plan → Executable Plan

Supports user-defined optimizations and extensibility via rules.

Abstracts SQL, DataFrame, and Dataset APIs into a unified optimization flow.

INTERVIEW QUESTIONS & WITH ANSWER

Tungsten Execution Engine — Physical Execution Layer

Purpose: Provides low-level, memory-efficient execution of the query plan.

Layer: Execution Engine (part of the runtime phase).

Introduced in: Spark 1.4+ for improved performance.

Key Features:

Whole-stage code generation (WSG): Compiles parts of the query into Java bytecode to avoid virtual function calls and for-loop overhead.

Off-heap memory management: Reduces garbage collection overhead.

Cache-friendly and CPU-efficient algorithms

Improves performance by using binary processing and vectorization.

Feature	Catalyst Optimizer	Tungsten Execution Engine
Function	Query planning and optimization	Efficient physical query execution
Phase	Compile-time (Planning)	Run-time (Execution)

Optimizes	Logical and physical plans	Memory usage, CPU efficiency
Techniques Used	Rule-based and cost-based optimization	Whole-stage codegen, off-heap memory
Target	SQL, DataFrame, Dataset	JVM bytecode, CPU, memory

Bonus: Practical Coding Challenges

💻 Challenge 1: Write a PySpark function to remove duplicate rows from a DataFrame based on specific columns.

💻 Challenge 2: Create a PySpark pipeline to read a CSV file, filter out rows with null values, and write the result to a Parquet file.

💻 Challenge 3: Implement a window function to rank salespeople based on total sales by region. 💻 Challenge 4: Write a PySpark SQL query to calculate the average salary by department, including only employees with more than 3 years of experience.

💻 Challenge 5: Implement a PySpark function to split a large DataFrame into smaller DataFrames based on a specific column value.

Quick Tips for Interviews

INTERVIEW QUESTIONS & WITH ANSWER

Tip 1: Be ready to explain real-world scenarios where you've used PySpark.

Tip 2: Know how to optimize Spark jobs using caching, partitioning, and broadcasting.

Tip 3: Understand the trade-offs between RDDs, DataFrames, and Datasets.