1️⃣ What's the difference between a DataFrame and an RDD in PySpark?

Answer:

- **RDD (Resilient Distributed Dataset):**
  - Low-level API that provides fine-grained control over data and transformations.
  - Immutable and distributed collection of objects.
  - No schema — purely objects or records.
  - Transformations are functional (e.g., map, flatMap, filter).
  - More code required to express data manipulations.
- **DataFrame:**
  - High-level abstraction built on top of RDDs.
  - Similar to a table in a relational database, with schema (column names and types).
  - Optimized by Catalyst optimizer.
  - Supports SQL-like operations (select, groupBy, filter, etc.).
  - Easier to use and significantly faster for structured data.

✅ **Use DataFrames** when dealing with structured data — cleaner syntax and better performance due to Catalyst & Tungsten engines.

2️⃣ How do you optimize PySpark code for performance in production pipelines?

Answer:

Some key PySpark optimization strategies include:

- **Broadcast joins:** When one table is small (<10MB), use broadcast() to prevent shuffle.
- **Avoid shuffles:** Use mapPartitions, reduceByKey over groupByKey.
- **Repartitioning:** Use repartition() or coalesce() wisely to balance load.
- **Caching:** Use cache() or persist() if a DataFrame is reused.
- **Avoid UDFs:** Prefer Spark built-in functions for better optimization.
- **Column pruning & predicate pushdown:** Leverage .select() and .filter() early.

- **Monitor with Spark UI:** Identify slow stages, skewed tasks, GC overheads.

🎯 Always test performance on large datasets in staging before production deployment.

3️⃣ How does the Catalyst Optimizer impact query execution?

Answer:

The **Catalyst Optimizer** is the core of Spark SQL's execution engine.

- **Optimizes logical plans** into physical plans before execution.
- Performs:
    - Constant folding
    - Predicate pushdown
    - Column pruning
    - Join reordering
    - Expression simplification
- Converts transformations into **efficient execution strategies** (e.g., broadcast joins).
- Works with both DataFrame and Spark SQL APIs.

💡 This leads to **faster execution** and **fewer resources consumed** — without changing user code.

4️⃣ Common serialization formats used in PySpark — and why?

Answer:

Serialization is key for performance in distributed systems like Spark.

- **Java Serialization:**
    - Default in older Spark versions.
    - Slower and larger in size.
- **Kryo Serialization:**
    - More compact and faster than Java.
    - Needs to register custom classes for best performance.
    - Enabled via:

```
spark.conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
```

✅ **Use Kryo** for complex object graphs or when serialization overhead becomes a bottleneck.

5️⃣ How do you fix skewed data issues in large datasets?

Answer:

Skew happens when some partitions have significantly more data than others, causing imbalance.

To fix:

- **Salting the keys:** Add a random prefix to skewed join keys to distribute load, then remove after join.
- **Skew hint in Spark 3.0+:**

```
df.hint("skew")
```

- **Broadcast joins:** Broadcast the small side of the join.
- **Repartitioning:** Use repartition() before join to redistribute.
- **Custom partitioning:** Use hash-based or range partitioning to spread records evenly.

🧠 Always use **Spark UI** to detect skew — look for slow/stalled tasks.

6️⃣ Explain memory management in PySpark.

Answer:

PySpark runs on top of the JVM, so memory management depends on **Spark's unified memory management model**.

- **Memory is split into two areas:**
    1. **Execution Memory** – Used for shuffles, joins, aggregations.
    2. **Storage Memory** – Used for caching and broadcasting.
- **Tunable Configs:**
    - spark.executor.memory: Total memory per executor.
    - spark.memory.fraction: Fraction of heap used for execution + storage (default 0.6).

- spark.memory.storageFraction: Fraction reserved for storage within the above.

💡 Proper tuning helps prevent **OOM (Out of Memory)** errors and **GC overhead** issues.

7️⃣ List all types of joins in PySpark and when to use them.

Answer:

PySpark supports the following types of joins:

Join Type	Description	Use Case
inner	Matches records from both tables based on join key	Default and most used
left / left_outer	Keeps all records from left, matching from right	When left table is primary
right / right_outer	Keeps all records from right, matching from left	When right table is primary
full / full_outer	Returns all records from both sides, NULLs for unmatched	For full reconciliation
left_semi	Returns only left table records that have a match in right	For filtering existence
left_anti	Returns only left table records with no match in right	For exclusion filters
cross	Cartesian join (all combinations)	Used with caution; high cost

📌 Syntax Example:

```
df1.join(df2, on="id", how="left")
```

8️⃣ What is broadcast() in PySpark and when should you use it?

Answer:

broadcast() is used to **replicate a small DataFrame across all worker nodes** to avoid shuffle during join.

✅ **Use when:**

- One side of join is small (<10MB)
- You want to avoid shuffling large datasets

📌 Example:

```
from pyspark.sql.functions import broadcast

df1.join(broadcast(df2), "id")
```

🚫 Avoid using on large datasets — may cause memory issues.

9️⃣ How do you define and use UDFs (User-Defined Functions) in PySpark?

Answer:

UDFs allow you to **apply custom Python logic** to DataFrame columns.

📌 Example:

```
from pyspark.sql.functions import udf

from pyspark.sql.types import IntegerType


def square(x):

    return x * x


square_udf = udf(square, IntegerType())

df = df.withColumn("square_val", square_udf("num"))
```

⚠️ **Downsides:**

- Breaks Catalyst optimization
- Slower than using built-in functions

✅ Prefer Spark SQL functions (F.col, F.when, etc.) over UDFs where possible.

🔟 What is lazy evaluation in PySpark? How does it affect job execution?

Answer:

Lazy evaluation means **Spark doesn't compute anything until an action is triggered.**

- Transformations (like filter, select, withColumn) are **recorded as lineage**.
- Only when you call an **action** (show(), count(), write(), etc.), Spark builds a DAG and executes.

✅ **Advantages:**

- Optimization via Catalyst
- Reduces redundant computations

📌 Helps Spark **optimize the job plan** and **combine stages for better performance**.

1️⃣1️⃣ Steps to create a DataFrame in PySpark?

Answer:

You can create DataFrames in 3 ways:

✅ **From list/dict:**

data = [("Alice", 25), ("Bob", 30)]

df = spark.createDataFrame(data, ["name", "age"])

✅ **From RDD:**

rdd = spark.sparkContext.parallelize(data)

df = rdd.toDF(["name", "age"])

✅ **From file:**

df = spark.read.csv("file.csv", header=True, inferSchema=True)

1️⃣2️⃣ What is an RDD (Resilient Distributed Dataset)?

Answer:

RDD is the **core abstraction of Spark** representing a distributed collection of objects.

- Immutable and fault-tolerant
- Supports low-level transformations (map, flatMap, filter)
- Operates in memory (but can spill to disk)

✅ Useful when:

- You need fine-grained control
- Working with unstructured or complex transformations

🧠 DataFrames and Datasets internally use RDDs.

1️⃣3️⃣ Difference between transformations and actions in PySpark?

Answer:

Transformations	Actions
Return a new RDD/DataFrame	Trigger computation
Lazy evaluated	Immediate execution
Examples: filter(), map(), select()	Examples: show(), count(), collect()

✅ **Transformations** build lineage; **Actions** submit the job to Spark engine.

1️⃣4️⃣ How to handle null values in DataFrames?

Answer:

Use PySpark functions to handle nulls:

✅ **Drop rows with nulls:**

```
df.dropna()
```

✅ **Fill nulls:**

```
df.fillna({"age": 0, "city": "Unknown"})
```

✅ **Filter nulls:**

```
df.filter(df["age"].isNotNull())
```

✅ **Replace with default values using when:**

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

df.withColumn("age", when(col("age").isNull(), 0).otherwise(col("age")))
```

1️⃣5️⃣ What is a partition in PySpark? How to optimize it?

Answer:

A **partition** is the smallest unit of parallelism in Spark — a subset of data processed by a single task on an executor.

🔹 **Why it matters:**

- More partitions → better parallelism (up to a point)
- Too few → underutilization
- Too many → overhead

✅ **Optimization Tips:**

- Use .repartition(n) for even distribution and shuffles
- Use .coalesce(n) for reducing number of partitions without shuffle (useful before writing)
- Use partitioning on columns (.partitionBy("col")) when writing to disk for faster reads

---

1️⃣6️⃣ Difference between narrow vs wide transformations?

Answer:

Type	Description	Example	Shuffling?
Narrow	Data required for transformation is in a single partition	map(), filter()	❌ No
Wide	Requires data from multiple partitions (involves shuffle)	groupByKey(), join(), reduceByKey()	✅ Yes

✅ **Narrow transformations** are more efficient
🚫 **Wide transformations** involve **shuffles**, which are costly

---

1️⃣7️⃣ How does PySpark infer schema and why it matters?

Answer:

PySpark **infers schema automatically** using data types from input sources (CSV, JSON, etc.) if inferSchema=True.

✅ **Why it's important:**

- Helps Spark understand column types (int, float, string)
- Enables optimizations via Catalyst

- Without schema, all columns are treated as strings → errors in processing

📌 **Best practice:**
Manually define schema for large files or production jobs to avoid incorrect inference and improve performance.

1️⃣ 8️⃣ What is the role of SparkContext in a PySpark app?

Answer:

SparkContext is the **entry point** for Spark functionality.

🔹 It connects to the cluster manager and initializes:

- Executors
- Jobs
- Tasks
- RDDs

📌 In PySpark, SparkSession is built on top of SparkContext:

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

sc = spark.sparkContext
```

✅ It is responsible for:

- Job coordination
- Resource allocation
- Communication with cluster

1️⃣ 9️⃣ How do you perform aggregations in PySpark efficiently?

Answer:

✅ Use **groupBy + agg**:

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

df.groupBy("region").agg(sum("sales"), avg("profit"))
```

✅ Use **reduceByKey** for RDDs when keys are already paired

✅ Use **window functions** for rolling/sessional aggregations

✅ **Avoid groupByKey()** – it causes full shuffle and high memory usage

📌 Partition data properly and **cache** intermediate steps if reused

---

2️⃣0️⃣ When & how do you cache or persist data in PySpark?

Answer:

Use .cache() or .persist() to avoid recomputation of expensive transformations.

Method	Description
cache()	Stores in memory (MEMORY_AND_DISK by default)
persist(StorageLevel)	Allows control: MEMORY_ONLY, DISK_ONLY, etc.

✅ **Use when:**

- Same DataFrame is reused multiple times
- Heavy transformations
- During iterative algorithms (e.g., ML, joins)

📌 **Tip:** Always monitor memory with Spark UI — caching too much can evict useful data.