🔹 Q1. How do you tune executor and driver configuration for a Spark job?

Goal:
Evaluate your understanding of cluster resource utilization — especially executor sizing, memory vs core trade-offs, and GC behavior.

L4 Expectation:

- You know defaults aren't optimal.
- You reason using:
    - Number of partitions
    - Shuffle size
    - Caching needs
    - GC % and Spark UI stats

What to Cover:

- `spark.executor.instances`, `executor.memory`, `executor.cores`, `driver.memory`
- Sweet spot: 4–5 cores/executor, 8–16 GB memory
- When to reduce memory (to reduce GC stalls)

Quantified Answer Example:
"Reducing cores/executor from 8 to 4 decreased GC time from 38% to 8% and job time from 55 min to 27 min."

🔹 Q2. What is task serialization/deserialization in Spark and how does it impact performance?

Goal:
Check if you understand Spark's internal communication overhead.

L4 Expectation:

- You differentiate between Java and Kryo serializers
- You optimize by avoiding deep nesting, using broadcast variables, etc.

What to Cover:

- Why Kryo is 10x faster
- How task deserialization shows up in Spark UI
- When to register classes

Quantified Answer:
"Switching to Kryo reduced task deserialization time from 1.5s to 0.3s and improved CPU usage by 30%."

🔹 Q3. How does Spark manage memory internally and how can you tune it?

Goal:
See if you know about Spark's **unified memory model** and how execution vs storage memory is split.

L4 Expectation:

- You know how `spark.memory.fraction` works
- You recognize symptoms of poor memory tuning (e.g., excessive spilling, cache eviction)

What to Cover:

- 60% of JVM heap used by unified memory
- Further split: execution vs storage
- Tuning `memory.fraction`, `memory.storageFraction`

Quantified Example:
"Cache hit ratio improved from 64% to 92% after tuning memory fraction. Disk spill reduced from 3.2 GB to 400 MB."

🔹 Q4. How do you tune garbage collection in Spark executors?

Goal:
Evaluate JVM GC understanding and its impact on Spark performance.

L4 Expectation:

- You've analyzed GC time in Spark UI
- You can explain G1GC vs CMS
- You know large heap + many cores = longer GC pauses

What to Cover:

- Use G1GC for large memory jobs
- Lower executor memory → more stable GC
- Enable `GC logs` for tuning

Quantified Answer:
"GC overhead dropped from 30% to 7% by switching to G1GC and resizing executor memory."

Tactic	Why It Works
✅ Switch to G1GC	Handles large heaps better, parallel GC threads
✅ Reduce executor memory	Less heap = shorter GC pause time
✅ Limit executor cores to 4–5	Fewer concurrent tasks → better GC behavior
✅ Avoid nested objects/UDF leaks	Fewer objects = less GC pressure
✅ Enable GC logging	To measure and debug actual GC overhead

🔹 Q5. DataFrame vs RDD vs SQL performance — when to use what?

Goal:
Test your understanding of **Catalyst optimization vs manual control**.

L4 Expectation:

- You favor DataFrames/SQL for most cases
- You use RDDs only when needed (e.g., custom partitioner, low-level control)

What to Cover:

- Catalyst: logical → physical plan optimization
- DF/SQL > RDD in most ETL jobs
- Avoid RDDs unless DataFrame is too limited

Quantified Example:
"Same pipeline using SQL ran in 12 mins, vs 32 mins with RDDs due to Catalyst optimizations."

✅ 1. Predicate Pushdown = Filter Pushdown (Same Thing)

🔍 Definition:

> Spark **pushes filtering conditions** (WHERE clauses) **down to the storage layer** (like Parquet/ORC), so only matching **rows** are read from disk.

🔹 Example:

python

CopyEdit

```
df = spark.read.parquet("user_data/")

df.filter("age > 30").select("name").show()
```

Instead of:

- Reading **all rows**
- Then filtering in memory

Spark:

- Tells **Parquet reader**: "Only give me rows where age > 30"

✅ **Result**: Less I/O, faster reads, smaller memory usage

🧠 Supported Formats:

Format	Predicate Pushdown
Parquet	✅ Yes
ORC	✅ Yes
Delta Lake	✅ Yes
CSV, JSON	❌ No

✅ 2. Column Pruning / Projection Pruning

Spark **reads only the needed columns** from disk instead of reading all columns in a dataset.

🔹 Example:

python

CopyEdit

```
df = spark.read.parquet("user_data/")

df.select("name").show()
```

If the file has columns like: `name`, `age`, `email`, `salary`

❌ Without column pruning: all 4 columns are read
✅ With column pruning: only `name` column is loaded from disk

This is called:

- **Projection pruning** → in SQL language
- **Column pruning** → in Spark/DataFrame context

✅ Result: Reduced disk read, faster query

🔹 Q6. When should you avoid UDFs in Spark?

Goal:
Test if you know UDFs **break Catalyst optimizations** and run in a slower interpreted mode.

L4 Expectation:

- You push for built-in functions
- You understand that Python UDFs slow down the JVM
- You avoid UDFs in joins, filters, and projections

What to Cover:

- Use `F.when`, `F.col`, `F.expr` instead of UDFs
- Avoid UDFs especially in select & where clauses

- UDFs = black box to Catalyst

Quantified Answer:
"Rewriting UDFs using built-in functions reduced runtime by 52% — 44 min → 21 min."

Problem	Impact
❌ **No Catalyst Optimization**	UDFs are treated as black boxes → no predicate pushdown, column pruning
❌ **No JVM Codegen (Tungsten)**	Spark cannot generate optimized bytecode for UDF logic
❌ **Serialization Overhead**	Python UDFs require data to move from JVM → Python (via Py4J) → JVM
❌ **Slower Execution**	UDFs run row-by-row, interpreted mode → slower for large datasets
❌ **Hard to Debug**	UDFs are opaque in Spark UI → no insight into what's happening

---

🔹 Q7. What are common causes of executor OutOfMemory (OOM) and how do you fix them?

Goal:
Assess your debugging skills for **real-world failures** — especially memory crashes.

L4 Expectation:

- You give multiple root causes (e.g., caching large DFs, skew, serialization issues)

- You give tuning or refactoring-based solutions

What to Cover:

- Use `.unpersist()`
- Repartition large DFs
- Avoid caching intermediate stages
- Reduce cores/executor for better GC

Quantified Example:
"OOMs reduced from 9 to 0 after switching to Kryo, reducing executor cores from 8 to 4, and unpersisting."

🔹 2. Common Root Causes of OOM

Cause	Why It Happens
❌ **Large DataFrame Cached Without Unpersisting**	**Caches eat up memory and stay unless manually released**
❌ **Data Skew (e.g., few keys = huge partitions)**	**A single task ends up processing most of the data**
❌ **Too Many Cores/Executor**	**More parallel tasks = more memory pressure = GC overhead**
❌ **Using Default Java Serializer**	**Slower, creates more garbage objects**
❌ **Huge Wide Transformations**	**Large shuffles (joins/groupBy) overload memory**
❌ **Nested UDFs or large closures**	**Leads to bloated task size and high object creation rate**

🔹 3. How to Fix or Prevent OOMs

Fix	Why It Helps
✅ Use `.unpersist()` after caching	Frees up storage memory
✅ Switch to Kryo serializer	More compact, faster → less heap pressure
✅ Reduce `executor.cores` from 8 → 4–5	Fewer concurrent tasks = better GC performance
✅ Avoid unnecessary caching	Don't cache everything — cache only reused data
✅ Repartition skewed or wide DFs	Avoid hot partitions that blow memory
✅ Use broadcast joins when possible	Avoid shuffle-heavy joins
✅ Use G1GC + memoryOverhead	Better JVM memory control

🔹 Q8. What is partition pruning and how does it improve performance?

Goal: Partition Pruning is different from Predicate Performance
Check if you understand **I/O reduction via predicate pushdown**. (Partition pruning is

L4 Expectation:

- You mention static and dynamic pruning
- You explain enabling via config

- You give real query example with filter

What to Cover:

- Only read necessary partitions during query execution
- Works best on partitioned Parquet/Delta tables
- Config: `spark.sql.optimizer.dynamicPartitionPruning.enabled`

Quantified Answer:
"Read reduced from 2000 partitions to 6. Runtime dropped from 40 min → 7 min."

✅ **Definition:**

Partition pruning means Spark **skips reading entire partitions** if it knows from the query that those partitions are not needed.

✅ **When it happens:**

- Your data is **partitioned on disk** (like in Parquet or Hive tables).
- Your query **filters on partition column**.

✅ **Example:**

Imagine this Parquet table:

python

CopyEdit

```
# Data partitioned by 'country'

df = spark.read.parquet("/data/events")  # partitioned by 'country'

df.filter("country = 'IN'").show()
```

- Here, **Spark reads only the folder** `/data/events/country=IN/`.
- It **doesn't read any other countries' data** — that's **partition pruning**.

🔍 **Happens at file level (before reading data).**

Type	Example	Works With
Partition Pruning	`country = 'IN'`	✅ = only (equality on **partition columns**)
Predicate Pushdown	`age > 30`, `salary <= 5000`	✅ Inequalities, ranges, complex expressions — on **non-partition columns**

Types of Partition Pruning

Type	How It Works	When It Happens
Static	Filter value is known at compile time	`df.filter("country = 'IN'")`
Dynamic	Filter value comes from a **JOIN key**	e.g., `WHERE f.country = d.country_code`

🔹 Q9. Dynamic vs Static Executor Allocation — which one and when?

Goal:
Evaluate your understanding of **resource elasticity** and **cost optimization**.

L4 Expectation:

- You know when dynamic is good (streaming) vs when static is better (batch)
- You mention executor idle timeout, bounds

What to Cover:

- Dynamic Allocation needs shuffle service
- Config: `spark.dynamicAllocation.enabled`
- For batch → static gives predictability

Quantified Example:
"Saved ~$200/week in EMR by enabling dynamic allocation; reduced idle executors from 30 to 5."