🔹 1. Spark DAG, Jobs, Stages, and Tasks

Context: You're analyzing job performance in Spark UI and need to identify slow stages, task skew, and shuffles.

Concept:

- **DAG (Directed Acyclic Graph)**: Built from transformations. Spark lazily builds this logical plan.
- **Job**: Created when an action is triggered (e.g., `collect()`, `count()`).
- **Stage**: A job is split into multiple stages based on shuffle boundaries. Narrow transformations stay in the same stage.
- **Task**: Each stage is split into tasks, one per partition.

Trade-offs:

- Too many tasks = overhead; too few = under-utilization
- Poor stage division = inefficient shuffle & skew
- Avoid unnecessary actions → they create multiple jobs

Production Example:
A job with `groupBy` followed by a `join` showed slowdowns. The Spark UI revealed 3 stages; stage 2 had massive skew and shuffle (due to groupBy). Salting fixed the skew and rebalanced tasks.

Key Takeaways:

- Every action creates a job
- Watch for shuffle boundaries → expensive
- Debug slow stages & skew via Spark UI

🔹 2. Catalyst Optimizer – Logical & Physical Plans

Context: You write a complex DataFrame query, and Spark rewrites it differently in the plan.

Concept:

- Catalyst is Spark's query optimization framework for DFs/Datasets.
- Logical Plan → Optimized Logical Plan → Physical Plan → Executed Plan
- Optimizations: Predicate pushdown, constant folding, projection pruning

Trade-offs:

- Can't control Catalyst fully
- May mis-optimize in rare edge cases
- Needs to work with supported functions for Catalyst to optimize

Example:
A `filter().select()` is internally reordered as `select().filter()` for performance.

Key Takeaways:

- Spark transforms queries under the hood for performance
- Understand plans using `explain(mode="formatted")`

🔹 **3. Tungsten Engine – Memory & Code Gen**

Context: You're optimizing CPU and memory usage for large ETL pipelines.

Concept:

- **Tungsten** = physical execution engine.
- Features:
    - Off-heap memory management
    - **Whole-stage code generation**
    - Binary format processing

Trade-offs:

- Off-heap = faster, but harder to debug

- May lead to GC issues if poorly managed
- Codegen fails on UDFs, complex types

Example:
Enabling whole-stage codegen sped up a join-heavy ETL job by 40%. But memory errors appeared with large structs — resolved via tuning `spark.sql.codegen.wholeStage`.

Takeaways:

- Tungsten enables Spark's speed
- Avoid UDFs — blocks optimization/codegen
- Monitor memory/GC via Spark UI & logs

🔹 4. Serialization – Kryo vs Java

Context: You need to cache a custom class or large object graph.

Concept:

- **Java Serialization**: Default but slow & verbose
- **Kryo**: 10x faster, compact, requires class registration

Trade-offs:

- Java = easy, but heavy
- Kryo = fast, but strict
- Kryo can OOM if not sized properly

Example:
Switching to Kryo reduced serialization overhead from 30% to 10% CPU on a graph ETL job. Required registering custom classes.

Takeaways:

- Use Kryo in production

- Register classes with `spark.kryo.classesToRegister`
- Monitor cache spills due to poor serialization

🔹 **5. Join Optimization – Sort Merge, Broadcast, Shuffle Hash**

Context: You're joining a huge fact table with a small dimension table.

Concept:

- **Broadcast Join**: For small table (<10MB), avoids shuffle
- **Shuffle Hash Join**: For medium-sized tables
- **Sort Merge Join**: For large sorted tables with known keys

Trade-offs:

- Broadcast join → OOM if table too big
- Shuffle joins = costly shuffles
- Sort Merge = good for range joins, bad for small data

Example:
Broadcasting a 5MB lookup table sped up join by 3x; job failed when table grew to 50MB (OOM) → switched to shuffle join.

Takeaways:

- Always profile table size
- Use `spark.sql.autoBroadcastJoinThreshold` wisely
- Avoid shuffle if broadcast is possible

🔹 **6. Skewed Joins – Mitigation Techniques**

Context: 90% of join keys point to a single value.

Techniques:

- **Salting** keys
- **Skew join hints**
- **Repartitioning skewed keys separately**
- **Broadcast the small side**

Trade-offs:

- Salting increases shuffle
- Requires application-level awareness
- Can't always eliminate skew, only reduce impact

Example:
Join on `"country_id"` had 80% "IN" → salted `"IN_0"`, `"IN_1"` → rewrote join → improved stage execution time by 50%.

Takeaways:

- Skew = Spark killer
- Use salting, filters, or skew hints early
- Validate impact via stage/task time

🔹 7. Shuffle Mechanics – Internals

Context: A job has a 40-minute stage due to excessive shuffle writes/reads.

Concept:

- Shuffle = repartitioning data across nodes
- Happens in wide transformations
- Data is written to disk → transferred → read → deserialized

Trade-offs:

- High I/O

- Memory pressure (if buffers not tuned)
- Can cause stage retries, executor OOMs

Example:
`groupBy().agg()` caused 4TB shuffle; optimized with `reduceByKey()` and `mapPartitions` → reduced shuffle by 70%.

Takeaways:

- Reduce shuffle keys
- Avoid wide transformations when possible
- Tune `spark.shuffle.file.buffer`, `spark.reducer.maxSizeInFlight`

🔹 8. Custom Partitioner

Context: You want to group logs by userId to minimize shuffle.

Concept:

- Use `partitionBy()` in write OR custom RDD partitioner
- Ensures related data goes to same executor

Trade-offs:

- Works only at RDD or write level
- Poor partitioning = skew
- Over-partitioning = overhead

Example:
Custom hash partitioner used to ensure 1M events per user were colocated for aggregation.

Takeaways:

- Partition with intent
- Validate output data distribution

🔹 9. StorageLevel.MEMORY_AND_DISK_SER

Context: You want to cache, but dataset is large and won't fit in memory.

Concept:

- `MEMORY_AND_DISK_SER` stores serialized objects in memory, spills rest to disk
- Saves memory, but increases CPU (due to serialization)

Trade-offs:

- Better for large datasets
- Slower due to CPU serialization overhead
- Requires tuning memory fraction

Example:
Used in a pipeline with 12 GB intermediate join output; prevented executor OOMs.

Takeaways:

- Prefer this over plain `cache()` for large data
- Monitor spill size and GC logs

🔹 10. OutOfMemory in Executors – Handling**

Context: Job fails with "GC overhead limit exceeded".

Strategies:

- Reduce number of partitions (avoid over-parallelism)
- Use `persist(MEMORY_AND_DISK_SER)`
- Use `mapPartitions` to process in chunks
- Increase executor memory (`spark.executor.memory`)

- Avoid UDFs (they bypass optimizations)

Example:
Replacing UDF with native Spark functions reduced executor memory usage by 40%. Also reduced task retry count.

Takeaways:

- Monitor via Spark UI → GC time, spills, task retries
- Memory tuning is iterative — no one-shot fix

Scenario Question (from today):

In a production environment, how would you decide on the **number of partitions** for your Spark job? What factors would influence that decision?

ANS : You need to tune the partition count in a Spark job for optimal performance — avoiding both under-parallelism (few partitions) and overhead (too many tiny tasks).

Example:

In one job, 4TB input with 500 partitions → slow tasks and skew. Increasing to 3000 partitions balanced tasks across executors and reduced total job time by 25%.