Q1. What is a DAG in Spark? How is it different from a physical execution plan?

1. **Problem Context:**
 Spark represents your entire job as a **Directed Acyclic Graph** (DAG) — a logical flow of transformations.

2. **Concept Explanation:**

   ○ **Logical DAG**: Formed from transformations before an action. It's like: `df.filter().groupBy().map()`

   ○ **Physical Plan**: Once an action triggers, Spark converts the DAG into **Stages** and **Tasks**.

   ○ **Stage**: A set of pipelined operations (e.g., map/filter) that don't require a shuffle

   ○ **Task**: Unit of work per partition in a stage

3. **Design & Trade-offs:**

   ○ DAG is **recomputed from lineage** if there's a failure

   ○ If you persist intermediate results, Spark will not recompute from scratch

   ○ Checkpointing truncates the lineage when it's too long or risky (e.g., >100 stages deep) (9th Question explained deeply)

4. **Real-world Example:**
 "In one pipeline, lineage caused 4 re-computations of expensive joins when a worker failed. We persisted before the join and saved 40 minutes per failure."

5. **Takeaway:**
 DAG helps with fault tolerance, but deep lineage without persistence = risk of recomputing huge pipelines.

Q2. How does Spark handle memory, GC, and spill to disk?

1. **Problem Context:**
 Long-running jobs with wide transformations can spill intermediate data to disk or trigger GC stalls.

2. **Concept Explanation:**

- **Spark memory = Storage + Execution**
  - `spark.memory.fraction`: total usable memory (default 60%)
  - `spark.memory.storageFraction`: % reserved for cache vs computation
- **GC tuning:** Triggered when memory is tight — causes task delays
- **Spill to disk**: Happens during wide ops (joins, aggregations) when memory isn't enough

3. **Design & Trade-offs:**
   - Use Kryo serialization for faster (and smaller) object encoding
   - Monitor GC time in Spark UI → If high, reduce memory per executor or use more cores
   - Don't oversubscribe memory — better to **increase executors** with lower memory than a few with high memory
4. **Real-world Example:**
   "One job took 5h due to GC stalls. GC time was 50%. Switched from Java to Kryo, reduced executor memory to 4g, added more executors → GC time dropped to 3%, job time to 40 min."
5. **Takeaway:**
   GC overhead = hidden killer. Spark UI → Executors tab is your debugging friend.

Q3. How does shuffle work and what causes shuffle spills?

1. **Problem Context:**
   Wide transformations (join, groupBy) require shuffling — expensive, disk-heavy, and often bottlenecks.
2. **Concept Explanation:**
   - Data is **written by mappers** → **read by reducers**, creating temporary files
   - **Spill occurs** when memory is insufficient → Spark spills sort buffers to disk
   - Shuffle files are cleaned up *only after* job completes (except with external shuffle service)

3. **Design & Trade-offs:**
    - Tune with `spark.sql.shuffle.partitions` (default 200 is often too low)
    - Enable `spark.shuffle.compress=true` to save I/O
    - External shuffle service keeps shuffle data across executor loss
4. **Real-world Example:**
   "At Uber-scale job, shuffle read = 1.5 TB, spilled data = 700 GB. Increased partitions from 200 to 800, added compression → spill dropped to 180 GB, job time cut by 2h."
5. **Takeaway:**
   Shuffle = most expensive phase. Minimize it or optimize it — or your job dies at 95% progress.

Q4. A stage is stuck at 99% for 15 mins. What do you do?

1. **Initial Suspects:**
    - A task is **skewed** → taking longer than others
    - GC overhead or memory pressure on that executor
    - Shuffle fetch failure or disk spill on slow node
2. **Actions Taken:**
    - Check Spark UI → see stage DAG, task duration, skewed input size
    - Look for tasks with 10x runtime or shuffle read
    - Use `spark.sql.adaptive.skewJoin.enabled=true` in Spark 3+
    - Repartition by range to reduce skew
3. **If reproducible:**
    - Add salting or bucket join strategy
    - Use persistent caching for upstream if re-computations are observed
4. **Real-world Fix:**
   "One job was stuck on task 17/100 for 25 mins. Task was reading 20x more data.

Salting + AQE skew join enabled solved it — saved 70% runtime."

5. **Takeaway:**
 Always check Spark UI → task duration, input size, GC time, shuffle reads.

✅ Q5. What is the External Shuffle Service (ESS), and when is it required in production?

1. **Problem Context:**
 When an executor crashes or gets decommissioned mid-job, its shuffle files can be lost — causing job failure or retries.

2. **Concept Explanation:**
   - ESS is a daemon that runs outside the executor and manages shuffle file lifecycle
   - Allows **other executors** or **speculative tasks** to fetch shuffle data even if original executor dies
   - **Required when**: Using dynamic allocation (executor churn is common), or in long shuffle-heavy jobs

3. **Design & Trade-offs:**
   - ESS increases fault tolerance
   - Needs correct config (`spark.shuffle.service.enabled=true`, port setup)
   - Extra daemon process = more ops overhead

4. **Real-world Example:**
 "A job crashed at stage 6/7 after 3 hours — root cause: lost executor with intermediate shuffle files. Enabled ESS → retried only failed tasks, not whole job → saved 2h runtime."

5. **Takeaway:**
 If you're running on YARN or with dynamic allocation — ESS is non-negotiable for resilience.

✅ Q6. What is Task Speculation in Spark? When do you use or avoid it?

1. **Problem Context:**
   In large clusters, a few straggling tasks can delay job completion significantly.

   ### What is Task Speculation in Spark?

   Speculative execution is Spark's way of saying:

   "Some tasks are running too slow. Maybe they're on a bad node or got unlucky. Let's start a duplicate of that task elsewhere and keep whichever finishes first."

2. **Concept Explanation:**
   - Speculation means **launching duplicate tasks** for slow-running ones
   - Whichever completes first, result is accepted
   - Enabled via `spark.speculation=true`
3. **Design & Trade-offs:**
   - ✅ Good when slow nodes cause tail latency
   - ❌ Bad if slowness is due to external API, DB, or skew (wastes resources)
   - Causes extra compute pressure if misused
4. **Real-world Example:**
   "Job had 99/100 tasks complete in 5 mins; last one took 27 min. Enabled speculation → duplicated last task → job time reduced to 6 mins. Later avoided on DB-write job where speculation caused duplicate inserts."
5. **Takeaway:**
   Use it **only for CPU-bound workloads** with large task parallelism and occasional tail latency.

Task speculation = Spark **duplicates slow-running tasks** (called *stragglers*) and **launches them on different executors**, not on the same one.

❗ Key Point:

Speculative tasks are always run on different executors, not on the same one.

Why?

- If the original executor is slow (e.g., GC issue, disk I/O, data skew), launching the same task again **on the same executor** would **not help**.
- Spark instead tries to launch the duplicate **on a faster, healthier executor** that still has available CPU cores and memory.

✅ Q7. GC Overhead Limit Exceeded – What Causes It? How Do You Fix It?

1. **Problem Context:**
   Spark job fails due to JVM error: **GC Overhead Limit Exceeded** — meaning >98% of CPU time is spent in GC, and <2% heap is recovered.
2. **Causes:**
   - Too little memory per executor
   - Too many objects (especially with map-side aggregations or large joins)
   - Poor serialization (e.g., Java serializer with complex nested data)
3. **Fixes:**
   - Switch to **Kryo serializer** (`spark.serializer=org.apache.spark.serializer.KryoSerializer`)
   - Reduce memory per executor to allow more containers
   - Tune GC: G1GC for large heaps
   - Repartition to reduce data per task
4. **Real-world Fix:**
   "1.2B-row join caused GC spike. GC logs showed >90% time in GC. Switched to Kryo, cut executor memory from 12g → 6g, increased executors → job time dropped from 3.5h to 48 mins."
5. **Takeaway:**
   GC tuning isn't optional at scale. Use Spark UI → GC Time per executor, or review

GC logs.

✅ Q8. How Do You Decide Spark Executor Config in Production?

1. **Problem Context:**
   Wrong executor settings = wasted resources, OOMs, slow jobs.
2. **Rules of Thumb:**
   - Start with: `spark.executor.instances`, `spark.executor.memory`, `spark.executor.cores`
   - Use **1 executor per node** for large-memory nodes, or **2–5 per node** on standard
   - Each executor → 3–5 cores ideally, too many = parallelism issues
   - Memory = ~90% of available *container* memory – 1 GB for overhead
3. **Real-world Config Comparison:**

Config	Result
1 exec, 16g memory, 8 cores	❌ GC pause spikes, 1 task slow
4 execs, 4g memory, 2 cores	✅ Better GC, parallelism improved
Kryo + 8g + 3 cores + 4 execs	✅ Final config — balanced

4. **Tuning Tip:**
   Add `--conf spark.memory.fraction=0.7` if storage vs compute is unbalanced.
5. **Takeaway:**
   Every workload is different. Benchmark, test in staging, monitor GC, shuffle, CPU.

---

9) CHECKPOINTING

🎯 Final State:

Aspect	Location	Notes
✅ **New lineage**	From HDFS file	Starts from checkpointed file as the base
❌ **Old lineage (1–49)**	Discarded (from DAG)	Not used anymore after checkpoint
✅ **Checkpoint data**	Stored in HDFS	Parquet-like format, managed by Spark

---

🔍 What does "new lineage" mean?

- Spark now **treats the checkpointed file as a fresh input source** (like reading from Parquet)
- All further transformations (51–100) are applied **on this file**, not on the original raw source
- So if failure happens later (e.g. transformation 70 fails), Spark **does NOT go back to step 1**
- It starts from the **checkpointed HDFS file**

ONLY AFTER AN ACTION CHECKPOINT SHOULD BE ENABLED, NOT BEFORE THAT