🔍 1. How do you choose the right cluster configuration for 100 GB of data?

✅ **Answer:**

To configure a Databricks cluster for processing 100 GB of data, you must consider:

- **Data size**: 100 GB is moderate; doesn't require massive scale.
- **Job complexity**: Simple transformations need fewer resources; joins, shuffles, ML models may require more.
- **Frequency**: For scheduled jobs, use **Job Clusters**; for exploration, **All-Purpose Clusters**.
- **Recommended setup**:
  - 4–8 workers (each with 8–16 GB RAM)
  - Auto-scaling enabled (handles variable workload)
  - Photon engine (if using DBR 9.1+ for improved performance)
  - Use spot instances to reduce cost (if SLA allows)

📌 **Tip:** Always test with a smaller dataset and monitor performance using Spark UI before scaling up.

🔍 2. What happens internally when you submit a Spark job?

✅ **Answer:**

Here's the **execution flow** of a Spark job:

1. **Code Submission**: Your transformations/actions are submitted via the driver.
2. **DAG Creation**: Spark creates a **Directed Acyclic Graph** of logical execution stages.
3. **Logical Plan**: Spark builds a logical plan from user code (e.g., select, join).
4. **Optimization**: Spark's Catalyst optimizer transforms the logical plan into a **physical plan**.
5. **Task Scheduling**: The physical plan is divided into **stages and tasks**.
6. **Execution**: Tasks are sent to **executors** on worker nodes via SparkContext.
7. **Result Handling**: Results are returned to the **driver**, cached, or written to storage.

📌 **Driver = orchestrator**, **Executors = workers**

🔍 3. What is driver memory, and when does it spill to disk?

✅ **Answer:**

- **Driver memory** is the memory used by the Spark **driver node** (where your job is submitted).
- It stores:
  - DAG metadata
  - RDD lineage
  - Broadcast variables
  - Collected results (collect(), toPandas())

💥 **Spills occur when**:

- You run collect() on a huge dataset
- You cache large metadata in the driver
- Memory exceeds spark.driver.memory

📌 Avoid spilling by using take(), limit(), and not materializing full datasets in the driver.

🔍 4. How does the Spark memory manager work?

✅ **Answer:**

Spark uses a **Unified Memory Management** system which divides memory into:

- **Execution memory** (for joins, aggregations, shuffles)
- **Storage memory** (for caching and persisting DataFrames/RDDs)
- **User memory** (misc overhead like JVM allocations)

🔄 **Dynamic sharing**: If storage isn't full, execution can borrow memory and vice versa.

📌 When either region is full:

- **Storage** → Evicts old cached blocks (LRU policy)
- **Execution** → Spills intermediate data to disk

🔍 5. When do you get an OutOfMemory (OOM) in driver or executor?

✅ **Answer:**

💥 **Driver OOM occurs when**:

- You use collect() or toPandas() on a huge DataFrame
- You cache large results or broadcast huge variables

💥 **Executor OOM occurs when**:

- There are wide transformations like shuffles or joins without enough memory
- The dataset doesn't fit into cache
- Improper partitioning leads to skew

📌 Fixes:

- Avoid wide transformations without partitioning
- Tune memory: spark.executor.memory, spark.memory.fraction
- Replace collect() with take() or save to storage

🔍 6. What is executor memory, how is it distributed, and when does it spill to disk?

✅ **Answer:**

Executor memory is used for executing tasks. It's split into:

- **Execution Memory** – for shuffle, sort, join, and aggregation operations
- **Storage Memory** – for caching/persisted datasets
- **User Memory** – JVM overhead, data structures, internal functions

Spill to disk happens when:

- Execution memory is exceeded during join/shuffle
- Caching exceeds storage limit and triggers eviction
- Memory fraction (default 60%) is exhausted and disk is the only option

📌 You can monitor spills in Spark UI > Stages > Tasks > Metrics

---

🔍 7. What is a pool in Databricks and why is it useful?

✅ **Answer:**

A **Databricks pool** is a set of pre-warmed virtual machines (VMs) that help:

- **Reduce cluster start time** (especially for short jobs)
- **Improve developer productivity** by avoiding cold-start latency
- **Reuse VMs** across jobs to lower cost

📌 Created under Compute > Pools, and assigned to clusters in config.

Used heavily in:

- UAT environments
- Frequent job testing
- Notebook-heavy analysis sessions

🔍 8. What workloads can run on a standard Databricks cluster?

✅ **Answer:**

Standard clusters (aka All-Purpose clusters) can run:

- **Batch ETL pipelines**
- **Machine Learning training (using MLlib or sklearn)**
- **Ad-hoc exploratory analysis (SQL, Python, Scala, R)**
- **Streaming (for light workloads using Structured Streaming)**

📌 Ideal for development, team collaboration, and running notebooks interactively.

🔍 9. What are the types of clusters in Databricks and how do you choose the right one?

✅ **Answer:**

Databricks provides **3 main cluster types**:

🔹 **All-Purpose Cluster**

- Use Case: Dev work, notebook execution
- Multiple users, long-running sessions
- Higher cost but more flexibility

🔹 **Job Cluster**

- Use Case: Scheduled production jobs

- Spun up for the job → terminated afterward
- Better cost efficiency for automation

🔹 **Pool-Based Cluster**

- Uses pre-warmed VMs for faster start-up
- Useful for short, frequent jobs

📌 **Choosing Tips**:

- For production jobs: **Job Clusters**
- For dev teams & analysis: **All-Purpose Clusters**
- For cost-saving + performance: Use **pools** with either cluster type