🔥 1️⃣ Executor Memory vs. Driver Memory

👉 What's the difference?

- **Driver Memory:**
  The driver is the **master node** that runs the Spark application, handles the **DAG scheduling**, tracks tasks, and manages metadata. Driver memory is used for things like Spark context, job coordination, task tracking, and data structures collected back to the driver (e.g., .collect() or .take()).

- **Executor Memory:**
  Executors are the **worker nodes** that run the tasks and perform computations. Executor memory is used for processing data (transformations/actions), storing shuffle data, and caching/persisted datasets (.cache() or .persist()).

👉 Impact on Garbage Collection, Job Stability, and Performance:

- If **driver memory is too low**, the application can fail due to **OutOfMemory errors** when collecting too much data to the driver (common with .collect() misuse).
- If **executor memory is insufficient**, tasks may fail due to **OOM during shuffles, joins, aggregations, or caching.**
- **Large executor memory can cause longer garbage collection (GC) pauses**, affecting stability. It's sometimes better to have **more executors with less memory each** to reduce GC overhead.
- Poor memory configuration leads to job retries, slowdowns, or even crashes.

🔥 2️⃣ Databricks Clusters – Job vs. Standard

👉 How do you decide?

- **Job Cluster:**
  - Spin-up just for a single job or notebook run and then terminates.
  - Used for **production jobs, scheduled tasks, or CI/CD pipelines.**
  - **Cost-effective** because it auto-terminates when the job finishes.
- **Standard Cluster:**
  - Long-running interactive cluster.
  - Used for **ad-hoc analysis, development, and collaborative notebooks.**
  - Supports multiple users and multiple jobs at once.

👉 **Config for a 200GB Pipeline:**

- **Cluster Type:** Job cluster for ETL/Batch jobs.
- **Worker Type:** Choose **compute-optimized (like Standard_DS3_v2)** or memory-optimized based on transformation types.
- **Autoscaling:**
    - Enable autoscaling between **min 4 to max 12 workers** (depending on job parallelism needs).
- **Photon:**
    - Enable **Photon acceleration** if the workload involves SQL-heavy transformations — it provides massive speed improvements for SQL and Delta workloads.
- **Driver:** Medium-size driver (matching worker specs unless driver-heavy operations are involved).

🔥 3️⃣ **Dynamic Allocation – ON or OFF?**

👉 **When does it actually hurt performance?**

- **Dynamic Allocation ON:**
    - Scales executors up/down based on workload.
    - Hurts when tasks require **persistent parallelism**, like **large shuffles, broadcast joins, or streaming jobs**, where releasing executors frequently leads to recomputing cached shuffle data.
- **Dynamic Allocation OFF:**
    - Better for jobs with **large joins, shuffles, or stages that maintain high load throughout.**
    - Avoids frequent executor loss, reduces shuffle overhead, and improves stability.

👉 **Trade-offs:**

- **ON:** Saves costs for idle time, but can introduce shuffle data loss and recomputation.
- **OFF:** More stable for large pipelines but keeps the cluster size fixed, leading to higher costs if not tuned properly.

🔥 4️⃣ Skewed Joins

👉 **What causes them?**

- When one or more keys in a join have **significantly more data** than others, leading to an uneven distribution of work across tasks. Some tasks finish quickly, while others take a long time — causing performance bottlenecks.

👉 **Solutions:**

- **Salting:**
    - Add a random number (salt) to skewed keys to spread them across partitions, then remove the salt after the join.
- **Broadcast Join:**
    - Broadcast the smaller dataset to all executors, avoiding shuffles entirely.
    - Works when one table fits within **driver/executor memory (~2GB limit typically).**
- **Repartitioning:**
    - Use repartition() or skew join hints to redistribute the skewed data evenly before join operations.
- **Skew Join Hints (Databricks/Spark 3.0+):**
    - Use JOIN /*+ SKEW */ syntax to let Spark handle skewed keys automatically.

🔥 5️⃣ Spark DAG & UI Bottleneck Analysis

👉 **How does DAG visualization help debugging?**

- **DAG (Directed Acyclic Graph)** shows how Spark breaks a job into stages and tasks.
- Helps identify:
    - Expensive shuffles (shown as stage boundaries).
    - Unnecessary wide transformations (groupByKey vs. reduceByKey).
    - Long-running stages due to data skew or lack of parallelism.
    - Whether caching/persisting is working effectively.

👉 **Key Spark UI Metrics to Check First:**

1. **Stage Duration:** Look for stages taking much longer than others.

2. **Task Time Variance:** Check if some tasks are slow (possible skew).
3. **Shuffle Read/Write Size:** Large shuffle data can indicate poorly planned joins or aggregations.
4. **Executor CPU/Memory Utilization:** See if tasks are compute-bound or memory-bound.
5. **GC Time:** High garbage collection time affects job throughput.

🎯 **Final Thoughts:**

If you understand these deeply, you are not just a Spark user — you're an optimizer. 🧠 🔥