**Debugging & Logs in Databricks UI — Senior Engineer’s Approach**

**1. Where to Start Debugging**

When a job fails (or underperforms), think in layers:

1. **Driver logs** → Did the application set up correctly? Is SparkSession alive?
2. **Executor logs** → Did tasks fail? Are there out-of-memory errors?
3. **Spark UI (Jobs/Stages/Tasks tabs)** → Are there bottlenecks, shuffles, or skew?
4. **Cluster logs (Driver & Worker nodes)** → Any JVM/Python crashes or library import issues?
5. **Databricks Jobs UI / Run Output** → Quick summary of notebook/job run.

**2. Databricks Job Runs → Navigating Logs**

**🔸 From Jobs Page**

* Go to **Workflows → Jobs**.
* Click a specific **Run** → shows **Run Output** with:
  + **Driver notebook output** (stdout, cell output).
  + **Cluster event logs** (startup failures, autoscaling events).
  + **Links into Spark UI** for deeper inspection.

**🔸 Common quick checks**

* **Red error stacktrace** → Python/Scala exception with line number.
* **Execution time per cell** → Identifies bottlenecks.
* **Task retries count** → Shows flaky partition/data issues.

**3. Spark UI (built into Databricks)**

Accessible via:

* Job run → **Spark UI** link.
* Or, from **Clusters → Spark UI** (for live clusters).

**🔸 Tabs to know**

**Jobs Tab**

* High-level view: each Spark action = one job.
* Check if a job has many stages (indicates multiple shuffles).

**Stages Tab**

* Each job split into stages at shuffle boundaries.
* Look for:
  + **Shuffle read/write size** → Big = possible bottleneck.
  + **Skewed stage** → Some tasks much slower than others.

**Tasks Tab**

* Drill into per-task metrics:
  + **Duration** → Outliers = skew.
  + **GC time** → High = memory misconfiguration.
  + **Shuffle spill to disk** → Indicates insufficient executor memory.
  + **Input size vs Output size** → Explains data explosion.

👉 As a senior engineer, you’d say: *“I always check the slowest stage in the Stages tab, then open the Tasks view to see if I’m dealing with skew or memory pressure.”*

**SQL Tab**

* If using Spark SQL/DataFrames, shows the **query plan**.
* Inspect:
  + **Logical Plan vs Physical Plan**.
  + **Exchange nodes** = shuffle.
  + **BroadcastHashJoin markers** = optimizer applied broadcast join.

**4. Executor & Driver Logs**

**🔸 Accessing Logs**

* Navigate: **Clusters → Driver Logs** or **Worker Logs**.
* You’ll see:
  + **stdout** → print/logging outputs.
  + **stderr** → stacktraces, JVM/GC errors.
  + **log4j output** → Spark internal logs.

**🔸 Common Patterns**

* OutOfMemoryError in executor logs → executor memory too small.
* Long GC pauses in driver logs → driver under-provisioned.
* “Task killed due to stage failure” → often shuffle fetch failure or node loss.

👉 As a senior engineer, mention: *“I check stderr first for JVM OOMs, then stdout for my app logs, and finally cluster event logs for infrastructure-level issues.”*

**5. Cluster Event Logs**

**🔸 Location**

* **Clusters → Event Log** tab.

**🔸 Shows**

* Autoscaling events (new executors added/removed).
* Spot/preemptible instance losses.
* Node restarts or provisioning issues.

👉 If executors keep dropping, this log explains *why* (e.g., AWS spot termination).

**6. Debugging Techniques**

**🔸 Handling Skew**

* In Spark UI, look for **task duration variance** → skewed partitions.
* Fix via:
  + Salting key.
  + Enabling **Adaptive Query Execution (AQE)**.
  + Repartitioning data.

**🔸 Handling OOM**

* Look for java.lang.OutOfMemoryError or large **shuffle spill**.
* Fix:
  + Increase spark.executor.memory.
  + Reduce data skew.
  + Use .persist(DISK\_ONLY) instead of caching huge DataFrames in memory.

**🔸 Debugging Long Jobs**

* Check **Stage DAG visualization** in Spark UI.
* Identify shuffle boundaries → optimize joins, reduce shuffle partitions.
* Use **Query Plan** (df.explain(True)) in notebooks.

**7. Pro Tips (Answer Like a Senior Engineer)**

* *“I always start with the Spark UI → Stages tab, because 90% of Spark performance issues show up there.”*
* *“If tasks are retrying multiple times, I drill into worker logs — usually it’s bad data (corrupted row, skew) or insufficient memory.”*
* *“In Databricks, I check the Event Log tab for scaling issues — if executors are dropping, that’s an infra-level cause, not a code bug.”*
* *“I use df.explain(True) and the SQL tab to see if Catalyst Optimizer applied broadcast joins and filter pushdown as expected.”*
* *“When debugging, I always correlate cluster metrics (CPU/memory) with Spark UI metrics to decide if it’s tuning vs data problem.”*

**✅ Quick Debug Checklist (Interview-Ready)**

1. **Jobs UI** → See if failure is code-level or cluster-level.
2. **Spark UI → Stages tab** → Look for shuffle, skew, stragglers.
3. **Tasks tab** → GC, spill, skew metrics.
4. **SQL tab** → Check optimizer plan (broadcasts, exchanges).
5. **Driver & Executor logs** → stdout/stderr for stacktraces & OOMs.
6. **Cluster Event Log** → Scaling issues, node failures.

**Databricks Debug Playbook (by failure type)**

**1) Driver OOM (Out-Of-Memory) / Driver crash**

**Typical signatures**

* java.lang.OutOfMemoryError: Java heap space (driver logs)
* org.apache.spark.SparkException: Job aborted due to stage failure
* SparkDriver: Result of task is bigger than spark.driver.maxResultSize
* Notebook stuck/terminated right after a big collect() / toPandas() / display() on huge DataFrame

**First 60 seconds (triage)**

1. **Workflows → Jobs → (Run) → Output**: confirm last cell and error stacktrace.
2. **Open Spark UI → Jobs → last job**: confirm the action (often collect) that pulled too much data to the driver.
3. **Clusters → (cluster) → Driver Logs → stderr**: look for OutOfMemoryError, GC overhead limit exceeded.

**Deep diagnosis (where & what to check)**

* **Driver logs (stderr/stdout)**: OOM messages, large result serialization errors.
* **Spark UI → SQL tab**: confirm plan; see if massive **Exchange** (shuffle) or CollectLimit is missing.
* **Spark UI → Environment**: check spark.driver.memory, spark.driver.maxResultSize.
* **Notebook code**: search for collect(), toPandas(), display(df) without limit().

**Quick fixes (now)**

* Replace collect()/toPandas() with:
  + df.limit(1000).toPandas() for sampling
  + df.write.format("delta").mode("overwrite").save(...) and inspect downstream
  + df.take(1000) for quick peek
* Add limit() before display() during exploration.

**Durable fixes (next run)**

* **Right-size driver** (cluster config): raise spark.driver.memory and memory overhead.
* Set/raise spark.driver.maxResultSize (careful: this protects the driver, but does not solve poor patterns).
* Move logic to executors:
  + Replace Python collect() post-processing with Spark SQL/DataFrame operations
  + Use aggregations on cluster, persist results to Delta, then sample
* Cache only what you reuse, and **unpersist**:  
  df\_cached.unpersist(blocking=True)

**Prevention checklist**

* Enforce review for collect()/toPandas() in PRs.
* Add utility wrappers that log row counts and enforce limit() in notebooks for ad-hoc analysis.
* For BI extracts, write to tables/files rather than materializing to driver.

**2) Executor skew / stragglers (long-tail tasks)**

**Typical signatures**

* Stage is 99% complete, but a few tasks run 10–100× longer.
* Spark UI → Stages: huge skew in task durations.
* Errors later: TaskKilled, Container killed by YARN (or equivalent on cloud) due to long running tasks.
* Heavy **GC (Garbage Collection)** time on a few executors.

**First 60 seconds (triage)**

1. **Spark UI → Stages → (slowest stage)**: open **Tasks**; sort by **Duration**.
2. Check **Input/Shuffle Read size** for slow tasks vs median—if outliers exist, it’s **skew**.
3. Enable **Adaptive Query Execution (AQE)** if not already:
4. spark.conf.set("spark.sql.adaptive.enabled", "true")
5. spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")
6. If joining, try **broadcasting** the small side: /\*+ BROADCAST(dim) \*/.

**Deep diagnosis**

* **SQL tab → Physical plan**: look for SortMergeJoin with large **Exchange**; AQE notes like SplitSkewedPartition.
* **Tasks tab**: compare **Shuffle Read**, **Spill** and **GC time** for slow tasks.
* **Data profiling**: identify hot keys causing skew.
* from pyspark.sql.functions import col, count
* df.groupBy("join\_key").agg(count("\*").alias("n")).orderBy(col("n").desc()).show(20)

**Quick fixes**

* **Salting hotspot keys** (when join/groupBy on a heavily skewed key):
* from pyspark.sql.functions import col, rand
* SALT\_BUCKETS = 16
* big = big.withColumn("salt", (rand()\*SALT\_BUCKETS).cast("int"))
* small = small.withColumn("salt", (rand()\*SALT\_BUCKETS).cast("int"))
* joined = big.join(small, on=["join\_key", "salt"], how="inner")
* **Repartition** to increase parallelism (but beware extra shuffle):
* df = df.repartition(1000, "join\_key")
* **Pre-aggregate** before join to shrink the large side.

**Durable fixes**

* Keep **AQE** on; tune:
* spark.conf.set("spark.sql.adaptive.advisoryPartitionSizeInBytes", "256MB")
* spark.conf.set("spark.sql.shuffle.partitions", "auto") # DBR supports auto with AQE
* Data model: **bucket** by join keys if repeatedly joined, or use **Z-ORDER** (Delta Lake) for skipping on filters.
* Apply **dynamic partition pruning** for partitioned fact/dim joins:
* spark.conf.set("spark.sql.optimizer.dynamicPartitionPruning.enabled", "true")

**Prevention checklist**

* Profile key distributions early (top-N frequencies).
* Broadcast dimension tables (< ~100–300 MB) instead of big-big joins.
* Standardize salting utilities for known skewed domains.

**3) Shuffle fetch failure**

**Typical signatures**

* org.apache.spark.shuffle.FetchFailedException
* Stream is corrupted / Too large frame / Connection reset
* Stage fails at a **reduce** side after mappers finished; retries don’t help

**First 60 seconds (triage)**

1. **Spark UI → Failed Stage → Details**: confirm FetchFailed and the source map task IDs.
2. **Clusters → Worker Logs → stderr** on the **map-side** executor that produced the missing shuffle block—look for I/O errors, disk full, executor lost.
3. **Clusters → Event Log**: spot preemptions (spot instance reclaimed), disk failures, node loss.

**Deep diagnosis**

* Was a map-side executor lost after writing shuffle files? (external shuffle service should keep files)
* Are shuffle files enormous (huge partitions)? Check **Shuffle Write** sizes.
* Network instability or timeouts? Check **network timeout** in environment.

**Quick fixes**

* **Retry with higher I/O retries/timeouts**:
* spark.conf.set("spark.shuffle.io.maxRetries", "10")
* spark.conf.set("spark.shuffle.io.retryWait", "5s")
* spark.conf.set("spark.network.timeout", "600s")
* spark.conf.set("spark.executor.heartbeatInterval", "60s")
* Reduce shuffle pressure:
  + Increase parallelism: spark.sql.shuffle.partitions = 400 (or enable AQE auto)
  + **Repartition** earlier by join/group keys to balance
  + Prefer **broadcast join** when possible

**Durable fixes**

* Keep **AQE** enabled; it can pick **BroadcastHashJoin** and **split skewed partitions**.
* Break a single gargantuan shuffle into **two smaller stages** with intermediate **cache/persist(DISK\_ONLY)**.
* Pick node types with **larger local SSDs** if spills are huge; monitor disk usage.
* Upgrade to a recent **Databricks Runtime** (shuffle stability, push-based shuffle improvements).

**Prevention checklist**

* Avoid all-to-all wide shuffles when a pre-aggregation or filter could shrink data early.
* Cap input split size for very large files:
* spark.conf.set("spark.sql.files.maxPartitionBytes", "128MB")
* Use **checkpointing** for ultra-long **DAG (Directed Acyclic Graph)** lineages.

**4) Job stuck in *Pending* (never starts or stuck at “Waiting for cluster”)**

**Typical signatures**

* Job run shows **Pending** for minutes
* Cluster remains at **Starting** / **Acquiring instances**
* Pool at capacity / policy denies node type / cloud quota errors
* Init script/library install hangs

**First 60 seconds (triage)**

1. **Workflows → Jobs → (Run) → View details**:
   * Check **Run status**: Queued (concurrency cap) vs Waiting for cluster.
2. Click **Cluster** in the run → **Event Log**:
   * Look for: *Insufficient capacity*, *Quota exceeded*, *Policy violation*, *Spot instance terminated*, *Init script running*.
3. **Compute → Instance Pools** (if used): pool capacity/idle state.

**Deep diagnosis**

* **Concurrency limits**: Job or workspace **max concurrent runs** reached (queueing).
* **Policy**: Cluster policy blocks requested node type/size.
* **Cloud quota**: vCPU or IP exhaustion in the region/VNet/Subnet.
* **Spot capacity**: no spot available.
* **Init scripts / Libraries**: long-running or failing; open **Driver logs** during cluster start.

**Quick fixes**

* Drop to a **smaller/allowed node** or switch **region/zone** if capacity constrained.
* Temporarily **disable spot** (use on-demand) for critical runs.
* **Attach to an existing all-purpose cluster** (if policy allows) to bypass startup.
* Reduce **max concurrent runs** contention (stagger schedules).
* Comment out **init scripts/libraries** to validate they’re the blocker.

**Durable fixes**

* Use **Instance Pools** sized to peak (warm capacity).
* Rework cluster **policy** to permit required families/sizes, or add an alternative fallback node type list.
* Increase cloud **quotas** (vCPU, IPs, disks) in target region.
* Make init scripts **idempotent/fast**; host artifacts on reliable storage; add timeouts/logging.

**Prevention checklist**

* For production jobs, use **job clusters** with a **pool** to minimize cold starts.
* Define **fallback node types** (if policy supports).
* Regular capacity/quotas reviews per region/VNet/subnet.

**Bonus: Two more common failure modes**

**5) Python worker crash / Pandas UDF OOM**

* **Signs**: Python worker exited unexpectedly, Arrow memory errors, SIGKILL
* **Fixes**: use **iterator-based Pandas UDFs**, reduce batch size:
* spark.conf.set("spark.sql.execution.arrow.maxRecordsPerBatch", "50000")

Avoid toPandas() on big frames; raise executor memory; rewrite UDF logic in Spark SQL where possible.

**6) Task deserialization / classpath conflicts**

* **Signs**: Task not serializable, NoSuchMethodError, ClassNotFoundException
* **Fixes**: keep closures serializable; pin library versions; use **cluster-installed** libraries (not per-notebook) for consistency; avoid shadowing stdlib packages.

**Fast “Where to click” cheat-sheet**

* **Job summary & stacktrace**: Workflows → Jobs → (Run) → Output
* **Spark UI**: From the run or cluster → **Spark UI**
  + **Jobs**: which action failed
  + **Stages**: shuffle boundaries, long-tail stages
  + **Tasks**: skew, spill, GC time outliers
  + **SQL**: logical/physical plan; **Exchange** (shuffle), **BroadcastHashJoin**
  + **Environment**: effective configs
* **Logs**: Compute → (Cluster) → Driver/Worker logs → stdout/stderr
* **Cluster Events**: Compute → (Cluster) → Event Log (autoscaling, spot loss, policy violations, capacity)

**Config snippets you’ll actually use**

# Turn on Adaptive Query Execution (AQE) & skew handling

spark.conf.set("spark.sql.adaptive.enabled", "true")

spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")

spark.conf.set("spark.sql.adaptive.advisoryPartitionSizeInBytes", "256MB") # tune

# Tame shuffles & network

spark.conf.set("spark.sql.shuffle.partitions", "auto") # or a concrete number like "400"

spark.conf.set("spark.shuffle.io.maxRetries", "10")

spark.conf.set("spark.shuffle.io.retryWait", "5s")

spark.conf.set("spark.network.timeout", "600s")

spark.conf.set("spark.executor.heartbeatInterval", "60s")

# Safer exploration (avoid driver OOM)

spark.conf.set("spark.driver.maxResultSize", "1g") # guardrail, not a silver bullet

# …and prefer limit() + writes over collect()/toPandas()