10 Spark Performance Questions for $180k+ Databricks Data Engineer Roles

Spark optimization is THE interview differentiator - every $180k+ Databricks role tests your ability to diagnose and fix performance problems, not just write queries.

Senior interviews expect you to explain WHY a job is slow, WHAT metrics to check, and HOW to fix it without trial-and-error guessing.

🎯 1: Your Spark job has 200 tasks, but 1 takes 30 minutes while others finish in 10 seconds. What's happening?

⚠️ **The Junior Answer (Rejected)**:

"One task is processing more data. I'd add more executors."

Why this fails: Doesn't identify root cause - data skew. More executors won't help if one key has 90% of the data.

✅ **The Senior Answer (Offer-Winning)**:

"This is classic data skew - one partition has disproportionate data volume. I'd check Spark UI's Stage view for task duration distribution, identify the skewed key using df.groupBy('key').count(), then apply salting: append random suffix to skewed keys, perform the join, then remove the suffix. For aggregations, I'd use two-phase aggregation with partial pre-aggregation before the final shuffle."

Key concepts to mention:

• Task duration distribution in Spark UI Stage view

• Data skew identification via groupBy analysis

• Salting technique for join skew

• Two-phase aggregation for aggregation skew

• Adaptive Query Execution (AQE) skew join optimization

💡 **What Interviewers Are Testing**:

Diagnostic methodology and understanding of distributed data problems. Shows you can identify AND solve performance bottlenecks systematically.

🎯 2: When would you use broadcast join vs sort-merge join, and how do you force the decision?

⚠️ **The Junior Answer** (**Rejected**):

"Broadcast join for small tables, sort-merge for large tables."

Why this fails: Doesn't explain thresholds, trade-offs, or how to override Catalyst decisions.

✅ **The Senior Answer** (**Offer-Winning**):

"Broadcast join copies the smaller table to all executors, eliminating shuffle - ideal when one side fits in executor memory (default threshold is 10MB via spark.sql.autoBroadcastJoinThreshold). Sort-merge join shuffles both tables by join key, sorts them, then merges - better for large-large joins. I'd force broadcast with broadcast() hint when I know the dimension table is small but stats are stale, or disable it entirely for memory-constrained clusters. The key trade-off: broadcast eliminates shuffle cost but requires sufficient executor memory (each executor stores a copy) and network broadcast overhead."

Key concepts to mention:

• autoBroadcastJoinThreshold configuration (default 10MB)

• broadcast() hint to force broadcast join

• Shuffle cost vs memory/network trade-off

• Statistics staleness affecting Catalyst decisions

• Executor memory pressure from broadcast replication

💡 **What Interviewers Are Testing**:

Understanding of join internals and when to override optimizer decisions. Shows you can tune performance beyond default configurations.

🎯 3: How do you diagnose why a Spark job suddenly got 5x slower without code changes?

⚠️ **The Junior Answer (Rejected):**

"Something changed in the data. I'd check if the table got bigger."

Why this fails: Too vague - doesn't provide a systematic diagnostic approach.

✅ **The Senior Answer (Offer-Winning):**

"I'd systematically check: 1) Data volume changes - did input size increase significantly? 2) Data distribution changes - did a previously uniform key become skewed? 3) Statistics staleness - did ANALYZE TABLE stop running, causing bad join strategies? 4) Cluster resource contention - are other jobs competing for the same resources? 5) Small file proliferation - did recent ingestion create thousands of tiny files? I'd compare Spark UI metrics between good and bad runs: shuffle read/write sizes, task counts, and stage durations."

Key concepts to mention:

• Systematic diagnostic methodology

• Statistics staleness affecting query plans

• Small file problem from streaming ingestion

• Spark UI comparison between runs

• Resource contention from multi-tenant clusters

💡 **What Interviewers Are Testing:**

Production debugging skills and systematic thinking. Shows you can diagnose real-world issues without panicking.

🎯 4: What's the difference between narrow and wide transformations, and why does it matter for performance?

⚠️ **The Junior Answer (Rejected)**:

"Narrow transformations are faster because they don't shuffle data."

Why this fails: Doesn't explain the underlying mechanism or provide examples.

✅ **The Senior Answer (Offer-Winning)**:

"Narrow transformations (map, filter, select) operate on data within the same partition - one input partition maps to one output partition, allowing pipelining without data movement. Wide transformations (groupBy, join, repartition) require shuffle - data from all input partitions may contribute to any output partition, creating a shuffle stage boundary. This matters because shuffles are the most expensive operation: they write to disk, transfer data across the network, and create stage boundaries that prevent pipelining. I'd minimize wide transformations by filtering early, using broadcast joins when possible, and combining multiple aggregations."

Key concepts to mention:

• Partition dependency: 1:1 (narrow) vs N:N (wide)

• Stage boundaries at shuffle points

• Shuffle involves disk I/O and network transfer

• Pipelining only works within a stage

• Filter pushdown to reduce shuffle volume

💡 **What Interviewers Are Testing**:

Fundamental understanding of Spark's execution model. Shows you understand why certain operations are expensive.

🎯 5: How do you choose the right number of shuffle partitions (spark.sql.shuffle.partitions)?

⚠️ **The Junior Answer (Rejected)**:

"The default is 200, I just leave it at that."

Why this fails: Default is almost never optimal - shows no understanding of tuning.

✅ **The Senior Answer (Offer-Winning)**:

"A widely-used rule of thumb is to target 100-200MB per partition for balanced parallelism and memory efficiency. For a 100GB shuffle, that's 500-1000 partitions. Too few partitions cause OOM errors and poor parallelism; too many create scheduling overhead and small file problems. With Adaptive Query Execution enabled, I'd set an upper bound and let AQE coalesce small partitions automatically. The formula: shuffle_data_size / target_partition_size, rounded to match available cores. I'd also consider downstream operations - if writing to Delta, align with target file size to avoid additional compaction."

Key concepts to mention:

• Target 100-200MB per partition rule of thumb

• AQE automatic partition coalescing

• OOM risk from too few partitions

• Scheduling overhead from too many partitions

• Alignment with downstream file size targets

💡 **What Interviewers Are Testing**:

Practical tuning knowledge and understanding of resource trade-offs. Shows you can optimize jobs for specific workloads.

🎯 6: Your Spark job is hitting OOM errors. Walk me through your debugging process.

⚠️ **The Junior Answer (Rejected)**:

"I'd increase executor memory."

Why this fails: Throwing memory at the problem without diagnosis - wasteful and often doesn't fix root cause.

✅ **The Senior Answer (Offer-Winning)**:

"First, I'd identify WHERE the OOM occurs - driver or executor? Driver OOM often means collecting too much data (avoid collect() on large datasets) or broadcast tables too large. Executor OOM during shuffle suggests data skew or insufficient shuffle partitions. I'd check Spark UI for: memory usage per executor, GC time percentage (>10% indicates memory pressure), and which stage fails. Solutions depend on cause: increase partitions for skew, leverage shuffle spill to disk (automatic when memory thresholds exceeded), reduce broadcast threshold, or optimize the query to reduce intermediate data size."

Key concepts to mention:

• Driver vs executor OOM have different causes

• collect() and broadcast as driver OOM causes

• GC time as memory pressure indicator

• Automatic shuffle spill to disk when memory thresholds exceeded

• Partition count affecting per-task memory

💡 **What Interviewers Are Testing**:

Systematic debugging skills and understanding of Spark's memory model. Shows you diagnose before throwing resources at problems.

🎯 7: How does Adaptive Query Execution (AQE) improve performance, and when might you disable it?

⚠️ **The Junior Answer (Rejected):**

"AQE makes queries faster automatically, I always leave it on."

Why this fails: Doesn't understand the mechanisms or edge cases.

✅ **The Senior Answer (Offer-Winning):**

"AQE re-optimizes the query plan at stage boundaries using runtime statistics. Three key optimizations: 1) Coalescing shuffle partitions - combines small partitions to reduce scheduling overhead, 2) Converting sort-merge joins to broadcast joins when one side is smaller than expected, 3) Optimizing skew joins by splitting skewed partitions. I might disable it when: I need deterministic partition counts for downstream dependencies, the overhead of re-optimization exceeds benefits for simple queries, or debugging requires predictable execution plans. It's enabled by default in recent Databricks runtimes."

Key concepts to mention:

• Runtime statistics collection at shuffle boundaries

• Partition coalescing for small partition problem

• Dynamic join strategy conversion

• Skew join optimization via partition splitting

• Determinism considerations for downstream systems

💡 **What Interviewers Are Testing:**

Understanding of modern Spark optimizations and their trade-offs. Shows you stay current with platform features.

🎯 8: What's the impact of spark.sql.files.maxPartitionBytes and when would you change it?

⚠️ **The Junior Answer (Rejected)**:

"It controls partition size. I haven't needed to change it."

Why this fails: Doesn't understand when tuning is necessary or the implications.

✅ **The Senior Answer (Offer-Winning)**:

"This setting controls how Spark bins input files into partitions (default 128MB). I'd increase it when: reading many small files to reduce task count and scheduling overhead, or when executor memory is plentiful and I want fewer, larger partitions. I'd decrease it when: files are very large and I need more parallelism, or memory is constrained and I need smaller partitions to avoid OOM. For a table with 10,000 files of 10MB each, default creates ~800 partitions; increasing to 256MB reduces to ~400 partitions, cutting scheduling overhead in half while maintaining sufficient parallelism."

Key concepts to mention:

• File binning for partition creation

• Relationship to task count and parallelism

• Trade-off between parallelism and scheduling overhead

• Interaction with small file problem

• Memory implications of partition size

💡 **What Interviewers Are Testing**:

Understanding of file-to-partition mapping and practical tuning. Shows you can optimize read performance for specific scenarios.

🎯 9: How do you identify and fix a Cartesian product in a Spark query?

⚠️ **The Junior Answer (Rejecteed):**

"Cartesian products happen when you forget the join condition."

Why this fails: Doesn't explain how to identify or fix the issue.

✅ **The Senior Answer (Offer-Winning):**

"Cartesian products explode row counts (table A rows × table B rows) and usually indicate a bug. To identify: check query plan for 'BroadcastNestedLoopJoin' or 'CartesianProduct', look for joins with no equality condition (only non-equality predicates like < or >), or notice sudden 1000x output row increase. Common causes: missing join key, join on nullable columns with null=null matching, or unintended cross joins in complex CTEs. To fix: add proper equality join conditions, filter nulls before joining, or if intentional (rare), ensure small table sizes and use explicit CROSS JOIN for clarity."

Key concepts to mention:

• BroadcastNestedLoopJoin in query plan as warning sign

• Row explosion: A rows × B rows

• Non-equality predicates forcing Cartesian

• Null handling in join conditions

• Explicit CROSS JOIN for intentional cases

💡 **What Interviewers Are Testing:**

Query plan interpretation and debugging skills. Shows you can identify dangerous patterns before they hit production.

🎯 10: What's the difference between cache(), persist(), and checkpoint() - when would you use each?

⚠️ **The Junior Answer (Rejected)**:

"cache() stores data in memory for reuse."

Why this fails: Doesn't distinguish between the three or explain use cases.

✅ **The Senior Answer (Offer-Winning)**:

"cache() is shorthand for persist(MEMORY_AND_DISK) for DataFrames - stores DataFrame for reuse within the same job, maintains lineage for recovery. persist() allows storage level customization (MEMORY_ONLY, DISK_ONLY, MEMORY_AND_DISK_SER for compressed). checkpoint() writes to reliable storage (HDFS/cloud storage) and TRUNCATES lineage - used when lineage gets too long and recomputation would be expensive. I'd use cache() for iterative ML algorithms with multiple passes over data, persist(MEMORY_AND_DISK_SER) when memory is tight, and checkpoint() in long streaming jobs or iterative graph algorithms where lineage causes driver OOM."

Key concepts to mention:

• cache() = persist(MEMORY_AND_DISK) for DataFrames

• Storage level options (MEMORY_ONLY, DISK_ONLY, _SER variants)

• Lineage preservation (cache/persist) vs truncation (checkpoint)

• Checkpoint requires reliable distributed storage

• Lineage length as driver memory concern

💡 **What Interviewers Are Testing**:

Understanding of Spark's caching mechanisms and fault tolerance model. Shows you can optimize iterative workloads and manage memory effectively.