**8. SQL Query Optimization**

**Scenario:**  
Your downstream analytics team complains that their queries on the transactions fact table are **too slow** (minutes instead of seconds). As a Data Engineer, how do you diagnose and optimize?

**Baseline Approach (Expected in Interviews):**

* **Step 1 – Diagnosis**
  + Check query plan (EXPLAIN or Spark UI query plan).
  + Identify full table scans, skewed joins, redundant stages.
  + Check partition pruning — are filters aligned with partition columns?
* **Step 2 – Common optimizations**
  + **Partitioning & clustering:**
    - Partition facts on event\_date.
    - Use Z-ORDER or clustering on high-cardinality filter columns (customer\_id).
  + **Indexing / materialization:**
    - If using BigQuery/Redshift: clustering, materialized views.
    - If Delta Lake: Delta cache, Z-ORDER.
  + **Column pruning:**
    - Select only required columns (SELECT customer\_id, amount vs SELECT \*).
    - Store in Parquet/Delta for columnar reads.
  + **Join optimization:**
    - Broadcast small dimension tables (broadcast(dim\_customers)).
    - Use join hints where appropriate.
  + **Pre-aggregation / summary tables:**
    - Build aggregate tables (e.g., daily\_sales, monthly\_sales).
    - Expose BI queries to these instead of raw fact tables.
  + **File layout optimization:**
    - Compact small files (optimize to larger parquet files ~128MB).
    - Use auto-optimize & auto-compaction in Databricks.

**Code snippets:**

**1. Check query plan**

EXPLAIN SELECT customer\_id, SUM(amount)

FROM transactions

WHERE event\_date BETWEEN '2025-01-01' AND '2025-01-31'

GROUP BY customer\_id;

**2. Partitioned Delta Table**

(df.write

.format("delta")

.partitionBy("event\_date")

.mode("overwrite")

.save("/mnt/delta/transactions"))

**3. Z-ORDER for query acceleration**

OPTIMIZE transactions

ZORDER BY (customer\_id)

**4. Broadcast join**

from pyspark.sql.functions import broadcast

result = fact\_df.join(broadcast(dim\_df), "customer\_id")

**Advanced considerations (Interview “extra mile” answers):**

* **Predicate pushdown:** Ensure filters push down to Parquet/Delta storage.
* **Caching:** Cache dimension tables in memory for repeated joins.
* **Skew handling:** If a customer has 10M rows but most have few → add salting (customer\_id || random\_salt).
* **Star schema modeling:** Normalize dimension tables, keep fact narrow.
* **BigQuery/Redshift specifics:**
  + BigQuery: partition by ingestion\_time, cluster on customer\_id, use partition filters.
  + Redshift: diststyle & sortkeys, vacuum analyze.

**Follow-up Q&A**

**Q1. How do you decide between partitioning and clustering?**

* Partition on low-cardinality columns used in filters (event\_date, region).
* Cluster on high-cardinality columns with frequent filters (customer\_id).
* Too many partitions = small files problem. Balance is key.

**Q2. What if analysts query wide time ranges, e.g., 2 years?**

* Large scan → solution: pre-aggregate summary tables (monthly aggregates).
* Use caching for common query sets.
* Z-ORDER clustering helps for column filters beyond time.

**Q3. How would you optimize a join where one table has skewed keys?**

* Apply salting technique (concat(customer\_id, random())).
* Use skew hints (/\*+ SKEWJOIN \*/).
* Broadcast if one side is small enough.

**Q4. What’s your approach if queries are still slow after optimizations?**

* Profile workloads: are queries ad hoc or repeated?
* Build a semantic layer (Cube, dbt metrics, Databricks SQL).
* Materialize hot queries into views or cache results.
* Use adaptive query execution (AQE) in Spark for better join strategy at runtime.

**One-page cheat sheet (memorize):**

* **Diagnosis:** EXPLAIN → partition pruning → joins → file sizes.
* **Techniques:** partition (low-cardinality), Z-ORDER/cluster (high-cardinality), column pruning, broadcast joins, compaction, pre-aggregates.
* **Skew:** salting, skew hints, AQE.
* **Tools:** Databricks OPTIMIZE, BigQuery clustering, Redshift sort/dist keys.
* **Tradeoffs:** partition explosion vs scan efficiency, pre-aggregates vs freshness.

✅ With this, you now have ingestion → CDC → schema evolution → optimizations → DQ → query tuning stitched together.