**🔑 Spark Optimizations in Relation to Executor Memory & Performance**

**1. Z-Ordering (Databricks / Delta Lake specific)**

**Concept:** Multi-dimensional clustering. Data is written in a way that rows with similar values across multiple columns are colocated in the same files.

* **When to use:**
  + Query patterns filter on **multiple columns** (e.g., WHERE region = 'APAC' AND product = 'Loan').
  + Useful when partitioning on multiple dimensions is impractical.
* **Pros:**
  + Reduces I/O by pruning files intelligently.
  + Better than single-column partitioning for high-cardinality queries.
* **Cons:**
  + Needs rewrite/maintenance (expensive for huge tables).
  + Only available in Delta Lake.
* **Relation to Executor Memory:**
  + Since fewer files are scanned, **less storage memory pressure** for caching and less execution memory required during scans.
  + Caching z-ordered data is efficient since filtered subsets are smaller.
* **Real Example:**
  + BFSI: Customer transactions queried by **account\_id + date** → Z-ordering speeds up fraud detection queries.

**2. Liquid Clustering (Databricks advanced feature)**

**Concept:** Dynamic clustering where Delta Lake reorganizes data continuously in the background without heavy rewrite jobs.

* **When to use:**
  + On **slowly evolving large fact tables** where data arrives continuously (e.g., daily trades, transactions).
* **Pros:**
  + No need for static partitioning upfront.
  + Handles schema evolution + re-clustering automatically.
* **Cons:**
  + Proprietary to Databricks (not open-source Spark).
  + May introduce overhead in write throughput.
* **Relation to Executor Memory:**
  + Reduces **shuffle-heavy re-clustering jobs**, so execution memory usage goes down.
  + Cached queries benefit since data layout stays optimized over time.
* **Real Example:**
  + Market trades table in BFSI (billions of rows daily) → Liquid clustering ensures queries by **symbol + trade\_date** remain performant without manual partitioning.

**3. Salting (Skew handling)**

**Concept:** Add an artificial "salt" key to distribute skewed data (e.g., single key with millions of rows).

* **When to use:**
  + Skewed joins (e.g., one customer with 90% of records).
  + Aggregations with heavy key skew.
* **Pros:**
  + Prevents executor OOM caused by single executor holding all skewed records.
  + Better parallelism across executors.
* **Cons:**
  + Extra complexity in ETL logic.
  + Must "desalt" during final aggregations.
  + Overhead of new salt column being present
* **Relation to Executor Memory:**
  + Prevents one executor’s **execution memory** from blowing up (OOM in joins).
  + Balances shuffle blocks, reducing memory pressure during shuffle writes/reads.
* **Real Example:**
  + Insurance claims aggregation where **1 hospital contributes 60%** of records. Adding salt distributes that hospital’s claims across multiple executors.

**4. Broadcast Join**

**Concept:** Send a small table to all executors to avoid shuffle.

* **When to use:**
  + One table is small (<10–500 MB, configurable by spark.sql.autoBroadcastJoinThreshold).
  + Star-schema joins (dimension → fact).
* **Pros:**
  + Eliminates shuffle → huge speedup.
  + Simple config (broadcast(df) or auto by optimizer).
* **Cons:**
  + Small table must fit in **executor storage memory**.
  + Risk of OOM if broadcast size exceeds available memory.
* **Relation to Executor Memory:**
  + Broadcasted data lives in **storage memory** (like cache).
  + If memory is tight, LRU eviction may remove broadcast data, forcing recomputation.
* **Real Example:**
  + BFSI: Join a **customer dimension (200 MB)** with a **transactions fact table (10 TB)** → broadcasting avoids a massive shuffle.

**5. Partitioning (Data Layout / ETL)**

**Concept:** Store data in physical folders based on a column (e.g., date=2025-08-31).

* **When to use:**
  + Large append-only fact tables.
  + Queries often filter by partition keys (date, region, etc.).
* **Pros:**
  + Prunes files → less I/O.
  + Easy to implement.
* **Cons:**
  + Over-partitioning → too many small files.
  + Not efficient for high-cardinality keys (e.g., customer\_id).
* **Relation to Executor Memory:**
  + Reduces amount of data loaded into execution/storage memory (only relevant partitions scanned).
  + Over-partitioning can increase overhead, since executors load lots of small files into memory.
* **Real Example:**
  + Credit card transactions table partitioned by **transaction\_date** for daily reporting.

**6. Bucketing**

**Concept:** Hash-based distribution of rows into fixed number of buckets (files) by column.

* **When to use:**
  + Frequent joins on the same column.
  + When partitioning is impractical (high cardinality).
* **Pros:**
  + Avoids shuffle for bucketed joins (if both tables bucketed on same column).
  + Good balance between partitioning and shuffling.
* **Cons:**
  + Bucket count is fixed → inflexible with data growth.
  + Write cost is higher.
* **Relation to Executor Memory:**
  + Avoids shuffle → reduces **execution memory** pressure.
  + Cached bucketed tables are efficient since files align with join keys.
* **Real Example:**
  + Customer and transactions tables bucketed by **customer\_id** → Spark avoids shuffle during joins.