DATABRICKS OPTIMIZATION

AUTHOR: BIBHU

Introduction

Optimizing Databricks workloads is crucial for enhancing performance, reducing costs, and minimizing errors caused by inefficient resource utilization. Based on my experience, performance degradation typically occurs in the following key phases. While these points address significant optimization aspects, they do not cover 100% of all scenarios—I will continue to refine and expand this list as I encounter new challenges.

1. **Data Shuffling Phase** – Optimizing how data is redistributed across partitions to minimize shuffle overhead.
2. **Data Transformation Phase** – Enhancing data processing efficiency by optimizing operations such as joins, aggregations, and filtering.
3. **Cluster Configuration Phase** – Adjusting compute resources to balance cost and performance, including instance types, autoscaling, and parallelism.
4. **Storage & Query Performance Phase** – Improving data storage structures and query execution plans to accelerate processing.
5. **Rewrite Phase** – Managing data modifications, including updates, deletions, and merges, to ensure efficient reprocessing and storage performance.

Each section provides insights into common performance bottlenecks, their underlying causes, and step-by-step solutions to optimize workloads effectively.

1. Data Shuffling Optimization

What is Shuffle in SQL Queries?

Shuffle occurs when data is redistributed across multiple nodes or partitions during query execution. This is often triggered by operations such as **GROUP BY, JOIN, ORDER BY, and DISTINCT**, where data needs to be grouped or sorted based on different keys.

Why Does Shuffle Happen?

- **GROUP BY**: Data needs to be grouped based on a column that may not be pre-partitioned.
- **JOIN**: When two tables are joined on a non-partitioned column, data from both tables needs to be shuffled to bring matching keys together.
- **ORDER BY**: Sorting requires merging and redistributing data across multiple nodes.
- **DISTINCT**: Similar to GROUP BY, data needs to be shuffled to ensure unique values are identified globally.

Common Issues with Data Shuffling:

✕ High network overhead due to excessive data movement.

✕ Slow execution times caused by inefficient joins and aggregations.

✕ Data spilling to disk when memory is insufficient, leading to performance degradation.

Example: Detecting Shuffle in SQL Queries

We analyze the following SQL query using **EXPLAIN FORMATTED** to check for data shuffling.

```
EXPLAIN FORMATTED

SELECT customer_id, COUNT(*)

FROM transactions

GROUP BY customer_id;
```

Step-by-Step Execution Analysis

Understanding the Query

- We are counting the number of transactions for each customer_id.
- The database needs to group the data by customer_id, which may require **shuffling** if the data is not already partitioned on this column.

Running EXPLAIN FORMATTED

- EXPLAIN FORMATTED provides a **detailed execution plan**, showing how the query is processed.
- We look for **EXCHANGE** operators in the execution plan, which indicate **data shuffling**.

Sample Execution Plan

(Your actual output may differ based on the database engine, but it will have similar components.)

```
== Physical Plan ==

*HashAggregate (2)

- Exchange (1)  << SHUFFLE DETECTED

 - HashAggregate (0)

  - Scan [transactions]
```

Breaking it down:

- **(0) HashAggregate**: The first stage of aggregation occurs at the local node level.
- **(1) Exchange**: Data is shuffled across different partitions to group all records of the same customer_id together.
- **(2) HashAggregate**: The final aggregation is performed after shuffling.

1.1 How to Reduce Shuffle?

Optimize Partitioning

- If your data is stored in a **partitioned table**, ensure that the **partition key matches the GROUP BY column**.
- Example: If transactions is partitioned by customer_id, then **GROUP BY customer_id** will have minimal shuffle.

```
CREATE TABLE transactions_partitioned
USING PARQUET
PARTITIONED BY (customer_id)
AS SELECT * FROM transactions;
```

Use BROADCAST JOIN (for JOIN queries)

- If one table is small, you can use a **BROADCAST JOIN** to avoid shuffling.

```
SELECT /*+ BROADCAST(customers) */
t.customer_id, COUNT(*)
FROM transactions t
JOIN customers c ON t.customer_id = c.customer_id
GROUP BY t.customer_id;
```

Reduce the Data Before Shuffle

- Apply filters (WHERE clause) **before GROUP BY** to reduce the amount of shuffled data.

```
SELECT customer_id, COUNT(*)
FROM transactions
WHERE transaction_date >= '2024-01-01'
GROUP BY customer_id;
```

1.2 Broadcast Hash Join to Avoid Shuffling (Some additional points)

- **Why Use Broadcast Join?**

Broadcasting small tables (<10MB) prevents unnecessary shuffling by sending the table to all worker nodes.

- **Steps to Enable Broadcast Join**

```
Enable automatic broadcasting (default threshold: 10MB
SET spark.sql.autoBroadcastJoinThreshold = 10485760; -- 10MB
```

- **Force Broadcast Join using Query Hint**

DATABRICKS OPTIMIZATION AUTHOR: BIBHU

A Broadcast Join is a join optimization technique used in distributed computing frameworks like Apache Spark and Databricks. It works by replicating (broadcasting) a smaller dataset to all nodes where the larger dataset is stored, reducing data shuffling and improving query performance.

By forcing a Broadcast Join using a query hint, we explicitly instruct the SQL engine to treat the specified table as a broadcasted table, even if it might not choose to do so automatically.

Query Explanation:

```
SELECT /*+ BROADCAST(customer_data) */ *

FROM transactions

JOIN customer_data ON transactions.customer_id = customer_data.customer_id;
```

/*+ BROADCAST(customer_data) */ → This is a query hint that forces the customer_data table to be broadcasted across all nodes.

transactions → This is the larger table.

customer_data → This is the smaller table (broadcasted to all nodes).

The JOIN condition is based on customer_id.

When to Use Force Broadcast Join

1. **Smaller Table**: The table being broadcasted (customer_data) should be **small enough to fit into memory** on each node.
2. **Avoiding Data Shuffle**: Helps reduce **network transfer costs** and **execution time** in a distributed system.
3. **Improving Performance**: Useful when **joining a large table with a much smaller table**, as it eliminates the need for a full shuffle.

When NOT to Use Force Broadcast Join

1. **Large Table as Broadcast**: If customer_data is too large, it can cause **memory overflow** or **performance degradation**.
2. **Cluster Constraints**: If memory is limited, broadcasting can **slow down execution** instead of speeding it up.
3. **Default Optimizer Decision is Better**: If the query optimizer **automatically selects a more efficient join strategy**, forcing a broadcast might not be necessary.

Verify Broadcast in Spark UI

Open Spark UI

1. Run your Spark application.
2. Open **Spark UI** in your browser:
    - If using **Databricks**, go to **"Query Details"** under "SQL Query History".
    - If using **standalone Spark**, access Spark UI via “http://<driver-node-ip>:4040”
    - Navigate to the "SQL" Tab In the Spark UI, go to the "SQL" tab.
    - Click on the executed query to open "Query Details".
    - Look for "BroadcastExchange" Operator Under the "DAG Visualization" or "Query Plan", find BroadcastExchange (HashedRelationBroadcastMode)
    - This confirms that Spark successfully broadcasted the customer_data table.

1.3 Adaptive Query Execution (AQE) for Dynamic Optimization

What is AQE?

Adaptive Query Execution (AQE) is a feature in Apache Spark 3.0+ that dynamically optimizes query execution at runtime based on actual data statistics. Unlike traditional static query plans, AQE allows Spark to adjust join strategies, optimize shuffle partitions, and change execution plans dynamically.

Traditional vs. Adaptive Query Execution

Feature	Without AQE (Static Execution)	With AQE (Dynamic Execution)
Join Strategy	Fixed at compile time	Adjusts based on data size
Shuffle Partitioning	Predefined by spark.sql.shuffle.partitions	Dynamically adjusted to avoid skew
Skewed Joins Handling	Not optimized	Detects skewed data and optimizes joins
Coalesce Shuffle Partitions	Manual tuning required	Automatically reduces partitions for efficiency

Key Optimizations Done by AQE

1. Dynamically Switch Join Strategies → Changes between Sort-Merge Join (SMJ) and Broadcast Hash Join (BHJ) based on data size.
2. Optimize Shuffle Partitions → Adjusts the number of partitions dynamically to reduce shuffle size and improve performance.
3. Handle Skewed Joins → Detects data skew and applies optimized join strategies to avoid slow execution.
4. Eliminate Unnecessary Shuffles → Removes redundant shuffle operations.

Steps to Enable AQE

- **Ensure AQE is enabled (default in Databricks)**

  SET spark.sql.adaptive.enabled = true;

- **Enable automatic broadcast join threshold for AQE**

  SET spark.databricks.adaptive.autoBroadcastJoinThreshold = 209715200; -- 200MB

- **Rerun the query and check shuffle size reduction in Spark UI.**

Check the Query Plan for AQE Optimizations

Look for the following indicators in Spark UI → SQL Tab → Query Execution Details:

1. "Adaptive Query Stage" → Confirms that AQE is applied.
2. "BroadcastExchange" → Shows that AQE automatically enabled broadcast joins.
3. "Shuffle Partition Coalesce" → Confirms that AQE optimized shuffle partitions.

1.4 Shuffle Hash Join vs. Sort Merge Join in Apache Spark

Step 1: Understanding the Differences

What is Sort Merge Join?

- Sort Merge Join (SMJ) is the default join strategy in Spark when two large tables are joined.
- It first sorts both datasets on the join key and then merges them efficiently.
- Best suited for large datasets where one table does not fit in memory.
- Downside: Sorting is expensive, and it requires shuffling before joining.

Example Execution Steps of Sort Merge Join:

1. Spark sorts both tables based on the join key.
2. Shuffle operation occurs to bring matching keys to the same partition.
3. The sorted tables are merged using an efficient merge operation.

What is Shuffle Hash Join?

- Shuffle Hash Join (SHJ) is faster than Sort Merge Join when one of the datasets can fit into memory.
- Instead of sorting, it creates a hash table for the smaller dataset and performs lookups while scanning the larger dataset.

- Best suited when at least one table is small enough to fit in memory.
- Downside: If the table is too large to fit in memory, it may cause out-of-memory errors.

Example Execution Steps of Shuffle Hash Join:

1. Shuffle occurs to ensure that matching keys are in the same partition.
2. The smaller table is loaded into memory as a hash table.
3. The larger table is scanned, and the join is performed using hash lookups instead of sorting.

Enabling Shuffle Hash Join in Spark:

1. **Disable Sort Merge Join (Force Shuffle Hash Join)**
   By default, Spark prefers Sort Merge Join. To force Shuffle Hash Join, disable Sort Merge Join:
   SET spark.sql.join.preferSortMergeJoin = false;
   This tells Spark not to use Sort Merge Join, allowing Shuffle Hash Join to be chosen automatically when possible.
2. **Adjust Auto Broadcast Join Threshold**
   For small tables, enabling Broadcast Join can further optimize performance.
   SET spark.sql.autoBroadcastJoinThreshold = -1; -- Disable broadcast join to ensure SHJ is used.
3. **Comparing Performance Before and After**

Join Type	Best Use Case	Execution Overhead	Spark UI Indicators
Sort Merge Join	Large datasets that do not fit in memory	Sorting and shuffling	"SortMergeJoin" in Query Plan
Shuffle Hash Join	When one dataset fits in memory	Shuffle only (No Sorting)	"ShuffledHashJoin" in Query Plan

4. **Expected Results After Forcing Shuffle Hash Join:**
   Faster execution time when the smaller table fits in memory.
   No sorting operations in the query plan.
   Lower shuffle write size compared to Sort Merge Join

1.5 Cost-Based Optimizer (CBO)

What is Cost-Based Optimization (CBO)?

- **Cost-Based Optimizer (CBO)** is a query optimization technique in Spark that uses **table statistics** to choose the most efficient execution plan.
- **Without CBO:** Spark uses **Rule-Based Optimization (RBO)**, which follows **static rules** without considering data distribution.
- **With CBO:** Spark selects the **best join order, join type, and execution plan** by analyzing **data size, cardinality, and distribution.**

Benefits of CBO:

- **Reduces query execution time** by selecting efficient join strategies.
- **Minimizes shuffle and memory usage** by optimizing data distribution.
- **Improves parallelism** by balancing partition workloads.

Step-by-Step Guide to Enable & Use CBO in Spark:

- **Enable Cost-Based Optimization (CBO)**

  By default, CBO is disabled in Spark. Enable it using the following SQL command:

  SET spark.sql.cbo.enabled = true;

Why Enable CBO?

- Allows Spark to analyze **table statistics** and select an **optimal execution plan**.
- Helps **choose the best join order** in multi-table joins.

Summary of CBO Optimizations

Optimization	Command / Setting	Benefit
Enable CBO	SET spark.sql.cbo.enabled = true;	Allows Spark to use statistics for query optimization
Compute Table Statistics	ANALYZE TABLE transactions COMPUTE STATISTICS;	Helps Spark decide best join strategy
Compute Column Statistics	ANALYZE TABLE transactions COMPUTE STATISTICS FOR COLUMNS customer_id;	Improves filter pushdown & partition pruning
Enable Join Reordering	SET spark.sql.cbo.joinReorder.enabled = true;	Ensures Spark processes smaller tables first
Check Execution Plan in UI	**Spark UI → SQL Tab → Query Plan**	Verify if optimizations are applied

2. Data Transformation Optimization

Common Issues

- High storage costs due to excessive small files.
- Poor query performance due to unoptimized file structures.
- Slow ingestion speed impacting job SLAs.

2.1 Identify the Small Files Problem

Definition:

Small files problem occurs when a large number of tiny files are written to storage, leading to inefficient queries, high metadata overhead, and increased costs.

How to Identify Small Files?

1. **Check the Storage System:**
   - Use the **Databricks UI**, Azure Data Lake, or AWS S3 console to inspect the file sizes.
   - Look for a large number of files < 128MB.
2. **Run the following query to count files in a Delta table:**

DESCRIBE DETAIL table_name;

   - This returns metadata, including numFiles, sizeInBytes, and min/maxFileSize.
   - If numFiles is **significantly high**, the table needs **optimization**.

Example Output:

numFiles	sizeInBytes	minFileSize	maxFileSize
5000	1TB	50KB	150MB

Explanation of Each Column:

1. **numFiles (5000):**
   - The total number of Parquet files in the Delta table.
   - **5000 files** indicate a **high number of small files**, which may impact query performance and storage costs.
2. **sizeInBytes (1TB):**
   - The total size of the table in **bytes** (converted here to **1TB**).
   - This means **5000 files together occupy 1TB** of storage.

3. **minFileSize (50KB):**
    - The **smallest** file in the table is **50KB**.
    - This is **too small** for efficient querying, leading to **high metadata overhead**.
4. **maxFileSize (150MB):**
    - The **largest** file in the table is **150MB**.
    - Ideally, **Delta tables should have file sizes close to 128MB–256MB** for optimal performance.

What Does This Indicate?

- The table has **too many small files**, causing:
    - **High storage costs** (due to inefficient file distribution).
    - **Slow query performance** (more files → more metadata reads).
    - **Longer job execution times** (excessive I/O operations).

Recommended Action:

- **Run OPTIMIZE to merge small files**:

  OPTIMIZE table_name;

- **Enable Auto-Optimize to prevent future small files**:

  ALTER TABLE table_name SET TBLPROPERTIES (

  'delta.autoOptimize.optimizeWrite' = true,

  'delta.autoOptimize.autoCompact' = true

  );

- **Consider partitioning if appropriate**, but avoid over-partitioning.

  This will help **reduce the number of files**, improve **query speed**, and lower **storage costs**.

3. Cluster Configuration Optimization

Common Issues

1. **Underutilized or Over-Provisioned Clusters**
    - Clusters are either too large (wasting resources) or too small (causing slow execution).

2. **High Compute Costs due to Misconfigured Resources**
    - Inefficient configurations lead to **unnecessary spending**.
3. **Frequent Data Spilling to Disk**
    - When memory runs out, Spark writes data to disk, **slowing down performance**.

3.1 Tune Shuffle Partitions

Definition

Shuffle partitions control how data is **distributed and processed** across executors. A wrong number of partitions can cause **memory issues and slow performance**.

Steps to Optimize

1. **Set shuffle partitions to auto-adjust (Databricks 12+ supports this)**

```
set spark.sql.shuffle.partitions = auto;
```

2. **Manually adjust based on cluster size (if needed)**
    - Default is **200** partitions, but adjust based on data size:

```
set spark.sql.shuffle.partitions = <number>;
```

    - General rule:
        - **Small dataset (<10GB) → 50-100 partitions**
        - **Medium dataset (10GB-1TB) → 200-1000 partitions**
        - **Large dataset (>1TB) → 1000+ partitions**

3.2 Preventing Data Spilling

Definition

Data spilling occurs when Spark runs out of memory and **writes intermediate data to disk**, slowing down execution.

Steps to Reduce Spilling

1. **Check the Spark UI for spill metrics**
    - Look at **"Tasks" → "Spill (disk)"** in **Spark UI**.
    - If spilling is frequent, adjust memory allocation.
2. **Adjust Adaptive Query Execution (AQE) shuffle partition size**

```
set spark.databricks.adaptive.autoOptimizeShuffle.preshufflePartitionSizeInBytes =
16777216;
```

    - This **reduces partition size** to **16MB** (adjust based on workload).
    - Larger partitions reduce overhead but can cause memory pressure.

3.3 Enable Cluster Autoscaling

Definition

Autoscaling **automatically adjusts** the number of workers in a cluster based on the workload, ensuring **efficient resource use**.

Steps to Enable Autoscaling:

1. **Enable autoscaling when creating a cluster**

- Set **Min Workers** and **Max Workers** to allow Databricks to scale up/down as needed.

2. **Monitor utilization in Spark UI**
   - If **CPU usage is low**, reduce the max worker count to save costs.
   - If **jobs are slow**, increase the min worker count.

3.4 Leverage Photon Engine

Definition

Photon is a **high-performance query engine** optimized for **Apache Spark SQL**. It improves query execution speed by using **vectorized processing** and **native execution**.

Steps to Enable Photon Engine

1. **Use Photon when creating a cluster**
   - In **Databricks UI → Compute → Create Cluster**, enable **"Use Photon"**.
2. **Run SQL workloads on Photon for faster execution**
   - Works best with **Delta Lake and SQL-heavy workloads**.

4. Storage and Query Performance Optimization

Common Issues

1. **Slow Query Performance due to Fragmented Storage**
   - Data is spread across multiple small files, making reads inefficient.
2. **High I/O Costs from Inefficient File Structures**
   - Poor file management leads to increased read/write operations, increasing costs.
3. **Suboptimal Caching Leading to Redundant Computations**
   - Data is repeatedly recomputed instead of being cached, slowing down processing.

4.1 Table Maintenance with ANALYZE TABLE

Definition

ANALYZE TABLE collects metadata and statistics about a table to help the query optimizer generate **efficient execution plans**.

Steps to Optimize

1. **Compute statistics for query optimization:**

```
ANALYZE TABLE table_name COMPUTE STATISTICS;
```

- This updates **column-level statistics**, improving **filtering and join operations**.

2. **For column-specific statistics (improves query pruning and indexing):**

```
ANALYZE TABLE table_name COMPUTE STATISTICS FOR COLUMNS column1, column2;
```

- Helps **Databricks' Catalyst Optimizer** determine the **best execution plan**.

4.2 Delta Lake Optimization

Definition

Delta Lake **optimizes file storage** by automatically managing file sizes and **compacting small files into larger ones**, reducing **fragmentation and read latencies**.

Steps to Optimize

1. **Enable automatic file size management:**

```
ALTER TABLE table_name SET TBLPROPERTIES (
'delta.tuneFileSizesForRewrite' = true
);
```

- Automatically adjusts **file sizes** during **compaction** to balance **performance and cost**.

2. **Run OPTIMIZE to merge small files (manually triggered optimization):**

```
OPTIMIZE table_name;
```

- Merges small files into **larger Parquet files**, improving **query performance**.

3. **Apply Z-Ordering (for faster lookups on frequently queried columns):**

```
OPTIMIZE table_name ZORDER BY (high_cardinality_column);
```

- **Reorders data layout** to improve performance when filtering by the given column.

4.3 Efficient Data Caching

Definition

Caching **stores frequently used data in memory**, reducing **the need for repeated computations and disk reads**, significantly improving **query performance**.

Steps to Optimize

1. **Cache frequently accessed tables:**

```
CACHE TABLE table_name;
```

- Stores the table **in memory** for faster retrieval.

2. **Use the persist() method for selective caching in PySpark (if needed):**

   ```
   df = spark.read.table("table_name").persist()
   df.show()
   ```

   - This keeps the **DataFrame in memory**, avoiding reloading from disk.

3. **Clear the cache if it's not needed to free memory:**

   ```
   CLEAR CACHE;
   ```

   - Removes all cached tables, **preventing memory overflow**.

5.Rewrite Phase

The **Rewrite Phase** in Databricks refers to the process of modifying and reprocessing data in storage due to updates, deletions, or merges. Since Parquet files (used in Delta Lake) are immutable, any updates or modifications require rewriting the affected data files instead of modifying records in place. This can lead to **high resource consumption, memory spills, and increased execution time** if not optimized properly.

Breakdown of the Rewrite Phase

1. Identifying When Rewrite Happens

Rewrite occurs in scenarios like:

- **MERGE operations** (e.g., updating Delta tables with new data).
- **DELETE operations** (deleting specific records in Delta tables).
- **UPDATE operations** (modifying existing records).

2. Understanding Data Rewriting Mechanics

- Delta Lake stores data in Parquet files, which **cannot be updated directly**.
- When a record is updated, **the entire affected Parquet file(s) must be rewritten**.
- If a small number of records require modification, but they belong to large files, the entire file must be rewritten, leading to **write amplification** (rewriting more data than necessary).

3. Key Challenges in the Rewrite Phase

- **High Memory Spillage**: Large-scale rewrites can cause **shuffle spills** and increased memory consumption.
- **Write Amplification**: Updating a few rows may lead to **rewriting millions or billions of rows**, significantly increasing execution time.
- **Concurrency Issues**: Running **OPTIMIZE and ZORDER** on actively used partitions can cause **conflicts with streaming jobs**.

Solution: Deletion Vectors

A Deletion Vector is a logical deletion mechanism in Delta Lake that allows records to be marked as deleted without rewriting entire Parquet files. Instead of physically removing records and rewriting files, deletion vectors maintain a lightweight metadata index, significantly improving performance and reducing write amplification.

Step-by-Step Implementation of Deletion Vectors in Delta Lake

1. Enable Deletion Vectors in Delta Lake

To take advantage of deletion vectors, enable them at the table level using:

```
ALTER TABLE my_table SET TBLPROPERTIES ('delta.enableDeletionVectors' = 'true');
```

This setting ensures that instead of rewriting Parquet files during DELETE or MERGE operations, **Delta Lake marks the rows as deleted** and avoids unnecessary rewrites.

2. Handling DELETE Operations Efficiently

Instead of physically deleting rows and triggering a full rewrite, Delta Lake marks them as deleted using deletion vectors.

Example: Delete Old Records Without Full Rewrite

```
DELETE FROM my_table WHERE event_date < '2023-01-01';
```

- With deletion vectors enabled, **only metadata is updated**, and affected rows are excluded from queries.
- **No Parquet files are rewritten**, reducing resource consumption.

3. Querying Data with Deletion Vectors

Since deleted rows are logically removed, they are automatically excluded from queries.

Example: Query Without Deleted Rows

SELECT * FROM my_table;

Queries automatically ignore logically deleted rows, improving read performance.

Additional Parameters which can optimize

Parameter	Definition	Optimization Category	When to Use?
spark.sql.autoBroadcastJoinThreshold	Controls the size limit for automatically broadcasting tables in joins.	Data Shuffling Optimization	Increase when small lookup tables (default 10MB) are not being broadcasted efficiently in joins.
spark.sql.files.openCostInBytes	Defines the cost associated with opening a file during query execution.	Storage & Query Performance Optimization	Reduce when dealing with many small files to improve query performance.
spark.sql.adaptive.enabled	Enables Adaptive Query Execution (AQE) for dynamic query optimizations.	Data Transformation Optimization	Set to **true** to allow Spark to optimize shuffle partitions and join strategies dynamically.
spark.databricks.delta.optimizeWrite.enabled	Enables optimized writes to reduce the number of small files written to Delta tables.	Rewrite Optimization	Set to **true** when frequent writes cause small files in Delta tables.

spark.databricks.delta.autoCompact.enabled	Enables automatic compaction of small Delta files.	Rewrite Optimization	Set to **true** when Delta table performance degrades due to too many small files.
spark.databricks.delta.optimize.zorder.autoCompact.enabled	Enables automatic Z-Ordering compaction on Delta tables.	Storage & Query Performance Optimization	Use when queries frequently filter on specific columns, reducing unnecessary scans.
spark.databricks.delta.merge.optimizeWrite.enabled	Optimizes MERGE operations to avoid excessive rewrites.	Rewrite Optimization	Set to **true** when MERGE operations are rewriting large portions of the table.
spark.sql.parquet.filterPushdown	Pushes filters down to the Parquet file scan.	Storage & Query Performance Optimization	Set to **true** to minimize data read from Parquet files by applying filters at the file scan level.
spark.sql.join.preferSortMergeJoin	Controls whether Spark prefers Sort-Merge Joins over Broadcast Joins.	Data Shuffling Optimization	Set to **false** when dealing with smaller tables to allow Broadcast Joins instead.
spark.sql.sources.bucketing.enabled	Enables bucketing for efficient joins and aggregations.	Data Shuffling Optimization	Use when working with **large joins or aggregations** that benefit from bucketed tables.

spark.databricks.delta.checkpointInterval	Defines how frequently Delta table checkpoints are written.	Rewrite Optimization	Increase interval for high-frequency updates to reduce checkpoint overhead.
spark.sql.codegen.wholeStage	Enables whole-stage code generation for query execution.	Storage & Query Performance Optimization	Use when queries are CPU-intensive and involve complex transformations.
spark.sql.merge.treeMergeThreshold	Controls the threshold for tree-based merging in Delta Lake.	Rewrite Optimization	Increase this value when **MERGE operations** cause excessive file rewrites.