Delta Lake & Spark | Day 10

Performance Optimization

Databricks 14-Days AI Challenge

January 2026

Agenda

- Introduction to Performance Optimization
- Query Execution Plans
- Partitioning Strategies
- OPTIMIZE & ZORDER
- Caching Techniques
- Best Practices

Why Performance Optimization Matters

The Library Analogy:

- Scattered books = slow searches
- Organized by genre/author = fast lookups

Key Benefits:

- Reduces compute costs
- Improves user experience
- Enables larger dataset processing
- Reduces resource contention

The Four Pillars of Optimization

Pillar	What It Does	When to Use
Query Plans	Shows HOW Spark executes your query	Debugging slow queries
Partitioning	Physically organizes data on disk	Large tables with filter patterns
ZORDER	Optimizes data layout within files	Multi-column filter queries
Caching	Keeps data in memory	Repeated access to same data

Query Execution Plans

A "Recipe" for Your Data: Shows every step Spark takes to transform your query into results.

The Four Phases of Query Planning

Phase 1: Parsed Logical Plan

- Parses SQL into abstract syntax tree
- Checks for syntax errors
- No validation against tables yet

Phase 2: Analyzed Logical Plan

- Validates table/column names
- Resolves data types
- Checks permissions

Phase 3: Optimized Logical Plan

- Catalyst optimizer rules
- Predicate pushdown
- Projection pruning

Phase 4: Physical Plan

- Converts to physical operations
- Chooses join strategies
- Determines data exchange

Understanding explain(True) Output

```
spark.sql("SELECT * FROM silver.events WHERE event_type='purchase'").explain(True)
```

Sample Output:

```
== Physical Plan ==
*(1) Filter (event_type = purchase)
+- *(1) ColumnarToRow
   +- FileScan parquet
      PushedFilters: [EqualTo(event_type,purchase)]
```

Key Indicators:

- `*(1)` = Whole-stage codegen (good!)
- `PushedFilters` = Filter optimized
- `PartitionFilters` = Partition pruning

Key Terms in Physical Plans

Term	Meaning	Performance Impact
FileScan	Reading from disk	Base cost
Filter	Removing rows	Good if pushed to scan
Project	Selecting columns	Good if reduces data
Exchange	Shuffle between nodes	Expensive!
BroadcastExchange	Small table broadcast	Efficient for joins
SortMergeJoin	Both sides sorted	Moderate cost
BroadcastHashJoin	One side broadcast	Best for small tables
HashAggregate	Grouping operation	Memory intensive

Red Flags: Multiple Exchanges | SortMergeJoin on small tables | No PushedFilters

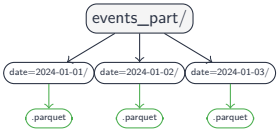
What is Partitioning?

Definition: Divides table into separate folders on disk based on column values.

Filing Cabinet Analogy:

- **Without:** All documents in one drawer
- **With:** Separate drawers for 2022, 2023, 2024
- Finding "2024 docs" = open only one drawer

Benefit: Spark reads only relevant folders instead of scanning everything.



Creating a Partitioned Table

```
CREATE TABLE silver.events_part
USING DELTA
PARTITIONED BY (event_date, event_type)
AS SELECT * FROM silver.events
```

Component	Purpose
`CREATE TABLE`	Creates new table in silver schema
`USING DELTA`	Specifies Delta Lake format (ACID, time travel)
`PARTITIONED BY`	Creates folder hierarchy: event_date → event_type
`AS SELECT *`	Populates with data from source table

Partition Pruning in Action

```
SELECT * FROM silver.events_part
WHERE event_date = '2024-01-15' AND event_type = 'purchase'
```

Without Partition Pruning:

- Scans ALL data files
- Reads then filters
- Slow and expensive

With Partition Pruning:

- Reads ONLY matching folder
- Skips irrelevant partitions
- Fast and efficient

Spark reads only:

`events_part/event_date=2024-01-15/event_type=purchase/`

Choosing Partition Columns

Good Partition Columns:

- Low cardinality (10-1000 unique values)
- Frequently used in WHERE clauses
- Used in joins as keys
- Time-based (date, month, year)

Bad Partition Columns:

- High cardinality (millions of values)
  - Creates millions of tiny folders
- Columns rarely filtered on
- Columns that change frequently

Cardinality Guidelines

Cardinality	Example	Recommendation
Very Low (<10)	status, region	Good for 2nd-level partition
Low (10-100)	country, event_type	Good partition column
Medium (100-1000)	city, product_category	Use with caution
High (1000-10000)	user_segment	Consider ZORDER instead
Very High (>10000)	user_id, timestamp	Never partition, use ZORDER

Best Practice: Order partitions from lowest to highest cardinality

The Small Files Problem

Causes:

- Streaming micro-batches
- Frequent small updates
- Upsert operations
- Individual inserts

Why It's Bad:

- Overhead per file (metadata ops)
- Inefficient reads
- Memory pressure on driver
- Slower query planning