Databricks Optimization: Performance, Cost, and Governance Best Practices

Mandar Baxi Follow 7 min read · Jul 10, 2025

In the evolving landscape of data platforms, performance and cost-efficiency are not optional — they're foundational. As data teams push the limits of scale and complexity, tuning your Databricks Lakehouse environment can unlock significant gains. Optimizing your Databricks environment is both a science and an art. From tuning Delta Lake file layouts to leveraging Photon,

every tweak can drastically improve query performance and reduce cost. In this guide, we walk through best practices across Databricks implementation.

🗄️ Storage Layer Optimization

🔹 Small File Handling

Use OPTIMIZE to compact many small files into larger ones (~100–250 MB) to reduce file listing and read overhead.

Delta Lake tables often accumulate many small files due to streaming or frequent writes. Running OPTIMIZE rewrites these into larger files, improving scan efficiency and query parallelism.

🔹 Z-Ordering

Apply ZORDER BY on up to 4 high-cardinality columns (e.g., customer_id) to improve data skipping and filtering.

Z-Ordering reorders data on disk to colocate similar values. This allows Databricks to scan fewer files when queries apply filters to those columns.

🔹 Auto-Optimize

Enable the configs delta.autoOptimize.optimizeWrite=true and delta.autoOptimize.autoCompact=true to manage file sizes during write operations.

These settings ensure optimal file sizes (~128 MB) and auto-compaction, minimizing the need for manual OPTIMIZE commands.

- **Delta Lake Features**

Use Delta Lake's ACID transactions, time travel, and data skipping for scalable, consistent, and performant data operations.

Delta Lake overlays transactional consistency and intelligent query optimization (like min/max statistics for skipping files) on top of Parquet, delivering a scalable and performant data layer.

- **Liquid Clustering**

Use Liquid Clustering to dynamically organize large datasets without rigid partitioning.

- **Predictive Optimization**

Unity Catalog-managed tables receive recommended OPTIMIZE and VACUUM operations based on query patterns and data characteristics. These optimizations are not executed automatically — Databricks provides actionable suggestions via UI alerts in the Databricks workspace

Databricks surfaces insights from telemetry like query frequency and file size distributions to suggest compaction or cleanup. This balances performance gains with governance control.

⚙️ Compute Layer Optimizations

🔹 Photon Engine

Use Photon for SQL/DataFrame workloads to get 2–4× performance boost and reduce TCO.

Photon is Databricks' native vectorized engine. Benchmarks show 2× speedup on TPC-DS; customers report 3–8× performance boosts in real workloads.

🔹 Cluster Sizing

Enable autoscaling with a min of 2–4 nodes; set auto-termination for idle clusters (15–30 min).

Autoscaling dynamically adjusts capacity to fit workload demands. Auto-termination prevents runaway cost from idle compute.

🔹 Spot Instances

Use spot VMs for fault-tolerant jobs.

Spot instances can slash costs by 60–70%. With fallback support, they're ideal for non-critical or retryable pipelines.

🔹 Photon Selectivity

Avoid using Photon for small (<10 GB) datasets or jobs dominated by Python UDFs.

Photon benefits diminish for lightweight or UDF-heavy workloads. Use it strategically.

- **Spot Hybrid Strategy**

Use spot instances with on-demand fallback in production pipelines.

Fallback ensures resilience while leveraging cost savings — Databricks gracefully handles spot VM preemption.

- **Serverless SQL Warehouses**

Leverage with Photon for BI workloads with bursty usage patterns.

Serverless warehouses scale to zero when idle, and auto-resume instantly. Ideal for dashboards with spiky or unpredictable usage.
When they don't help: *For heavy continuous workloads, serverless may be costlier than a well-tuned classic warehouse due to higher per-second DBU rates. Serverless also incurs higher list price per DBU, but it auto-scales so aggressively that total spend can still be lower.*

- **Serverless Job Clusters**

Use for short-lived jobs like nightly ETLs, scheduled reports, or GenAI prompts.

Serverless job compute eliminates cold starts and charges only for active runtime, bringing saving workflows like nightly ETLs, GenAI jobs, or micro-batch pipelines.

When they don't help: *You have long-running, resource-heavy pipelines better served by tuned classic or spot clusters.*

🧩 Partitioning & Clustering Strategies

🔹 Partitioning

Physically partition using low-cardinality columns (e.g., date, region) to enable pruning.

Partitioning organizes files into separate directories based on a column's values. When queries filter on that column, Databricks can skip reading entire partitions — improving performance and reducing I/O. However, it works best when the column has limited unique values

🔹 Z-Ordering

Logically cluster data on high-cardinality columns (customer_id, zipcode) to boost data skipping.

Z-Ordering sorts files using a multi-column layout (via Z-order curves) to colocate rows with similar values, improving performance when filtering or joining on those columns. It works best for large datasets with frequent queries on specific high-cardinality fields.

- **Liquid Clustering**

Use CLUSTER BY instead of rigid partitioning.

Liquid Clustering replaces traditional static partitioning with a flexible CLUSTER BY approach. It incrementally reclusters data in the background during OPTIMIZE, making it ideal for large, growing datasets with evolving schemas or unpredictable query patterns. It avoids the rigidity and overhead of static partitioning.

🔄 MERGE Optimization

- **Temporal Filters**

Apply timestamp conditions in the ON clause (e.g., target.updated_at >= '2024–01–01') during MERGE.

MERGE operations in Delta Lake compare every row in the source and target tables based on the ON condition. If the target table is large (e.g., months or years of data), this can be expensive. When the target table is partitioned by time (e.g., updated_at, ingestion_date, etc.), including a temporal predicate in the ON clause allows Delta Lake to prune partitions and skip reading irrelevant files.

- **Low-Shuffle Merge**

Enable spark.databricks.delta.merge.optimizeMatchedOnly = true.

This optimization minimizes shuffles by pre-sorting and partitioning data, which improves performance and scalability for large merge jobs. Low-shuffle merge

works by: Coalescing rows within the same partition ; Minimizing movement of data across nodes ; Using bucketed sort merge logic, similar to hash partitioning

🔹 **Photon-Accelerated MERGE**

Run on Photon clusters for 2–4× improvement.

Photon leverages SIMD execution and vectorized logic for faster MERGE INTO operations involving large datasets or complex joins.

⚡ **Query Acceleration & Caching Strategies**

🔹 **Delta Cache**

Use SSD-backed worker nodes (e.g., i3, Ls-series) to enable Delta Cache.

Delta Cache stores data in local SSD memory on worker nodes. It speeds up repeated reads by avoiding remote storage I/O, improving performance by 2–3× for many analytical queries.

🔹 **Materialized Views**

Precompute and store aggregated query results to accelerate BI dashboards and reduce compute

Materialized views in Databricks store the results of pre-defined transformations or aggregations. They're automatically refreshed and help avoid re-running complex joins/aggregates on raw data — especially useful for BI dashboards and derived metrics.

🔹 **Predicate Pushdown**

Apply .filter() or WHERE clauses early in queries to reduce data scanned and optimize execution plans.

Predicate pushdown pushes filtering logic as close to the data source as possible (e.g., before joins or aggregations), allowing Spark or Delta to skip reading irrelevant data. This reduces I/O, memory usage, and overall compute cost.

🔹 **Metric Views (Unity Catalog)**

Define reusable, governed business metrics to reduce query duplication and optimize BI/dashboard workloads.

Metric Views allow you to centrally define KPIs (like revenue, churn, LTV) in Unity Catalog using YAML or SQL. These metrics are automatically optimized, support query-time slicing by dimensions, and integrate seamlessly with BI tools and SQL endpoints. By eliminating repetitive aggregations and logic in dashboards, Metric Views cut compute cost, improve query performance, and ensure semantic consistency across users.

🔐 **Governance & FinOps Enablement**

🔹 **Unity Catalog Tags**

Use metadata tags (e.g., cost_center, department) for chargeback/showback models.

Unity Catalog supports key-value tagging on tables, views, and catalogs. These tags can represent business metadata like cost center or project ID. When combined

with audit logs and query history, tags enable fine-grained chargeback accounting by mapping resource consumption back to teams or functions — critical in FinOps or multi-tenant data platforms.

🔹 **System Tables**

Track DBU usage — Query system tables in Unity Catalog to analyze DBU consumption by user, table, cluster, or query over time.

Databricks System Tables provide structured telemetry data across your platform. You can analyze these tables using SQL to identify high-cost queries, inefficient workloads, or underutilized clusters. Ideal for building custom FinOps dashboards, trend reports, or automated optimization triggers.

🔹 **Cluster Policies**

Set guardrails on VM types, autoscaling, and termination. Enforce instance types/auto-termination — Use Cluster Policies to enforce standard instance types, node limits, and auto-termination rules across teams.

Prevents overspending and enforces org-wide compute standards — ensuring efficient, compliant usage across teams.

💡 **Final Thoughts**

Databricks optimization isn't one-size-fits-all. But these tactics, when applied intentionally, drive measurable gains — faster queries, lower cloud bills, and happier data teams. As workloads evolve and governance demands rise, these best practices become essential levers for platform stability, agility,

and cost transparency. It's not about squeezing the last millisecond — it's about making Databricks resilient, responsive, and ready for what's next.

References:

1. Databricks. "Best practices for performance efficiency." *docs.databricks.com*, 2025.
2. Databricks. "Comprehensive Guide to Optimize Databricks, Spark and Delta Lake Workloads." *databricks.com*, 2025.
3. Databricks. "Cost optimization for the data lakehouse." *docs.databricks.com*, 2025.
4. Microsoft. "Best practices for cost optimization — Azure Databricks." *learn.microsoft.com*, 2025.
5. Microsoft. "Best practices for performance efficiency — Azure Databricks." *learn.microsoft.com*, 2025.

Databricks | Optimization | Performance | Cost

Written by Mandar Baxi

1 follower · 1 following

Chief Practice Partner at Persistent Systems

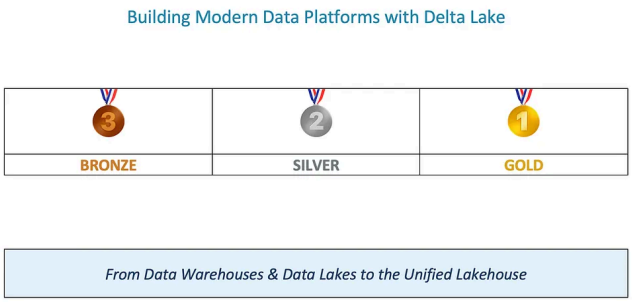
Follow

No responses yet

Write a response

What are your thoughts?

Recommended from Medium

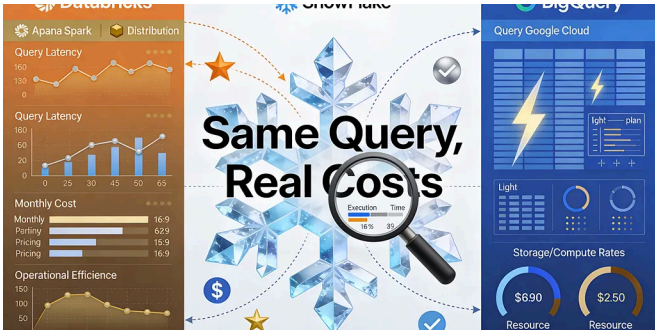


In EndToEndData by Prem Vishnoi(cloudvala)

The Complete Guide To Lakehouse Architecture

1.2 Data Lakes: Scale Without Trust

Dec 30, 2025 75



Reliable Data Engineering

Comparing Databricks, Snowflake, and BigQuery: Same Query, Real...

A side-by-side analysis of query performance, pricing, and operational differences across...

5d ago 30

Cloud With Azeem

LinkedIn Is Replacing Kafka—Here's Why the Streaming Giant is...

Inside LinkedIn's Bold Move to a New Data Pipeline That Could Change the Future of...

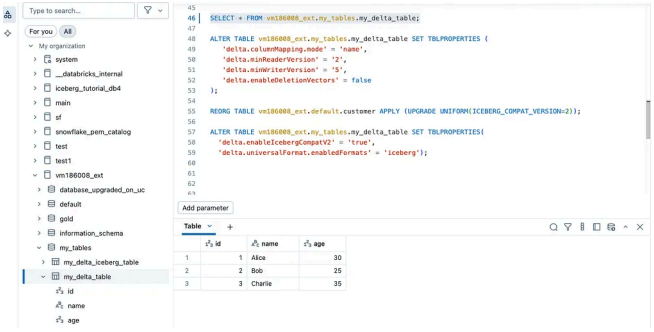
6d ago 55

In AWS in Plain English by Taffarel de Lima Oliveira

How to build a Data Lakehouse using Kafka, Flink, Iceberg, and...

Utilizing a data warehouse provided enhanced performance and user-friendlines...

Aug 18, 2025 1

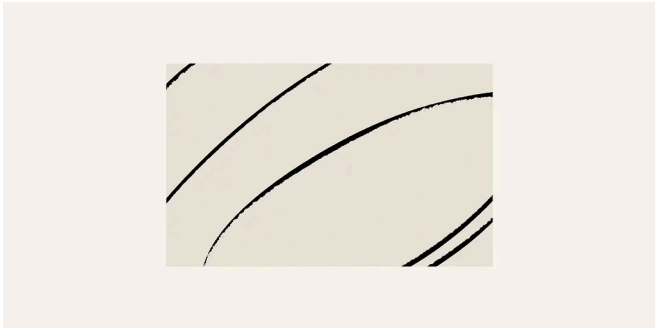


Vik Malhotra

Unlocking Data Interoperability: Reading Unity Catalog Tables in...

By: Vik Malhotra & Jaideep Patel

Sep 7, 2025



Umesh Pawar

Delta Lake Change Data Feed (CDF) Reduces ETL Costs and...

GitHub repository: umeshpawar2188/Delta-change-data-feed

Jul 25, 2025

See more recommendations