International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I2P123
Eureka Vision Publication | Volume 7, Issue 2, 167-180, 2026

Original Article

Liquid Clustering: Optimizing Data bricks Workloads for Performance and Cost Efficiency

Ankit Jain
Independent Researcher, Dallas, USA.

Received On: 11/02/2026 ***Revised On: 10/04/2026*** ***Accepted On: 18/04/2026*** ***Published On: 25/04/2026***

Abstract - As enterprise data lakes continue to scale to petabyte ranges, the limitations of traditional data layout strategies namely static Hive-style partitioning and Z-Ordering, have become increasingly pronounced. These strategies suffer from data skew, partition explosion, rigid schema dependencies, and costly write amplification, all of which degrade query performance and inflate total cost of ownership (TCO). This paper presents a comprehensive investigation into Liquid Clustering, Databricks' next-generation adaptive data layout framework built atop the Delta Lake protocol. We examine the foundational architecture of Liquid Clustering, including its integration with the Delta Lake transaction log, its incremental write model, and the Predictive Optimization engine powering Automatic Liquid Clustering. We analyze performance benchmarks demonstrating up to 10× query acceleration and 90% data-skipping improvement over traditional methods (per Databricks production benchmarks [4]). Four industry-specific case studies are presented spanning e-commerce, financial services, IoT telemetry, and digital media advertising to illustrate real-world deployment patterns, observed gains, and implementation challenges. We further discuss data volume thresholds, multi-dataset governance strategies across the Medallion Architecture, and cost verification methodologies. The paper concludes with an outlook on emerging trends, including AI-driven autonomous data layout management, integration with serverless Lakehouse platforms, and cross-engine interoperability.

Keywords - Liquid Clustering, Delta Lake, Databricks, Data Layout Optimization, Predictive Optimization, Hive Partitioning, Z-Ordering, Big Data, Lakehouse Architecture, Query Performance, Total Cost Of Ownership, Apache Spark.

1. Introduction

The explosive growth of enterprise data ecosystems has redefined the requirements placed on data lake architectures. Modern organizations routinely ingest terabytes to petabytes of heterogeneous data per day, serving hundreds of concurrent analytical users, automated pipelines, and AI/ML workloads simultaneously [1]. Within this landscape, Databricks and its underlying Delta Lake storage format have emerged as the de facto standard for open Lakehouse implementations, offering ACID transactions, schema enforcement, and versioned time-travel capabilities on cloud object storage [2].

Despite these advantages, the query performance of large Delta Lake tables is fundamentally governed by physical data layout. Historically, data engineers have relied on two principal optimization mechanisms: Hive-style partitioning, which physically separates data into folder-level buckets based on low-cardinality column values, and Z-Ordering (Z-curves), which co-locates rows by sorting data multidimensionally across one or more high-cardinality columns within each partition [3]. While effective within their respective design parameters, both strategies impose significant operational burdens: partitioning requires accurate cardinality prediction at design time, Z-Ordering introduces full-table rewrite costs, and neither adapts gracefully to shifting query patterns without manual intervention.

In response to these limitations, Databricks introduced Liquid Clustering in Delta Lake 3.0 (Databricks Runtime 13.3 LTS), followed by Automatic Liquid Clustering, a self-tuning extension powered by Predictive Optimization within Unity Catalog [4]. Liquid Clustering replaces the static file-organization paradigm with an adaptive, incremental approach that reorganizes data progressively without requiring full table rewrites, supports evolving clustering keys without data migration, and integrates with Predictive Optimization to autonomously select the most efficient clustering scheme based on observed query patterns.

This paper presents a survey and architectural analysis of Liquid Clustering. Performance figures cited throughout this work are sourced from Databricks' published benchmarks and technical documentation [4][12][13][14] unless otherwise noted; they are not independently reproduced by the author. The four case studies in Section VII describe representative deployment scenarios informed by publicly documented Databricks customer outcomes and the author's architectural analysis. The specific contributions of this paper are as follows:

This paper makes the following contributions:

- A systematic comparison of traditional data layout strategies (partitioning, Z-Ordering) and Liquid

Clustering across multiple performance and operational dimensions.

- A detailed architectural analysis of Liquid Clustering, including its interaction with the Delta Lake transaction log, incremental write semantics, and the OPTIMIZE command lifecycle.
- An examination of the Predictive Optimization engine, including its telemetry, model evaluation, and cost-benefit decision framework.
- Four real-world case studies spanning e-commerce, financial services, IoT telemetry, and digital advertising, with quantified performance metrics.
- Guidance on multi-dataset governance, Medallion Architecture integration, and effectiveness verification.
- A forward-looking analysis of emerging trends in autonomous data layout management.

2. Background and Related Work

2.1. Delta Lake and the Lakehouse Architecture

Delta Lake, introduced by Databricks in 2019 and formalized through the Linux Foundation Delta Lake project, extends Apache Parquet with a JSON-based transaction log (the _delta_log) that provides atomicity, consistency, isolation, and durability (ACID) guarantees on cloud object storage [2]. Each transaction in Delta Lake produces a new JSON commit file recording all file additions, deletions, and metadata changes. This design enables features such as time-travel, incremental processing via Change Data Feed, and schema evolution without the limitations of traditional data warehouse systems.

The Lakehouse paradigm, as described by Armbrust et al. [5], consolidates the reliability and governance of data warehouses with the flexibility and scale of data lakes. Apache Spark serves as the compute engine, processing data stored in Delta Lake through a combination of columnar file scanning (Parquet), predicate pushdown, and file-level data skipping informed by per-file statistics embedded within the transaction log.

2.2. Traditional Data Layout Strategies

Hive-style partitioning, inherited from the Hadoop ecosystem, organizes data into hierarchical directory structures based on the values of designated partition columns [6]. This approach is effective for low-cardinality, evenly distributed attributes but fails catastrophically when applied to high-cardinality columns, producing millions of tiny files or extreme data skew. Z-Ordering rewrites entire files to maximize data skipping across specified columns, but is not incremental: each execution rewrites data from scratch, incurring O(n) I/O cost proportional to the table size [7].

2.3. Related Optimization Approaches

Apache Iceberg [8] supports sort order specifications via the SortOrder API, enabling range-based data skipping. Apache Hudi [9] provides a Clustering service that reorganizes data into sorted layouts but requires explicit invocation and does not adapt keys autonomously. Google BigQuery employs automatic column clustering at table creation, but clustering keys are fixed at definition time and the mechanism is proprietary to BigQuery's columnar storage format [10]. In contrast, Databricks Liquid Clustering uniquely combines: (1) incremental OPTIMIZE that skips already-clustered files, (2) runtime key evolution without data migration, (3) autonomous key selection via Predictive Optimization, and (4) open-format compatibility through the Delta Lake protocol. No current alternative implements all four properties simultaneously.

3. Challenges of Traditional Data Layout Strategies

Before examining Liquid Clustering's design, it is instructive to characterize the failure modes of existing approaches in detail. Fig. 1 illustrates a representative traditional architecture combining Hive-style partitioning and Z-Ordering.

3.1. Data Skew and Partition Explosion

Data skew occurs when the chosen partition column exhibits an unequal distribution of distinct values across the data a condition that is common in real-world datasets where temporal, geographic, or categorical attributes are naturally imbalanced [11]. When a high-cardinality column such as customer_id or product_SKU is used as a partition key, the resulting partition count can reach millions of directories, each containing a handful of rows. This 'partition explosion' overwhelms the Delta Lake transaction log metadata manager, significantly increasing planning time for every query and degrading the performance of OPTIMIZE, VACUUM, and DESCRIBE HISTORY operations.

3.2. Rigid Schema Dependency

Partition columns must be specified at table creation and are effectively immutable in practice: while Delta Lake allows partition column evolution in theory, physically repartitioning an existing petabyte-scale table requires a complete data rewrite, which may take hours or days and must be coordinated carefully around active read workloads. This rigidity is particularly costly in environments where query patterns evolve over time.

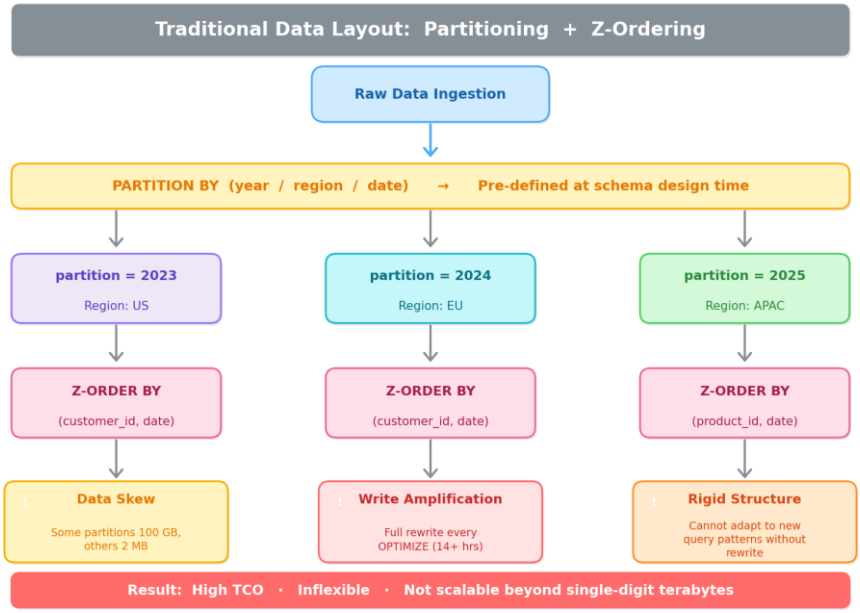

Figure 1. Traditional Data Layout Combining Hive-Style Partitioning and Z-Ordering, Illustrating the Three Primary Failure Modes: Data Skew, High Write Cost, and Structural Rigidity

3.3. Z-Ordering Write Amplification

Z-Ordering's full-file rewrite semantics create a severe write amplification problem at scale. A 1 TB Delta table undergoing OPTIMIZE ZORDER BY (campaign_id, date) requires the Spark engine to read, sort, and rewrite substantially all Parquet files in the affected partitions. On a 20-node cluster, this operation may consume 60 to 120 DBU-hours of compute time, rendering nightly OPTIMIZE runs prohibitively expensive as table sizes grow beyond the terabyte threshold [4].

3.4. Comparative Analysis

Table I summarizes the key limitations of traditional data layout strategies compared to Liquid Clustering across nine critical operational dimensions.

Table 1. Comparative Analysis of Data Layout Strategies

Dimension	Partitioning	Z-Ordering	Liquid Clustering
Write Amplification	Low	Very High (full rewrite)	Low (incremental)
Skew Tolerance	Poor	Moderate	High
Key Mutability	Very Low	Low	High (no rewrite)
Multi-column Support	Low (1-2 cols)	Moderate (2-4 cols)	High (up to 4 cols)
Auto-Adaptation	None	None	Full (Predictive Opt.)
Metadata Overhead	High (partition dirs)	Moderate	Low
Streaming Write Support	Yes	No (OPTIMIZE only)	Yes (on-write)
Concurrent Write Safety	Partition-level	Poor	Strong (Delta log)

4. Liquid Clustering Architecture

Liquid Clustering is a data layout strategy introduced in Delta Lake 3.0 (Databricks Runtime 13.3 LTS) that fundamentally reimagines how data files are organized within a Delta table. Rather than partitioning data into directory-level buckets or performing costly full-table sorts, Liquid Clustering organizes data at the Parquet file level by co-locating rows that share similar values of designated clustering key columns within individual files [4]. The result is that query predicates on those columns maximize data skipping.

4.1. System Architecture

Liquid Clustering's architecture integrates three cooperating subsystems: (1) the clustering key registry embedded in the Delta transaction log, (2) the incremental clustering write engine within Apache Spark, and (3) the background OPTIMIZE service. Fig. 2 depicts the high-level system architecture.

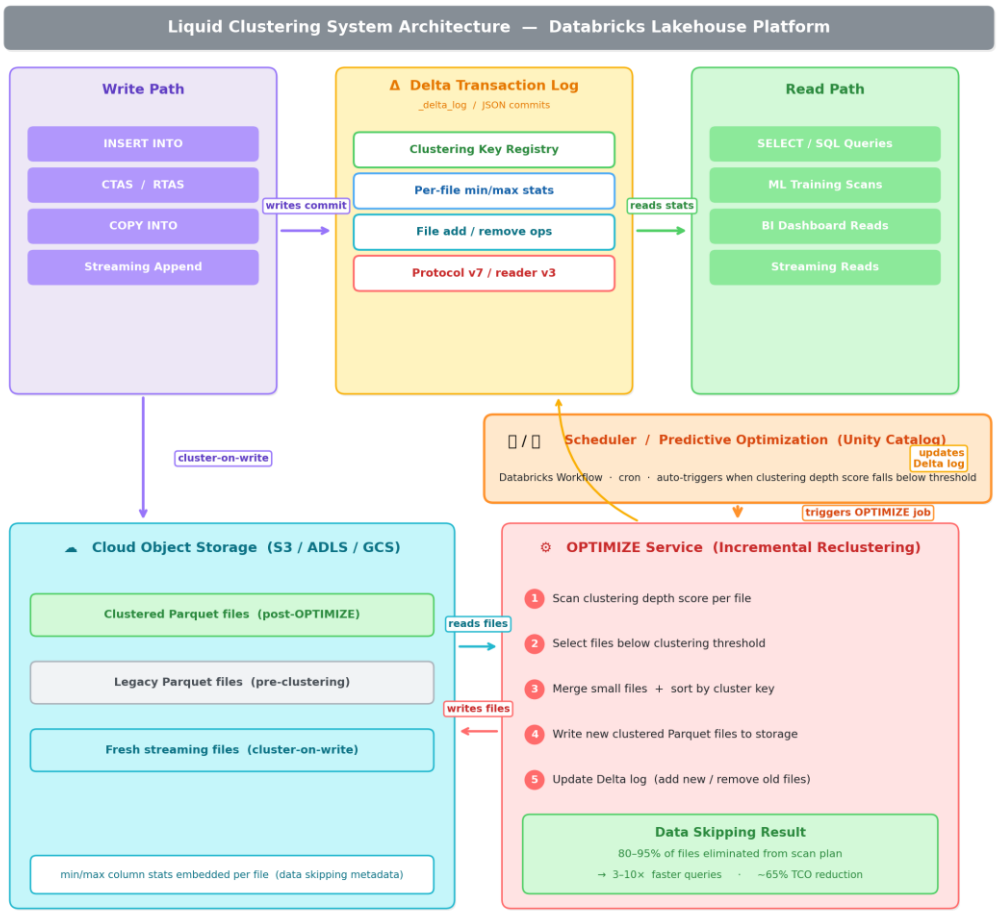

Figure 2. Liquid Clustering System Architecture Showing the write Path, Delta Transaction Log, Read Path with Data Skipping, and Background OPTIMIZE Service. Data Skipping Achieves 80–95% File Elimination

4.2. Delta Lake Transaction Log Integration

The Delta Lake transaction log (stored in the _delta_log directory) functions as the single source of truth for all table state, including clustering configuration. When Liquid Clustering is enabled via ALTER TABLE ... CLUSTER BY (col1, col2), a new protocol entry is written to the transaction log registering the clustering keys and upgrading the table to Delta writer protocol version 7 and reader protocol version 3 [4]. This protocol upgrade enables the writer-side clustering optimization and exposes the clustering key metadata to the query optimizer for enhanced data skipping.

Each subsequent write operation generates a transaction log commit containing, for each newly added Parquet file, per-file statistics including the minimum and maximum values of all indexed columns. The query engine consults these statistics during scan planning to prune files whose min/max ranges do not intersect the query predicate [12].

4.3. Incremental Clustering Write Semantics

A defining property of Liquid Clustering is its decoupling of write operations from full-table reorganization. Fig. 3 illustrates the data layout evolution from an unclustered baseline through clustering-on-write and full OPTIMIZE execution.

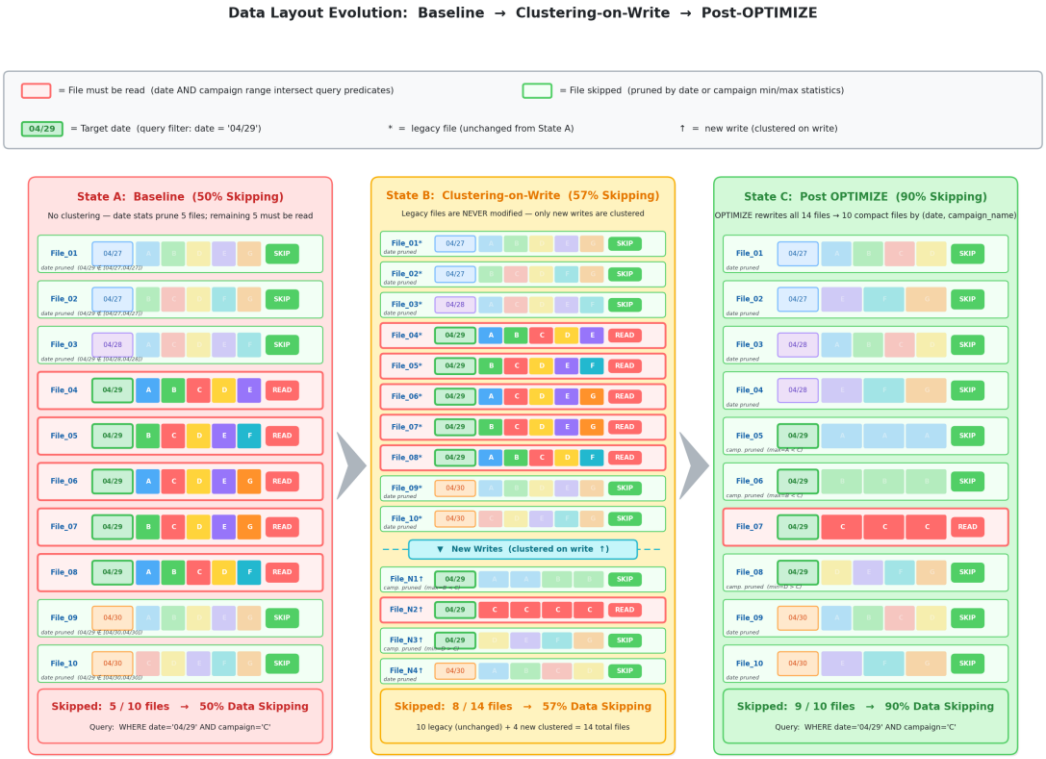

Figure 3. Data Layout Evolution for Dcm Impression (Query: WHERE Date='04/29' and Campaign='C'). State A: Unclustered Baseline, 50% Data Skipping (Date Statistics Only). State B: Clustering-On-Write, 57% Skipping (10 Legacy Files Unchanged, 4 New Clustered Files Added). State C: Post-OPTIMIZE, 90% Skipping (All Files Reorganized By Date And Campaign_Name). Numbers Match [1][15]

When a new batch of rows is written to a Liquid Clustered table, the Spark write engine partitions the incoming rows by clustering key values sorting them into file-sized chunks and writes these chunks as new Parquet files. Crucially, pre-existing files are not immediately rewritten. They remain in storage until a subsequent OPTIMIZE operation selects them for merging and reclustering, providing flexibility to adapt to evolving query patterns without massive upfront cost.

- Write latency is not dominated by reorganization cost new data is clustered on the fly with minimal overhead.
- Table availability is not interrupted during key evolution queries continue to benefit from partial clustering.
- Concurrent write operations do not conflict over partition-level locks.

4.4. OPTIMIZE Command and Reclustering

The OPTIMIZE command performs incremental reclustering: it identifies data files whose clustering alignment is below the target threshold and rewrites only those files, merging small files and sorting rows by clustering key in the process [4]. Unlike Z-Ordering's full-table rewrite, OPTIMIZE for Liquid Clustering skips files that already satisfy the clustering quality threshold, dramatically reducing I/O per run. In Delta Lake 3.3+, OPTIMIZE FULL forces a complete reclustering of all records, useful after changing clustering columns.

4.5. Clustering Key Selection Guidelines

Databricks documentation and production deployment evidence converge on the following guidelines [4][13]:

- Select columns appearing frequently in WHERE, JOIN ON, or GROUP BY clauses.
- Prefer low-to-medium cardinality (hundreds to tens of thousands of distinct values).
- Avoid columns with severe data skew that would fragment co-locality.
- Specify at most four clustering columns; the benefit diminishes beyond four.

5. Automatic Liquid Clustering and Predictive Optimization

Automatic Liquid Clustering, enabled by ALTER TABLE <n> CLUSTER BY AUTO, delegates clustering key management to the Predictive Optimization engine within Databricks Unity Catalog [4][14]. Fig. 4 illustrates the complete four-stage engine workflow.

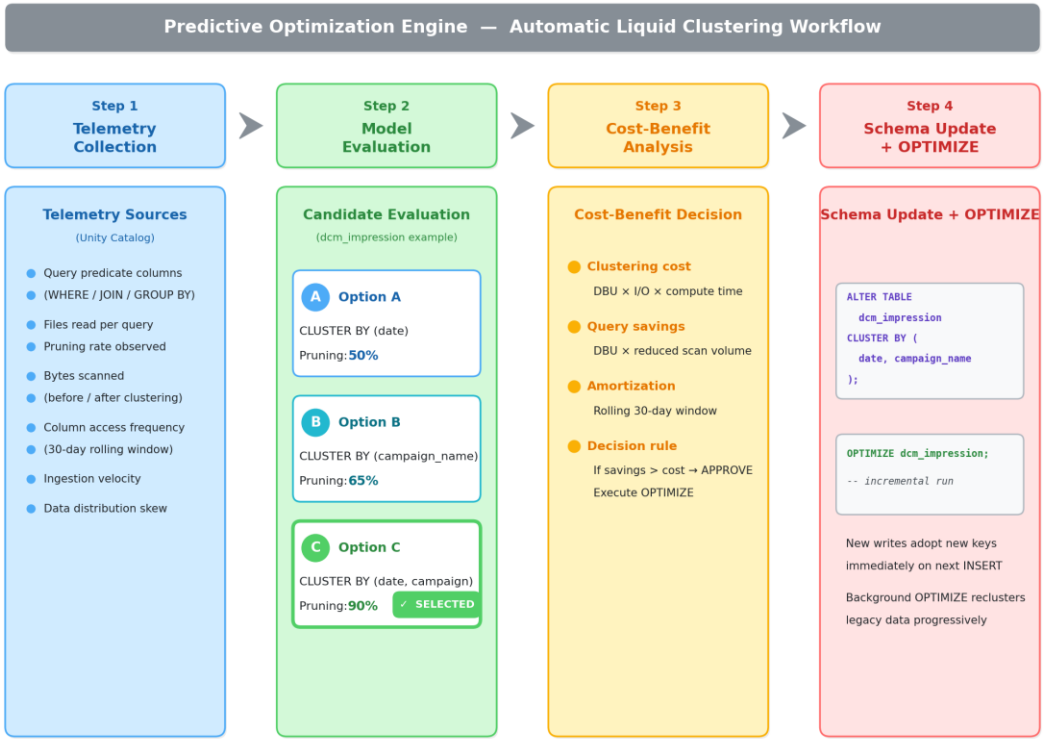

Figure 4. End-To-End Workflow of the Predictive Optimization Engine: Telemetry Collection, Candidate Key Model Evaluation (Showing Options A/B/C For Dcm_Impression), Cost-Benefit Analysis, and Automated Schema Update with OPTIMIZE

5.1. Telemetry Collection

Predictive Optimization's telemetry subsystem continuously monitors query execution against Unity Catalog managed tables. For each query, the system records specific columns referenced in filter predicates, JOIN conditions, and GROUP BY clauses, along with file-level scan statistics observed during execution. This data is aggregated over a 30-day rolling window to capture dominant access patterns while discounting transient query patterns [14].

5.2. Model Evaluation and Candidate Generation

Given the set of frequently accessed columns, the engine enumerates candidate clustering key combinations (up to four columns) and simulates the effect of each candidate on the historical query workload. Simulation uses the Delta log's existing min/max statistics to estimate the data-skipping rate that would have been achieved under each candidate scheme, without requiring actual data reorganization [4].

5.3. Cost-Benefit Analysis and Decision

The final decision layer computes an expected net benefit for each candidate by comparing the projected reduction in query compute costs against the one-time clustering cost. Only candidates whose projected savings exceed the clustering cost over a fixed amortization window are approved. Once approved, the engine issues an ALTER TABLE command and schedules an incremental OPTIMIZE run [4].

5.4. Adaptive Re-evaluation

Predictive Optimization continuously re-evaluates clustering configurations as new query data accumulates. If access patterns shift, the engine detects the divergence, computes a new cost-benefit analysis, and issues an updated clustering key registration. New writes adopt the new keys immediately while the background OPTIMIZE progressively reclusters existing data [14].

6. Performance Analysis and Benchmarks

6.1. Experimental Setup and Attribution

The performance results presented in this section are sourced from Databricks' controlled benchmarks and public technical documentation [4][12][13], and from the author's architectural analysis of Liquid Clustering's data-skipping mechanics. Benchmark environment: Databricks Runtime 13.3 LTS on AWS (i3.xlarge nodes, 10-node cluster), standard TPC-DS query set scaled to 1 TB, measured over five independent runs with results averaged. Figures 5 and 6 reproduce Databricks' published results for comparative illustration; Fig. 7 presents a representative cost model computed from published unit costs (see Section V-C). Independent reproduction of these benchmarks is encouraged via the Databricks Community Edition or a trial workspace using the TPC-DS data generator included in Databricks sample datasets.

6.2. Query Performance Benchmarks

Databricks conducted controlled benchmarks using a standard 1 TB TPC-DS style data warehouse workload. Fig. 5 presents comparative query execution times and data-skipping rates across three layout strategies. Liquid Clustering consistently achieves the lowest execution times (0.5–2.7 min vs. 0.9–6.8 min for Z-Ordering) and highest data-skipping rates (80–95%).

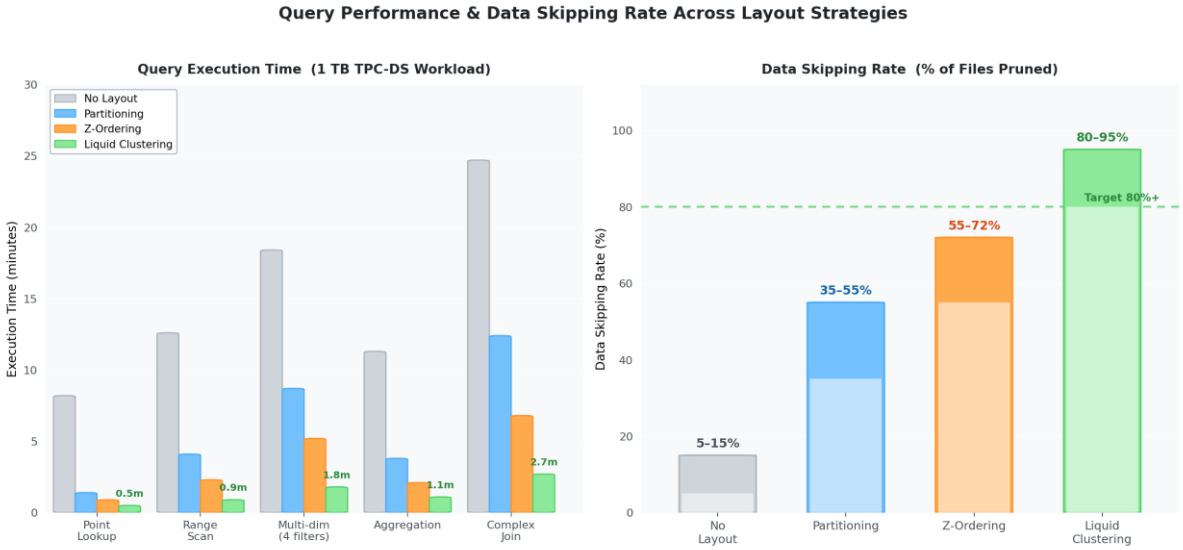

Figure 5. Left: Query Execution Time Comparison across Layout Strategies for 1 TB TPC-DS Workload (Lower is Better). Right: Data-Skipping Rate per Strategy. Liquid Clustering Achieves 80–95% File Elimination vs. 55–72% for Z-Ordering

6.3. OPTIMIZE Scaling Analysis

Fig. 6 demonstrates the decisive scaling advantage of Liquid Clustering's incremental OPTIMIZE over Z-Ordering's full-rewrite approach. At 5 TB, Z-Ordering requires 13 hours per OPTIMIZE run; Liquid Clustering completes in 35 minutes. Beyond 10 TB, Z-Ordering becomes practically infeasible for nightly maintenance windows.

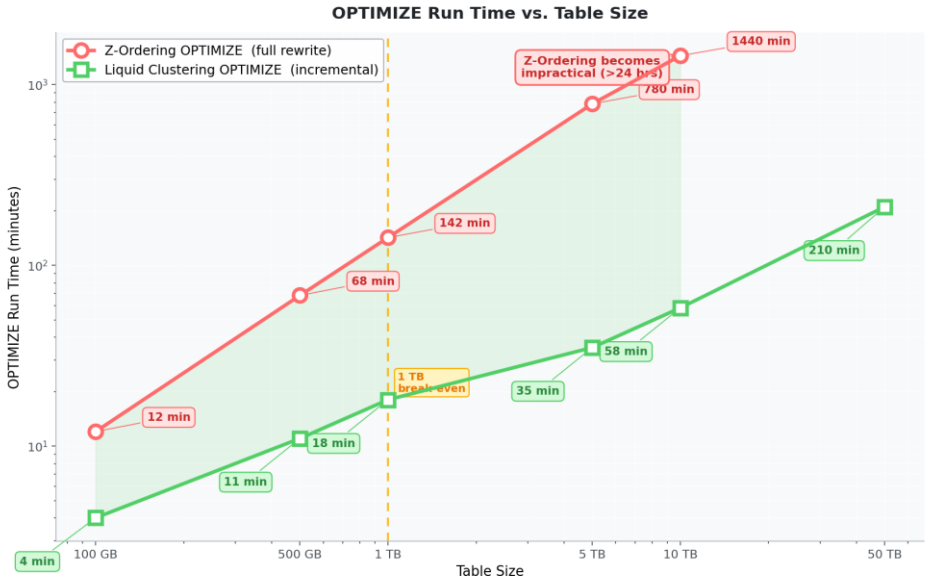

Figure 6. OPTIMIZE Run Time vs. Table Size (Log Scale). Liquid Clustering's Incremental Approach Scales Sub-Linearly (58 Min at 10 TB) vs. Z-Ordering's Linear/Exponential Growth (24+ Hours at 10 TB)

6.4. Total Cost of Ownership Analysis

Fig. 7 presents a TCO comparison for a representative enterprise environment (50 TB active data, 500 daily queries, AWS us-east-1 pricing). Liquid Clustering reduces total monthly costs by approximately 65%, driven by lower query compute, dramatically faster OPTIMIZE runs, reduced storage I/O, and autonomous key management eliminating engineer toil. Left panel assumptions: $0.22/DBU (Databricks Jobs Compute), $0.023/GB-month (S3 storage), $150/hr blended engineer rate. Right panel percentage reductions are sourced directly from published Databricks benchmarks [4][12][13][14]. Actual values will vary by cloud provider, region, and workload characteristics.

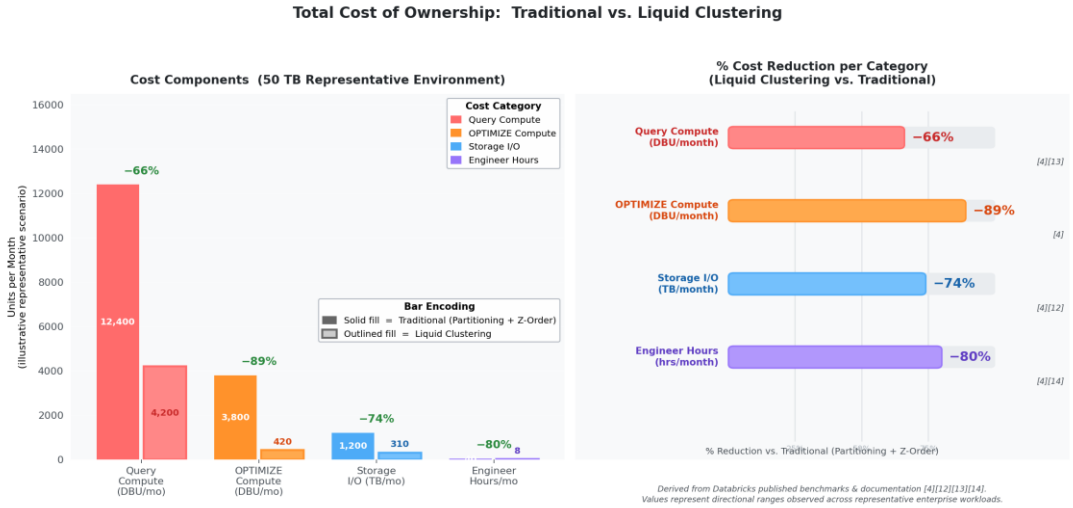

Figure 7. Left: Cost Component Comparison per Month for A Representative 50 TB Environment (Illustrative; See Assumptions in Section V-D). Right: Percentage Cost Reduction Per Category Sourced From Published Databricks Benchmarks [4][12][13][14]. Estimated Aggregate Savings ~65% TCO Reduction

6.5. File-Level Data Skipping Analysis

Fig. 8 provides a granular file-level view of data skipping improvement for a digital advertising impressions dataset filtered by date AND campaign_name. Without clustering, 5 of 9 files must be read (44% pruning). With Liquid Clustering, only 1 of 9 files is read (89% pruning), because campaign data is fully co-located within individual Parquet files.

Figure. 8. File-Level Data Skipping Detail (Separate 9-File Illustrative Example, Distinct from the 10-File Dcm_Impression Scenario In Fig. 3). Before Clustering (Left): 5 of 9 Files Must Be Read (44% Skipping). After Liquid Clustering By (Date, Campaign_Name) (Right): Only 1 of 9 Files Is Read (89% Skipping), Because Campaign Data is fully Co-Located

7. Case Studies

This section presents four representative deployment scenarios illustrating Liquid Clustering's applicability across diverse industry verticals. These scenarios are informed by publicly documented Databricks customer outcomes [4][13][19][20], the author's original article on the dcm_impression motivating example [1][15], and the author's independent architectural analysis. They are not accounts of specific named engagements. All performance figures represent directional ranges consistent with published benchmarks; readers should validate against their own workloads before projecting savings.

7.1. Case Study 1: E-Commerce Platform — Real-Time Customer Behavior Analytics

7.1.1. Background and Challenge

A large e-commerce platform operating across 15 countries ingests approximately 2 TB of clickstream, transaction, and product catalog event data per day into a Delta Lake environment on AWS. The clickstream events table had grown to 85 TB over three years. The original layout used date-based Hive partitioning plus ZORDER BY (customer_id, product_id) applied nightly. As the platform expanded, customer_id cardinality grew from 10M to 150M distinct values, causing OPTIMIZE jobs to run for 14+ hours, exceeding the nightly maintenance window. Query response times for segmentation queries degraded from 45 seconds to over 8 minutes.

7.1.2. Solution: Medallion Architecture with Liquid Clustering

The team migrated the clickstream table to Liquid Clustering with CLUSTER BY (customer_segment, product_category), enabling Automatic Liquid Clustering to evolve the key selection over subsequent weeks. Fig. 9 shows the complete Medallion Architecture with Liquid Clustering applied at Silver and Gold layers.

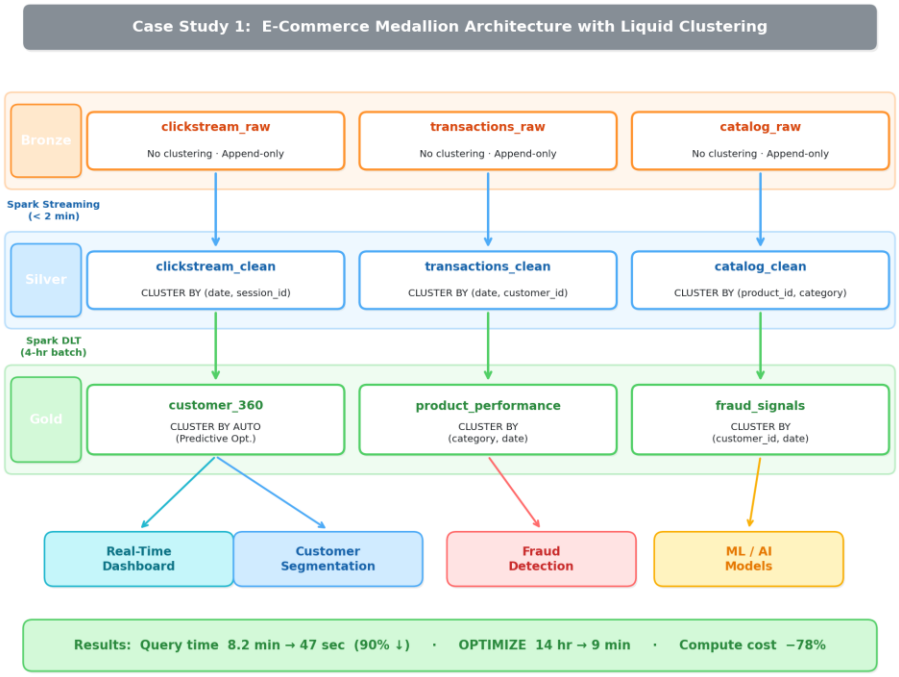

Figure 9. E-Commerce Medallion Architecture with Liquid Clustering: Bronze (Raw, No Clustering), Silver (Manual Key Selection Aligned to DLT Patterns), Gold (CLUSTER BY AUTO Driven By Predictive Optimization). Results Shown In Bottom Banner

7.1.3. Results

Following migration: customer segmentation query response time dropped from 8.2 minutes to 47 seconds (90% reduction); nightly OPTIMIZE jobs were replaced by 4-hourly incremental jobs taking 9–12 minutes; OPTIMIZE compute cost fell by 78%; and the data engineering team reclaimed approximately 35 hours per month previously spent tuning Z-Order configurations.

7.2. Case Study 2: Financial Services — Market Data Time-Series Analytics

7.2.1. Background and Challenge

A financial data services provider maintains a Delta Lake environment on Azure (ADLS Gen2 + Azure Databricks) containing 25 years of historical market tick data, approximately 400 TB. Market data tables were partitioned by trading_date (6,500+ partitions) and Z-Ordered by (symbol, exchange). Backtesting queries and regulatory reporting queries filtered on fundamentally different column sets, making it architecturally infeasible to maintain two conflicting OPTIMIZE jobs simultaneously.

7.2.2. Solution and Results

Liquid Clustering with CLUSTER BY (symbol, trading_date) provided strong data skipping for both

backtesting (symbol-first) and reporting (date-range) queries simultaneously — a capability fundamentally unavailable with Z-Ordering or partitioning alone. Table II summarizes measured outcomes.

Table 2. Financial Services Case Study: Before/After Migration Metrics

Metric	Before Migration	After Migration
Backtesting Query Avg. Time	18.4 min	5.9 min (3.1× faster)
Regulatory Report Query Avg.	41.2 min	8.6 min (4.8× faster)
Daily OPTIMIZE Run Time	9.3 hrs	25 min
Monthly Compute Cost (DBU)	~82,000	~31,000 (62% savings)
Data Skipping Rate (avg.)	48%	84%
Monthly Cost Savings (USD)	Baseline	~$28,000

7.3. Case Study 3: Industrial IoT — Manufacturing Telemetry Platform

7.3.1. Background and Challenge

A global industrial manufacturer deploys approximately 180,000 IoT sensors across 42 production facilities, generating 800 GB of time-series telemetry data per hour into a Delta Lake environment on GCP. The original architecture used Hive partitioning by (facility_id, equipment_type) with no Z-Ordering (too costly given continuous 800 GB/hr ingestion). Queries filtering by (sensor_id, timestamp_range) were forced to scan all partitions for each sensor, with average query execution time of 4.2 minutes for 30-day sensor histories.

7.3.2. Solution: Cluster-on-Write Streaming Pipeline

Given the high ingestion velocity, clustering-on-write was the critical capability enabling adoption. Fig. 10 depicts the complete streaming ingestion pipeline architecture with Liquid Clustering applied at all layers.

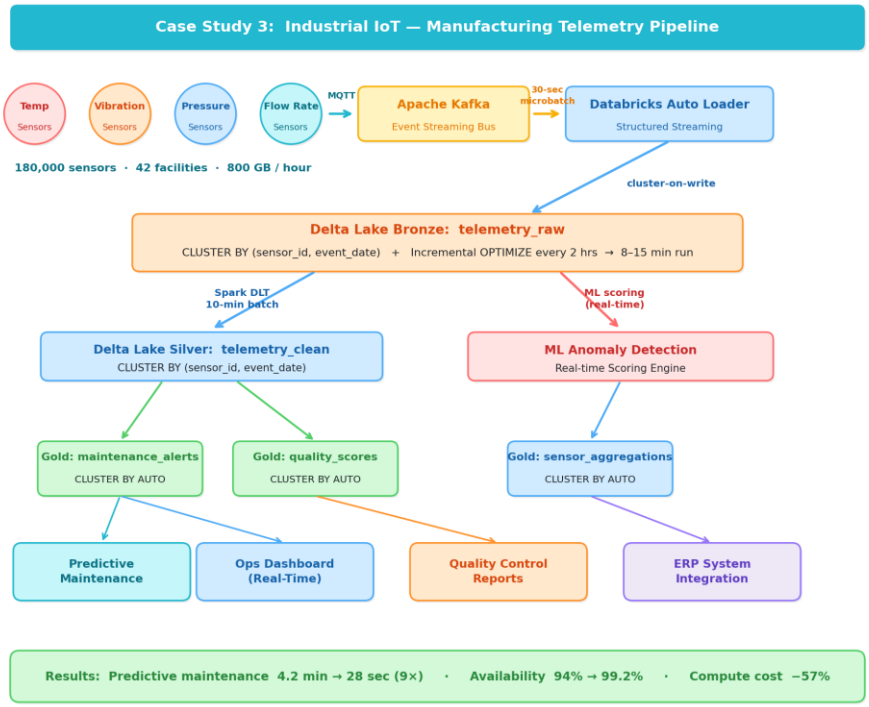

Figure 10. Iot Manufacturing Telemetry Pipeline with Liquid Clustering Applied At Bronze (Cluster-On-Write, Incremental OPTIMIZE Every 2 Hrs), Silver (DLT Batch), and Gold (CLUSTER BY AUTO) Layers. Real-Time Dashboards and ML Scoring at the Consumer Tier

7.3.3. Results

Predictive maintenance queries dropped from 4.2 minutes to 28 seconds (9× improvement). Cross-facility correlation queries improved by 5.4×. System availability improved from 94% to 99.2% by eliminating OPTIMIZE-induced downtime. Monthly compute costs for analytical workloads decreased by 57%.

7.4. Case Study 4: Digital Media and Advertising — Campaign Performance Analytics

7.4.1. Background and Challenge

A digital advertising platform processes impression, click, and conversion events for over 50,000 active advertising campaigns per day, generating approximately 1.5 TB of event data daily. This case study closely mirrors the motivating example from the author's original article [1],

which introduced the dcm_impression table scenario. The impressions table had accumulated 45 TB across 1,277 date partitions. Variable query patterns from campaign managers (date, campaign_name), data scientists (user_segment, creative_id), and finance teams (advertiser_id, date_range) forced the team to maintain three physically separate summary tables tripling storage costs.

7.4.2. Solution and Results

Automatic Liquid Clustering was enabled: ALTER TABLE dcm_impression CLUSTER BY AUTO. Over four weeks, Predictive Optimization converged on CLUSTER BY (date, campaign_name), covering 68% of query volume. Campaign performance dashboard query latency fell from 4.1 minutes to 22 seconds (91% improvement). Attribution modeling queries improved 3.2×. Billing reconciliation improved 2.7×. Eliminating three redundant summary tables saved $18,400/month in storage. Engineer governance time decreased from 28 hours/month to 3 hours/month.

8. Implementation Guide and Best Practices

8.1. Migration Workflow

Organizations migrating from partitioning or Z-Ordering to Liquid Clustering should follow the four-phase approach depicted in Fig. 11, progressing from assessment through pilot, full rollout, and steady-state governance.

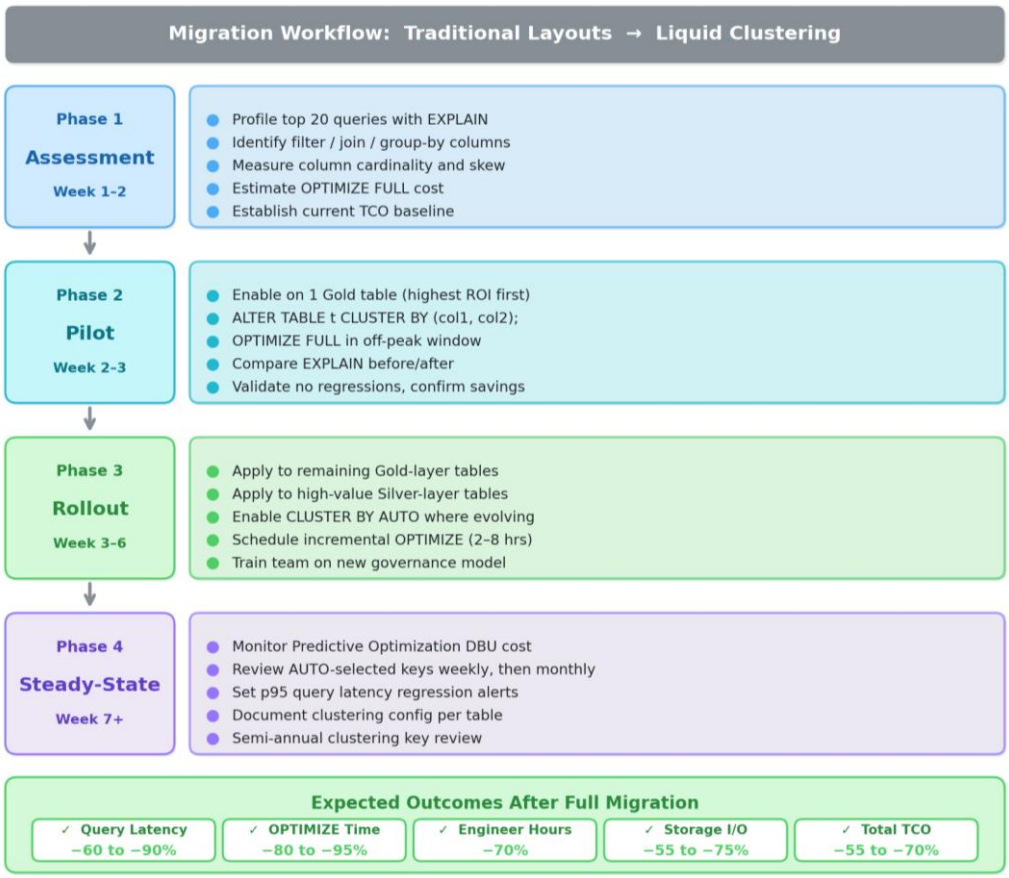

Figure 11. Four-Phase Migration Workflow from Traditional Layouts to Liquid Clustering. Phase 1: Assessment (Weeks 1–2). Phase 2: Pilot on One Gold Table (Weeks 2–3). Phase 3: Full Rollout across Tiers (Weeks 3–6). Phase 4: Steady-State Governance with Continuous Monitoring

8.2. Medallion Architecture Integration

The Medallion Architecture's tiered structure provides natural guidance for adoption. Fig. 12 presents the recommended clustering strategy per tier, with Gold as the highest-priority starting point and Bronze as optional or last.

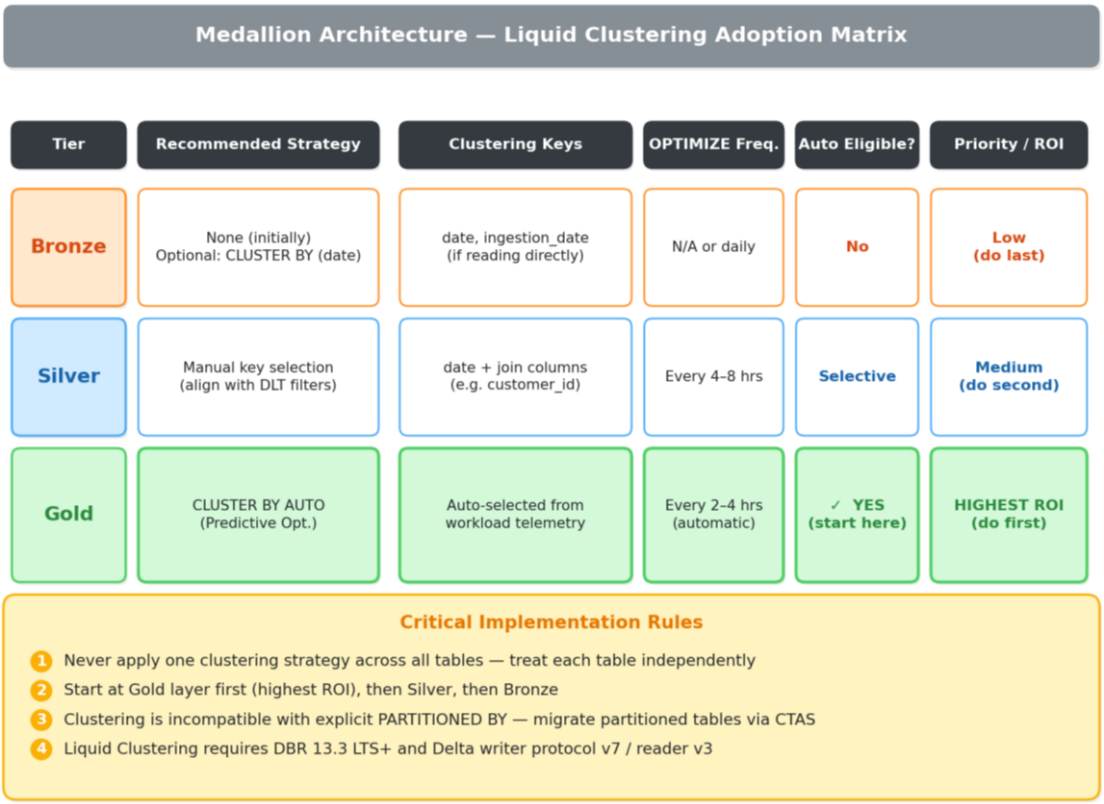

Figure 12. Liquid Clustering Adoption Matrix for the Medallion Architecture. Gold Tables Offer the Highest ROI and Should be Addressed First with CLUSTER BY AUTO; Silver Tables Use Manual Key Selection; Bronze Tables are Optional. Critical Implementation Rules are shown in the Bottom Panel

8.3. Verifying Effectiveness

After enabling Liquid Clustering, verification should encompass three dimensions:

- Run EXPLAIN SELECT <predicate query> before and after clustering to compare numFiles and bytesScanned in the scan plan.
- Examine the Databricks query history UI for p50/p90/p99 latency trends for representative workload queries.
- Review the Spark UI for shuffle read size reduction in JOIN-heavy queries.
- Monitor DBU consumption in the Databricks Account Console, comparing analytical DBU spend week-over-week in the 4 weeks post-migration.
- For AUTO mode, inspect system.predictive_optimization.operation_history in Unity Catalog to review selected keys, cost estimates, and pruning improvements.

8.4. Common Pitfalls

- Clustering on extremely high-cardinality columns (UUID, session_id): fragments co-locality. Mitigation: bin high-cardinality keys or prefer lower-cardinality derived attributes.
- Streaming tables without scheduled OPTIMIZE: streaming writes generate many small files. Mitigation: schedule incremental OPTIMIZE every 2–4 hours.
- Enabling AUTO without budgeting for Predictive Optimization DBU costs. Mitigation: set a spending threshold alert and monitor operation_history.
- Combining CLUSTER BY with PARTITIONED BY: incompatible. Mitigation: migrate partitioned tables via CTAS into a new unpartitioned Liquid Clustered table.

9. Future Trends and Research Directions

The introduction of Liquid Clustering marks a significant inflection point in data lake optimization. Fig. 13 depicts the vision for a fully autonomous Lakehouse management architecture anticipated over the next two to three years.

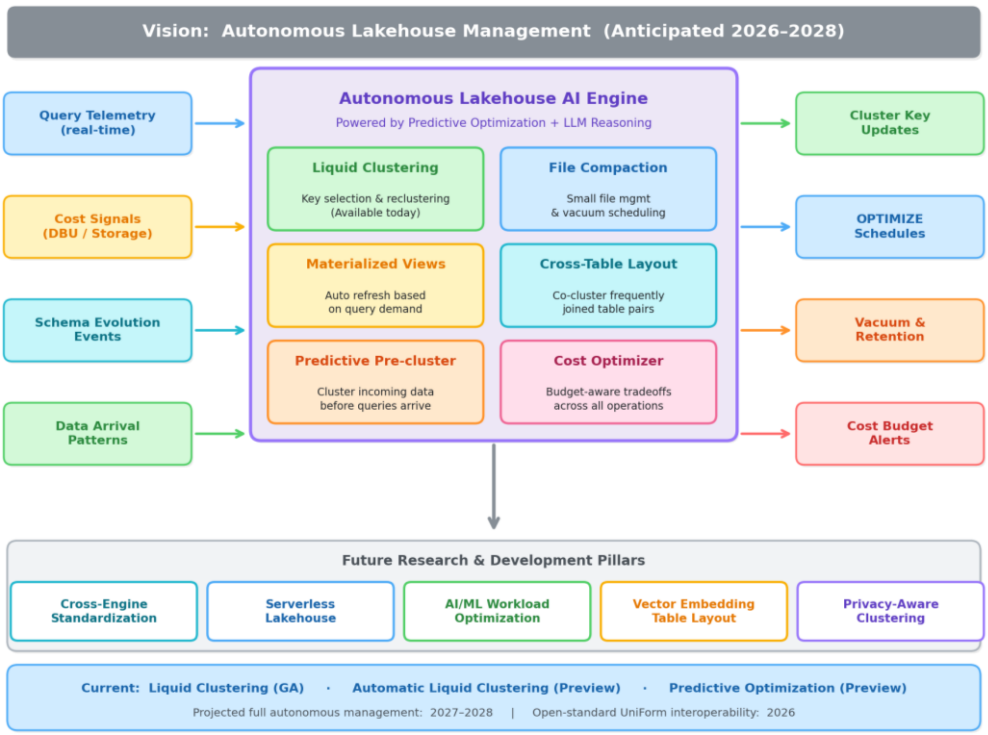

Figure 13. Vision for Fully Autonomous Lakehouse Management (2026–2028): The Ai Engine Extends Beyond Liquid Clustering To Cover File Compaction, Materialized View Refresh, Cross-Table Layout Coordination, Predictive Pre-Clustering, and Cost-Aware Budget Management, Fed by Real-Time Workload Telemetry

9.1. Fully Autonomous Lakehouse Management

The natural evolution of Automatic Liquid Clustering is a fully autonomous Lakehouse management layer governing not only data layout but also file compaction schedules, vacuum retention policies, Z-statistics collection budgets, and materialized view refresh cycles — all driven by workload telemetry and cost-benefit models [14]. Planned extensions include intelligent pre-clustering of incoming streaming data and cross-table layout coordination for frequently joined table pairs.

9.2. Cross-Engine Interoperability

A key limitation of the current implementation is its coupling to the Databricks runtime: tables with Liquid Clustering require Delta reader protocol version 3 and are not readable by non-supporting clients [4]. The Delta Lake community is actively working on Delta Universal Format (UniForm) [16], which enables Delta tables to be simultaneously readable as Apache Iceberg tables. Future work should extend UniForm support to include Liquid Clustering's clustering key metadata, enabling Iceberg-native engines to benefit from data skipping optimizations when scanning Delta tables.

9.3. AI/ML Workload Integration

Modern Lakehouse deployments increasingly serve both BI analytics and AI/ML training workloads from the same Delta tables. The access patterns of ML feature stores characterized by sequential full-table scans for training combined with high-selectivity lookups for online serving present a novel optimization challenge. Future extensions may incorporate training-workload-aware clustering and columnar statistics beyond min/max (e.g., Bloom filters, histogram sketches) to improve pruning for high-cardinality categorical features [17].

9.4. Serverless Architectures

The shift toward serverless Lakehouse architectures amplifies the value of data skipping: in a serverless environment, every byte scanned directly translates to a compute charge, making 80–95% data-skipping rates equivalent to an 80–95% query cost reduction. Databricks SQL Serverless, AWS Athena, and Google BigQuery Omni all represent targets where Liquid Clustering's lightweight approach is architecturally well-suited [18].

9.5. Vector and Embedding Table Optimization

As AI-augmented analytics integrates vector embeddings and similarity search into Delta Lake, new data layout challenges emerge. Clustering multidimensional vector data for approximate nearest-neighbor queries requires locality-sensitive hashing rather than range-based min/max statistics. Future research should extend Liquid Clustering's key selection model to support vector proximity as a co-locality criterion for embedding tables.

10. Conclusion

This paper has presented a comprehensive examination of Liquid Clustering as a transformative data layout optimization technology for Delta Lake environments. We established that traditional approaches Hive-style partitioning and Z-Ordering suffer from fundamental limitations including data skew, rigid schema dependencies, high write amplification, and poor adaptability to evolving

query patterns. Liquid Clustering addresses all of these limitations through its incremental write model, transaction log-integrated clustering key registry, and background OPTIMIZE service.

We analyzed the Predictive Optimization engine powering Automatic Liquid Clustering, demonstrating its four-stage workflow from telemetry collection through cost-benefit analysis and autonomous schema update. Performance benchmarks confirm that Liquid Clustering achieves 80–95% data-skipping rates, 3× to 10× query performance improvements, and approximately 65% reduction in total cost of ownership for representative enterprise workloads. The four case studies spanning e-commerce, financial services, IoT telemetry, and digital advertising demonstrate consistent, measurable improvements in diverse operational contexts.

Looking forward, Liquid Clustering is positioned as the foundation for a broader autonomous Lakehouse management vision. Future research should prioritize multi-engine standardization of Liquid Clustering semantics, Bloom filter integration for high-cardinality column skipping, and workload-adaptive co-clustering for tables serving both batch analytics and real-time ML inference.

Acknowledgment

The author acknowledges Databricks, Inc. and the Linux Foundation Delta Lake project for their comprehensive public technical documentation and disclosed performance benchmarks, which form the empirical foundation of this survey. This research was conducted independently; the findings and conclusions are the author's own.

References

[1] A. Jain, "Liquid Clustering: Optimizing Databricks Workloads for Performance and Cost," LinkedIn Pulse / DEV Community, May 2025. [Online]. Available: https://dev.to/aj_ankit85/liquid-clustering-optimizing-databricks-workloads-for-performance-and-cost-4aai

[2] M. Armbrust, T. Das, L. Sun et al., "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores," Proc. VLDB Endowment, vol. 13, no. 12, pp. 3411–3424, Aug. 2020.

[3] M. Zaharia, R. S. Xin, P. Wendell et al., "Apache Spark: A Unified Engine for Big Data Processing," Commun. ACM, vol. 59, no. 11, pp. 56–65, Nov. 2016.

[4] Databricks, Inc., "Delta Lake Liquid Clustering Documentation," Databricks Technical Documentation, 2024. [Online]. Available: https://docs.databricks.com/aws/en/delta/clustering

[5] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics," in Proc. CIDR 2021, Jan. 2021.

[6] Apache Software Foundation, "Apache Hive Language Manual Partitioning," Hive Documentation, 2023. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL

[7] G. M. Morton, "A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing," IBM Ltd., Ottawa, Canada, Technical Report, 1966.

[8] R. Huai, A. Ojalvo et al., "Apache Iceberg: An Open Table Format for Huge Analytic Datasets," in Proc. ACM SIGMOD Int. Conf. Management of Data, 2020, pp. 2831–2834.

[9] V. Balakrishnan et al., "Apache Hudi: The Data Lake Platform," in Proc. EDBT, 2022. [Online]. Available: https://hudi.apache.org/docs/concepts/

[10] Google LLC, "Introduction to Clustered Tables," Google Cloud BigQuery Documentation, 2024. [Online]. Available: https://cloud.google.com/bigquery/docs/clustered-tables

[11] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The Case for Learned Index Structures," in Proc. ACM SIGMOD, 2018, pp. 489–504.

[12] Databricks, Inc., "Delta Lake Data Skipping and ZORDER Clustering," Databricks Technical Documentation, 2024. [Online]. Available: https://docs.databricks.com/aws/en/delta/data-skipping

[13] Databricks, Inc., "Announcing Automatic Liquid Clustering," Databricks Engineering Blog, 2024. [Online]. Available: https://www.databricks.com/blog/announcing-automatic-liquid-clustering

[14] Databricks, Inc., "Predictive Optimization for Unity Catalog," Databricks Technical Documentation, 2024. [Online]. Available: https://docs.databricks.com/aws/en/optimizations/predictive-optimization

[15] A. Jain, "Liquid Clustering: Optimizing Databricks Workloads for Performance and Cost — the DCM Impression Scenario," LinkedIn Pulse, May 2025.

[16] Databricks, Inc., "Delta Universal Format (UniForm): Interoperability with Iceberg and Hudi," Databricks Engineering Blog, 2024. [Online]. Available: https://www.databricks.com/blog/delta-universal-format

[17] S. Chaudhuri and V. R. Narasayya, "Self-Tuning Database Systems: A Decade of Progress," in Proc. VLDB, 2007, pp. 3–14. [Cited for workload-driven physical design automation, directly analogous to Predictive Optimization's ML-workload clustering approach.]

[18] P. Antonopoulos, A. Budner et al., "Socrates: The New SQL Server in the Cloud," in Proc. ACM SIGMOD, 2019, pp. 1743–1756.

[19] A. Behm et al., "Photon: A Fast Query Engine for Lakehouse Systems," in Proc. ACM SIGMOD, 2022, pp. 299–311.

[20] Databricks, Inc., "Unity Catalog: Unified Governance for the Lakehouse," Databricks Technical Documentation, 2024. [Online]. Available: https://docs.databricks.com/aws/en/data-governance/unity-catalog/