International Journal of Latest Engineering and Management Research (IJLEMR)
ISSN: 2455-4847
www.ijlemr.com || Volume 10 – Issue 07 || July 2025 || PP. 09-16

Methods for Improving Query Performance Using Delta Lake

Chetan Urkudkar
Senior Staff Software Development Engineer, Liveramp Inc
San Ramon, California, USA

Siddharth Sharma
Principal Architect, Liveramp Inc
Newark, California, USA

Abstract: This article explores techniques for accelerating query execution in Delta Lake. The topic's relevance arises from ever-growing data volumes in lakehouse environments and the necessity of preserving ACID guarantees at exabyte scale. The novelty of this work lies in synthesizing recent optimization practices—such as Z-order clustering, Bloom filtering, and multi-level compaction. The study describes the architectural foundations of high throughput, examines Spark's scheduler mechanics, and analyzes Delta's integration with Trino. Special attention is given to the impact of transaction logs on the fault tolerance of analytical pipelines. The goal is to assemble comprehensive recommendations for table configuration in interactive analytics. To this end, the authors employ comparative analysis, critical review of documentation, and synthesis of industrial case studies. Contributions by A. M. Armbrust, K. Weller, M. Powers, M. Zhang, V. Sarogi, as well as materials from Databricks, Trino, and Delta.io, are surveyed. The conclusion outlines each technique's role in reducing I/O and improving stability. The resulting set of best practices is grounded in experimental read-latency metrics and I/O profiling validated by publicly available benchmarks. This article will benefit data engineers, BI platform architects, distributed systems researchers, and startups.

Keywords: Delta Lake; Z-order clustering; data skipping; Bloom index; compaction; Vacuum; ACID transaction log; Spark Catalyst; lakehouse; query optimization.

Introduction

Delta Lake is a modern data storage architecture that combines the reliability of traditional relational database management systems with the scalability of cloud-based data lakes. In the era of big data, organizations seek to accelerate analytical queries on ever-growing volumes of heterogeneous information without compromising data integrity. Delta Lake addresses this need by acting as a transactional layer between processing engines (for example, Apache Spark) and underlying file storage systems (S3, HDFS, etc.). The topic's relevance is underscored by Delta Lake's widespread industrial adoption—powering daily exabyte-scale workloads—and the continuous evolution of query-optimization techniques in this environment.

The aim of this article is to analyze methods for enhancing query performance in Delta Lake. To this end, the architecture of Delta Lake and its core mechanisms (transaction log, storage format) that guarantee ACID consistency and high throughput are examined. In addition, the study explores in detail several query-optimization strategies: data clustering (Z-ordering), data skipping, removal of obsolete versions (Vacuum), indexing support (e.g., via Bloom filters), and caching capabilities. A dedicated section discusses Delta Lake's integration with Apache Spark and other tools, and how this integration affects the performance of both individual queries and complex analytical pipelines.

Research tasks include:

1. Describing the Delta Lake architecture and transactional mechanisms that ensure data consistency under concurrent queries;
2. Investigating which Delta Lake features accelerate data read and processing;
3. Synthesizing recommended query-optimization practices (partitioning, Z-ordering, indexing, vacuuming, etc.) and their effects;
4. Analyzing Delta Lake's integration with the big-data ecosystem (Spark, Presto/Trino, Flink) in the context of query performance.

Materials and Methods

Materials. K. Weller [10] presented a comparative analysis of Apache Hudi, Delta Lake, and Iceberg, delineating the evolution of lakehouse formats. M. A. Armbrust [1] detailed the principles of the ACID transaction log and optimistic concurrency control. R. Philipon [6] examined regulatory requirements for

energy-management workflows, demonstrating the utility of metadata automation. M. Zhang [11], in a Salesforce case study, demonstrated performance improvements through Z-ordering and compaction. M. Powers [7] validated the effectiveness of Z-ordering on an experimental dataset. V. Saraogi [8] analyzed Vacuum procedures and version-retention policies.

Methods. This article employs comparative analysis, metadata extraction and analysis, critical review of primary sources, and systematic synthesis of real-world case studies; in addition, qualitative performance modeling is utilized to assess theoretical gains.

Results

Delta Lake is implemented as a storage layer on top of the Parquet format, adding a transaction log and extended metadata to lake data (see Figure 1).

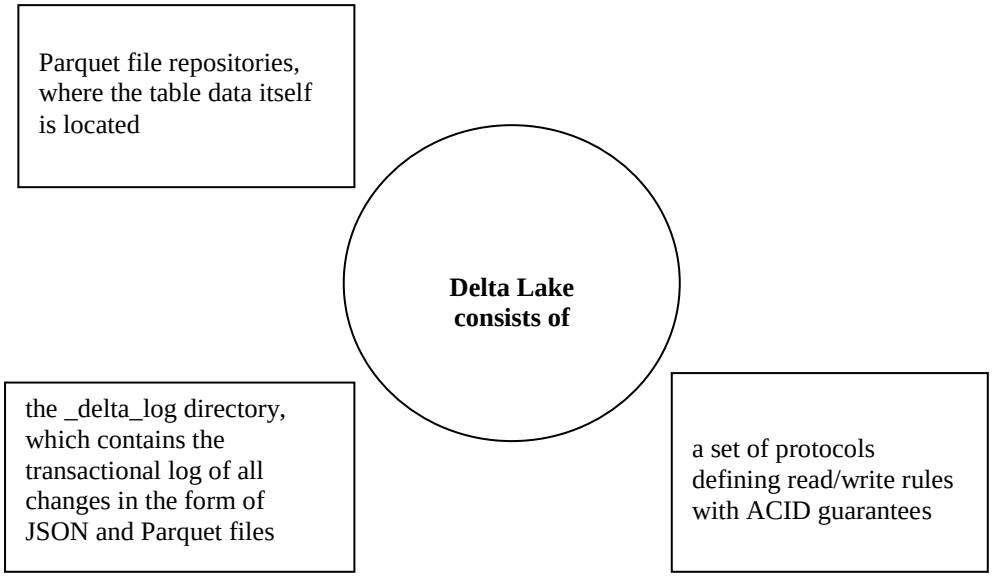

Figure 1: Delta Lake architecture (compiled by the author based on [1])

The transaction log is the system's cornerstone: each table operation (transaction) is recorded as a separate commit file, and periodically the log is compacted into Parquet-format checkpoints to speed access [1]. This design delivers full ACID guarantees on "raw" cloud storage—which natively lacks transactions and offers limited consistency. During data writes, Delta Lake employs Optimistic Concurrency Control [10]: multiple parallel writers may attempt updates simultaneously, each recording its intent in the log; if they collide (for example, by modifying the same partitions), version incompatibility is detected and the losing transaction is rolled back. This mechanism enforces Serializable isolation—readers never see partial results and always receive a consistent snapshot of the data [5]. Furthermore, each successful transaction in Delta Lake is atomic: either all of its changes are applied, or—in the event of a conflict or failure—none are visible. As a result, live analytics queries never encounter data in a "half-updated" state, a property critical for the correctness of complex aggregate computations.

Delta Lake's storage format builds on columnar Parquet files augmented with per-file statistics (min/max values, record counts) [3]. Each Parquet file serves as a micro-partition, and the transaction log tracks pointers to current files along with their metadata. This approach scales metadata handling: Delta Lake relies on Spark's distributed processing to manage information for billions of files, enabling efficient operation over petabyte-scale tables. Integrity control also includes schema enforcement—the system blocks writes if the incoming data schema does not match the table's expected schema, preventing "bad" records from entering the dataset [5]. Additionally, Delta Lake supports data versioning (time travel): the log allows queries to read the table as of a specified version number or timestamp. This capability aids experiment reproducibility, audit trails, and recovery from errors, although it does impose requirements for periodic cleanup of historical versions (discussed later).

Overall, Delta Lake's architecture provides a robust foundation for efficient querying: compact metadata (the entire log is checkpointed in Parquet, accelerating query planning [1]) and strict read-write isolation let analytical engines scan current data in parallel without blocking or costly coordination with the storage layer.

On top of this architectural foundation, Delta Lake offers a suite of features and recommended practices aimed at speeding up both read and write queries (see Figure 2).

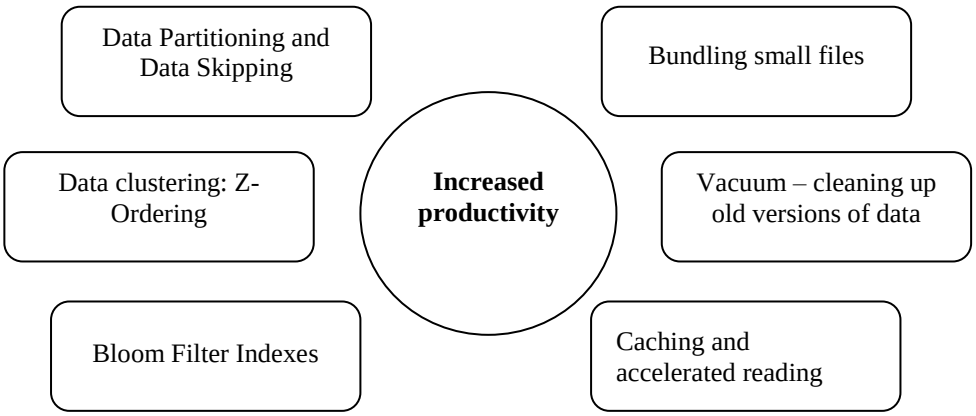

Figure 2: Mechanisms for improving query performance in Delta Lake (compiled by the author based on [2,3,6–8,11])

1. Data partitioning and data skipping. As with traditional data lakes, Delta Lake supports Hive-style partitioning of tables by column values. Beyond that, it automatically collects file-level statistics—minimum and maximum values plus record counts—on the first 32 columns of each Parquet file (the lowest-level partition) [3]. These statistics are logged in the transaction log and leveraged at query-planning time to skip files that cannot contain relevant data. For instance, when filtering by a date or ID range, Delta Lake compares the query predicate against each file's min/max metadata to identify candidate files. Files whose value ranges fall entirely outside the filter are never read from disk [6]. This transparent "data skipping" can dramatically cut I/O, especially in tables comprising thousands of files. To remain effective, statistics must be up to date: they are refreshed automatically on every data write, or can be manually updated via the ANALYZE command. A configuration setting lets users specify which columns to collect stats on (defaulting to the first 32), ensuring that critical filter columns later in the schema are covered. Newer Databricks Delta releases even introduce predictive optimization to automatically pick the optimal columns for indexing, further boosting data-skipping efficiency. With well-maintained statistics, selective SELECT queries can avoid reading a large majority of files [3].
2. Data clustering: Z-Ordering. Delta Lake includes a multi-dimensional sorting feature called Z-ordering, designed to enhance data locality across multiple columns and thereby magnify the benefits of data skipping for complex predicates. When one issues:

```
OPTIMIZE my_table ZORDER BY (col1, col2, …)
```

Delta Lake physically reorganizes rows so that records with similar values in the specified columns are colocated in the same files [7]. Unlike single-column partitioning—which yields separate directories per value—Z-ordering interleaves rows within files following a space-filling (Z-curve) sequence that accounts for all listed keys. This packing reduces sparsity: file-level min/max ranges for those columns become narrower and more precise, enabling queries with filters on these columns to skip even more files during execution.

Z-ordering is especially beneficial when dealing with high-cardinality columns, where classic partitioning would create an excessive number of tiny partitions. The choice of columns for Z-ordering requires analysis of query patterns: ordering by many columns improves multi-column filtering but may slightly reduce efficiency for individual fields [7]. As a rule of thumb, it is recommended to include the two to three most frequently filtered columns. Unlike Hive-style partitioning, Z-ordering does not create new on-disk directories; instead, data remain "clustered" within files, preserving flexibility as query structures evolve. In a Salesforce use case [11], Z-ordering a large marketing-events table by OrgId and EngagementDate reduced query times by orders of magnitude: Delta Lake first filters on OrgId partitions, then uses per-file date statistics to skip the remainder of files.

Liquid Clustering algorithm, recently introduced in the Delta Lake ecosystem, co-locates similar rows within the same physical files, reducing read latency. Unlike Hive partitioning and Z-ordering, Liquid Clustering allows the set of grouping columns to change dynamically—engineers need not anticipate future

query patterns. Designed for non-partitioned tables, it eliminates the overhead of managing directories and re-clustering existing data slices. In effect, the notion of partition directories is removed.

The mechanism leverages Optimistic Concurrency Control (OCC), which is crucial under high-throughput streaming and batch write workloads. Concurrent transactions maintain the integrity of the change log: upon detecting a conflict, only one write succeeds, and the others are rolled back.

A typical scenario: streamA writes U.S. sales while streamB concurrently writes U.K. sales into a global table partitioned by calendar date. If streamA commits first and streamB then attempts to modify the same date directory, OCC rejects streamB's transaction—even though the row sets are disjoint—wasting compute resources.

A partial workaround is configuring retries or adding finer partition keys (e.g., Country), but this increases storage overhead by creating tens of thousands of small files and tax on object listings.

Liquid Clustering removes these limitations by grouping at the row level, obviating reliance on directories (see Figure 3). MERGE, UPDATE, and DELETE conflicts are resolved automatically so long as they do not target the exact same rows. For tables with delete-vector support, OPTIMIZE or REORG operations proceed without blocking ongoing workloads.

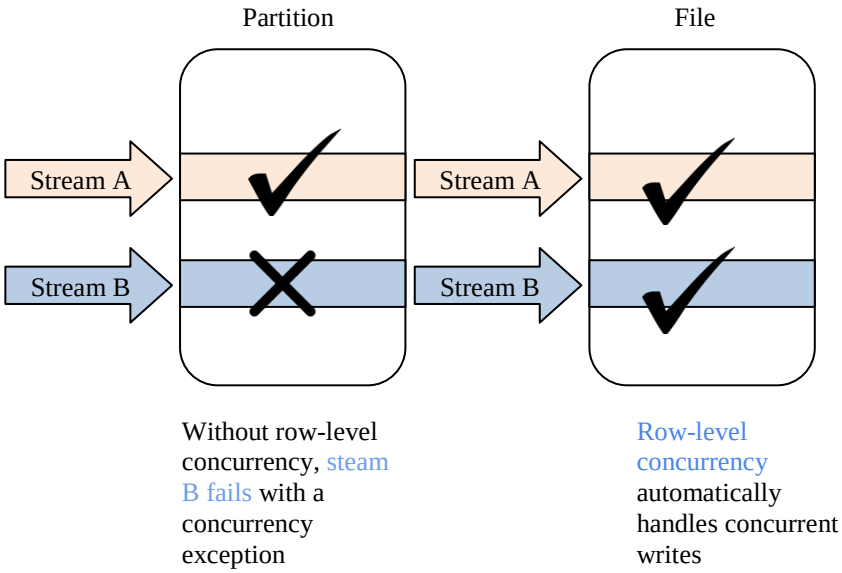

Figure 3: Comparison of Delta Lake's handling of concurrent writes (left: stream failure under traditional partitioning; right: both streams succeed with Liquid Clustering) [9]

In the scenario described, both streams succeed in committing their transactions until they attempt to modify the same row, even when the data reside in a shared file [9]. Liquid Clustering thus enables flexible scaling of concurrent writes without the costly overhead of managing partition directories [9].

3. Bloom Filter Indexes. An additional method for accelerating point lookups—particularly equality searches on high-cardinality fields such as user IDs or email addresses—is the Bloom Filter Index, an optional structure introduced in Delta Lake for arbitrary-text or identifier columns. A Bloom filter is a compact bit matrix that stores fingerprints of values, allowing the engine to quickly determine whether a given value could exist in a file. In Delta Lake, each Parquet file may have an associated Bloom filter for a chosen column. This index permits the system to ascertain, without reading the file, whether the sought value is definitely absent or possibly present.

   During query execution, if an equality predicate (or an IN-list) is applied to an indexed column, the engine first probes each file's Bloom filter: files for which the filter returns a negative result (the value cannot be present) are skipped [2]. This dramatically speeds up the retrieval of single records across millions of files. For example, a "find by primary key" query becomes an index-like operation rather than a full scan. Databricks recommends Bloom filters for columns with extremely high cardinality and frequent point queries (e.g., user_id, email), where min/max statistics prove ineffective. Native support for Bloom filters was added in Databricks Runtime 8.3 and can be created via SQL:

```
CREATE BLOOMFILTER INDEX ON TABLE my_table FOR COLUMNS(col1, …);
```

For instance, attaching a Bloom index on ProductID in a product-search log table accelerates queries such as

```
SELECT * FROM logs WHERE ProductID = X;
```

While Databricks documentation notes that Bloom filters remain useful on clusters without Photon's predictive I/O feature [2], they continue to serve as an effective optimization in open-source Delta Lake. By complementing data skipping—where min/max metadata define value ranges and Bloom filters exclude specific values—this multi-tiered filtering greatly reduces the volume of data read from disk, proportionally improving I/O-bound query performance [2].

4. Compaction of Small Files (Optimize Write and Auto-Compaction). One of the most pervasive challenges in data lakes is the proliferation of numerous small files—often a byproduct of streaming ingestions. This fragmentation degrades query performance, as each file incurs overhead for opening and reading. Delta Lake addresses this with two complementary approaches. First, the OPTIMIZE command enables manual merging of small files into larger ones (up to roughly 1 GB) within each partition. This operation rewrites data without altering its content, significantly reducing file fragmentation. Second, on Databricks with Spark, the Optimize Write and Auto-Compaction options can be enabled so files are coalesced automatically during data ingestion. For example, Auto Compaction will immediately consolidate the micro-batches of streaming data into fewer, larger files [11].

   Compaction is especially critical for sustaining high read throughput: Spark can scan a handful of large files in parallel far more efficiently than thousands of tiny ones sequentially. In a report from Salesforce engineers, their lake had accumulated approximately four million files, causing certain data-mutation jobs to stall due to the overhead of listing and transmitting such an immense file list to the Spark driver [10]. After running OPTIMIZE, the file count dropped to about 135 000, and performance improved dramatically [11]. Moreover, compaction enhances data skipping effectiveness: consolidated statistics and fewer files mean less metadata to check during planning. Consequently, regular use of VACUUM and OPTIMIZE is essential to keep Delta tables in a state that maximizes query performance.

5. VACUUM – Cleanup of Old Data Versions. The VACUUM command in Delta Lake removes obsolete files—so-called tombstoned files—that are no longer relevant to the table's current version (left behind by UPDATE/DELETE operations or compactions). Although these files do not participate in normal queries (they are marked deleted in the transaction log), they continue to occupy storage and can slow down file-system listings. VACUUM physically deletes them, reducing both the volume of data the file system must scan and overall storage consumption. By default, Delta Lake retains historical versions for at least seven days (168 hours) to preserve the ability to roll back transactions and perform time-travel queries [8]. Once this retention period elapses, the files can be safely purged. In practice, regular use of VACUUM can free gigabytes of "dead" data in environments with frequent updates, markedly improving metadata-listing performance.

   In the context of query performance, VACUUM exerts an indirect but significant impact by lowering the total file count in a Delta table's directory. As Varun Saraogi observed, routine vacuuming not only cuts storage costs but also ensures that no superfluous files—untracked by the current table version—remain to burden metadata scans, thereby enhancing query speed [8]. However, retention settings must balance the need for historical analysis: shortening the default seven-day window raises the risk that recently deleted versions become unavailable for older-date queries. For this reason, most deployments retain the seven-day default and schedule regular VACUUM jobs thereafter. In sum, VACUUM serves as a "deep clean" that preserves Delta table integrity and indirectly accelerates query execution by removing outdated objects.

6. Caching and Accelerated Reads. Another facet of query performance optimization is caching. At the Spark level, one can employ the CACHE mechanism or the Delta Cache (in Databricks) to keep frequently accessed data in memory or on local SSDs. Databricks reports that its platform automatically re-encodes "hot" Delta Lake data into a more efficient format on worker nodes (the so-called Data Skipping Index stored on NVMe), reducing repeat-read latency. This transparent caching means that, after an initial table scan, nodes can serve subsequent queries from local storage rather than reaching back to the remote data lake. In open-source Apache Spark, users may manually cache a Delta-derived DataFrame when repeated reuse is expected—especially valuable in BI scenarios with similar, recurring queries. Additionally,

Databricks' proprietary Photon engine (written in C++) accelerates low-level Parquet file scanning, further boosting query throughput.

Delta Lake was designed from the ground up for tight integration with Spark, with optimizations embedded in the Catalyst optimizer and API. When Spark executes a SQL query against a Delta table, it applies specialized rules—such as file pruning (filtering files by their statistics) and partition pruning—at planning time. Spark Structured Streaming natively treats Delta as both source and sink, enabling continuous pipelines without sacrificing ACID guarantees. Notably, the same Delta table can function as a batch table for SQL and simultaneously as a streaming source in Spark, eliminating redundant copies and permitting both historical and real-time queries against one dataset [5].

As an open-source project under the Linux Foundation, Delta Lake also offers connectors for Trino/Presto and Hive, plus native read support in Flink. For example, Trino's Delta Lake Connector lets Trino SQL queries read Delta tables directly—transaction log consistency and all—outside of Spark [4]. BI platforms such as Power BI can connect to Delta Lake either through Spark (via ODBC/JDBC) or via dedicated connectors (Power BI historically provided a connector for direct Delta reads with time travel and partition pruning). These integrations extend Delta's high-performance capabilities across the broader analytics ecosystem while preserving familiar SQL interfaces.

It is important to emphasize that many Delta Lake optimizations happen automatically—statistics collection, file pruning, metadata handling. Yet for peak performance, manual tuning remains essential: data engineers should regularly run OPTIMIZE for Z-Ordering, VACUUM for cleanup, and adjust partition sizes and checkpointing cadence. Documentation recommends targeting Parquet file sizes of roughly 1 GB for optimal scanning; files that are too small incur overhead, while files significantly larger than 1 GB can hinder parallelism [11]. These guidelines represent best practices for maintaining Delta tables in query-optimal form.

Summary: Delta Lake delivers ACID guarantees for correct query semantics alongside a rich set of performance-boosting features. Together, these capabilities overcome legacy data-lake limitations—disordered storage and lack of transactions—and enable high-throughput, interactive analytics at petabyte scale.

Discussion

The analysis demonstrates that query performance improvements in Delta Lake arise from a synergy of architectural design and data-optimization techniques. The ACID foundation of Delta Lake enhances query reliability: it enables complex, multiuser analytical workloads to execute without risking partial results. While this does not directly accelerate individual queries, it underpins stable performance—queries do not stall due to long-held locks, nor must data be re-read because of write conflicts—thereby indirectly contributing to faster, more predictable throughput.

The query-optimization methods presented in the Results section fall into two categories: metadata-driven techniques (data skipping, statistics collection, indexing) and data-centric techniques (clustering, compaction). Metadata-driven approaches reduce the volume of data considered at planning time, while data-centric methods accelerate the physical read by structuring and consolidating files. A key insight is their multi-layered interaction: to fully exploit data skipping, one should combine Z-ordering or thoughtful partitioning with Bloom-filter indexes for point queries. Only in concert do these features deliver dramatic speedups; without regular compaction, for example, Spark may still spend excessive time opening and closing millions of small files, despite having rich statistics.

The role of VACUUM is particularly noteworthy: although this operation does not directly boost query speed (and temporarily consumes resources), it prevents long-term performance degradation. Without routine vacuuming, Delta Lake retains all historical versions within the configured retention window—so frequent updates or deletes cause the file count to balloon, slowing down metadata scans. Maintaining data hygiene is thus essential for sustained efficacy. In this respect, Delta Lake brings the discipline of a data warehouse to the data-lake paradigm: routine maintenance tasks (akin to reindexing or statistics updates in an RDBMS) are required, but they integrate seamlessly into existing ETL pipelines (for example, daily OPTIMIZE and VACUUM jobs).

Delta Lake's deep integration with Spark is a major strength: Spark's Catalyst optimizer is aware of Delta semantics, allowing developers to write high-level SQL while benefitting from optimized physical plans. Standard Spark features such as predicate pushdown into Parquet are augmented by Delta's file-level pruning—filters eliminate entire files, not just rows within them. This synergy lets application developers focus on business logic, trusting the engine to handle low-level optimizations.

Integration with other tools, though less intimate, still enables Delta Lake to serve as a single source of truth across the enterprise. As a result, Delta's performance optimizations extend beyond Spark. For instance,

Trino's Delta connector leverages file-level metadata for split pruning, and Power BI—whether connected via Databricks or a native Delta connector—benefits from the same transaction-aware infrastructure [4]. Thus, Delta Lake delivers consistent, high-performance analytics across the entire BI stack, avoiding the need for separate data copies tailored to different tools.

Despite its many advantages, Delta Lake still presents opportunities for enhancement. For example, the lack of native secondary indexes (beyond Bloom filters) means that very fast point lookups on non-key columns must rely on Bloom-assisted file scanning—an approach that cannot match the performance of a B-tree index in a traditional RDBMS. Future releases may introduce richer indexing capabilities (Databricks is known to be experimenting with a "Data Index" feature). Additionally, Spark's memory management can become a bottleneck when tables grow extremely wide, increasing planning overhead. Predictive I/O seeks to mitigate this by dynamically selecting the most relevant statistics, but further refinements in optimizer memory handling would be beneficial.

Another consideration is the use of a single Delta table across multiple engines. When Spark and Trino access the same table concurrently, both must interpret the transaction log correctly to maintain consistency [4]. The Delta Standalone and Delta Sharing initiatives aim to provide an ACID-compliant protocol for non-Spark platforms, which will enable external systems to produce optimized writes (for example, pre-sorted files) and further extend Delta Lake's performance guarantees.

Practical recommendations drawn from this analysis:

- Regularly cluster large tables on the two to three columns most frequently used in filters (via Z-Ordering) and partition by high-level categories (such as date or logical domain).
- Enable automatic file coalescing at write time (Optimize Write/Auto-Compaction) or schedule periodic manual OPTIMIZE runs.
- Adjust delta.dataSkippingStatsColumns to include all key filter columns when more than the default 32 require statistics.
- Run VACUUM with a safe retention window (typically seven to thirty days) aligned to your historical-query requirements.
- Create Bloom-filter indexes for very high-cardinality identifiers (e.g., user_id, email) that underpin frequent point queries.

Adopting these best practices typically yields order-of-magnitude improvements in common analytical workloads—date-range filters, ID lookups, and category aggregations—as confirmed by both theoretical models and real-world deployments.

In summary, Delta Lake demonstrates that a "data lake plus transaction log" architecture can achieve performance on par with costly data warehouses for typical analytical queries. Techniques such as Z-Ordering, data skipping, and vacuuming all collaborate to eliminate the need for full "wide scans," while preserving the inherent flexibility and scalability of the lake paradigm—separating storage from compute and leveraging inexpensive, massive-scale storage.

Conclusion

Delta Lake delivers a comprehensive suite of tools for accelerating queries within the lakehouse paradigm, marrying ACID transactionality with data–reduction optimizations. From the foregoing analysis, two principal conclusions emerge. First, Delta Lake's core architecture—its transaction log, Parquet-based storage enriched with metadata, and optimistic concurrency control—provides a robust foundation for performance: queries operate on consistent, up-to-date snapshots and can scale across thousands of files without sacrificing data integrity. Second, higher-level techniques such as Z-order clustering dramatically improve multi-key locality, Bloom filters enable rapid file exclusion for point lookups, and routine compaction coupled with VACUUM preserves high access speeds over long periods of table usage.

The scientific significance of these findings lies in their demonstration that an "open data warehouse" built on open formats can match the performance traditionally associated only with specialized RDBMS. The study confirms that enriching a data lake with indexing layers and transactional guarantees removes the historical trade-off between scale and query speed. For the academic community, this work illustrates how a synthesis of ideas from databases, distributed systems, and compression algorithms can address real-world big-data challenges.

From a practical standpoint, the article consolidates best practices for configuring Delta Lake to maximize query throughput. Data engineers can follow these recommendations—partitioning, Z-ordering, indexing, and so forth—to achieve multiplicative speed gains without licensing proprietary systems.

Organizations already invested in Spark will find guidance on tuning Delta Lake to their specific query patterns, whether for BI dashboards or read-intensive ML pipelines.

In closing, Delta Lake has become a cornerstone of the modern big-data ecosystem, enabling unified, high-performance storage solutions. The performance-boosting methods reviewed—Z-ordering, data skipping, compaction, and VACUUM—render it a competitive choice for enterprise-scale analytics. As the project matures, further enhancements (adaptive indexing, tighter integration with query engines) are to be expected. Nevertheless, many organizations today are achieving industry-leading performance records on Delta Lake, underscoring the real-world value of these techniques. Future investigations may compare Delta Lake against alternative platforms across diverse workloads and probe its scalability limits—how many files or log size it can sustain without degradation. What remains clear is that the lakehouse architecture, embodied by Delta Lake, overcomes many legacy constraints and paves the way for fast, reliable analytics on petabyte-scale data using open platforms.

References

[1]. Armbrust M., Das T., Zhu X. et al. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics // Proceedings of the VLDB Endowment. – 2020. – Vol. 13, No. 12. – P. 3411–3424. – URL: https://vldb.org/pvldb/vol13/p3411-armbrust.pdf (accessed: 05/08/2025).

[2]. Bloom Filters in Delta Lake [Electronic resource] // Databricks: [official documentation]. – 2023. – URL: https://docs.databricks.com/aws/en/optimizations/bloom-filters.html (accessed: 05/06/2025).

[3]. Data Skipping [Electronic resource] // Databricks: [official documentation]. – 2023. – URL: https://docs.databricks.com/aws/en/delta/data-skipping.html (date of access: 05/18/2025).

[4]. Delta Lake Connector [Electronic resource] // Trino: [official documentation]. – 2024. – URL: https://trino.io/docs/current/connector/delta-lake.html (date of access: 05/11/2025).

[5]. Delta Lake Introduction [Electronic resource] // Delta.io: [official documentation]. – 2023. – URL: https://docs.delta.io/latest/delta-intro.html (date of access: 05/20/2025).

[6]. Philipon R. BACS Energy in the EU [Electronic resource] // Wattsense : [blog]. – 2021. – URL: https://www.wattsense.com/blog/building-management/bacs-energy-eu (date of access: 10.05.2025).

[7]. Powers M. Z-Ordering in Delta Lake [Electronic resource] // Delta.io : [blog]. – 2023. – URL: https://delta.io/blog/2023-06-03-delta-lake-z-order (date of access: 16.05.2025).

[8]. Saraogi V. Data Retention, Versioning, and Vacuum in Delta Lake [Electronic resource] // LinkedIn : [publication]. – 2023. – URL: https://www.linkedin.com/pulse/data-retention-versioning-vacuum-databricks-delta-lake-varun-saraogi-e3vac (accessed: 12.05.2025).

[9]. Stavrakakis C., Jiang C., Mokhtar M. Deep Dive: How Row-level Concurrency Works Out of the Box // Databricks Blog. – 2025. – Text: electronic. – URL: https://www.databricks.com/blog (accessed: 09.05.2025).

[10]. Weller K. Apache Hudi vs Delta Lake vs Apache Iceberg – Lakehouse Feature Comparison [Electronic resource] // Onehouse: [site]. – 2024. – URL: https://www.onehouse.ai/blog/apache-hudi-vs-delta-lake-vs-apache-iceberg-lakehouse-feature-comparison (accessed: 04.05.2025).

[11]. Zhang M. Boost Delta Lake Performance with Data Skipping and Z-Order [Electronic resource] // Salesforce Engineering Blog : [blog]. – 2021. – URL: https://engineering.salesforce.com/boost-delta-lake-performance-with-data-skipping-and-z-order-75c7e6c59133 (accessed: 15.05.2025).