DND

Delta Lake Performance Behind the Scenes: From Partitioning to Liquid Clustering

softserve

DND

About me

- Big Data Engineer with extensive experience in Python
- Enthusiastic about math and machine learning
- Big fan of Remembrance of Earth's Past trilogy by Liu Cixin

Roman Dryndik
Senior Big Data Software Engineer

softserve

Agenda

1. Intro
2. Compaction
3. Partitioning
4. VACUUM
5. Statistics and Data Skipping
6. Z-Ordering
7. Bloom Filter Index
8. Liquid Clustering
9. Summary
10. Q&A

Intro

Introduction

Why Delta Lake Optimization Matters

Key Problems:

- Small files problem: streaming writes, and DML operations generate thousands of tiny files → high metadata overhead, slow object-store listings, increased I/O costs, and poor compression
- Over-partitioning: high-cardinality partition keys (e.g., user_id) create thousands of directories with micro-files → metadata explosion and degraded read performance
- Updates and deletes leave behind outdated files → table size grows and performance declines
- Inefficient data scanning: without proper data layout, queries scan terabytes instead of gigabytes

DND

Compaction

Reducing Small Files to Speed Up Queries

softserve

Compaction

The Small Files Problem

Root cause:

- **Streaming Ingestion**: writing micro-batches every few seconds/minutes
- **DML Operations**: frequent MERGE, UPDATE, or DELETE actions produce new files
- **Over-partitioning**: splitting data into too many granular folders

Compaction

The Small Files Problem

Performance impact:

- Growth of metadata in the transaction log
- **Metadata overhead**: the driver spends more time listing files in object storage (S3/ADLS) than processing data
- **High I/O latency**: opening/closing thousands of tiny files is inefficient
- **Poor compression**: parquet creates massive overhead (headers/footers) when files are too small

Compaction

OPTIMIZE & Auto Optimize

The **OPTIMIZE** command:

- Triggers **bin-packing**: reads small files and coalesces them into larger files (target size: **1 GB** by default)
- **Idempotent**: running it twice on the same data does nothing
- **ACID**: does not block concurrent readers or writers

Auto Optimize (Automated approach):

- **Optimized writes**: shuffles data **before** writing to reduce file count (increases write latency, improves read)
- **Auto compact**: triggers a "mini-optimize" **after** a write transaction commits

Compaction

ss — vim compaction.sql — vim — vim compaction.sql — 80×24

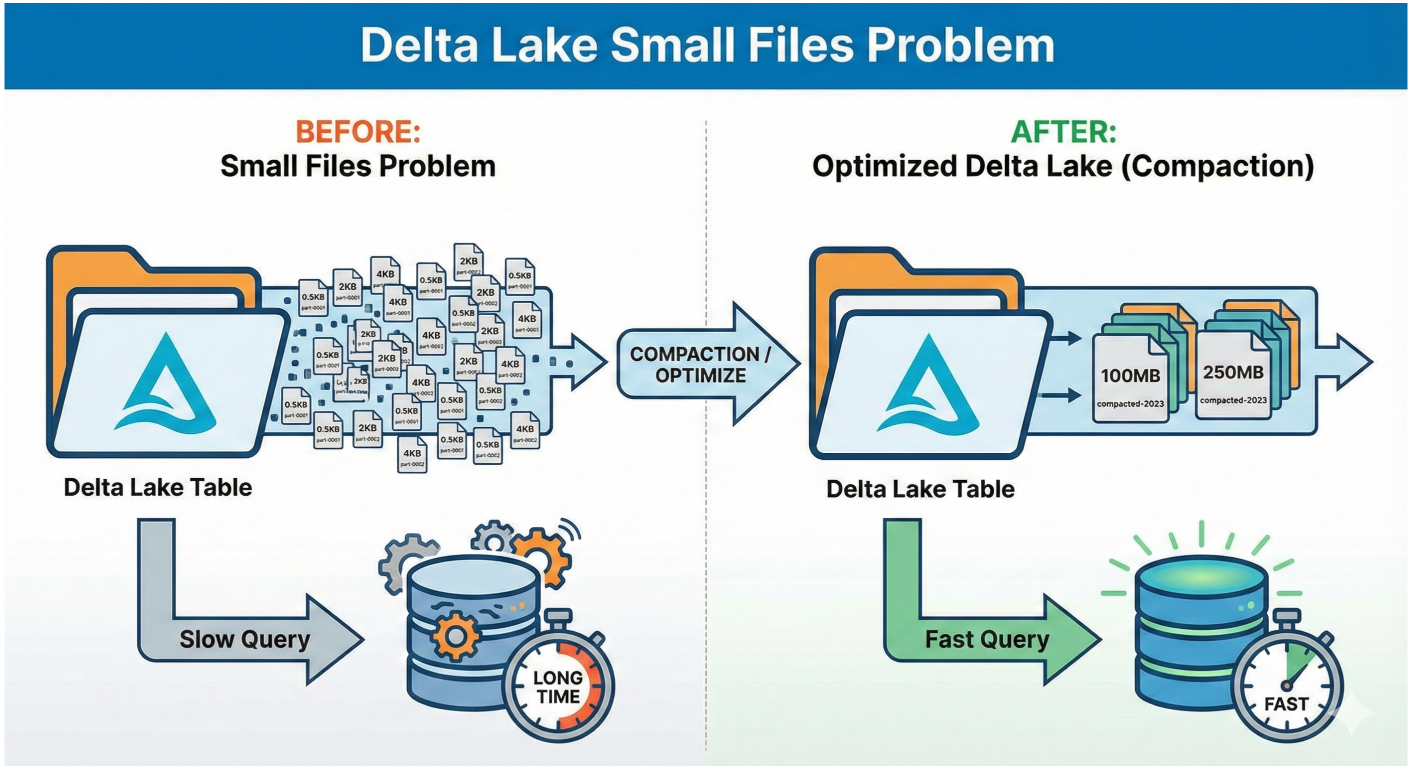
```sql
-- Standard bin-packing
OPTIMIZE events_table;

-- Incremental compaction (Save resources)
OPTIMIZE events_table WHERE date ≥ current_date() - 1;
```

```
"compaction.sql" 5L, 147B
```

ss — vim compaction.py — vim — vim compaction.py — 80×24

```python
spark.databricks.delta.optimizeWrite.enabled = true
spark.databricks.delta.autoCompact.enabled = true
```

Compaction


Delta Lake Small Files Problem
BEFORE:
Small Files Problem
AFTER:
Optimized Delta Lake (Compaction)
COMPACTION / OPTIMIZE
100MB
250MB
Delta Lake Table
Delta Lake Table
Slow Query
LONG TIME
Fast Query
FAST

Compaction

Best Practices

- **Streaming**: enable **Auto Compact** + run OPTIMIZE periodically (e.g., daily/weekly) for cleanup
- **Batch ETL**: run OPTIMIZE at the end of the daily job
- **Strategy**: use **predicates** (WHERE) to avoid compacting the entire table every time

Partitioning

Splitting Data by Keys for Faster Queries

Partitioning

Physical Data Layout

The concept:

- Splitting data into sub-directories based on high-level keys (e.g., Date, Region, Department)
- Structure: s3://bucket/table/date=2024-01-01/region=US/...

The benefit — partition pruning (aka data skipping):

- When a query includes a partition key in the WHERE clause, the engine completely ignores irrelevant directories
- Reduces scanned data from **terabytes** to **gigabytes**

The use case:

- Best for heavy filtering on specific columns (e.g., "Give me data for **Yesterday**").

Partitioning

partitioning.sql

```sql
-- Create a table partitioned by date
CREATE TABLE sales (
  id INT,
  amount DOUBLE,
  sale_date DATE
) USING DELTA
PARTITIONED BY (sale_date);

-- Query that triggers Partition Pruning
SELECT sum(amount)
FROM sales
WHERE sale_date = '2024-01-01'; -- Skips all other
```

Partitioning

Over Partitioning

The trap — over-partitioning:

- Partitioning by high-cardinality columns (e.g., user_id, timestamp, order_id)
- **Result**: thousands of tiny directories containing tiny files
- **Impact**: severe metadata overhead (driver node bottleneck) and loss of compression efficiency

Typical symptoms:

- Many partitions containing **1–10 files**
- File sizes < 32 MB
- Query plan showing **hundreds of partitions scanned**
- OPTIMIZE is taking hours because the data is too fragmented

Partitioning

How to Avoid Over-Partitioning

General recommendations:

- Partition only by **low- or medium-cardinality** columns (e.g., date, country, category)
- Prefer **Z-ordering** or **Bloom filters** instead of “deep partitioning”
- If you have already over partitioned:
  - **Repartition** the table and rewrite it
  - Consolidate partitions (e.g., daily → monthly)

Rule of thumb:

- Partitions should generally contain at least **1 GB** of data
- If your table is small (< 1 TB), you might **not** need partitioning at all

DND

VACUUM

Keeping Tables Clean and Performant

softserve

VACUUM

Problem

- UPDATE, DELETE, INSERT, and MERGE operations create new files
- Old files increase the table size
- When your tables are large, you have poor read performance

VACUUM

Solution

- VACUUM operation — removes old unused parquet files

vacuum.sql

```
-- Run VACUUM against table_name
VACUUM table_name [RETAIN num HOURS];
```

VACUUM

Solution

Retention period:

- Default: 7 days (RETAIN 168 HOURS)
- Can be reduced (e.g., for dev environments): RETAIN 24 HOURS
- Warning: too short → cannot time travel/rollback

vacuum.sql

```sql
-- Run VACUUM against table_name
VACUUM table_name [RETAIN num HOURS];
```

VACUUM

Impact on Table Size & Performance

Effects of VACUUM:

- Reduces table size on disk
- Speeds up table scans
- Reduces cluster workload during reads

Tips:

- Run VACUUM after large updates/deletes
- Avoid running too frequently to preserve history

DND

Statistics & Data Skipping

Query Faster Without Full Scans

softserve

Statistics & Data Skipping

File-Level Statistics

Delta Lake stores metadata for every data file:

- Min/max values per column
- Number of records
- Null counts
- Partition information (if applicable)

Why this matters:

- The client can understand the data distribution **without** opening the file
- Enables smarter, more targeted reading

DND

Statistics & Data Skipping

rdryn — admin@bozon: ~/projects/delta-lake-definitive-guide — ssh admin@192.168.0.10 — 123×32


```
\"fips\":1001,\"cases\":0,\"deaths\":0},\"maxValues\":{\"date\":\"2021-01-10\",\"county\":\"Ziebach\",\"state\":\"Wyoming\"
,\"fips\":78030,\"cases\":920560,\"deaths\":25562},\"nullCount\":{\"date\":0,\"county\":0,\"state\":0,\"fips\":1262,\"cases
\":0,\"deaths\":3588}}"
  }
}
{
  "add": {
    "path": "part-00006-d0ec7722-b30c-4e1c-92cd-b4fe8d3bb954-c000.snappy.parquet",
    "partitionValues": {},
    "size": 883342,
    "modificationTime": 1619121488000,
    "dataChange": true,
    "stats": "{\"numRecords\":147181,\"minValues\":{\"date\":\"2021-01-10\",\"county\":\"Abbeville\",\"state\":\"Alabama\",
\"fips\":1001,\"cases\":0,\"deaths\":0},\"maxValues\":{\"date\":\"2021-02-25\",\"county\":\"Ziebach\",\"state\":\"Wyoming\"
,\"fips\":78030,\"cases\":1188101,\"deaths\":29025},\"nullCount\":{\"date\":0,\"county\":0,\"state\":0,\"fips\":1250,\"case
s\":0,\"deaths\":3510}}"
  }
}
{
  "add": {
    "path": "part-00007-4582392f-9fc2-41b0-ba97-a74b3afc8239-c000.snappy.parquet",
    "partitionValues": {},
    "size": 325440,
    "modificationTime": 1619121487000,
    "dataChange": true,
    "stats": "{\"numRecords\":47559,\"minValues\":{\"date\":\"2021-02-25\",\"county\":\"Abbeville\",\"state\":\"Alabama\",\
"fips\":1001,\"cases\":0,\"deaths\":0},\"maxValues\":{\"date\":\"2021-03-11\",\"county\":\"Ziebach\",\"state\":\"Wyoming\",
\"fips\":78030,\"cases\":1208672,\"deaths\":30068},\"nullCount\":{\"date\":0,\"county\":0,\"state\":0,\"fips\":408,\"cases\
":0,\"deaths\":1170}}"
  }
}
admin@bozon:~/projects/delta-lake-definitive-guide$
```

Statistics & Data Skipping

```
rdryn — admin@bozon: ~/projects — ssh admin@192.168.0.10 — 123×32
Zellij (fascinating-brachiosaur)  Tab #1                                  Alt <[]>  VERTICAL
```


```
admin@bozon: ~/projects/delta-lake-definitive-guide —— 0
    "dataChange": true,
    "stats": "{\"numRecords\":147181,\"minValues\":{\"date\"
:\"2021-01-10\",\"county\":\"Abbeville\",\"state\":\"Alabama
\",\"fips\":1001,\"cases\":0,\"deaths\":0},\"maxValues\":{\"
date\":\"2021-02-25\",\"county\":\"Ziebach\",\"state\":\"Wyo
ming\",\"fips\":78030,\"cases\":1188101,\"deaths\":29025},\"
nullCount\":{\"date\":0,\"county\":0,\"state\":0,\"fips\":12
50,\"cases\":0,\"deaths\":3510}}"
  }
}
{
  "add": {
    "path": "part-00007-4582392f-9fc2-41b0-ba97-a74b3afc8239
-c000.snappy.parquet",
    "partitionValues": {},
    "size": 325440,
    "modificationTime": 1619121487000,
    "dataChange": true,
    "stats": "{\"numRecords\":47559,\"minValues\":{\"date\":
\"2021-02-25\",\"county\":\"Abbeville\",\"state\":\"Alabama\
",\"fips\":1001,\"cases\":0,\"deaths\":0},\"maxValues\":{\"d
ate\":\"2021-03-11\",\"county\":\"Ziebach\",\"state\":\"Wyom
ing\",\"fips\":78030,\"cases\":1208672,\"deaths\":30068},\"n
ullCount\":{\"date\":0,\"county\":0,\"state\":0,\"fips\":408
,\"cases\":0,\"deaths\":1170}}"
  }
}
admin@bozon:~/projects/delta-lake-definitive-guide$
```



```
admin@bozon: ~/projects/delta-lake-definitive-guide —— 0
{
  "numRecords": 47559,
  "minValues": {
    "date": "2021-02-25",
    "county": "Abbeville",
    "state": "Alabama",
    "fips": 1001,
    "cases": 0,
    "deaths": 0
  },
  "maxValues": {
    "date": "2021-03-11",
    "county": "Ziebach",
    "state": "Wyoming",
    "fips": 78030,
    "cases": 1208672,
    "deaths": 30068
  },
  "nullCount": {
    "date": 0,
    "county": 0,
    "state": 0,
    "fips": 408,
    "cases": 0,
    "deaths": 1170
  }
}
admin@bozon:~/projects/delta-lake-definitive-guide$
```


```
Ctrl + <g> LOCK  <p> PANE  <t> TAB  <n> RESIZE  <h> MOVE  <s> SEARCH  <o> SESSION  <q> QUIT  ...
```

Statistics & Data Skipping

Data Skipping Means Client Avoids Reading Files That Cannot Satisfy the Query

How it works:

- Client reads delta metadata (stats)
- Compares stats with query filters
- Only loads files whose min/max **overlaps** with the filter

Result:

- Fewer files scanned
- Less I/O
- Faster queries

DND

Statistics & Data Skipping

Data Skipping Means Client Avoids Reading Files That Cannot Satisfy the Query

What the client does:

- Checks file stats (min/max event_date)
- Skips all files where **max < '2023-06-01'**
- Reads only relevant files instead of scanning the whole table

Impact:

- Significant reduction in scanned data
- Lower latency
- Better cluster utilization

filter.sql

```sql
SELECT *
FROM events
WHERE event_date >= '2023-06-01';
```

Z-Ordering

Improving Multi-Column Filters

Z-Ordering

What Is Z-Ordering?

Z-Ordering is a data clustering algorithm:

- Reorders data within Delta files
- Groups related column values close together on disk
- Uses a space-filling “Z-curve” to interleave bits from multiple columns

Goal:

- Make multi-column filtering faster by improving data locality

Z-Ordering

Why Z-Ordering?

Benefits:

- Reduces the number of files a client needs to scan
- More efficient data skipping
- Works best for **high-cardinality columns** (e.g., user_id, timestamp)
- Particularly effective for multi-column predicates:

z-curve.sql

```sql
WHERE user_id = X AND event_date BETWEEN ...
```

Z-Ordering

How Z-Order Helps

Without Z-order:

- Data for user_id and event_date is spread across many files, which means poor locality
- Client scans many files just to find a small subset of rows

With Z-order:

- Related values of user_id + event_date are co-located in fewer files
- The client can skip most files

Result:

- Significant improvements in selective queries

Z-Ordering

How Z-Order Helps

Scenario:

- You have a user events table

Columns:

- user_id — high-cardinality identifier
- event_date — timestamp/date
- event_type — string (e.g., "click", "view", etc.)
- session_id
- country
- device_type

example.sql

```
SELECT *
FROM events
WHERE
    user_id = '65e1419e-59db-4c5e-9914-b69361bac2fb' AND
    event_date BETWEEN '2025-05-01' AND '2025-06-01';
```

Query pattern:

- Most queries filter by a combination of user_id and event_date

Z-Ordering

How Z-Order Helps

Without Z-order:

- Rows for the same user_id are spread across many files
- Date ranges overlap across files unpredictably
- The client must scan **hundreds** of files

With Z-order on (user_id, event_date):

- Rows for each user_id become physically close together
- Their dates fall into tight ranges
- The client can skip most files

z-curve-table-optimization-2.sql

```sql
-- Optimize table_name by user_id and event_date
OPTIMIZE events ZORDER BY (user_id, event_date);
```

Notes:

- Often combined with **OPTIMIZE** to compact small files and then apply Z-order
- Works best when you Z-order the columns most used together in filters
- Not necessary for partitioned columns

Bloom Filter Index

Fast Lookups on High-Cardinality Columns

Bloom Filter Index

Why Bloom Filters?

Use case:

- Columns with **high cardinality** (UUIDs, emails, session IDs, order IDs)
- Traditional data skipping (min/max stats) doesn't help for these columns
- Bloom filters give a fast, space-efficient way to check "**might** contain this value?"

Result:

- Avoids scanning files that do **not** contain the value

Bloom Filter Index

Bloom Filter = Probabilistic Index

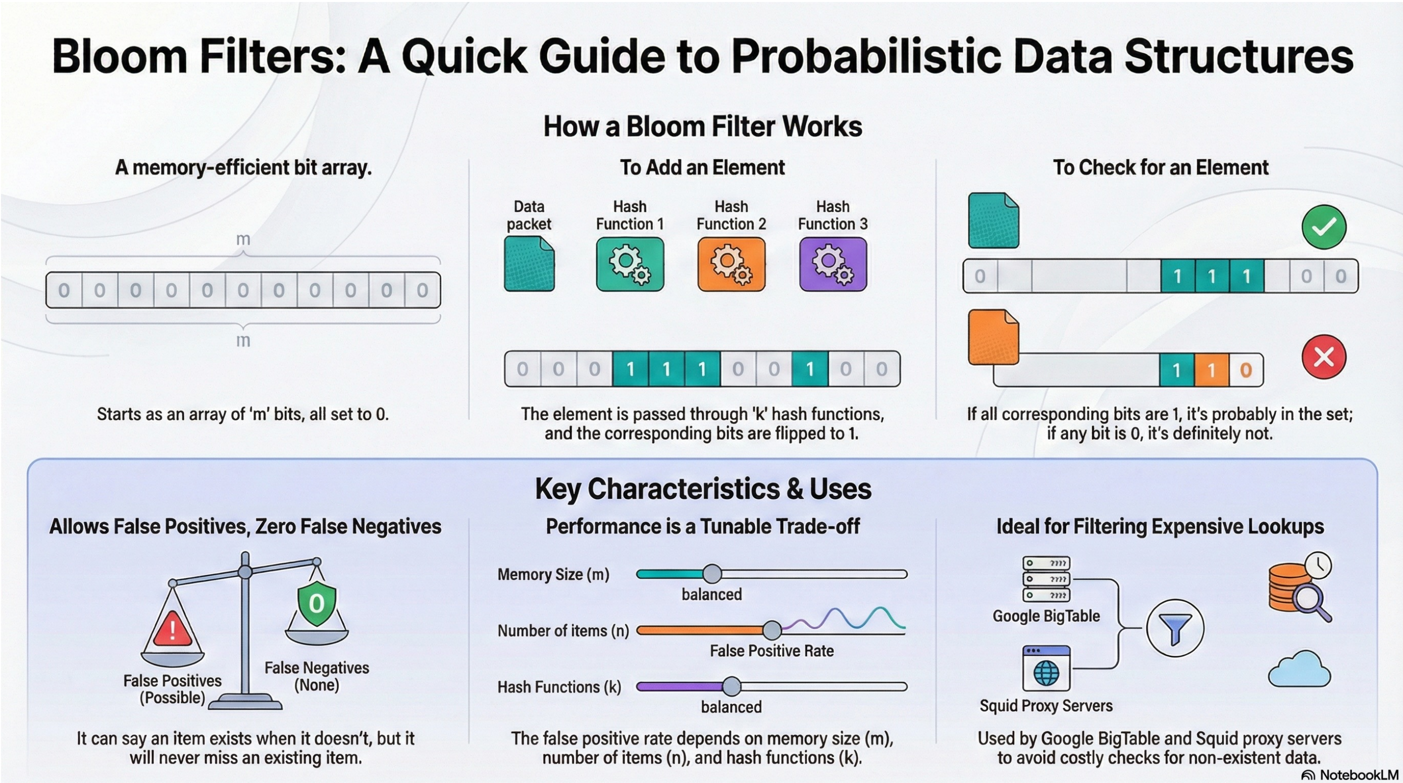
Benefits:

- Very compact bit array + multiple hash functions
- Can answer:
  - **“Definitely not present”**
  - **“Possibly present” (**false positives **possible**, false negatives **not** possible)

Why this helps:

- It is possible to check the Bloom filter **before** reading the file
- This eliminates large portion of irrelevant files → less I/O, faster reads

Bloom Filter Index



Bloom Filter Index

Example — UUID or Email Lookups

Consider a table:

- user_events

Columns:

- event_id (UUID)
- user_email (string)
- event_timestamp
- event_type

Bloom Filter Index

Example — UUID or Email Lookups

Without Bloom filters:

- UUIDs have no meaningful ordering
- min/max stats are useless
- Client may scan hundreds of files

bloom-filter.sql

```sql
SELECT *
FROM user_events
WHERE event_id = '550e8400-e29b-41d4-a716-446655440000';
```

Bloom Filter Index

Example — UUID or Email Lookups

With Bloom filter on event_id:

bloom-filrer-on-event-id.sql

```
ALTER TABLE user_events
ADD BLOOMFILTER INDEX
ON event_id
OPTIONS (fpp = 0.01, numItems = 1000000);
```

Bloom Filter Index

Example — UUID or Email Lookups

Effect:

- Client quickly checks each file's Bloom filter
- Most files are rejected immediately
- Only a few files are scanned

Moreover:

- Great for equality lookups on large, unique columns
- False positives are fine because it's still much faster than scanning
- Complements Z-order and statistics-based skipping
- **Small** index size means **low** overhead

DND

Liquid Clustering

Adaptive Layout for Fast Queries

softserve

Liquid Clustering

Traditional Optimization Challenges

- Table management requires careful partition design
- Changing partition keys is expensive and often requires a full table rewrite
- Z-ordering needs continuous monitoring to match evolving query patterns
- Data skew: uneven distribution creates imbalanced partition sizes, which slows down queries

Liquid Clustering

What Is Liquid Clustering?

A new dynamic clustering strategy:

- Continuously maintains clustering as data evolves
- No need for expensive, full-table OPTIMIZE operations
- Automatically adapts as new data arrives or existing data changes
- It replaces partitioning and Z-ordering
- No more expensive rewrites

Idea:

- Instead of periodically “reorganizing everything,” the table stays well-organized over time
- The data is organized using clustering keys to optimize layout and simplify table management

Liquid Clustering

When to Use Liquid Clustering

- Frequent filtering on high-cardinality columns where partitioning fails
- Tables with data skew that need balanced distribution
- Fast-growing tables requiring constant tuning
- High-concurrency writes, where clustering reduces conflicts
- Changing query access patterns over time
- Cases where partitioning would create too many or too few partitions

Liquid Clustering

Enabling Liquid Clustering

- Add CLUSTER BY (<columns>) in CREATE TABLE for existing tables
- ALTER TABLE <name> CLUSTER BY (<columns>) for new tables
- Updates metadata only — does **not** rewrite existing data
- SQL, Python, and Scala APIs are all supported
- DataFrame API note: clustering keys can be set only at creation or with overwrite mode — not in append mode

Liquid Clustering

Enabling Liquid Clustering

Clustering is **incompatible** with traditional partitioning and ZORDER.
It is designed to **replace** both.

Liquid Clustering

Choosing Clustering Keys

Selecting the right keys is crucial — good keys maximize data skipping and boost query performance.

- **Filter-first**: choose columns most used in WHERE clauses and joins
- **Stats required**: keys must be among columns with collected statistics (first 32 by default)
- **Avoid redundancy**: skip highly correlated columns — pick just one
- **Up to 4 keys**: more can hurt performance on tables < 10 TB

Liquid Clustering

Automatic Liquid Clustering

How it works:

- **Workload analysis**: identifies the most frequently filtered columns
- **Adaptive optimization**: updates keys as query patterns or data distribution change
- **Cost-aware decisions**: picks new keys only when the benefit outweighs the re-clustering cost
- **Powered by predictive optimization**: runs asynchronously in the background

clustering.sql

```
CREATE OR REPLACE TABLE table1(column01 int, column02 string)
CLUSTER BY AUTO;
```

Liquid Clustering

Automatic Liquid Clustering

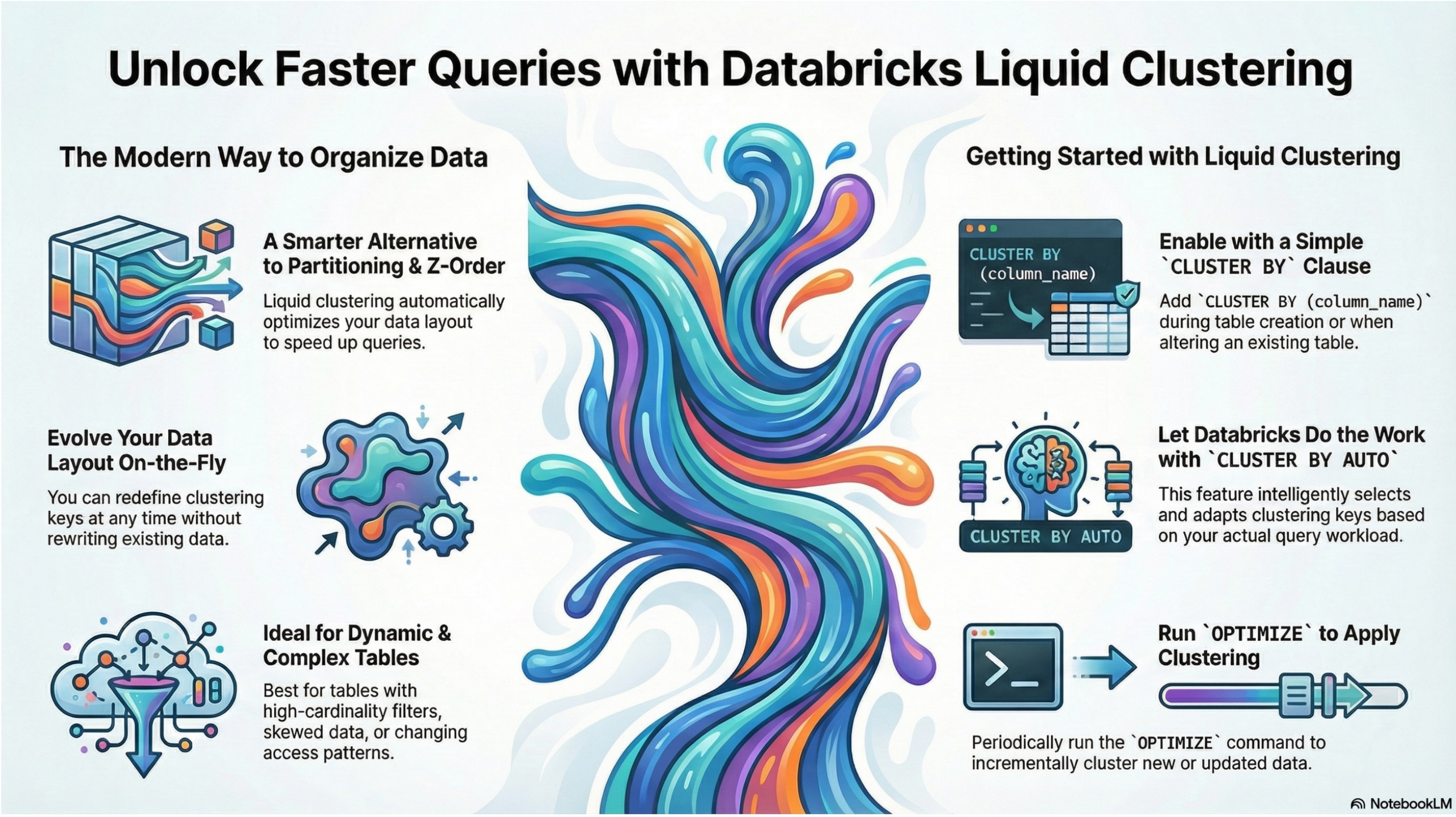
When keys may **not** be selected:

- The table is too small to benefit
- The existing layout is already effective
- Insufficient or inconsistent workload

Liquid Clustering

Key Benefits

- Stable performance without heavy maintenance jobs
- Faster queries with consistent data skipping
- Lower cost: avoids massive table rewrites
- Supports evolving datasets and dynamic clustering keys
- Redefine clustering keys without rewriting existing data is possible
- Simplify data layout: replaces manual partitioning and Z-order with one flexible technique
- Adapt to change: modify clustering keys without costly rewrites
- Automate with intelligence: automatic Liquid Clustering analyzes workloads and manages keys



Unlock Faster Queries with Databricks Liquid Clustering
The Modern Way to Organize Data
A Smarter Alternative to Partitioning & Z-Order
Liquid clustering automatically optimizes your data layout to speed up queries.
Evolve Your Data Layout On-the-Fly
You can redefine clustering keys at any time without rewriting existing data.
Ideal for Dynamic & Complex Tables
Best for tables with high-cardinality filters, skewed data, or changing access patterns.
Getting Started with Liquid Clustering
CLUSTER BY (column_name)
Enable with a Simple `CLUSTER BY` Clause
Add `CLUSTER BY (column_name)` during table creation or when altering an existing table.
CLUSTER BY AUTO
Let Databricks Do the Work with `CLUSTER BY AUTO`
This feature intelligently selects and adapts clustering keys based on your actual query workload.
Run `OPTIMIZE` to Apply Clustering
Periodically run the `OPTIMIZE` command to incrementally cluster new or updated data.
NotebookLM

Summary

Summary & Recommendations

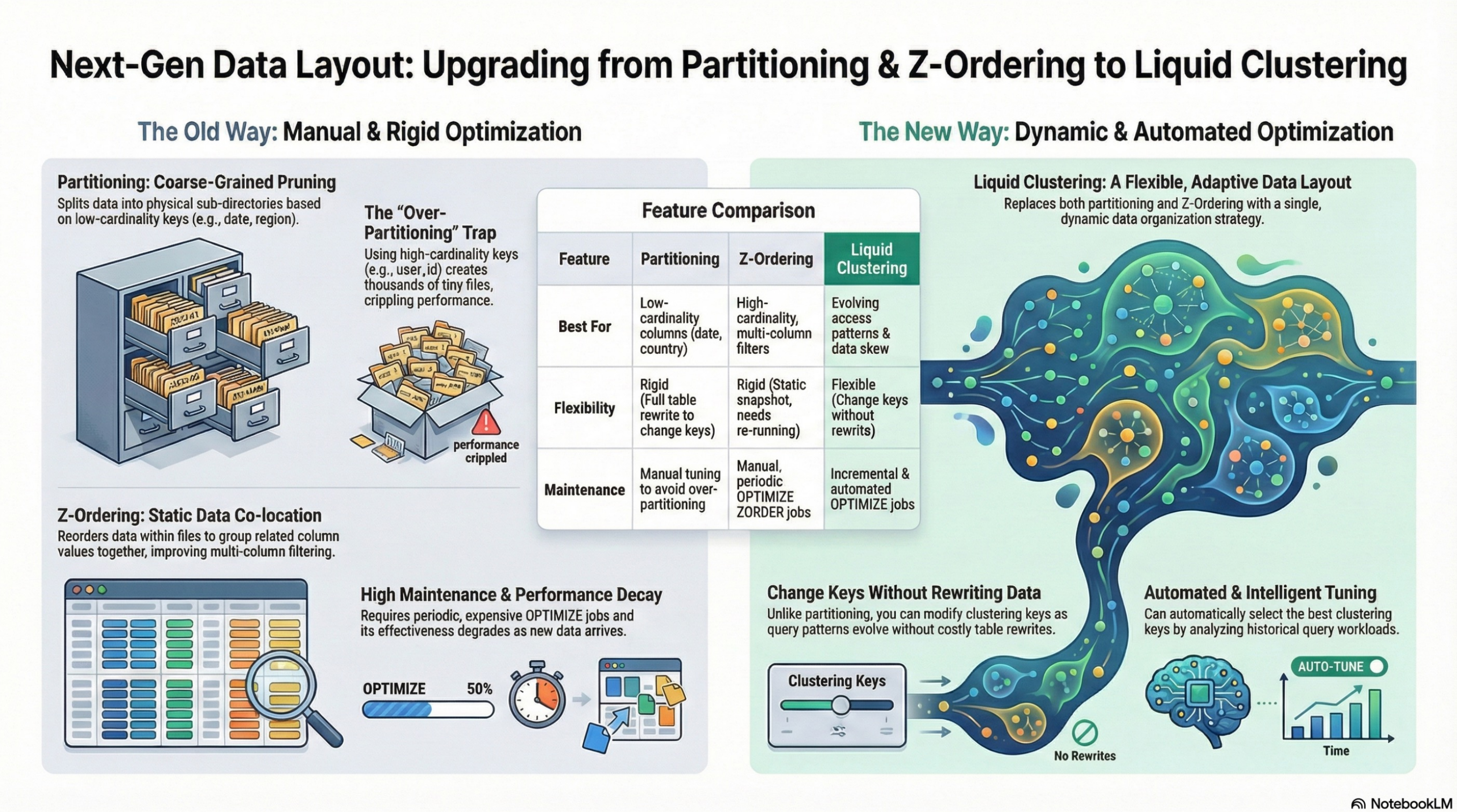
Combine Techniques for Best Performance

- No single optimization solves everything — Delta works best when techniques are combined
- Start simple: partition → compact → Z-order
- Add Bloom for faster lookups on high-cardinality columns
- Run **VACUUM** regularly
- Migrate to Liquid Clustering (if possible) for large, frequently-updated tables
- Re-evaluate periodically as data volume grows

DND

Any Questions?

softserve


Next-Gen Data Layout: Upgrading from Partitioning & Z-Ordering to Liquid Clustering
The Old Way: Manual & Rigid Optimization
The New Way: Dynamic & Automated Optimization
Partitioning: Coarse-Grained Pruning
Splits data into physical sub-directories based on low-cardinality keys (e.g., date, region).
The "Over-Partitioning" Trap
Using high-cardinality keys (e.g., user_id) creates thousands of tiny files, crippling performance.
performance crippled
Z-Ordering: Static Data Co-location
Reorders data within files to group related column values together, improving multi-column filtering.
High Maintenance & Performance Decay
Requires periodic, expensive OPTIMIZE jobs and its effectiveness degrades as new data arrives.
OPTIMIZE
50%
Feature Comparison
Feature
Partitioning
Z-Ordering
Liquid Clustering
Best For
Low-cardinality columns (date, country)
High-cardinality, multi-column filters
Evolving access patterns & data skew
Flexibility
Rigid (Full table rewrite to change keys)
Rigid (Static snapshot, needs re-running)
Flexible (Change keys without rewrites)
Maintenance
Manual tuning to avoid over-partitioning
Manual, periodic OPTIMIZE ZORDER jobs
Incremental & automated OPTIMIZE jobs
Liquid Clustering: A Flexible, Adaptive Data Layout
Replaces both partitioning and Z-Ordering with a single, dynamic data organization strategy.
Change Keys Without Rewriting Data
Unlike partitioning, you can modify clustering keys as query patterns evolve without costly table rewrites.
Clustering Keys
No Rewrites
Automated & Intelligent Tuning
Can automatically select the best clustering keys by analyzing historical query workloads.
AUTO-TUNE
Time
NotebookLM

DND

FOR THE FUTURE

softserve