DELTA LAKE

Overview and Technical Deep Dive

databricks

Get started with Delta using Spark APIs

Instead of **parquet**...

```
CREATE TABLE
default.customer
USING parquet

LOCATION "/data/customer"
```

... simply say **delta**

```
CREATE TABLE
default.customer
USING delta

LOCATION "/data/customer"
```

Agenda

1. Overview

- Promise of the Data Lake
- Challenges
- Delta Lake Architecture
- Adoption
- Getting Started

2. Deep Dive

- Delta on Disk
- ACID
- Metadata Management
- State Management
- Time Travel
- CRUD
- Performance
- Pipelines

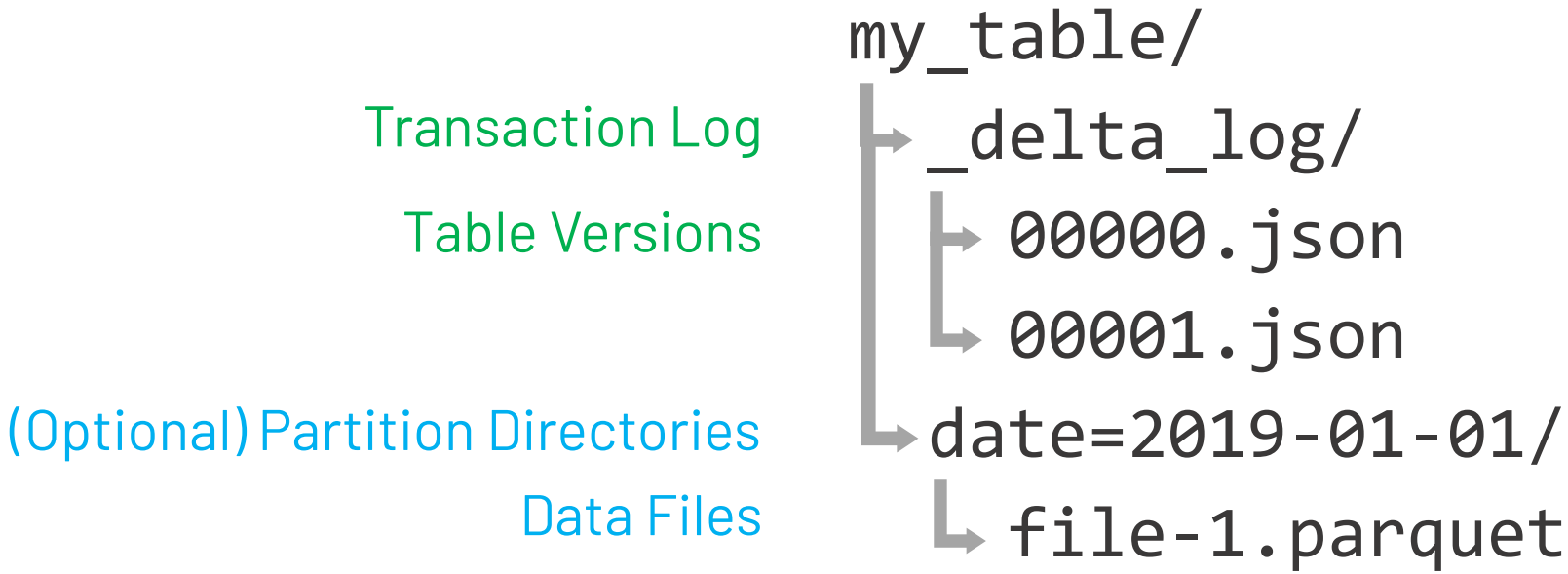
3. Next Steps

- Summary
- Questions
- Discussion

How does

work?

Delta on Disk



Version N = Version N-1 + Actions

Change Metadata – name, schema, partitioning, etc

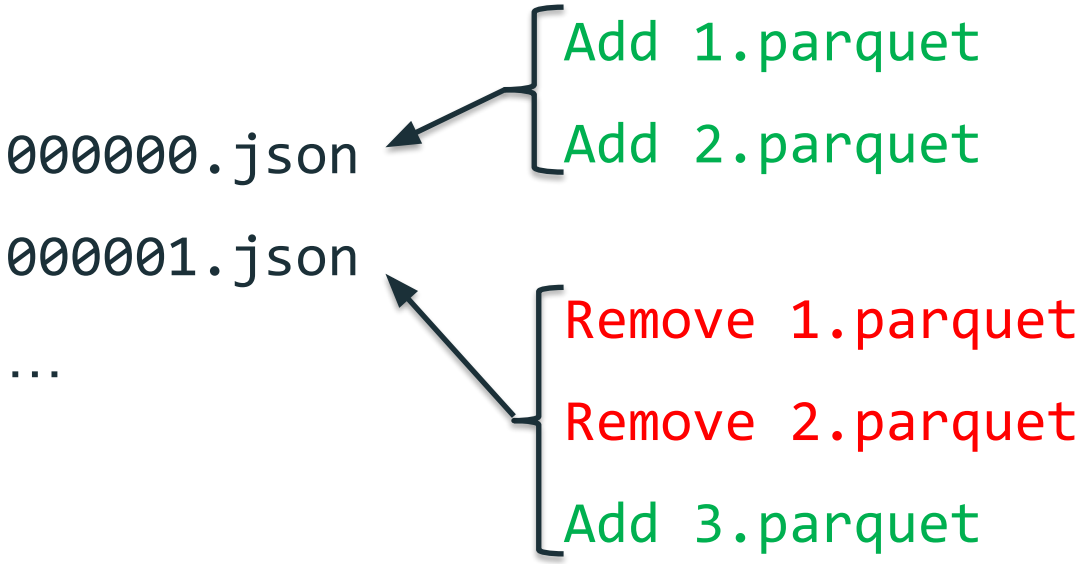
Add File – adds a file (with optional statistics)

Remove File – removes a file

Result: Current Metadata, List of Files

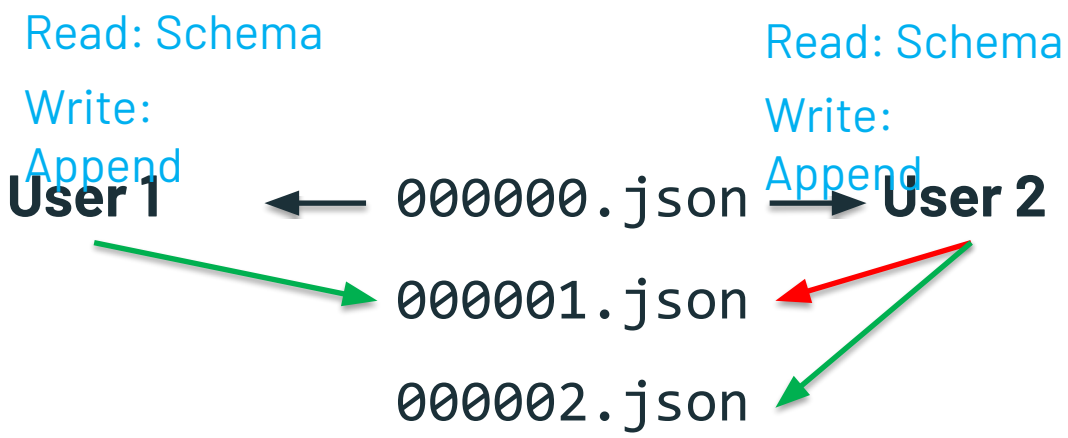
Implementing Atomicity

Changes to the table are stored as *ordered*, *atomic* units called commits



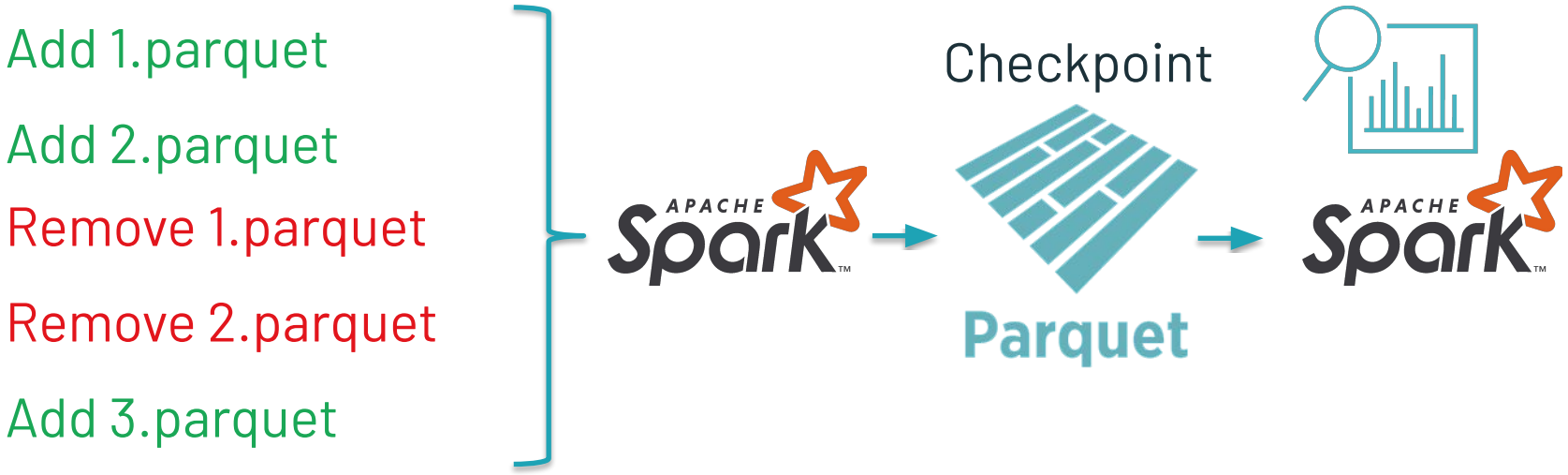
Solving conflicts optimistically

1. Record start version
2. Record reads/writes
3. Attempt commit
4. If someone else wins, check if anything you read has changed
5. Try again.



Handling massive metadata

Large tables can have millions of files in them! How do we scale the metadata? Use Spark for scaling!



Schema: Enforce & Evolve

- Parquet supports schema evolution, but will allow you to “evolve” it into an incompatible schema, breaking your table
- Delta enforces your schema by default, requiring an explicit config option to evolve it
- Delta prevents you from evolving it incompatibly and ensures your table always works
- Blog

Time Traveling by Version

```
SELECT * FROM my_table VERSION AS OF 1071;

SELECT * FROM my_table@v1071 -- no backticks to specify @
spark.read.option("versionAsOf", 1071).load("/some/path")

spark.read.load("/some/path@v1071")
```

Time Traveling by Timestamp

```
SELECT * FROM my_table TIMESTAMP AS OF '1492-10-28';

SELECT * FROM my_table@1492102800000000 -- yyyyMMddHHmmssSSS
spark.read.option("timestampAsOf", "1492-10-28").load("/some/path")

spark.read.load("/some/path@1492102800000000")
```

Data Skipping

- Simple, well-known I/O pruning technique
  - used by many DBMSes and Big Data systems
  - a.k.a Small Materialized Aggregates ZoneMaps ColumnStore Index
- Idea:
  - Track file-level stats like min & max
  - Leverage them to avoid scanning irrelevant files
- Example:

```
SELECT * FROM table WHERE col < 5
```

⬇

```
SELECT file_name FROM index
WHERE col_min < 5
```

file_name	col_min	col_max
file1.csv	6	8
file2.csv	3	10
file3.csv	1	4

Data Skipping with DELETE (GDPR, CCPA)

```
DELETE FROM my_table WHERE predicate
```

- If *predicate* is on partition columns only
  - Simple metadata operation
  - Simply create RemoveFile actions in the transaction log for all files in that partition

- If *predicate* is on data columns or contains a subquery
  - Leverage data skipping and figure out which files may match *predicate*
  - Run query that collects input_file_name for all rows that match *predicate*
  - Rewrite files without the rows that match the *predicate*

Data Skipping with DELETE (GDPR, CCPA)

```
DELETE FROM my_table WHERE predicate
```

- If *predicate* is on partition columns only
  - Simple metadata operation
  - Simply create RemoveFile actions in the transaction log for all files in that partition

- If *predicate* is on data columns or contains a subquery
  - Leverage data skipping and figure out which files may match *predicate*
  - Run query that collects input_file_name for all rows that match *predicate*
  - Rewrite files without the rows that match the *predicate*

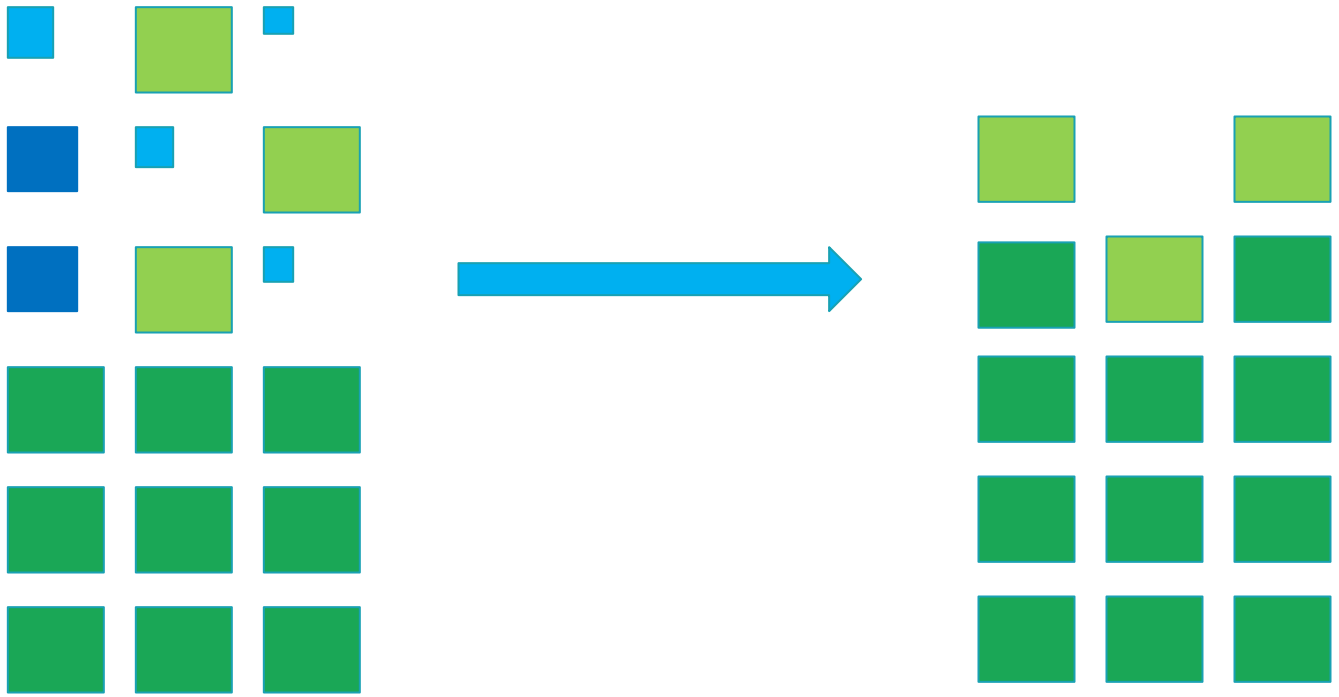
MERGE (Upsert) Basics

```
MERGE INTO my_table d USING my_view s ON condition
WHEN MATCHED AND d.status = 'CLOSED' THEN DELETE
WHEN MATCHED THEN UPDATE SET d.status = 'CLOSED'
WHEN NOT MATCHED THEN INSERT *
```

- Leverage data skipping (partition + stats) and figure out which files may match *predicate*
- Collect input_file_name for all rows that match *condition*. Ensure 1-1 mapping
- Rewrite files found above by updating the rows that match the *conditions*

OPTIMIZE: Compaction Built-in

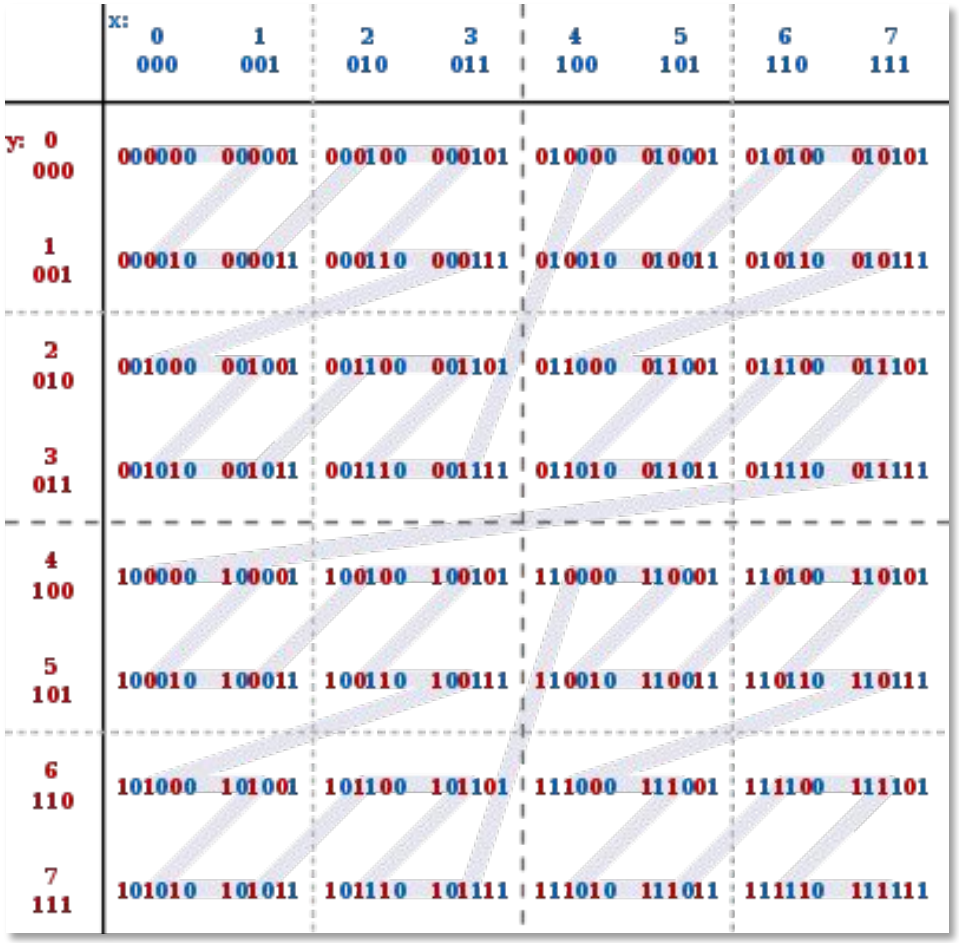
OPTIMIZE my_table



ZORDER: Multi-dimensional Clustering

OPTIMIZE my_table ZORDER BY (col1, col2)

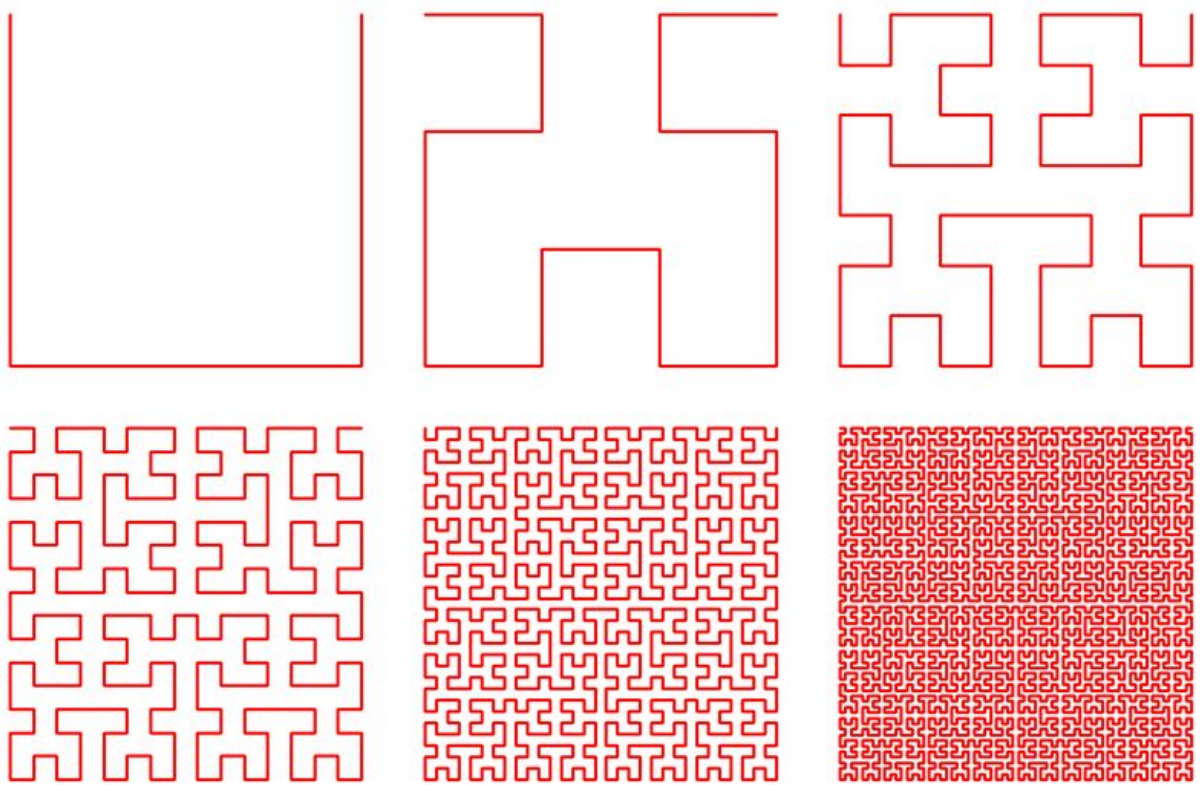
- Clusters data into Z-Cubes
- Each box is a separate file
- Effective on 3-5 columns



ZORDER v2: Hilbert Curve

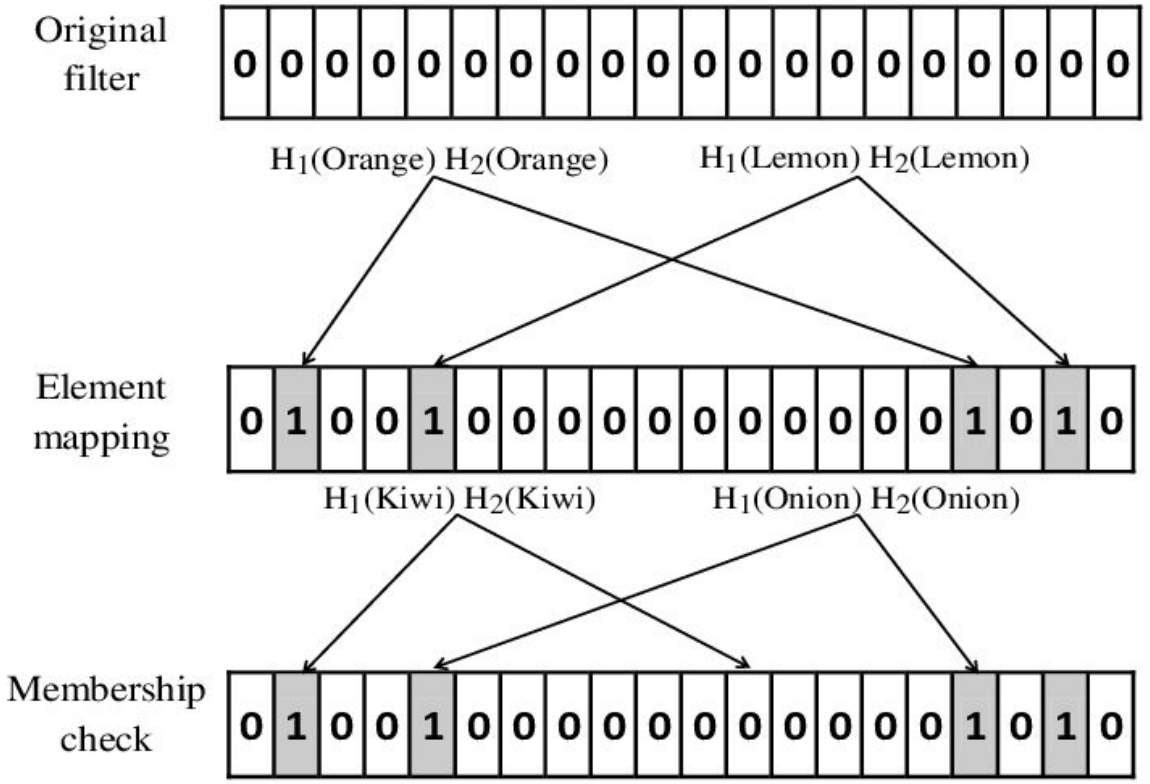
OPTIMIZE my_table ZORDER BY (col1, col2)

- DBR 7.6+
- Clusters data into H-Cubes
- Better clustering than ZORDER
- Effective on 5+ columns



Bloom Filters

- Useful for point lookups
- Data structure that returns either “definitely not in set” or “possibly in set”
- Additive with ZORDER (test ZORDER first, then add bloom filters)



Hadoop Creates Lots of Small Files

Traditional Writes

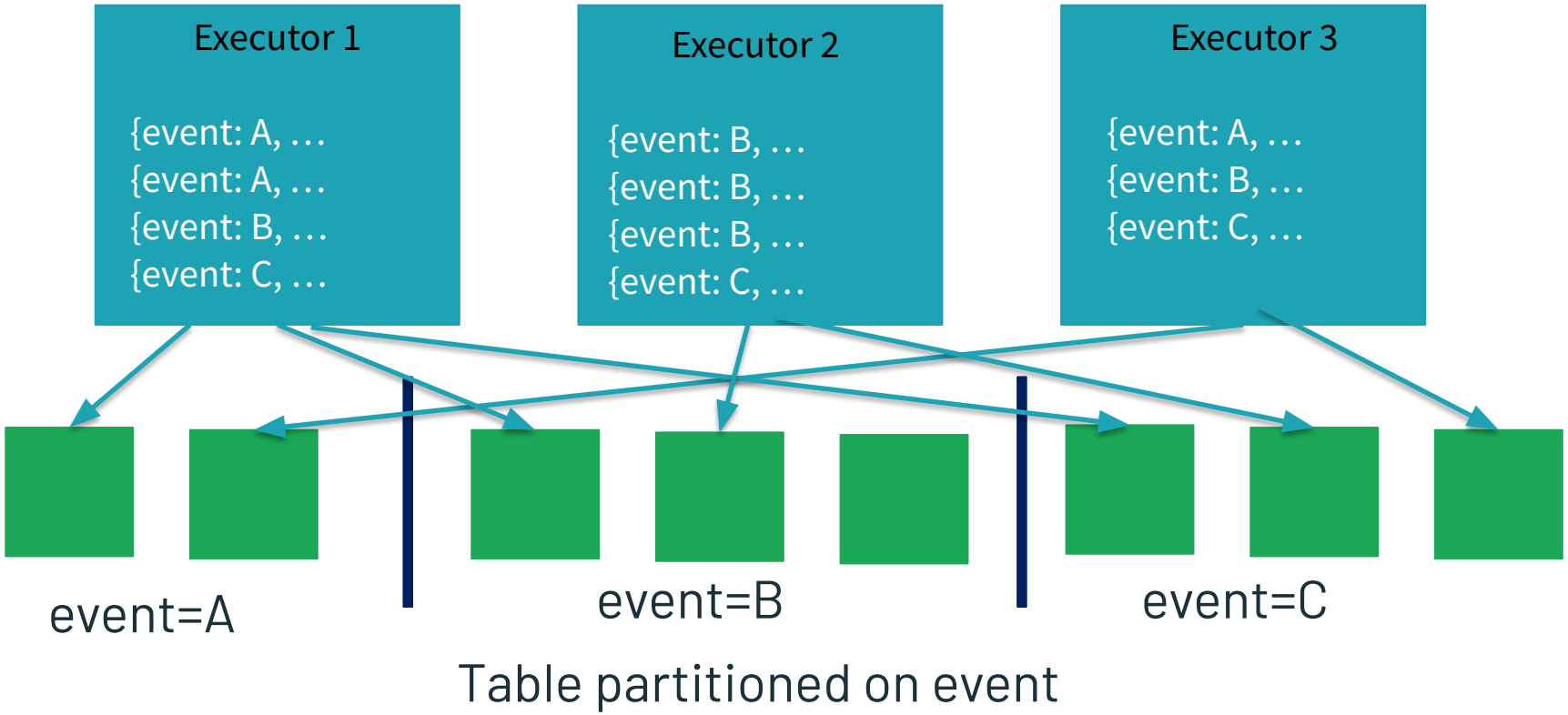
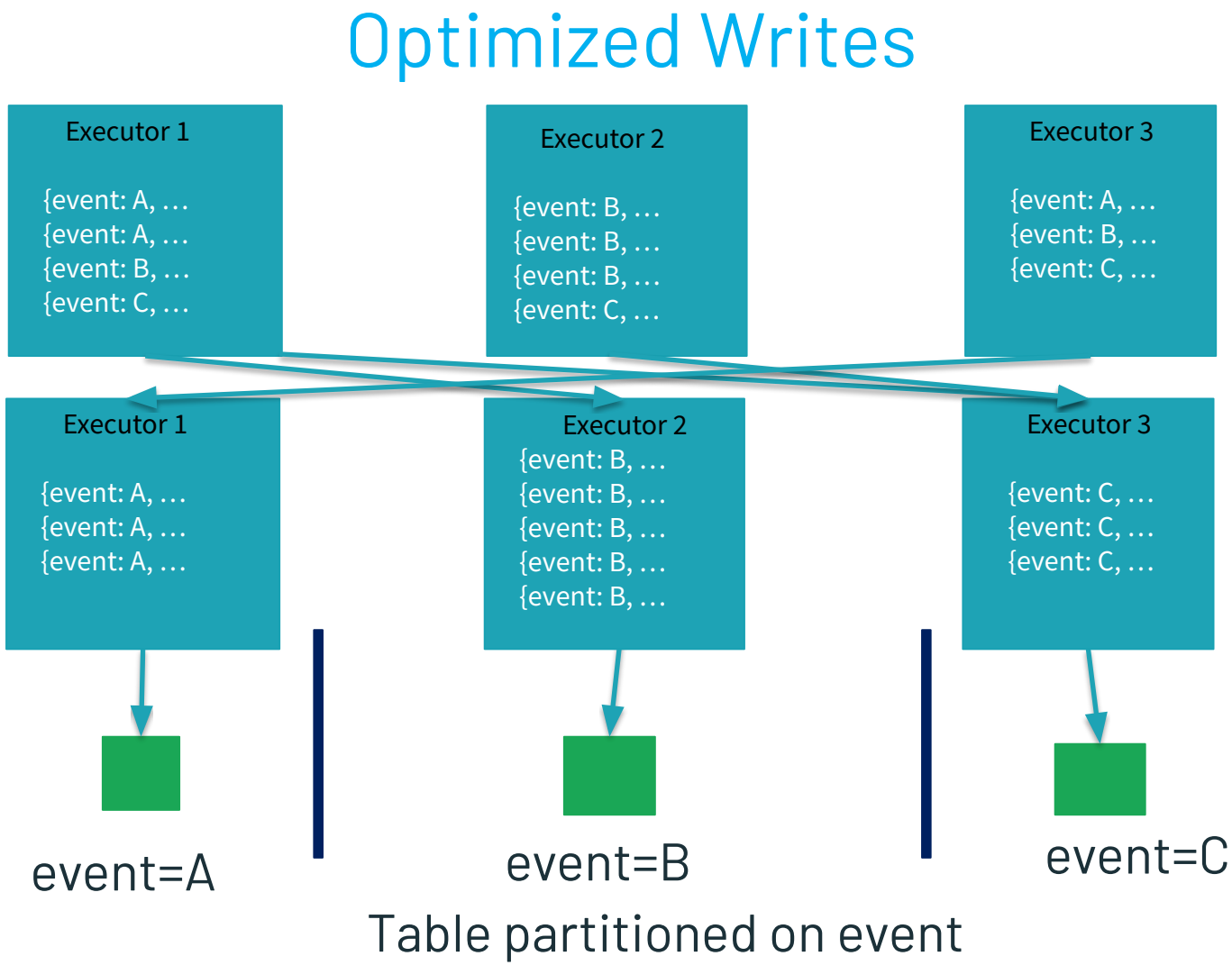


Table partitioned on event

Delta Manages File Size



Joining a Batch Delta Table to a Stream

- Every micro-batch, update the Delta table state
- If the version is equal to last batch, leverage map-side shuffle files of Delta table
- If version changed, perform computation, keep reference to shuffle files for next batch

Dynamic Data Skipping

- Common in star schema data models
- Joins involving fact and dimension tables where dimension tables are filtered
  - Dimension table should be small (broadcastable)
  - Both tables must be Z-ordered
- If the join column is a partition column, you get dynamic partition pruning
- If the join column is a data column, you get dynamic file skipping
  - If keys are few, we add WHERE key IN ('a1', 'a2', 'a3')
  - If keys are large, we add WHERE key <= max(a) AND key >= min(a)

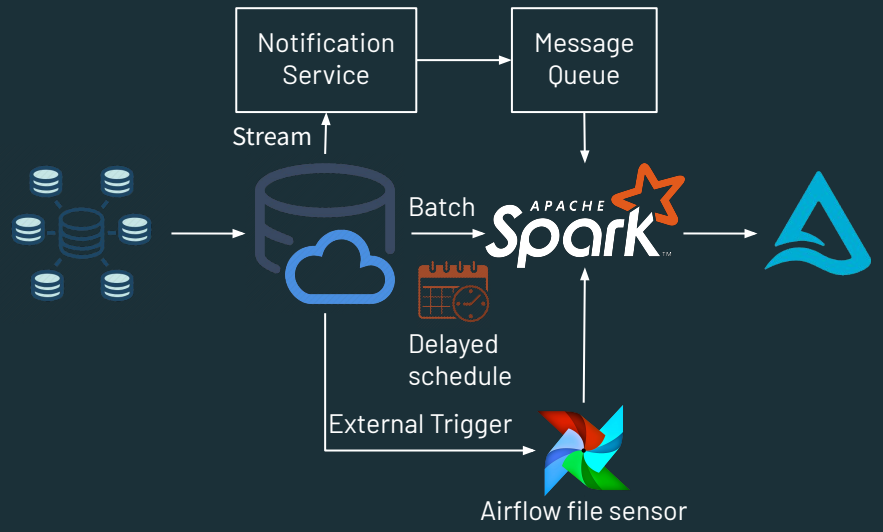
Delta Clones

- Shallow or Deep clones
- Clone a specific version (useful for Snapshots or ML)
- Clone for backups
- Stage changes to tables
- Blog

Databricks Ingest: Auto Loader

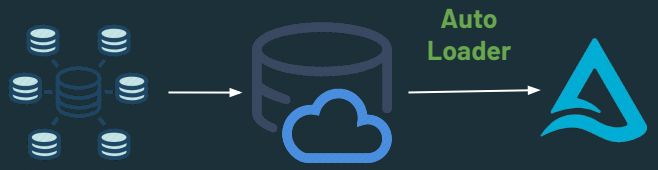
Load new data easily and efficiently as it arrives in cloud storage

Before



Gets too complicated for multiple jobs

After



- Pipe data from cloud storage into Delta Lake as it arrives
- "Set and forget" model eliminates complex setup

Launch Blog Post

databricks

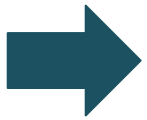
Change Data Feed: New in 8.2

Original Table (v1)

PK	B
A1	B1
A2	B2
A3	B3

Change data (Merged as v2)

PK	B
A2	Z2
A3	B3
A4	B4



Change Data Feed Output

PK	B	Change Type	Time	Version
A2	B2	Preimage	12:00:00	2
A2	Z2	Postimage	12:00:00	2
A3	B3	Delete	12:00:00	2
A4	B4	Insert	12:00:00	2

A1 record did not receive an update or delete.
So it will not be output by CDF.

Consuming the Delta Change Data Feed

Stream - micro batches

12:00 12:08 12:10:10

A2	B2	Preimage	12:00:00	2
A2	Z2	Postimage	12:00:00	2
A3	B3	Delete	12:00:00	2
A4	B4	Insert	12:00:00	2

A5	B5	Insert	12:08:00	3

A6	B6	Insert	12:09:00	4
A6	B6	Preimage	12:10:05	5
A6	Z6	Postimage	12:10:05	5

12:00

A5	B5	Insert	12:08:00	3
A6	B6	Insert	12:09:00	4

A6	B6	Preimage	12:10:05	5
A6	Z6	Postimage	12:10:05	5

12:10:00 12:20

Batch - every 10 mins

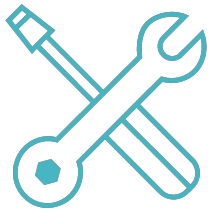
Stream-based Consumption

- Delta Change Feed is processed as each source commit completes
- Rapid source commits can result in multiple Delta versions being included in a single micro-batch

Batch Consumption

- Batches are constructed based in time-bound windows which may contain multiple Delta versions

In Progress: Merge Improvements



- Merge performance & Data Skipping effectiveness is affected by file size
- DBR 7.6 previewing a feature that dynamically tunes file size
- Configs:
  - spark.databricks.delta.tuneForRewrites = true
  - spark.databricks.delta.tuneForRewrites.autoDetect = true
- Merge can be more efficient
  - Low-Shuffle Merge preview in DBR 8.2+
  - Deletion Vectors

Normal Delta Merge

- Phase 1: Find the input files in target that are touched by the rows that satisfy the condition and verify that no two source rows match with the same target row [innerJoin]
- Phase 2: Read the touched files again and write new files with updated and/or inserted rows
- Phase 3: Use the Delta protocol to atomically remove the touched files and add the new files

Merge overview: phase 2 double click

Phase 2: Read the touched files again and write new files with updated and/or inserted rows.

The type of join can vary depending on the conditions of the merge:

- Insert only merge (e.g. no updates/deletes) → leftAntiJoin on the source to find the inserts
- Matched only clauses (e.g. when matched) → rightOuterJoin
- Else (e.g. you have updates, deletes, and inserts) → fullOuterJoin

Normal merge (merge on userId)

Changeset

userId	Name	type
1	Bilbo	Update
5	Maria	Insert
4	Fidel	Delete

Target file-001

userId	Name
1	Joe
2	Angela
3	Justin
4	Fidel
6	Zack
7	Valentino

Shuffle -> write to file-002

userId	Name
1	Bilbo
5	Maria
3	Justin
7	Valentino
6	Zack
2	Angela

Current latest version is file-002

Wasted shuffle and
Messed up ordering

Introducing lowShuffle

Simple premise: if we shuffle less data, we can do ops faster with less resources

1. Find the input files in target that are touched by the rows that satisfy the condition and verify that no two source rows match with the same target row [innerJoin]
2. leftOuter join between source and target to generate
   a. The new rows (updates and inserts)
   b. The file+rowids of the to-be-deleted rows (for deletes and updates)
3. Delta commit write the new rows to new files in this pass
4. Open the target files once again, and rewrite them by deleting the rowIds we found above (from 1b) (Delta commit)
   a. This should be done without any shuffle: just plain read-filter-write!

lowShuffle (merge on userId)

Changeset

userId	Name	type
1	Chris	Update
5	Maria	Insert
4	Fidel	Delete

Target file-001

userId	Name
~~1~~	~~Joe~~
2	Angela
3	Justin
~~4~~	~~Fidel~~
6	Zack
7	Valentino

Shuffle -> write to file-002

userId	Name
1	Chris
5	Maria

Shuffle -> write to file-003

userId	Name
2	Angela
3	Justin
6	Zack
7	Valentino

df.applyRowIndexFilter()

Current latest version is file-002 and file-003

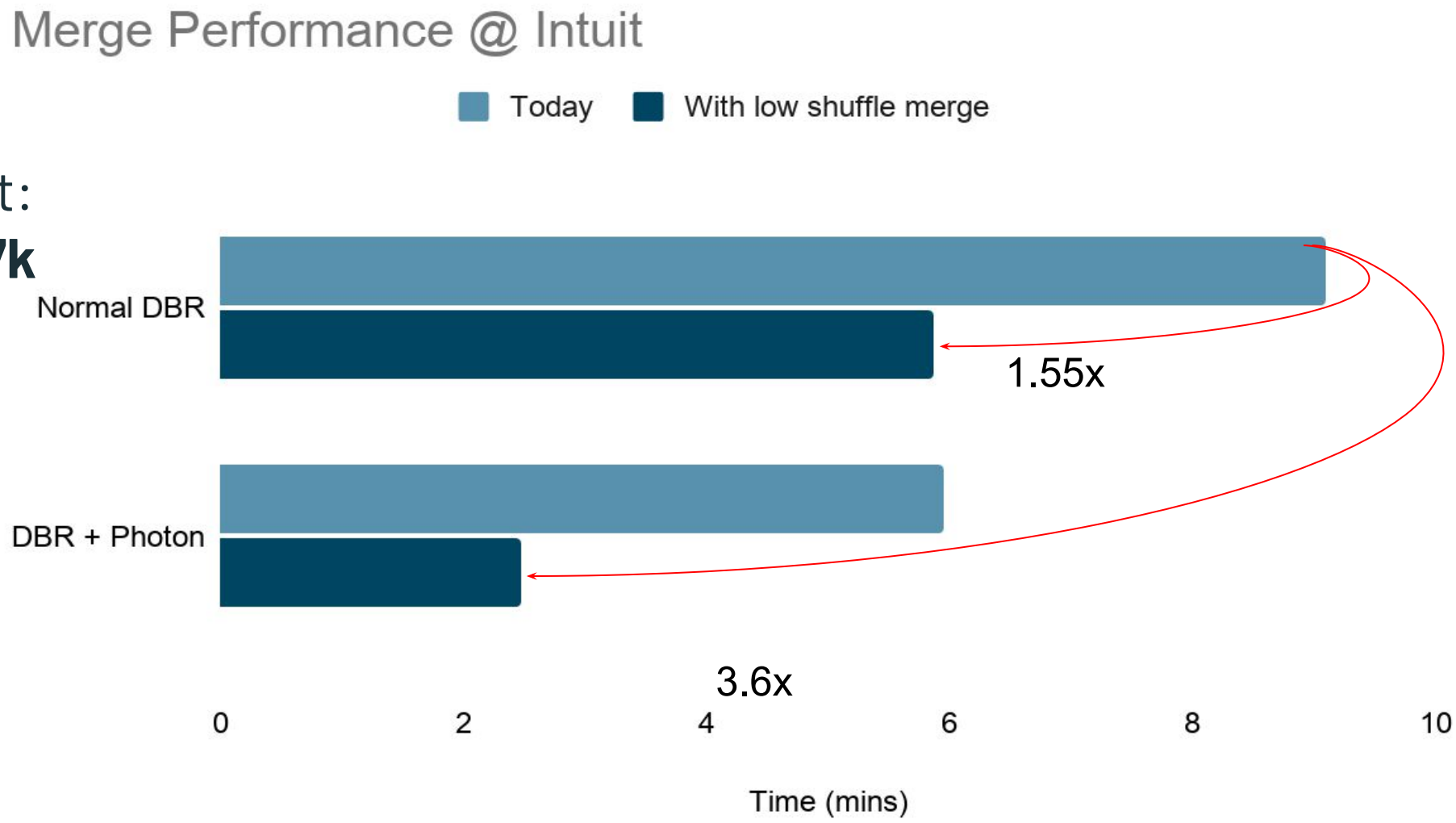
Testing results: 2-3x improvement in merge time

Real use case at Intuit: 1.5B row table with 27k row changeset: 64MB Delta files

Type	Duration	Speedup
DbrNormalMerge	9.1 minutes	(baseline)
DbrLowShuffleMerge	5.86 minutes	1.55x over baseline
PhotonNormalMerge	5.95 minutes	1.5x over baseline
PhotonLowShuffleMerge	2.48 minutes	3.6x over baseline

Testing results: 2-3x improvement in merge time

Real use case at Intuit:
1.5B row table with **27k** **row** changeset



Why it is lowShuffleMerge is great?

- Faster
- Cheaper
- Less shuffle = less messing up zOrder = less zOrder recomputation
- All of the cool kids are doing it

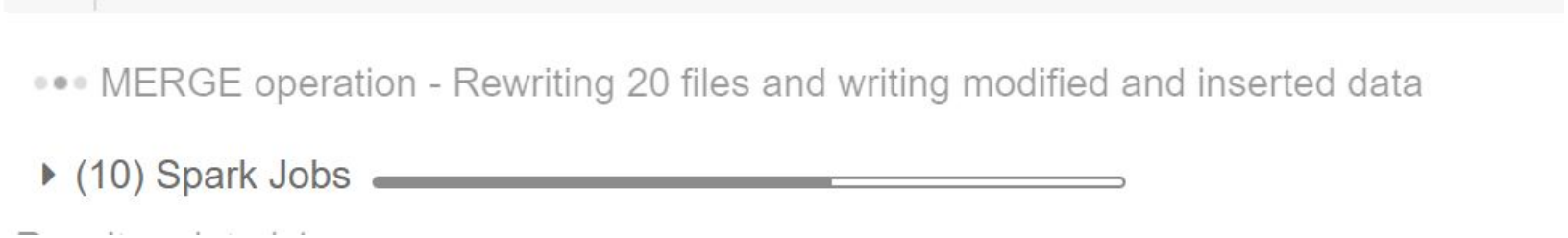
How do you know you use and how do you know it is working?

Use DBR 8.2+ *or* the most recent Photon image: enable by …

spark.databricks.delta.merge.enableLowShuffle true

Messages in the notebook include:

- MERGE operation - scanning files for matches
- MERGE operation - Rewriting ${filesToRewrite.size} files and writing modified and inserted data



Gotchas and more info

- CDC works as of DBR 9.1
- Feel free to check out more info and read the code
- No, LSM is not in OSS Delta Lake

What's next?

- autoFileTuning
- Deletion vectors!

Automatic File Size Tuning

- DBR 8.2+
  - delta.targetFileSize - easily set target file size
  - delta.tuneFileSizesForRewrites - detect if a table is receiving lots of MERGEs, then DBR will automatically switch to smaller file sizes to improve file pruning
- DBR 8.4+: We'll run these automatically
  - The target file size is based on the current size of the Delta table
  - Tables smaller than 2.56TB, the autotuned target file size is 256MB
  - Tables with a size between 2.56TB and 10TB, the target size will grow linearly from 256MB to 1GB
  - Tables larger than 10TB, the target file size is 1GB

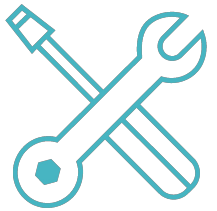
Automatic File Size Tuning

Table size	Target file size	Approximate number of files in table
10GB	256MB	40
1TB	256MB	4096
2.56TB	256MB	10240
3TB	307MB	12108
5TB	512MB	17339
7TB	716MB	20784
10TB	1GB	24437
20TB	1GB	34437
50TB	1GB	64437
100TB	1GB	114437

Deletion Vectors

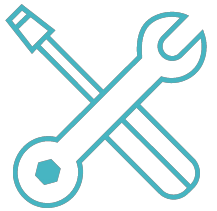
- Future format coming up
- Separates MERGE/UPDATEs into delete vector + inserts
- Easily Photonizable - data format is perfect for vectorization!
- Example:

SME Groups



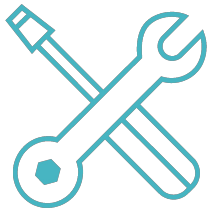
- #perf-delta-sme
- #perf-streaming-sme
- #help-photon

In Progress: Generated Columns



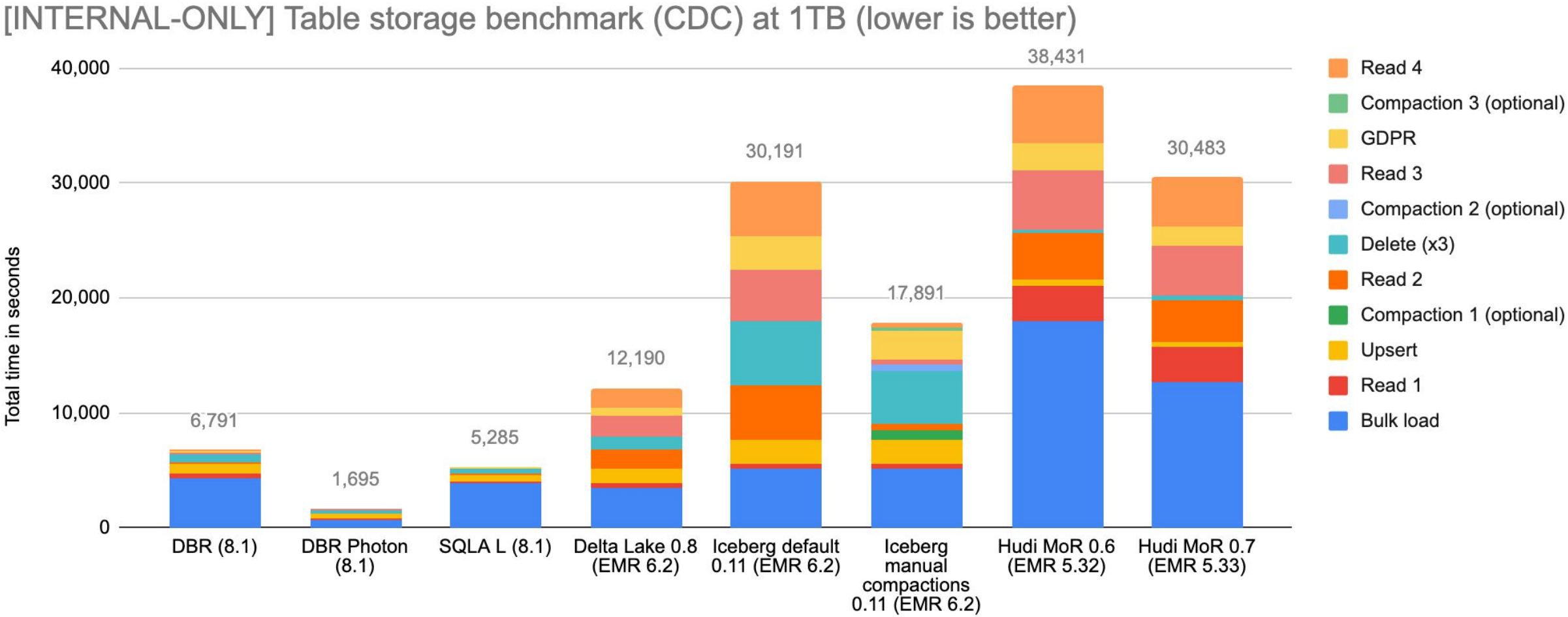
- Generated Columns let you create columns computed from others:
  - `date date GENERATED ALWAYS AS ( CAST(eventTime AS date) )`
- Simplifies timestamp <> date relationship management
- Query engine is aware of relationship
  - If the table is partitioned by date, but date is generated from timestamp, queries on timestamps will also do partition pruning
- Need bucketing - #perf-delta-sme
- Preview Doc

Competitors: Apache Hudi & Iceberg

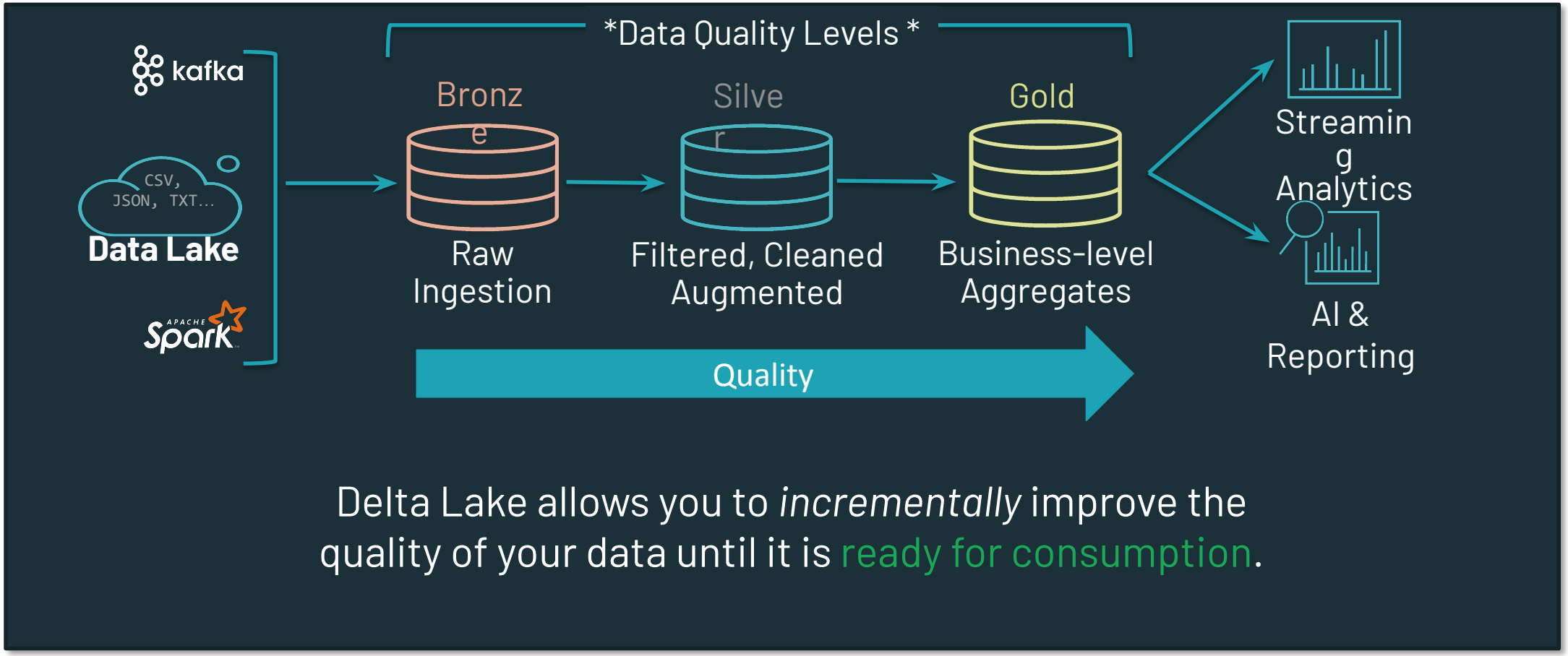


- Hudi
  - Originally from Uber & Designed for Updates against Primary Key only
  - Good integration with AWS services
  - Bad performance, bad APIs, and lack of feature completeness
  - Benchmarks & Competitive Positioning
  - Onehouse.ai
- Iceberg
  - Originally from Netflix & Designed for slowly-changing Petabyte-scale data lakes
  - Similar in concept to Delta but not originally designed for Streaming
  - Contributors from Apple, Adobe, LinkedIn, Expedia, Dremio, Starburst, and AWS
  - Better perf than Hudi, still slower than Delta, & requires lots of manual tuning
  - Lack of feature completeness
  - Benchmarks and Competitive Positioning
  - Tabular.io

Competitors: Apache Hudi & Iceberg



Summary, Questions, Discussion...



Agenda

1. Overview

- Promise of the Data Lake
- Challenges
- Delta Lake Architecture
- Adoption
- Getting Started

2. Deep Dive

- Delta on Disk
- ACID
- Metadata Management
- State Management
- Time Travel
- CRUD
- Performance
- Pipelines

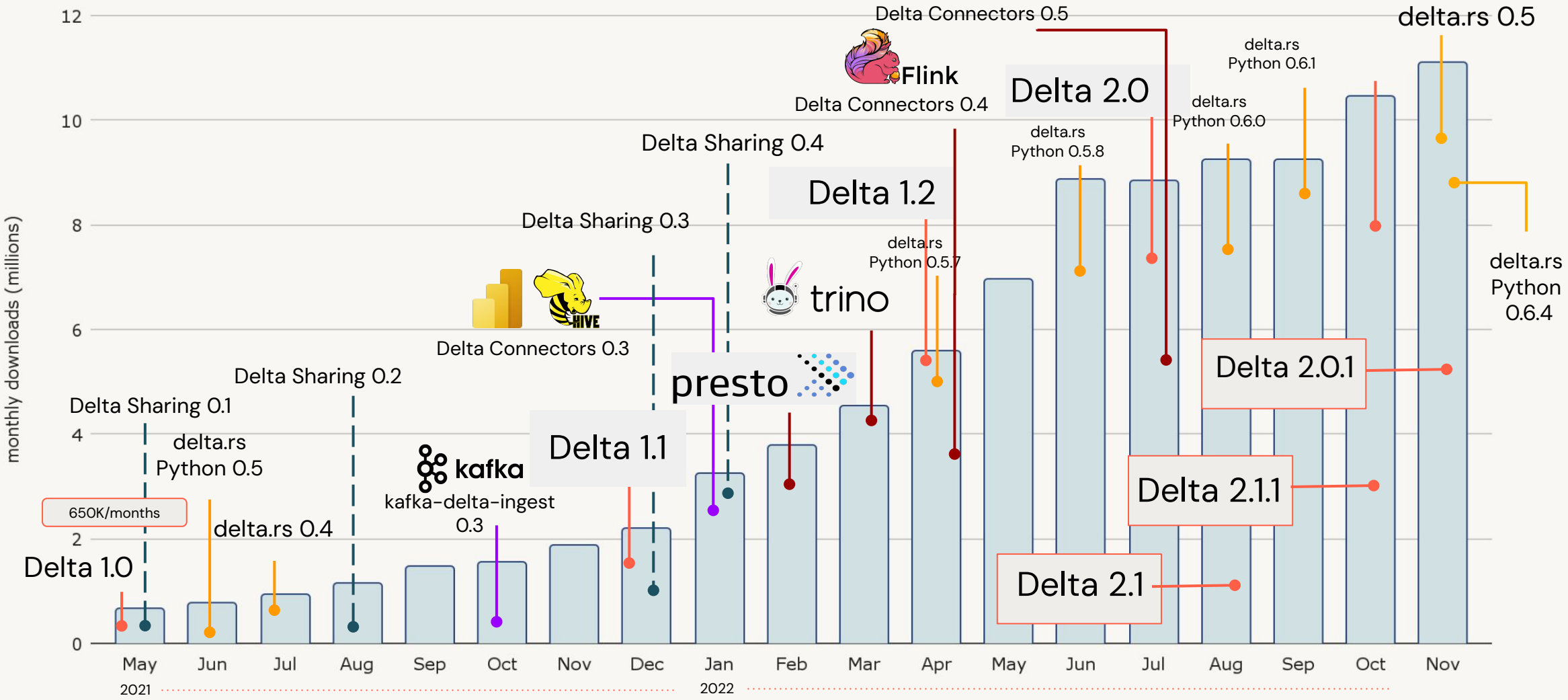
3. Next Steps

- Summary
- Questions
- Discussion

Delta Lake 2023 H1 Roadmap

The most widely used lakehouse format in the world

11.1M downloads/month



Key Differentiators

Performance, Community, Reliability

1.7+ Exabytes *processed* / day

663% Increase in contributor strength over last three years

7K+ Companies in Production

Recent Releases 2022

Delta 1.0.1

- Improvements to DeltaTableBuilder API
- Perf improvements

Delta 1.2.1

- Make the GCS LogStore configuration simpler
- Fix for S3 multi-cluster mode configuration

Delta Lake 2.1.0

- Spark 3.3
- Support for [TIMESTAMP | VERSION] AS OF in SQL
- Support for Trigger.AvailableNow when streaming from a Delta table
- Support for SHOW COLUMNS
- Support for DESCRIBE DETAIL in the Scala and Python DeltaTable API
- Optimize performance improvements

Delta 2.0.2

- Spark 3.2
- Support idempotent writes for DML operations
- Record VACUUM operation in the transaction log

Apr 2022 | Jul 2022 | Aug 2022 | Dec 2022 | Jan 2023

Delta 1.2.0

- Multi-cluster write in Delta Lake tables stored in S3
- Compacting small files (optimize) into larger files
- Data skipping using column statistics
- Restoring a Delta table to an earlier version
- Column renaming
- Arbitrary characters in column names
- Support for Google Cloud Storage

Delta Lake 2.0.0

- Spark 3.2
- Change Data Feed
- ZORDER
- Idempotent writes to Delta tables
- Dropping Columns
- Dynamic Partition overwrite
- Multi-part checkpoints

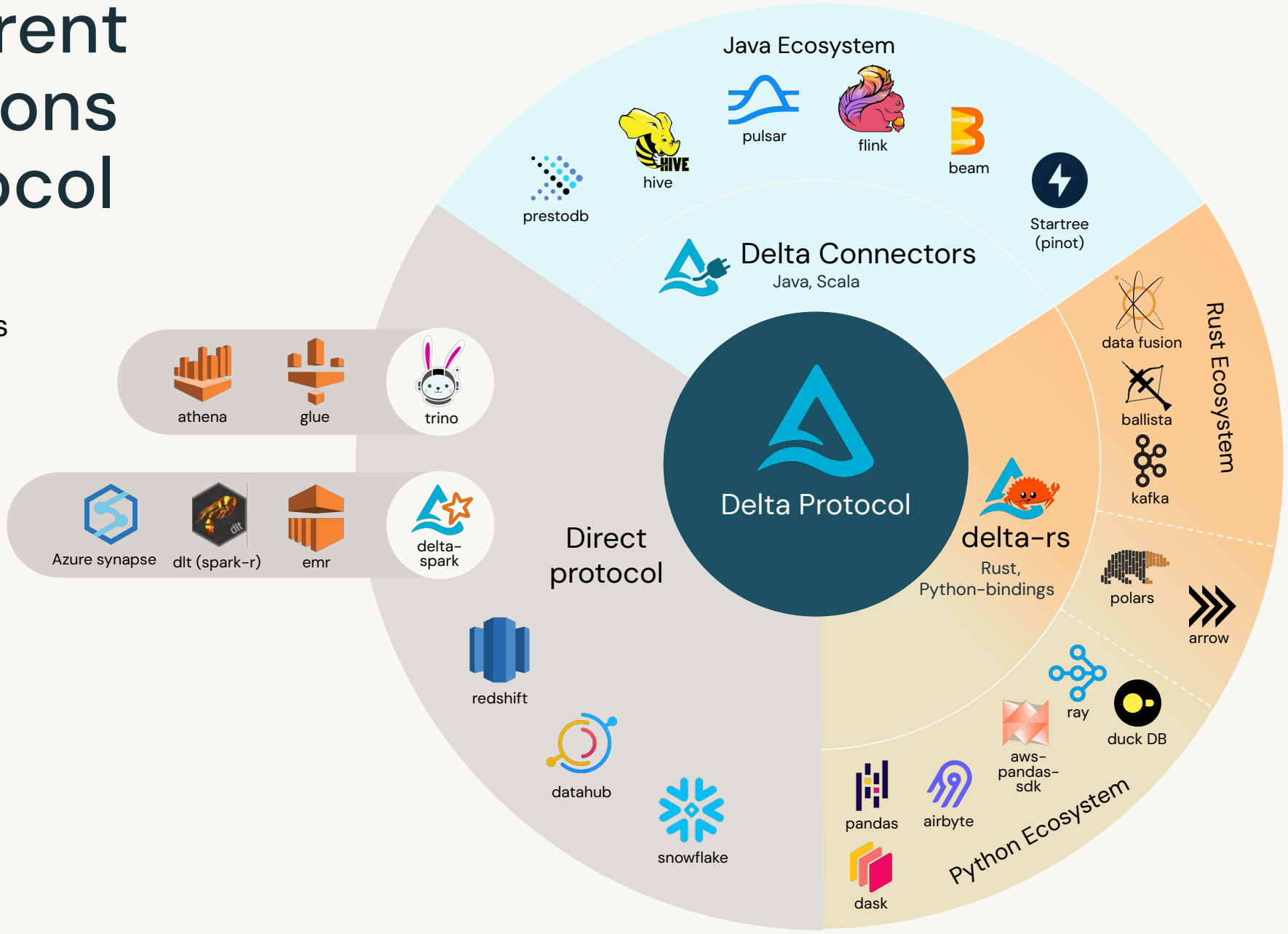
Delta 2.2.0

- Spark 3.3
- Improve query perf by LIMIT pushdown
- Aggregate pushdown into Delta scan for SELECT COUNT(*)
- Support for collecting file level statistics as part of the CONVERT TO DELTA command
- Improve performance of the DELETE command through column pruning
- Remove the restrictions for using Delta tables with column mapping in certain Streaming + CDF cases.

Multiple different implementations of Delta protocol

delta connectors and delta-rs have their own implementations of the Delta protocol that have to be maintained for the Java, Rust, and Python ecosystems.

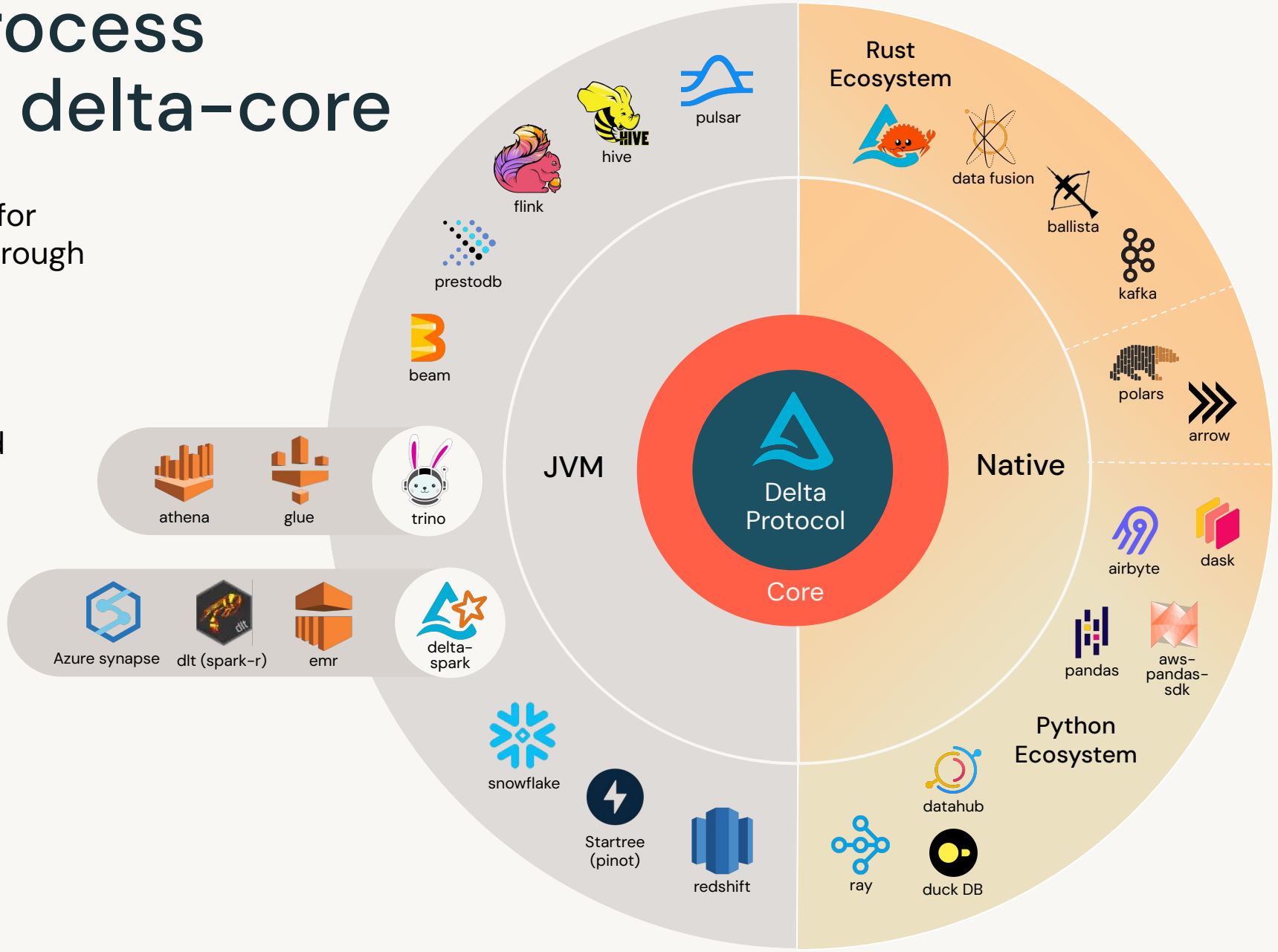
Trino, Snowflake, Redshift, Spark and others maintain their own implementation of the Delta protocol as well and none are quite in sync.



Simplify the process by introducing delta-core

delta-core maintains all the logic for working with the delta protocol through the transaction log wherever it is.

Clients need only maintain their connections to delta-core thus simplifying their development and maintenance



Delta Lake Roadmap

Delta Lake project's roadmap is publicly available in this GitHub issue

Key upcoming features

Support for Spark 3.4

Flink SQL support

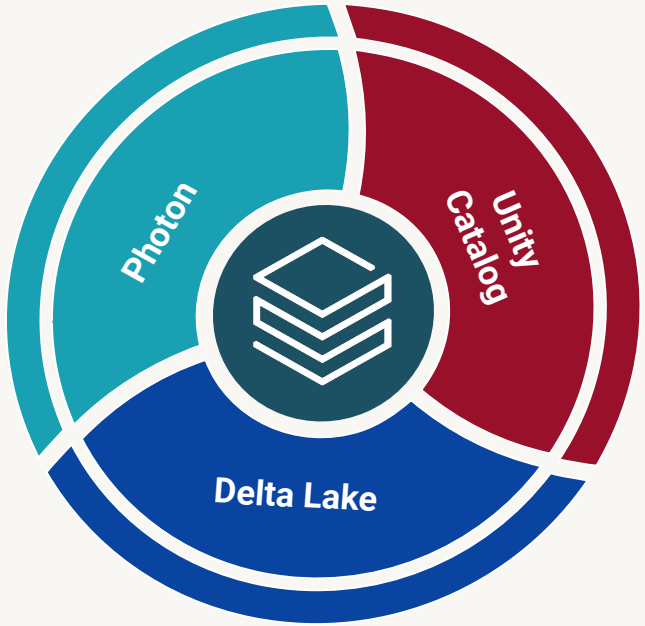
New Delta connector library

VISIT: https://delta.io/roadmap/

Building the Best Lakehouse on Linux Foundation Delta Lake

GOALS

1. Provide the best price-performance **out of the box** and as the workload **evolves**
2. Improve performance for **low latency** dashboards and BI use cases



PHOTON + UC + DELTA

1. Optimize Photon's native reader and write to Delta
2. Use UC to handle metadata to provide low latency query experience

Auto Tune for UC Managed Tables

Complete table management for best performance out of the box

Problem

For optimal performance tables require periodic tuning operations.

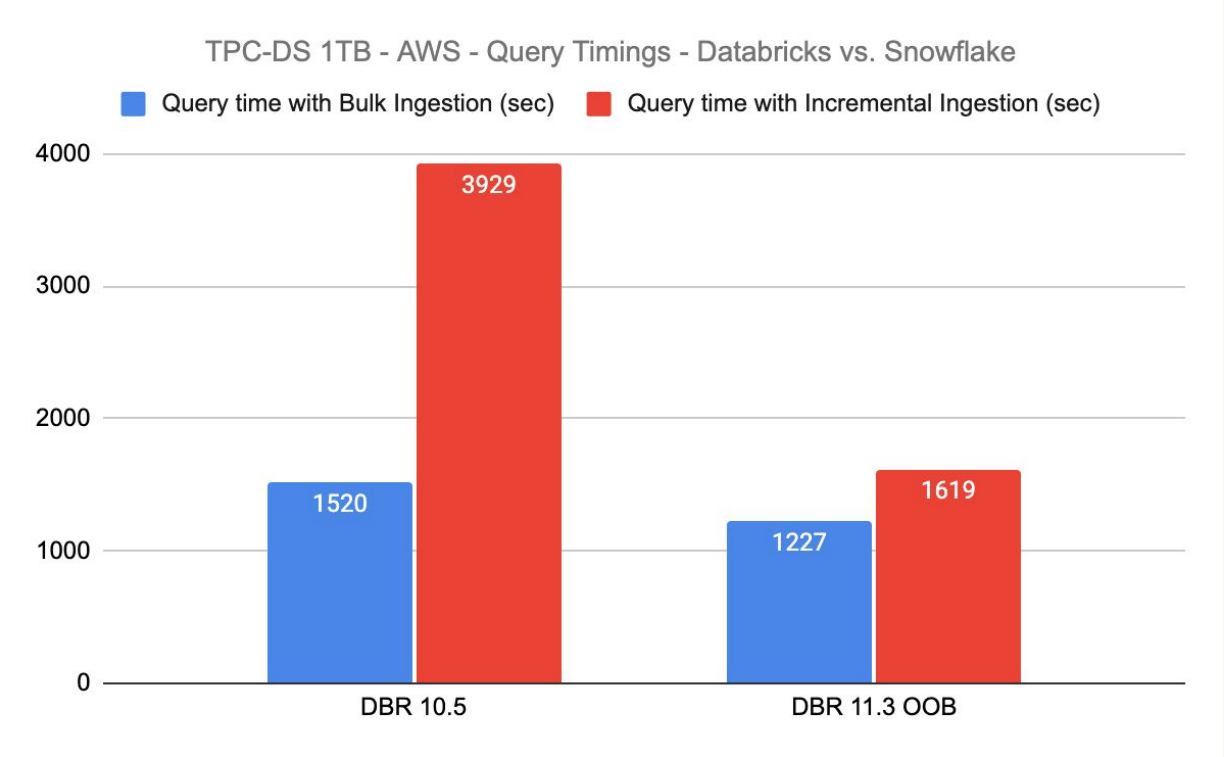
Solution

Unity Catalog will auto tune tables.

- For tables under 1TB, no tuning required (4x faster)
- Auto Tune takes care of the data layout

HOW?

- Optimized Writes for Unpartitioned Tables
- Asynchronous Auto Compaction
- Ingestion Time Clustering



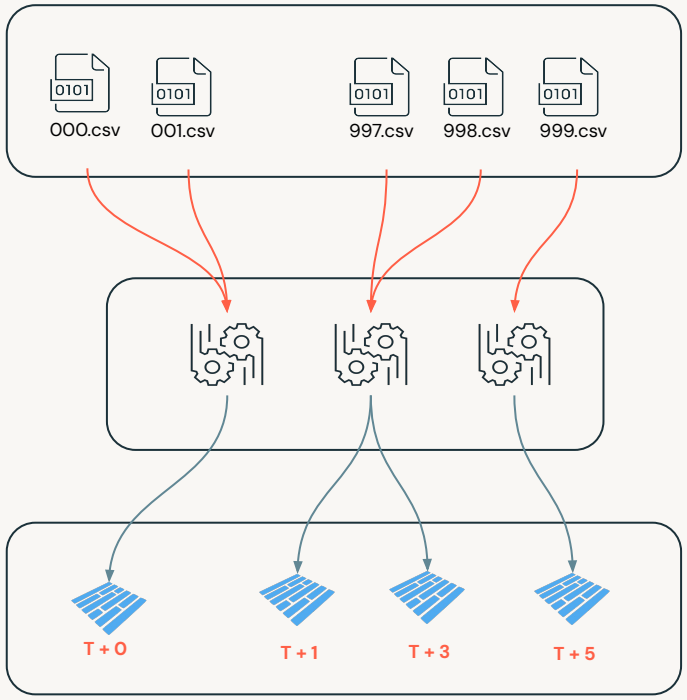
BEFORE AUTO TUNE **AFTER AUTO TUNE**

	Azure
Status	GA In Q4

Ingestion Time Clustering

Out of the box data skipping with no partitioning or z-order required

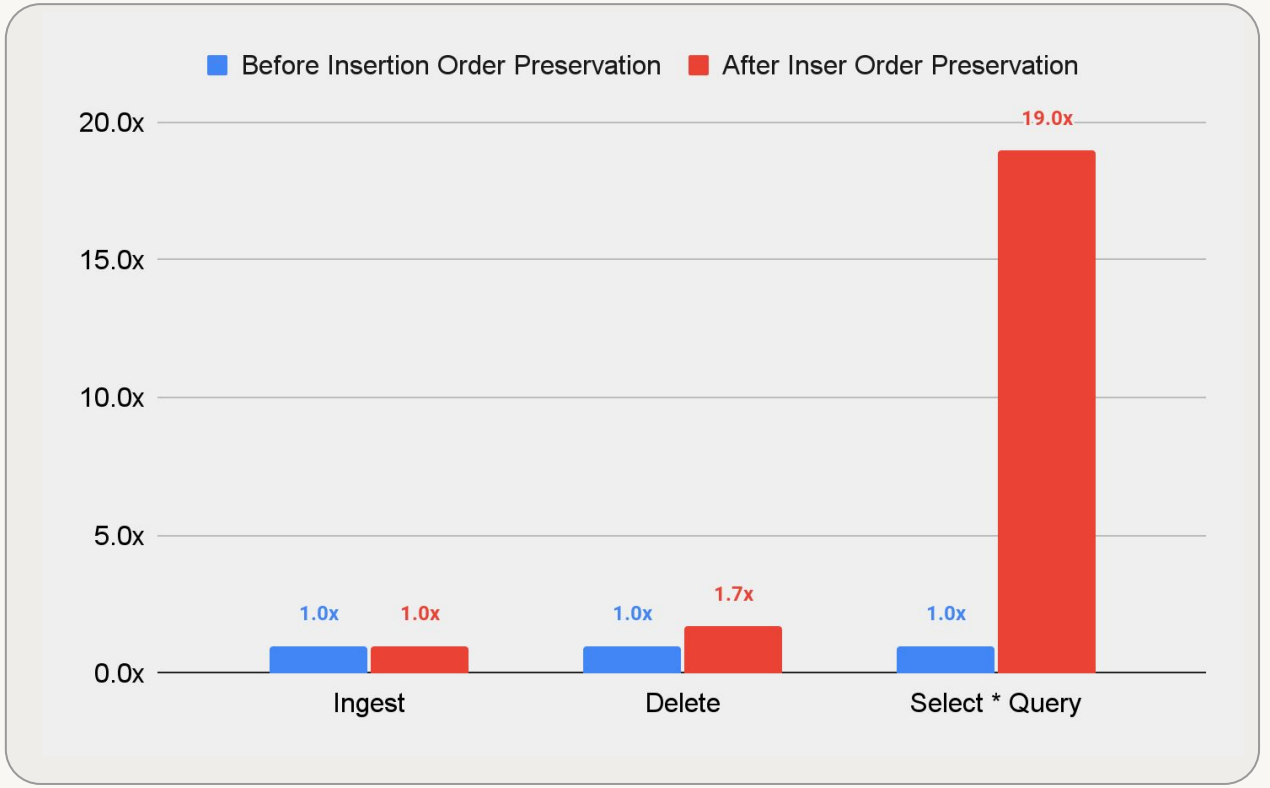
Preserves natural clustering across all Delta operations (DML, ingestion, maintenance)



	Azure
Status	GA

19x better query performance out of the box

Higher is better



Setup: Store Sales table with data naturally ordered by date

Faster Updates with Photon + Deletion Vectors

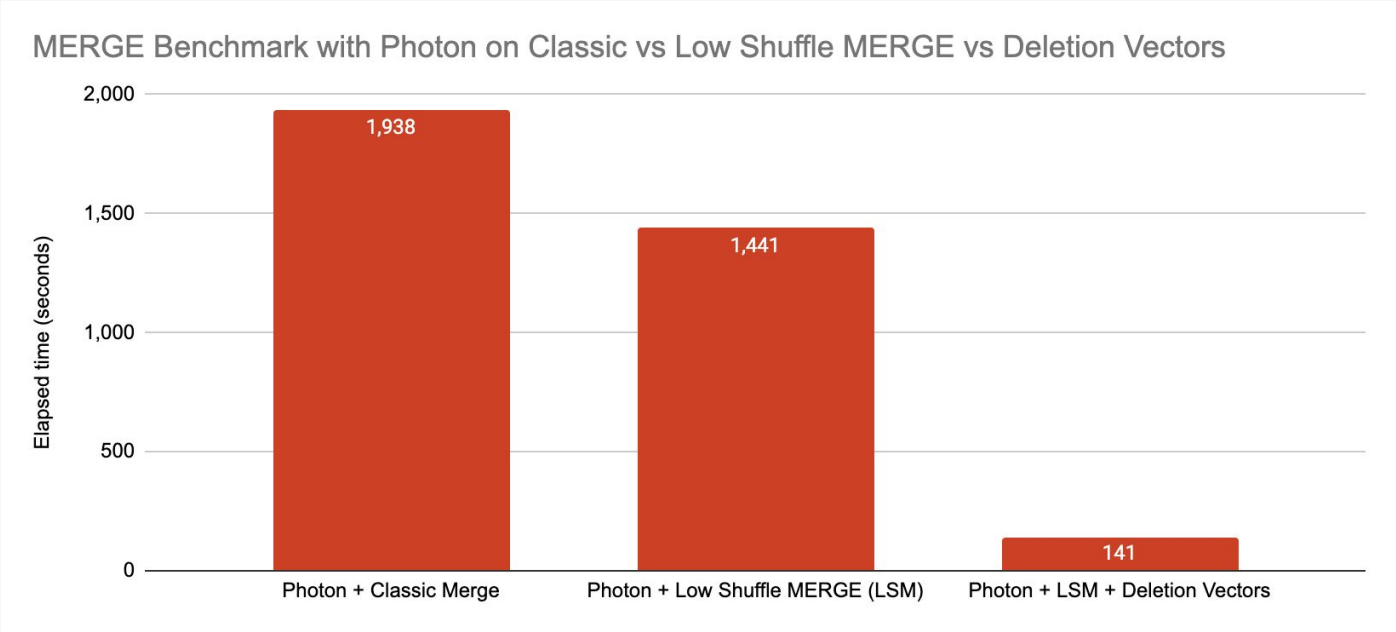
Up to 10x speed-up in MERGE, UPDATE & DELETE queries w/ Photon

Problem

- Updates to tables are expensive because of rewrites

Solution

- Using Deletion Vectors, Photon avoids the need to rewrite files during MERGE, UPDATE & DELETE, **speeding up updates by up to 10x.**



Azure
Q1 - Gated Public Preview

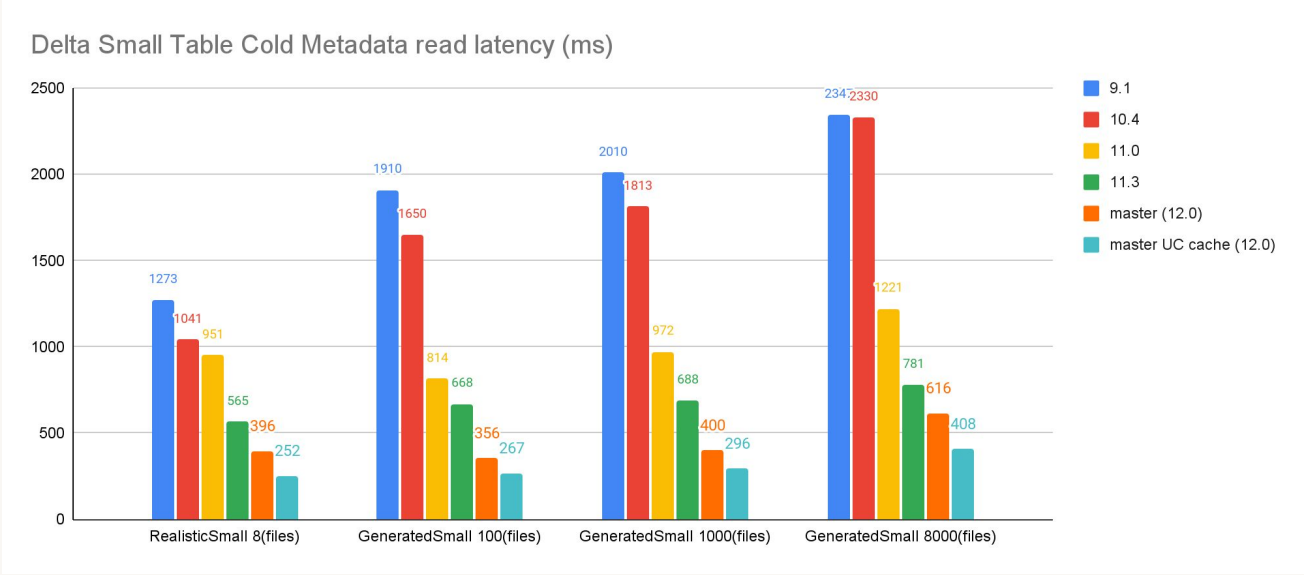
Making Delta Log Faster to Write and Read

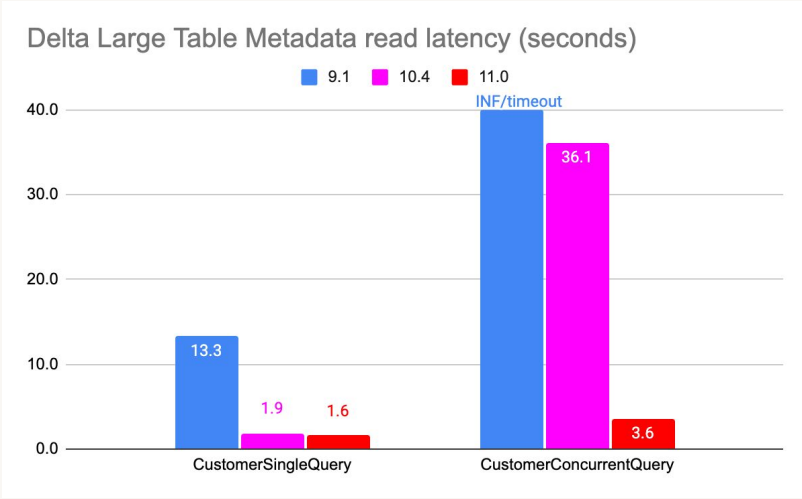
Benefitting all queries by spending less time to fetch metadata from cloud storage

Problem

On average, metadata processing accounts for 40% of query time for both large and small tables

- **Metadata Read Path Optimizations:** Combination of many small improvements when reading metadata
- **UC Metadata Caching:** Improved query perf for small tables
- **Incremental Commit:** Commit costs O(size of commit) instead of O(size of table)





Multi-Statement Controls - Atomic Publish

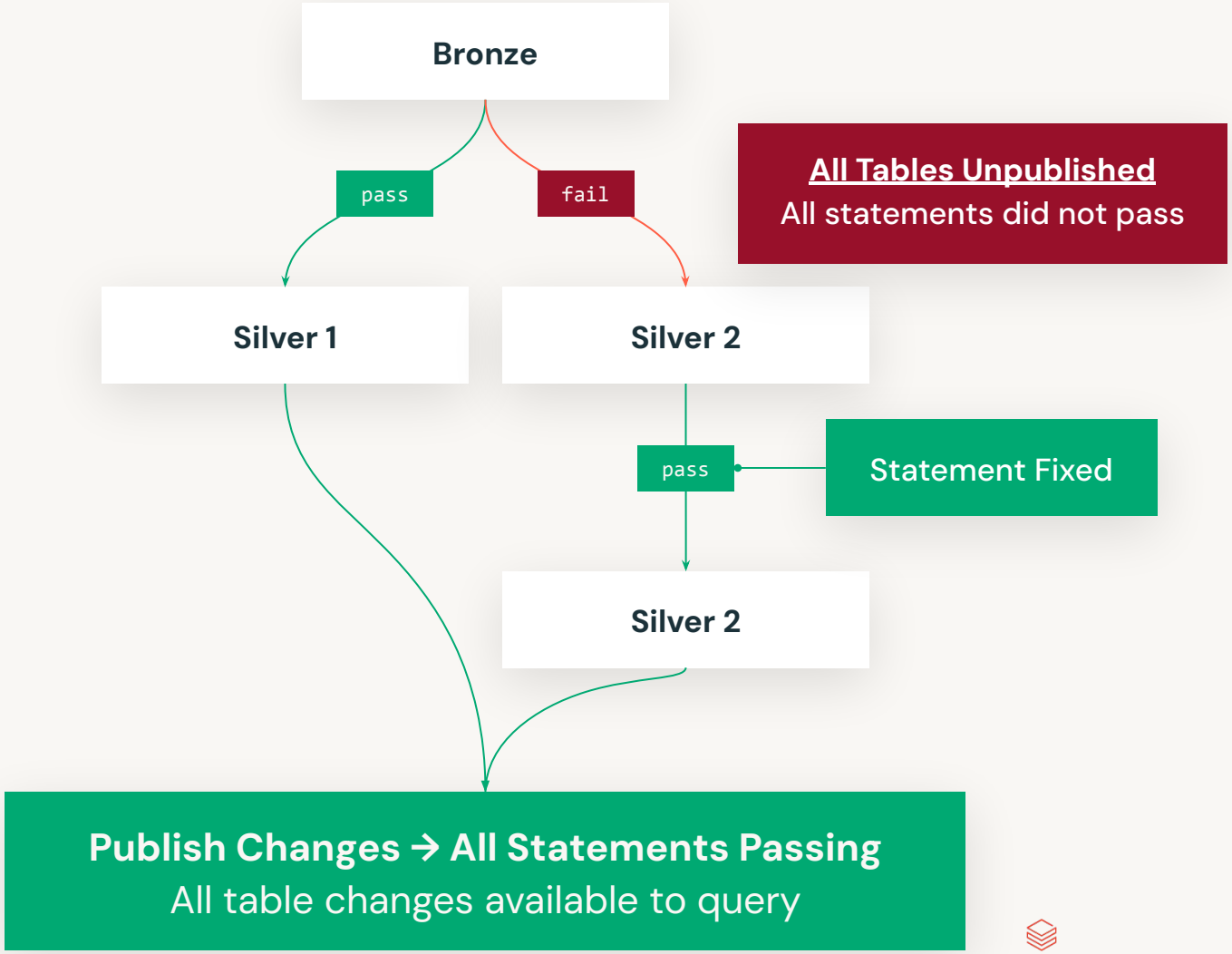
Ensure consistency across dependent tables using atomic publish

Problem

- Customers cannot ensure downstream read consistency across multiple related tables.

Solution

- Using Atomic Publish, data engineers can **granularly control when changes are query-able** by their downstream users.
- Can **rollback** or **fix-forward** failed statements.



Lifecycle Archival Support

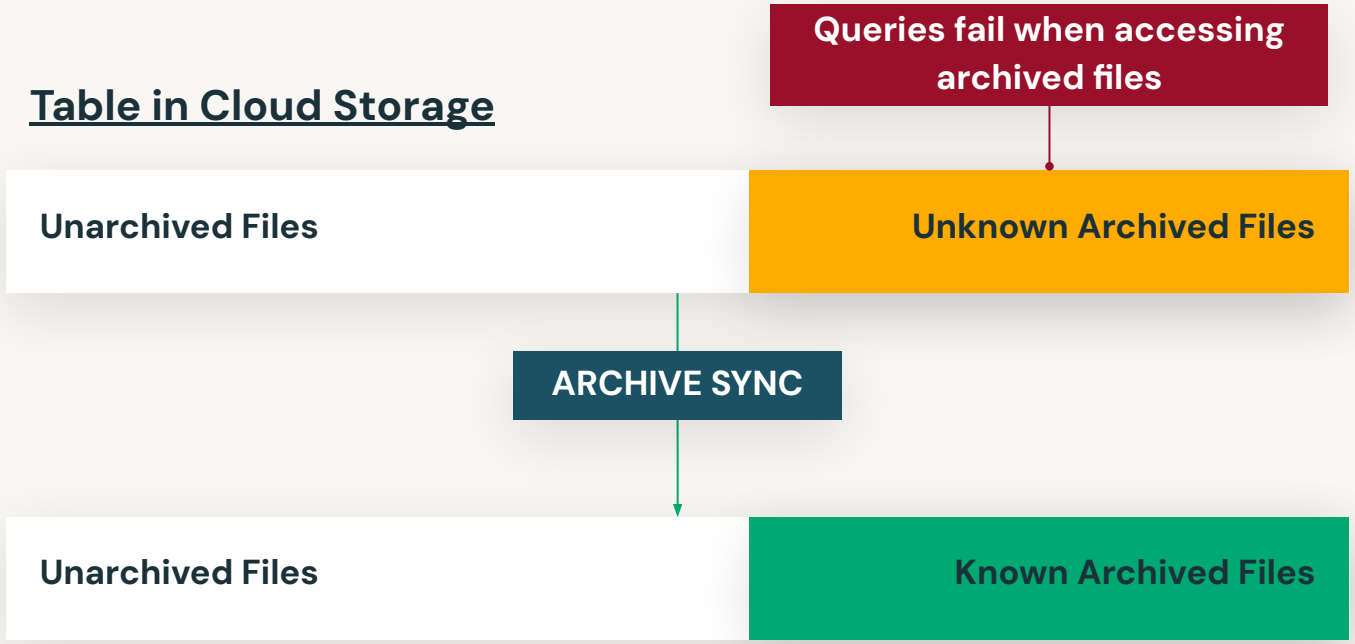
Store Delta files for cheaper in archive

Problem

- Delta fails with exceptions when users attempt to query tables with files in Archive.

Solution

- Provide a command to mark files in Delta tables as in Archive.
- Ignore marked files for SELECT * queries and maintenance operations.
- Fail queries early when attempting to access files in cloud storage archive.



✅ Ignores archived files for sampled queries

✅ Ignores archived files for maintenance

✅ No compute wasted - fails early on archive access

✅ Archival-aware error messaging

DFP/DPP for MERGE w/Broadcast Joins

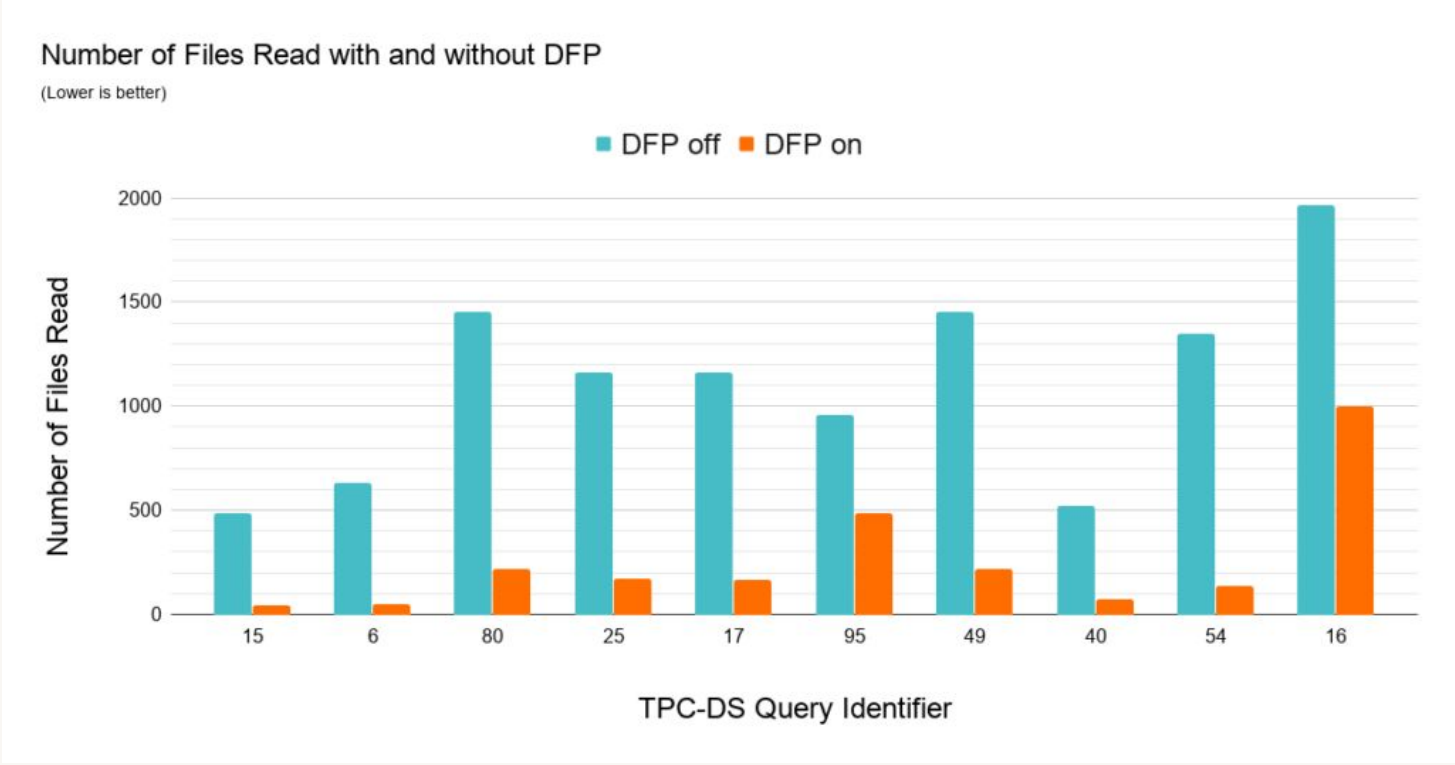
Automated pruning & faster MERGE execution for small UPSERT scenarios

Problem

- Customers have to do static pruning for optimal data skipping when merging into large target tables.

Solution

- Using file metadata and statistics available at runtime, we can automatically skip all irrelevant files/partitions for the merge, significantly improving data skipping.



Substantially Faster Clustering with Photon

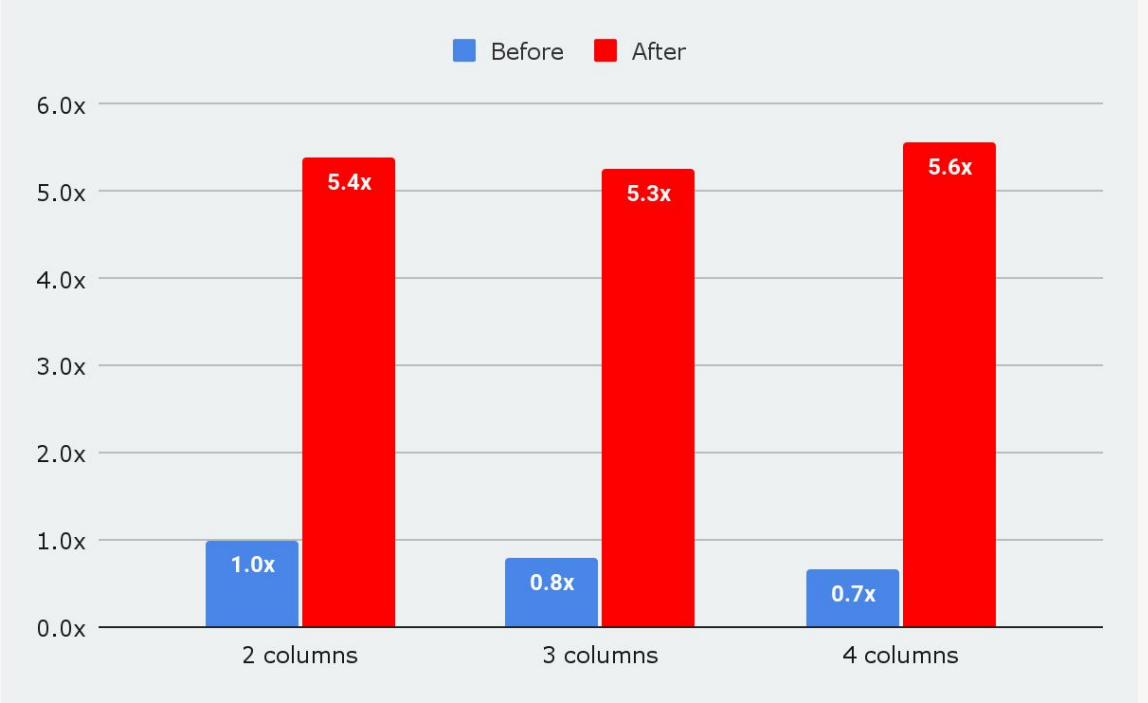
Cheaper and faster clustering for better read query performance with data skipping

ZORDER colocates columns in the same files

- Great read query performance when using same columns as predicates or JOINs
- Supports sorting by multiple columns

5x performance speed up when running ZORDER

Higher is better



Setup: 100 files, 83GB data, sorting on 1-3 columns