DATA+AI
SUMMIT 2022

Delta Lake 2.0

And the ever-growing ecosystem

Tathagata Das

Staff Engineer - Databricks

Denny Lee

Senior Staff Developer Advocate - Databricks

ORGANIZED BY databricks

DELTA LAKE

TODAY...

Delta Lake 2.0.0 is in preview

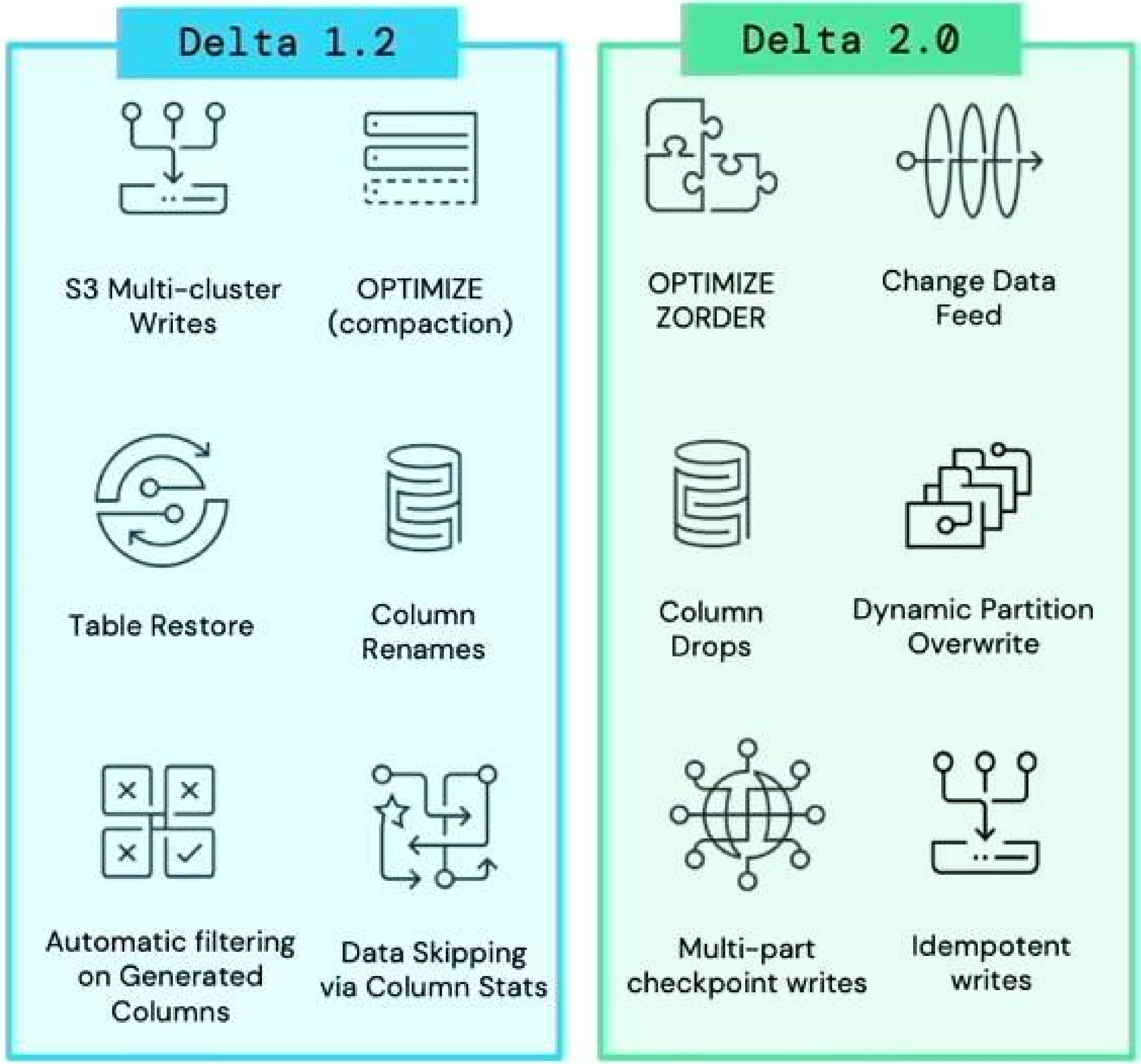
See https://delta.io for details on how to try it out

What is in Delta 2.0.0?

A lot of new features released in the last 1 year

This talk will focus on a few awesome features that are going to have a large impact on your workloads

For rest of the stuff, see the release notes and docs!



Data skipping via column stats

Don't read files unnecessarily!

Column min/max values automatically collected when writing files and stored in Delta Log

Read queries can skip files completely using min/max values

Much better than Parquet row-group filtering as you don't need to even read Parquet footer

```
SELECT * FROM events
WHERE year=2020 AND uid=24000
```

~~file1.parquet~~ year: min 2018, max 2019
uid: min 12000, max 23000

~~file2.parquet~~ year: min 2018, max 2020
uid: min 12000, max 14000

skipped as data range outside selected value

file3.parquet year: min 2020, max 2020
uid: min 23000, max 25000

Optimize ZOrder

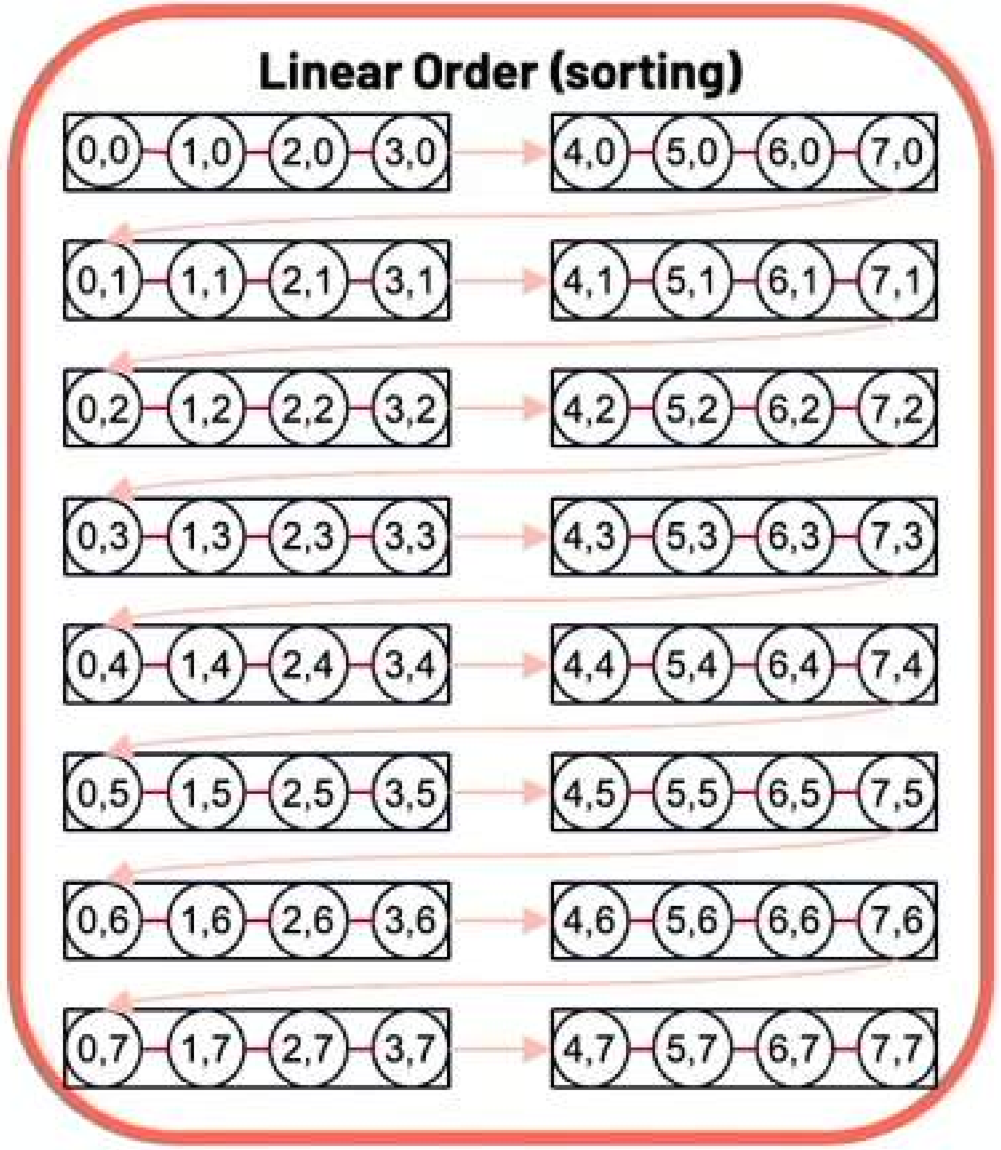
Maximize data skipping with data clustering

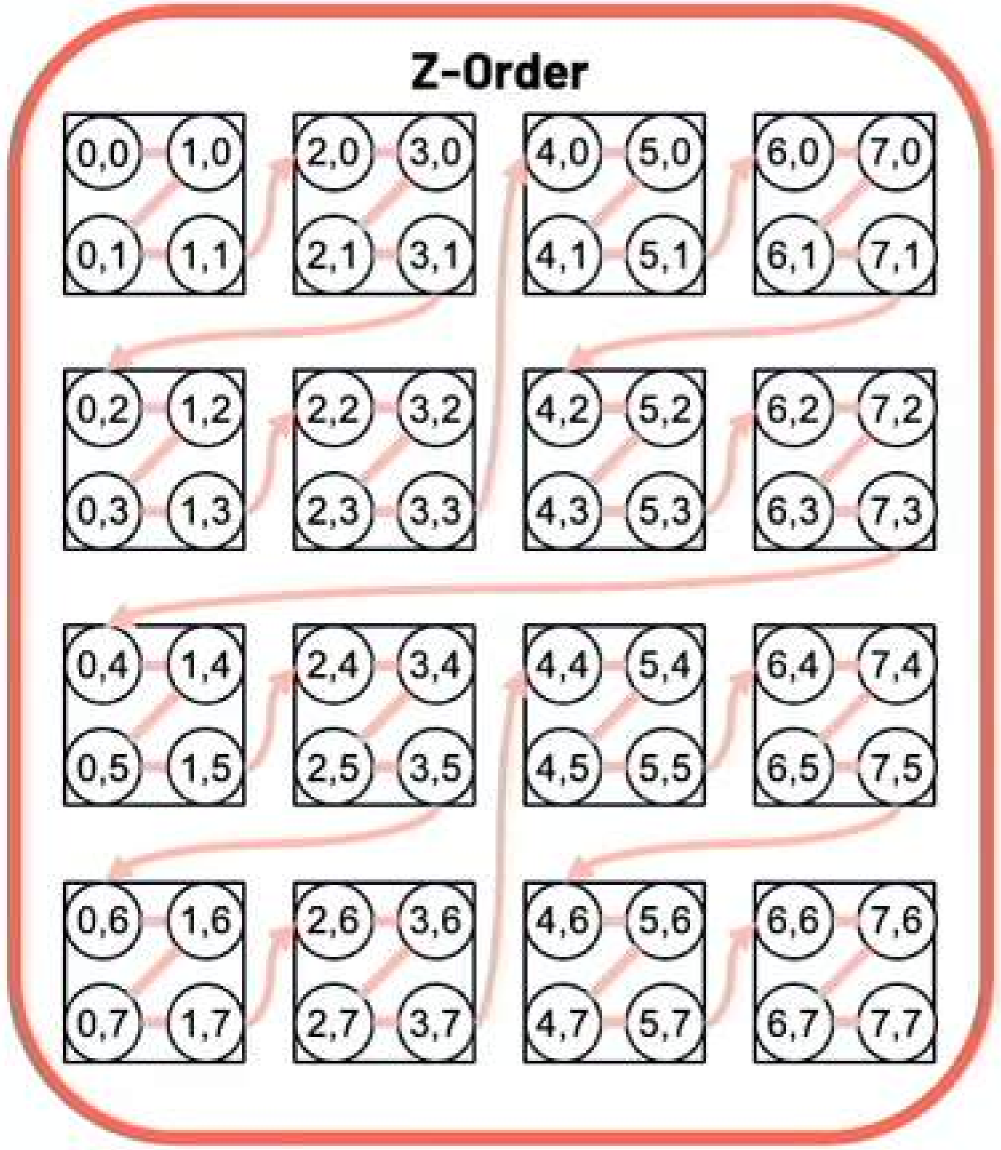
Data skipping most effective when files have very small min/max range

Sorting good for one column, not multiple

Zorder space filling curve gives better multi-column data clustering

```
OPTIMIZE deltaTable ZORDER BY (x, y)
```





Optimize ZOrder

Zorder enables great data skipping in queries with filters over multiple columns

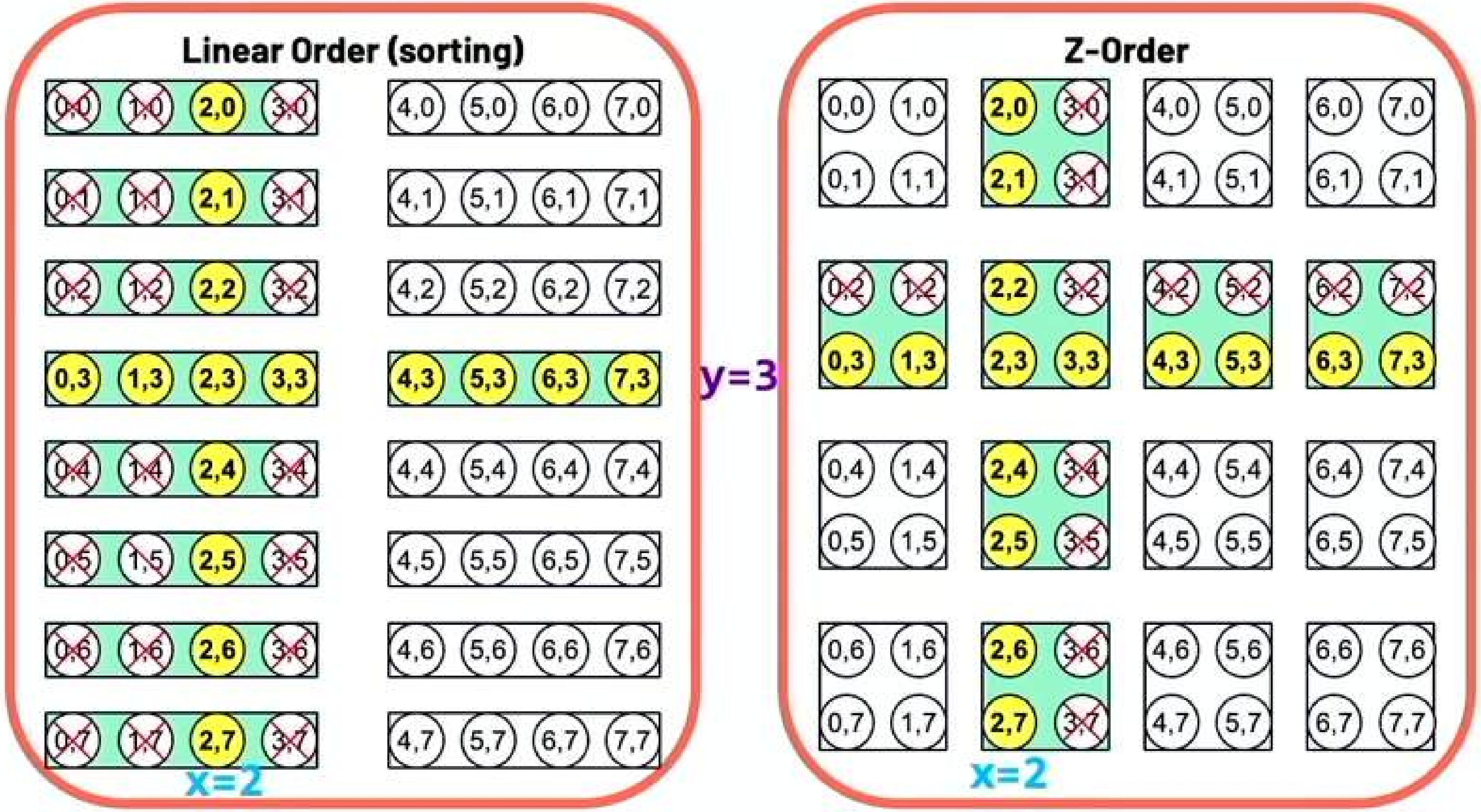
Choose Zorder columns based on query patterns

- Re-run zorder if query patterns change
- Evolve data layout based on your requirements!

```
SELECT * FROM deltaTable
WHERE x = 2 OR y = 3
```

9 files scanned in total 👎
21 false positives 👎

7 files scanned in total 👍
13 false positives 👍



Optimize compaction + Zorder: Perf results

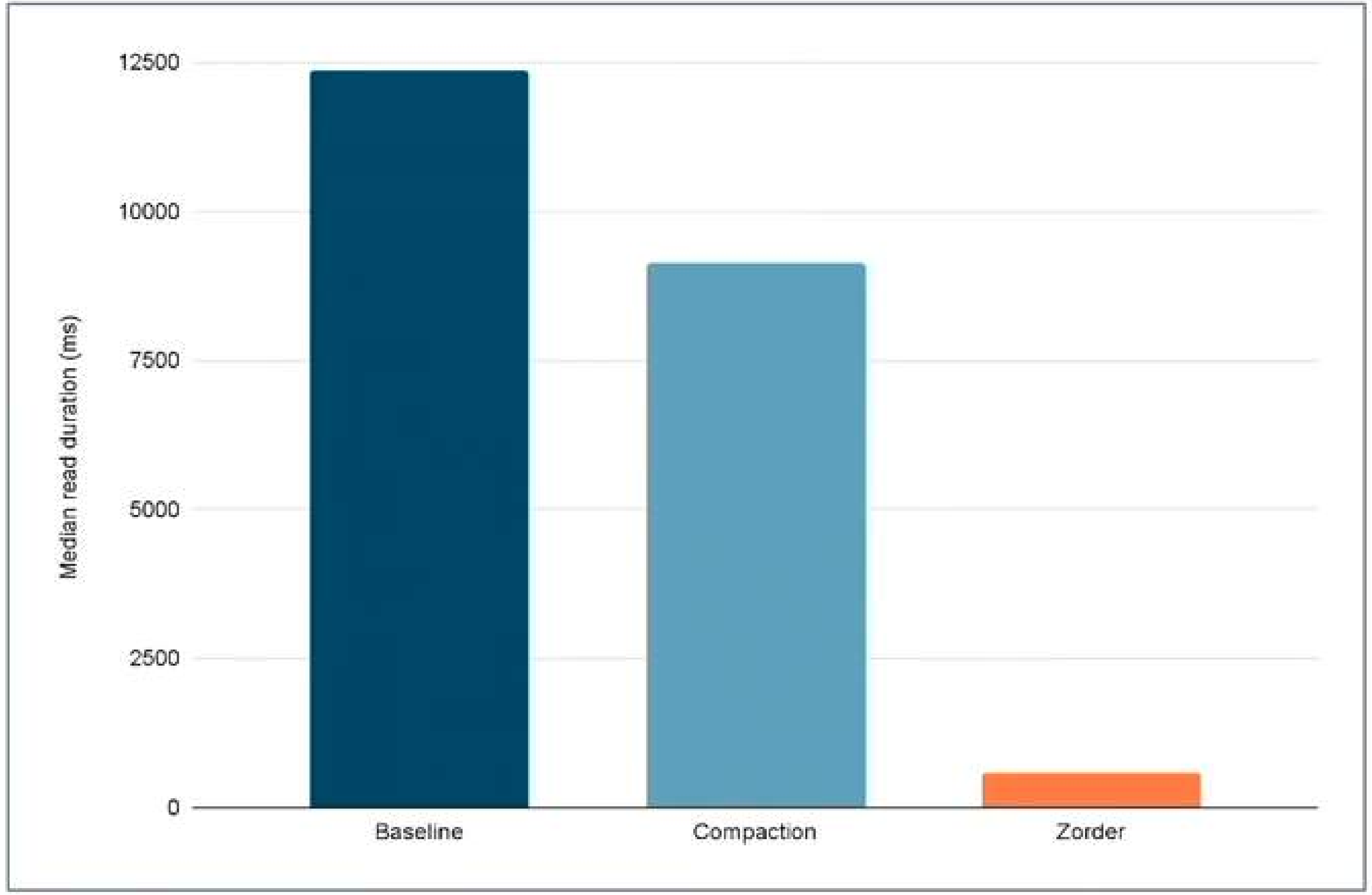
0.5TB store_sales table from 3TB TPCDS dataset

Baseline: ~40k files about ~13MB each

Compaction: ~1GB files

Zorder by ss_item_sk: ~1GB clustered files

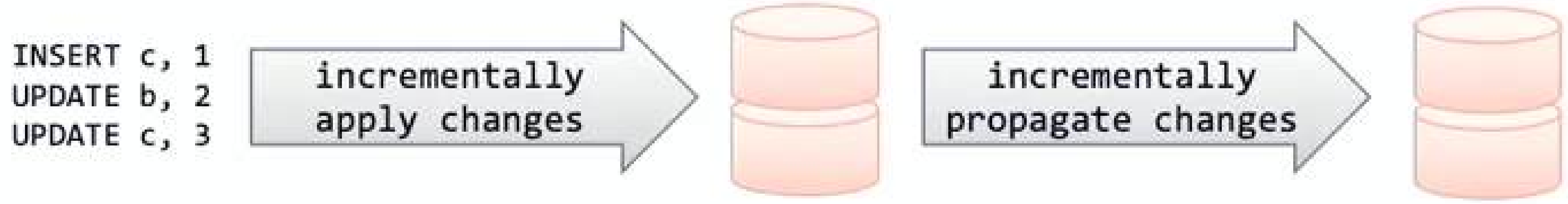
```
SELECT COUNT(*) FROM store_sales
WHERE ss_item_sk = 926
```



Change Data Feed : Motivation

Read row-level changes generated by update/delete/merge

Change Data Capture (CDC) is a common pattern where row-level changes are used to build incremental pipelines



end-to-end incremental pipelines

Change Data Feed: Motivation

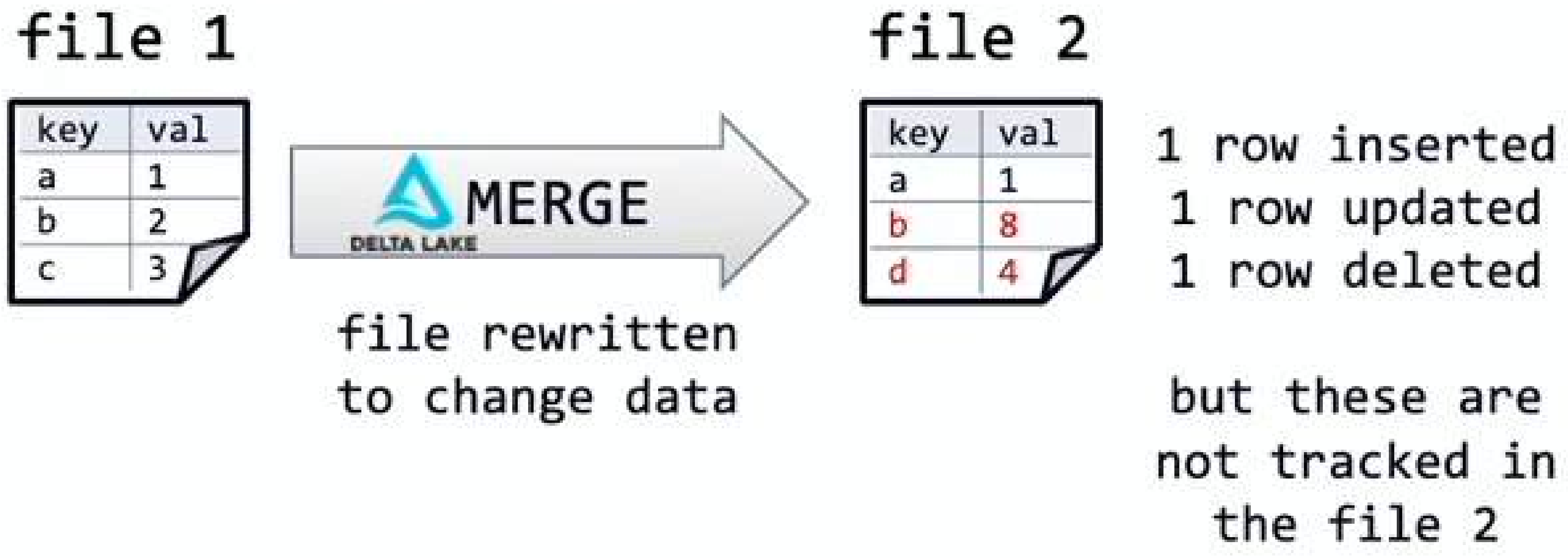
Read row-level changes generated by update/delete/merge

Applying external row-level changes is easy with MERGE

- SQL, Scala, Python APIs
- Automatic Schema Evolution to continuously evolve with your data

MERGE copy-on-write rewrites files to change data

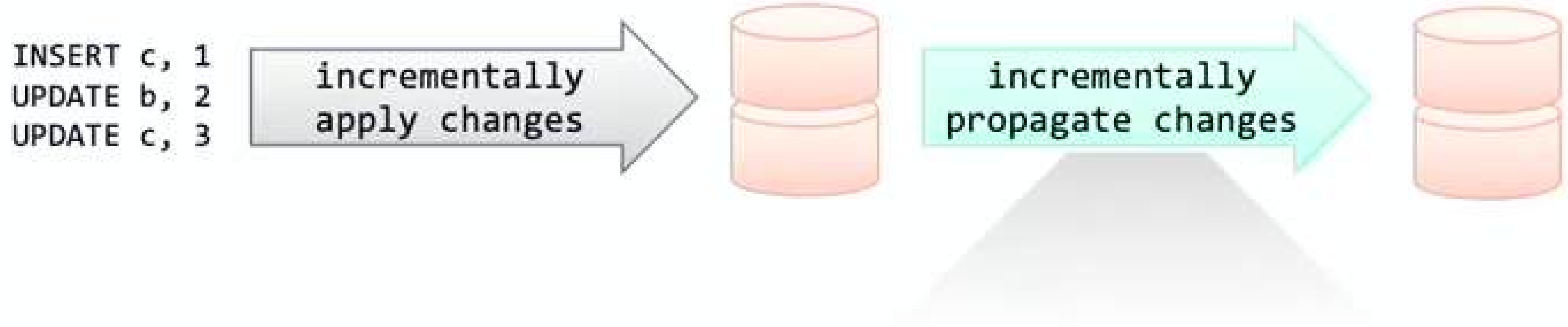
- optimized for fast reads
- but which rows changed are not tracked



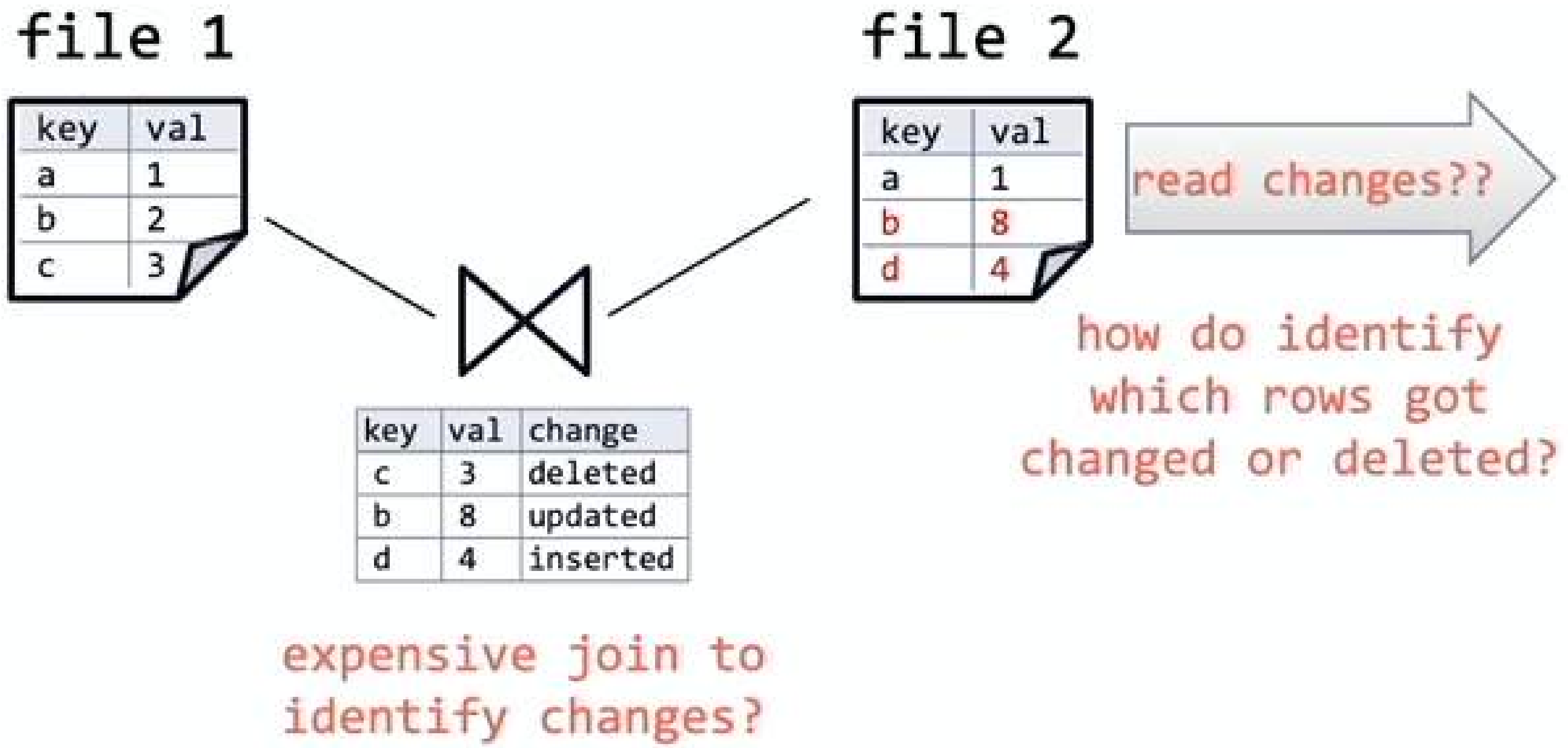
Change Data Feed: Problem

Read row-level changes generated by update/delete/merge

Reading just the changes rows is inefficient without more information



Joining between two versions can work if there are unique keys, but highly inefficient

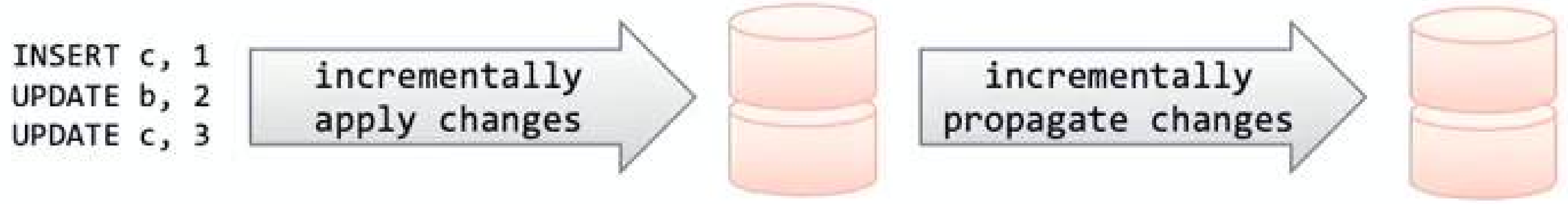


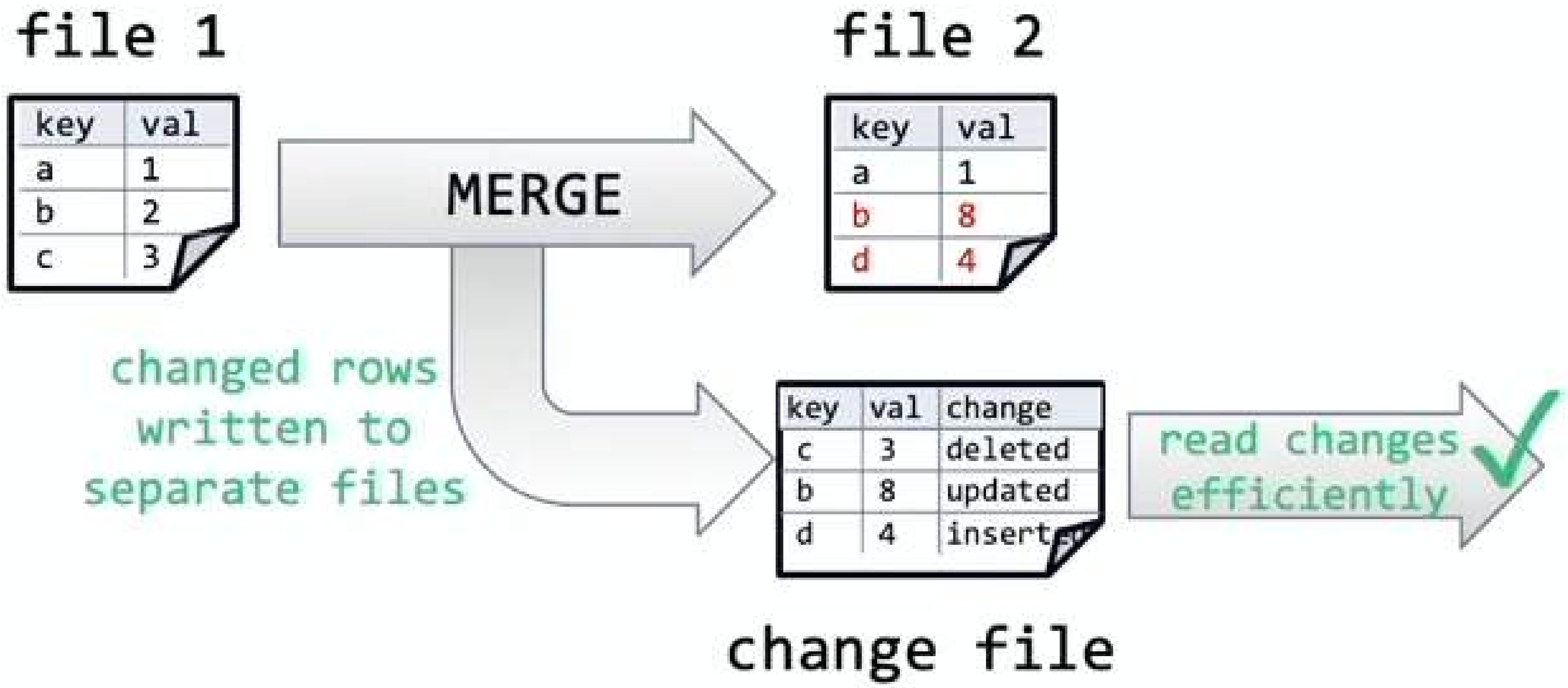
Change Data Feed: Solution

Read row-level changes generated by update/delete/merge

Store the row-level changes in a separate set of files

- Merge/update/delete will produce the additional change files
- Reading change data is efficient as they are separate files, no filtering needed
- Reading normal data unaffected and still efficient





Change Data Feed: Batch and Streaming APIs

Read row-level changes generated by update/delete/merge

Build incremental pipelines with Structured Streaming

- Read only latest changes or starting from a version


```
spark.readStream.format("delta")
  .option("readChangeFeed", "true")
  .load("/deltaTable")
```

Query changes between any table versions or timestamps

- DataFrame options
- SQL support in future

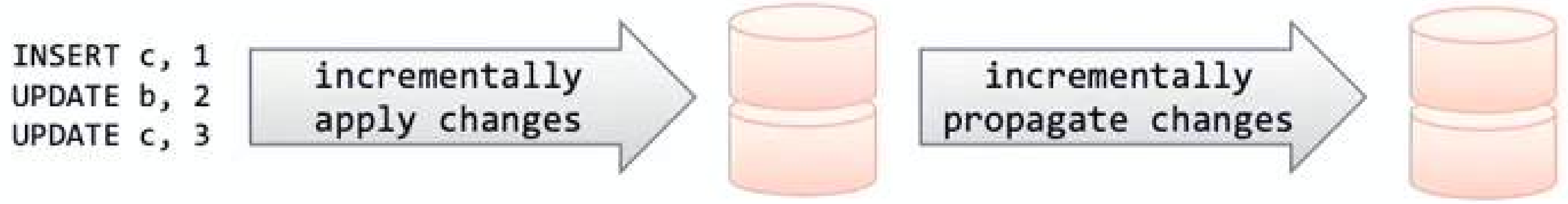
```
spark.read.format("delta")
  .option("readChangeFeed", "true")
  .option("startingTimestamp", "2021-04-21 05:45:46")
  .option("endingTimestamp", "2021-05-21 12:00:00")
  .load("/deltaTable")
```

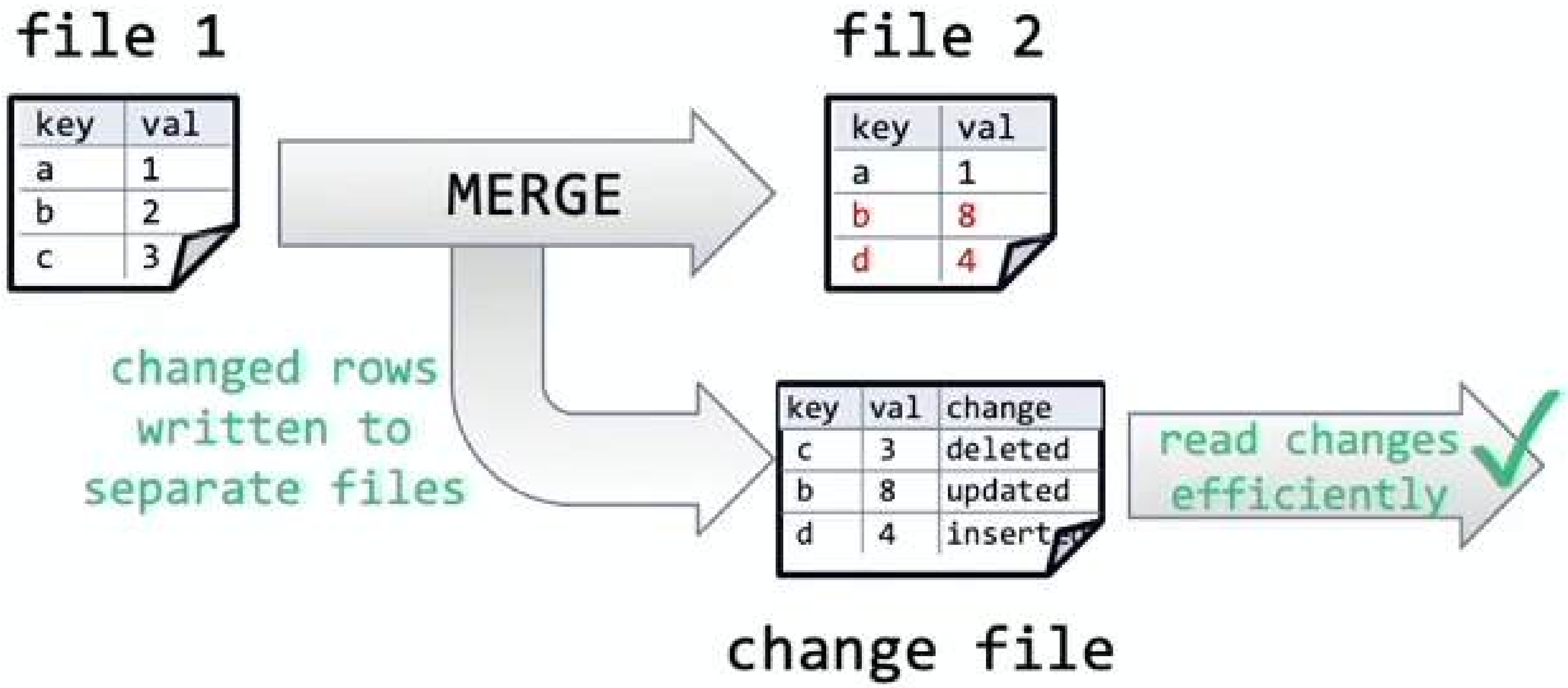
Change Data Feed: Solution

Read row-level changes generated by update/delete/merge

Store the row-level changes in a separate set of files

- Merge/update/delete will produce the additional change files
- Reading change data is efficient as they are separate files, no filtering needed
- Reading normal data unaffected and still efficient





Column Mapping: Problem

More flexibility in naming, renaming and dropping columns

Problem: Delta 1.1 and below required Parquet files to store data with same column name as table schema

- Cannot change column names without rewriting existing files
- Cannot have characters in column names not supported by Parquet (e.g., no spaces)

Before Column Mapping

Column Mapping: Solution

More flexibility in naming, renaming and dropping columns

Solution: Delta 1.2 introduced a mapping between the *logical column name* and the *physical column name* in the files

- Physical names are unique
- Logical column renames become a simple change in the mapping
- Logical column names can have arbitrary characters, physical name always Parquet-compliant

With Column Mapping

table data

key	val
a	1
b	8
d	4

logical col name	physical col name
key	uuid1
val	uuid2

file.parquet

uuid1	uuid2
a	1
b	8
d	4

Column Mapping: APIs

More flexibility in naming, renaming and dropping columns

[Delta 1.2]

Support for renaming columns

Support for arbitrary column names

Use special chars like ,;{}()\n\t=

```
ALTER TABLE table_name
RENAME COLUMN
old_col_name TO `{new,col,name}`
```

[Delta 2.0]

Support for dropping columns

```
ALTER TABLE table_name
DROP COLUMN col_name
```

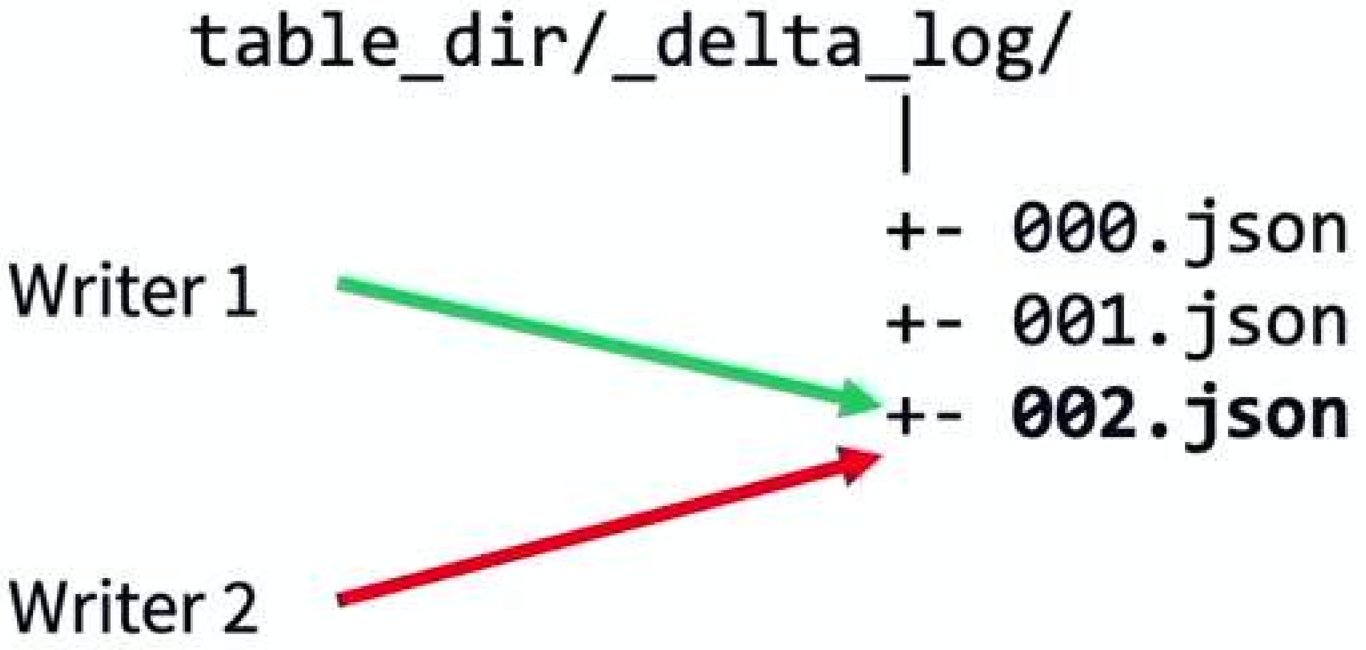
Multi-cluster writes on S3

Full ACID guarantees without maintaining your own infra

Delta Lake ACID guarantees rely on mutual exclusion guarantees from the file system

- Must be able to exclusively create a file in the Delta log only if absent
- Works great for HDFS, GCS, ADLS, etc.

Allows guarantees without using distributed locks or leases which are very hard to get right



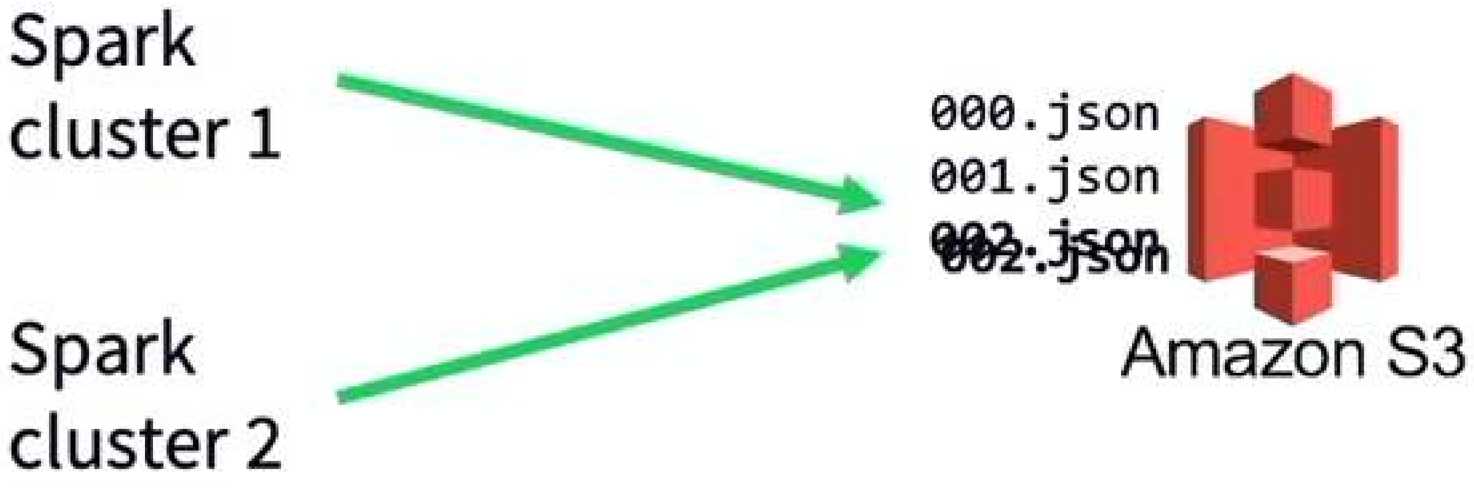
only one of the writers trying to concurrently write 002.json must succeed => only then all changes are serializable

Multi-cluster writes on S3: Problem

Full ACID guarantees without maintaining your own infra

Problem: S3 does not provide any mechanism for mutual exclusion

Delta 1.1 and below did not support concurrent writes from multiple Spark clusters



both concurrent writes from different clusters will succeed and overwrite each other's commits => no serializability

Multi-cluster writes on S3: Solution

Full ACID guarantees without maintaining your own infra

Solution: write with mutual exclusion to DynamoDB

1. Only one writer commits changes to DynamoDB

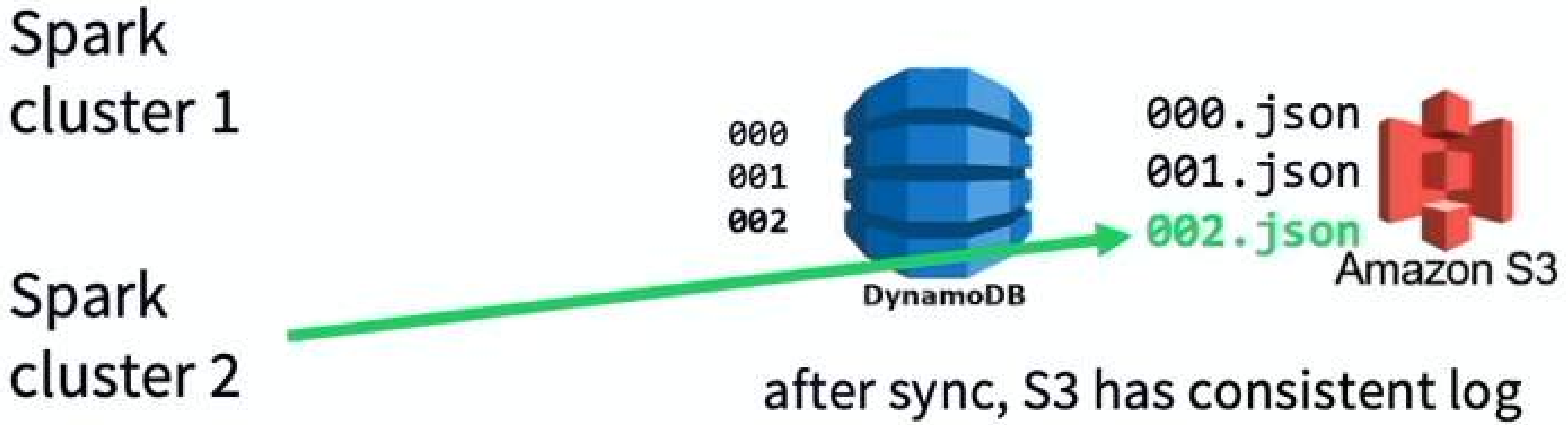
only one writer succeeds in committing to DynamoDB ensuring serializable changes

Multi-cluster writes on S3

Full ACID guarantees without maintaining your own infra

Solution: write with mutual exclusion to DynamoDB

1. Only one writer commits changes to DynamoDB
2. Committed writes synced from DynamoDB to S3



after sync, S3 has consistent log structure for all readers

Robust solution: no distributed locks or leases, no self-managed service or infra

Multi-cluster writes on S3

`Full ACID guarantees without maintaining your own infra`

Enable multi-cluster writes in Delta 1.2 and above by setting Spark configs

- *Log store type:*

```
spark.delta.logStore.s3.impl = io.delta.storage.S3DynamoDBLogStore
```

- *DynamoDB table details:*

```
spark.io.delta.storage.S3DynamoDBLogStore.ddb.tableName = <table name>

spark.io.delta.storage.S3DynamoDBLogStore.ddb.region = <AWS region>
```

All writers writing to the same Delta table must be configured with the same DynamoDB table for correctness

Many more features

See docs and release notes for details

Restore (aka rollback) to previous table versions

```
RESTORE TABLE deltaTable
  TO TIMESTAMP AS OF '2019-02-14 12:00:00'
```

Automatic filter generation on generated partition columns

Better filtering, faster queries

```
… WHERE eventTime < '2021-05-24 09:00:00.000'
```

generate extra filter if table is partitioned by `eventDate` generated from `eventTime`

```
… WHERE eventTime < '2021-05-24 09:00:00.000'
  AND eventDate < '2021-05-24'
```

Write impotently to a table

No duplicates on retries

```
dataframe.write.format("delta")
  .option("txnAppId", "myApp")
  .option("txnVersion", 10)
  .save("/deltaTable")
```

Flink: Delta Sink

Available since Delta Connectors 0.4

Writes from DataStream<RowData> in batch or streaming modes

Supports reading by table path on ADLS, GCS and S3 (single cluster)

- Support for S3 multi-cluster using DynamoDB coming in Connectors 0.5

Gives exactly once guarantees with replayable sources

```
DeltaSink<RowData> deltaSink = DeltaSink
    .forRowData(path, hadoopConf, rowType)
    .withPartitionColumns(...)
    .build();

datastream.sinkTo(deltaSink);
```

Flink: Delta Source

Coming with Delta Connectors 0.5

Reads as DataStream<RowData> in bounded or continuous mode

- For bounded, supports querying old table versions (aka Time Travel)
- For continuous, supports reading full table + changes, OR only changes since a version

Supports all file systems

Support for catalog tables + SQL + Table API in progress

```
DeltaSource
    .forBoundedRowData(path, hadoopConf)
    .build();

// Time travel
DeltaSource
    .forBoundedRowData(path, hadoopConf)
    .timestampAsOf("2022-02-24 04:55:00")
    .build();

// Streaming
DeltaSource
    .forContinuousRowData(path, hadoopConf)
    .build();
```

Trino / Presto: Delta connector

`Available since Presto 0.269 and Trino 0.375`

- [Presto and Trino] Supports reads on tables defined in Hive Metastore
- [Trino] Supports data skipping with column stats
- [Trino] Supports writes
- [Trino] Support Optimize compaction