Product safe harbor statement

This information is provided to outline Databricks' general product direction and is for **informational purposes only**. Customers who purchase Databricks services should make their purchase decisions relying solely upon services, features, and functions that are currently available. Unreleased features or functionality described in forward-looking statements are subject to change at Databricks discretion and may not be delivered as planned or at all

Deep Dive into Delta Lake and UniForm

Sirui Sun, Sunitha Beeram
June 2024

Who Are We?

Michelle Leon

- Staff Product Manager
  - Previously Webflow, Airbnb
- Based in San Francisco
- Talk to me about
  - Delta Lake: incl transactions, coordinated commits, Delta Kernel
  - Unity Catalog interoperability
  - Best burritos in the Mission neighborhood 🌯

Who Are We?

Joe Widen

- Principal Solutions Architect
  - Joined Databricks in 2017, previously at Hortonworks, Capital One
- Proud owner of the smallest contribution to Delta Lake
  - You can find it in Delta Lake 1.0.0
- Focused on
  - Helping clients push the limits of Delta Lake
  - Helping clients adopt the latest Delta Lake features

Agenda

1. Intro to Delta Lake and core capabilities
2. Unpacking the Transaction Log Protocol
3. Data Layout Innovations
4. Uniform
5. Use cases
6. Roadmap

Tech Check

Cats or Dogs?

Delta Lake

Delta Lake with UniForm is an open format that brings performance, interoperability, and ACID transactions to open data lakes.

Delta Lake Key Features

ACID Transactions
Protect your data with serializability, the strongest level of isolation.

Scalable Metadata
Handle petabyte-scale tables with billions of partitions and files at ease

Time Travel
Access/revert to earlier versions of data for audits, rollbacks, or reproduce

DML Operations
SQL, Scala/Java and Python APIs to merge, update and delete datasets

Unified Batch/Streaming
Exactly once semantics ingestion to backfill to interactive queries

Schema Evolution / Enforcement
Prevent bad data from causing data corruption

Audit History
Delta Lake log all change details providing a full audit trail

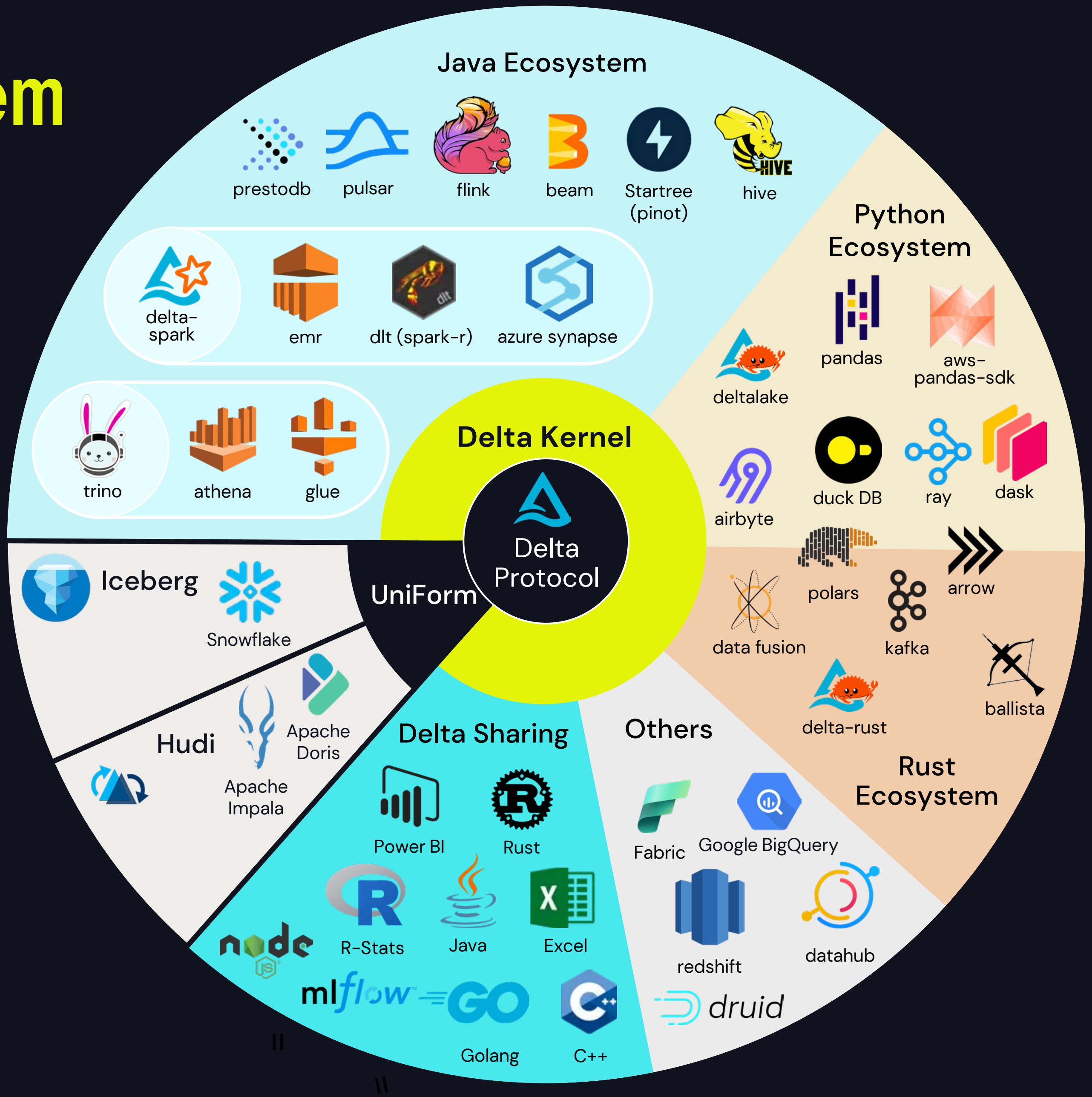
Open Source
Community driven, open standards, open protocol, open discussions

Delta Lake – quickstart

```
bin/spark-sql
    --packages io.delta:delta-spark_2.12:3.1.0
    --conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension"
    --conf "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

```
CREATE TABLE cat.sch.tbl` USING DELTA
AS SELECT col1 as id
FROM VALUES 0,1,2,3,4;
```

Thriving ecosystem

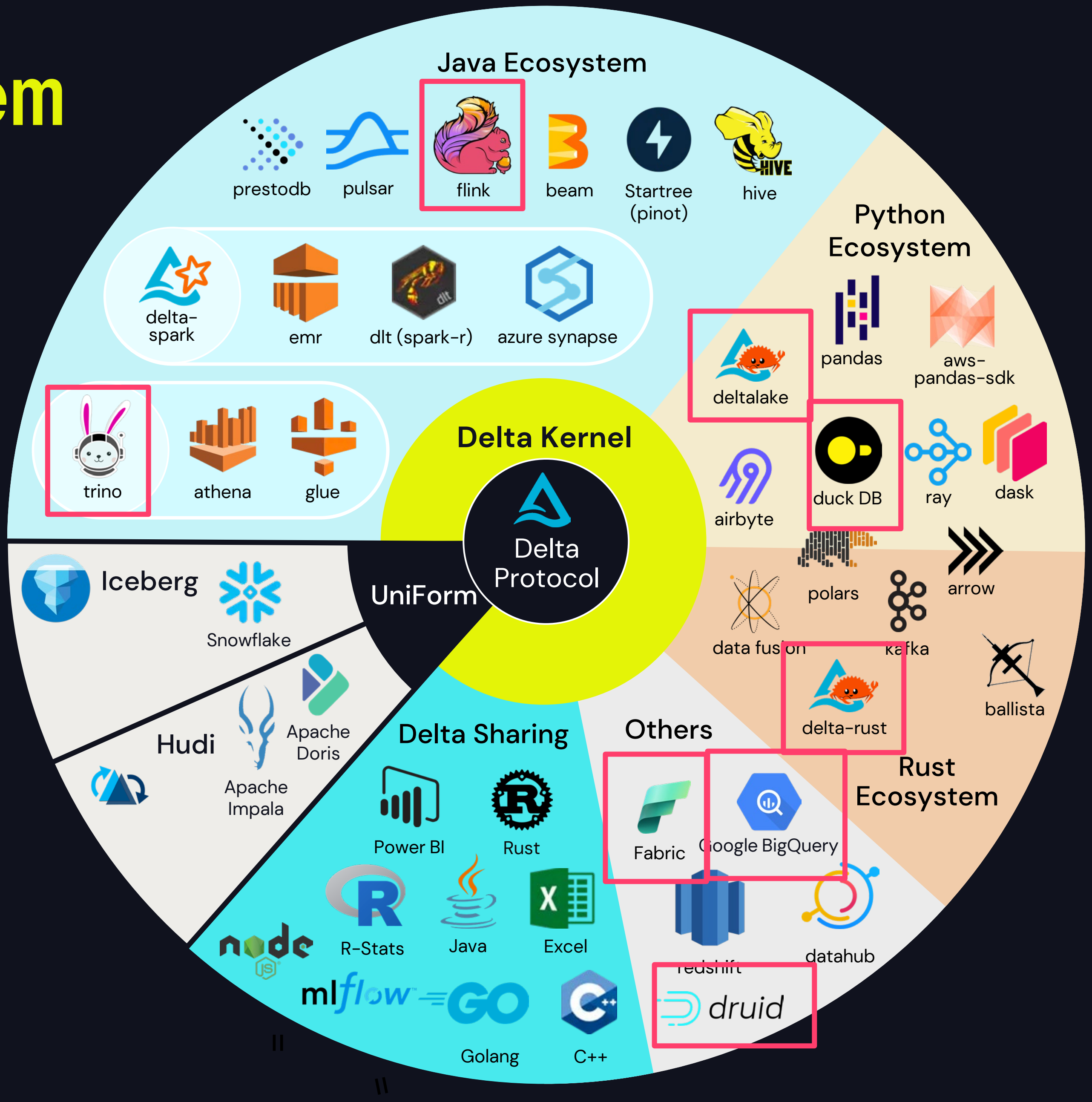


now i forget

Thriving ecosystem

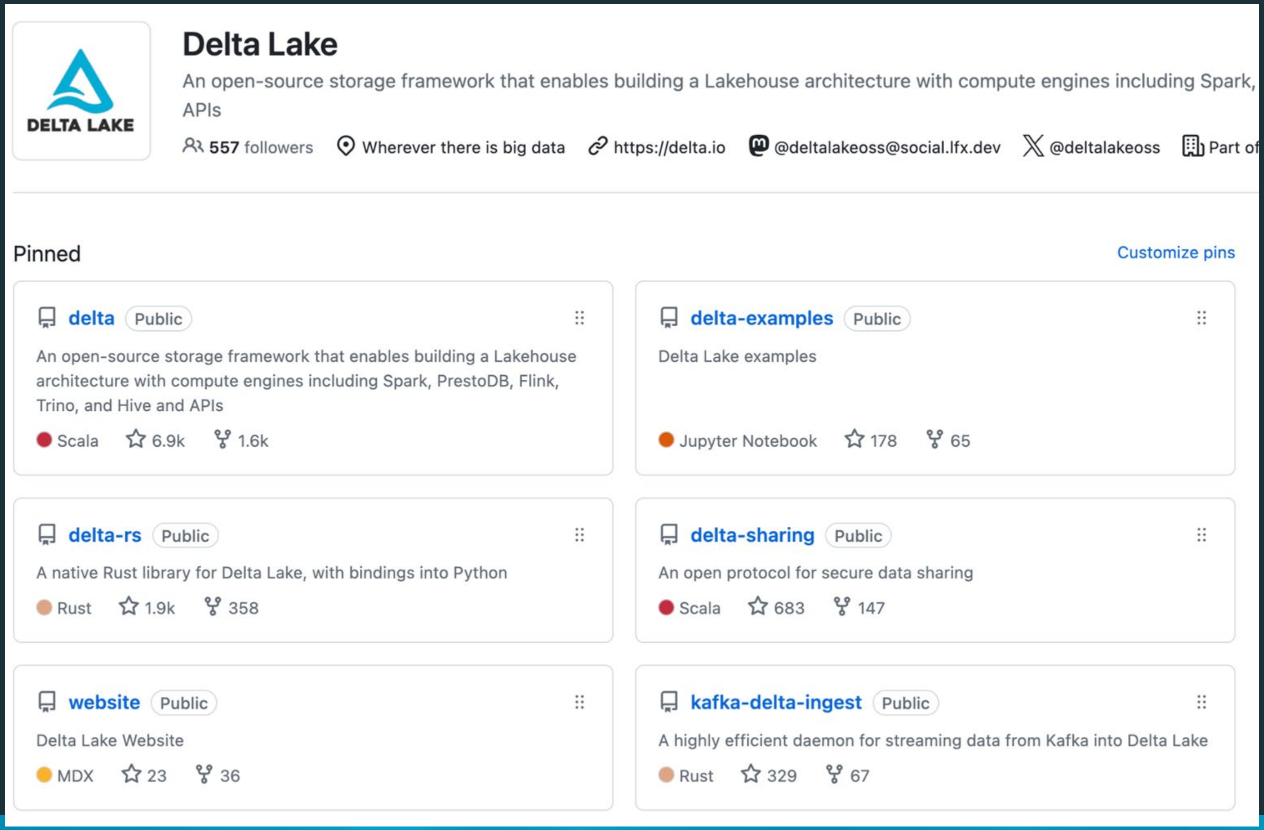
New or significantly updated

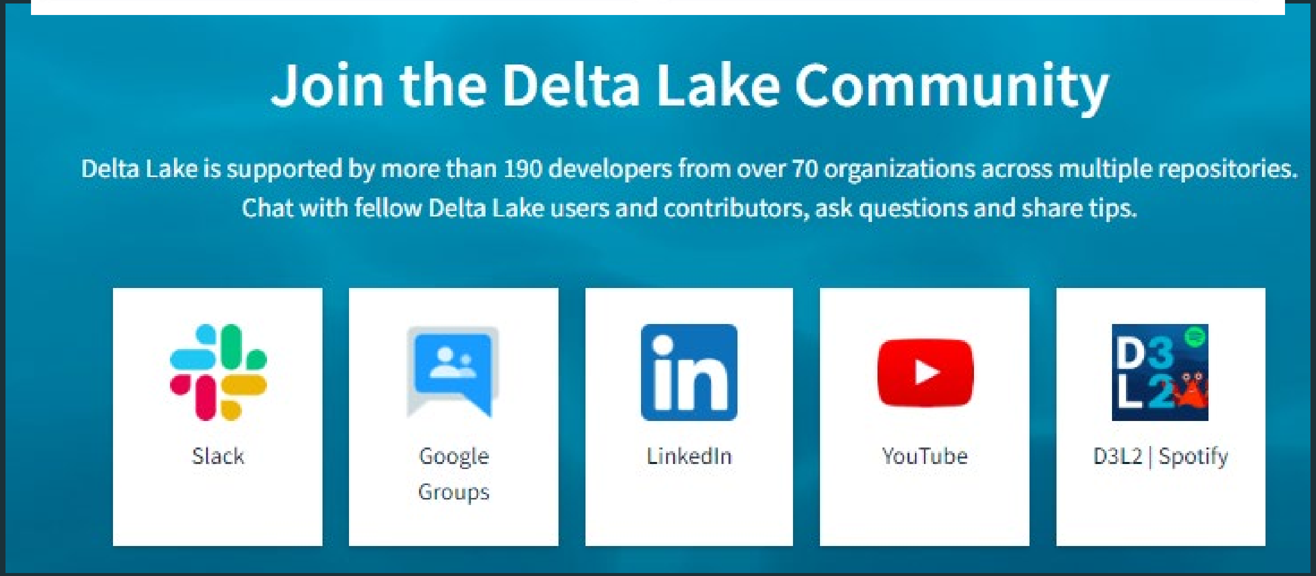
- Delta Flink
- Delta Trino
- Delta Rust / deltalake Python
- Apache Druid
- Google BigQuery
- DuckDB



Community

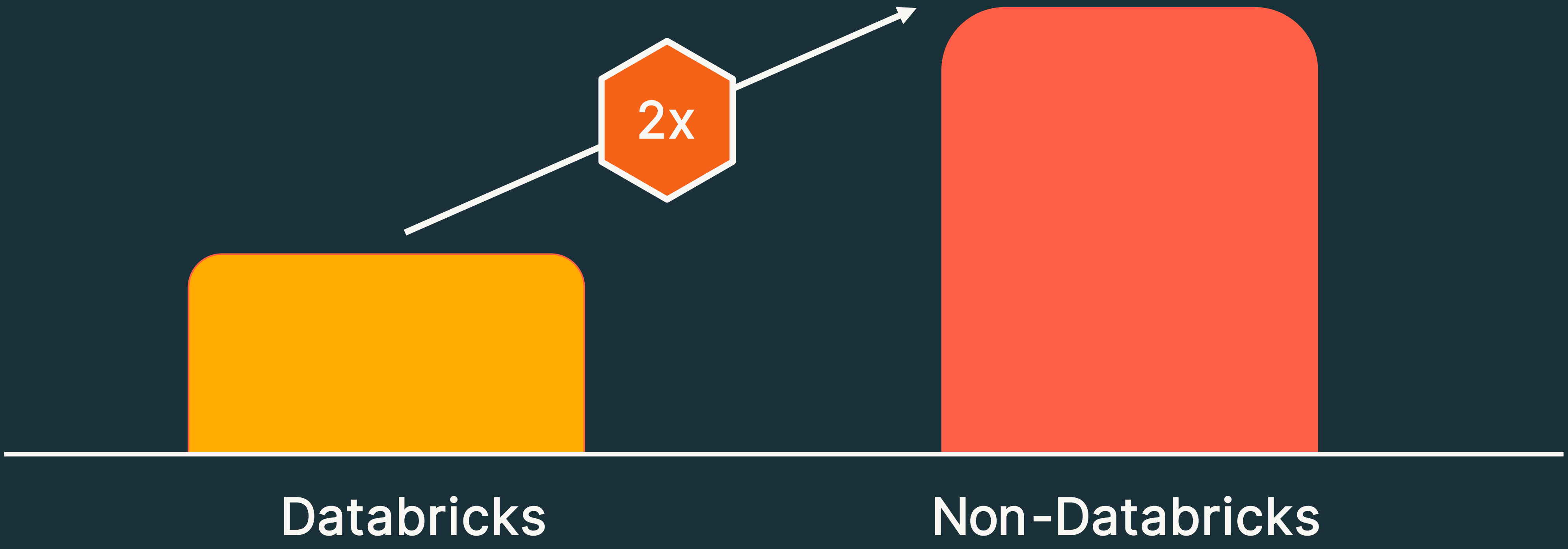
- 11+ repos in the project
  - production and incubator projects
- > 50 releases
  - Latest: Delta 3.2, Delta Rust 0.17
- Very active community
  - ~9K Github stars
  - ~500 contributors
  - Slack: ~10K members
  - LinkedIn: ~50K members
  - YouTube: ~2.5K subscribers





Delta Lake - Pull Requests Merged

Source: Linux Foundation Insights



Delta Lake:
The **most adopted** open lakehouse format

Scalable	Popular	Prevalent	Reliable	Innovative	Open
9+ exabytes processed per day	**1B+** Clusters per year	**60%+** Fortune 500 Adoption	**>10K+** Companies in production	**80+** New features / year	**>500** Contributors

2x yearly growth

The biggest Delta Lake release yet

Delta 3.0

- Deletion Vectors
- Liquid clustering
- Delta Kernel
- Optimized Writes
- Table features
- Incremental checkpoints
- Log compactions
- Row IDs
- Table cloning
- MERGE improvements
- CDF
- Auto-compaction

Delta 4.0

VARIANT
Lightning fast semi-structured data

UniForm GA
Write once, read as all formats

Liquid GA
Easy migration from partitioned tables

Coordinated Commits
Cross cloud, cross engines

Spark Connect
Better stability, upgradability

Collations
Flexible sort and comparison

Identity columns
Pain-free primary and foreign keys

Type widening
Data types expand with your data

• • •

POP QUIZ

Unpacking the transaction log

Delta Lake on Disk

```
/mytable/
      _delta_log/
            00010.checkpoint.parquet
            00011.json
            00012.json
            _last_checkpoint
      _change_data/
            cdc-file1.snappy.parquet
      date=2024-06-14/
            file-1.snappy.parquet
      deletion_vector1.bin
```

Delta Lake on Disk

Transaction Log	/mytable/ _delta_log/
Commits & Checkpoints	00010.checkpoint.parquet 00011.json 00012.json
Checkpoint pointer	_last_checkpoint
(Optional) Change Data	_change_data/ cdc-file1.snappy.parquet
(Optional) Partition Directories	date=2024-06-14/
Data	file-1.snappy.parquet
(Optional) Deletion Vectors	deletion_vector1.bin

Table = result of a set of actions

Metadata – name, schema, partitioning, etc

Add File – adds a file (with optional statistics)

Remove File – removes a file

Transaction Identifier – records an idempotent transaction id

Protocol Evolution – upgrades the version of the txn protocol

Commit Provenance – additional information about what higher-level operations was being performed as well as who executed it

Result: Current Metadata, List of Files, List of Txns, Version

Example of an addFile action

The add action is used to modify the data in the table by adding individual files respectively.

Path, partitionValues, size, modificationTime and dataChange are required fields. Other fields like stats, tags, and clusteringProvider are optional.

```
{
  "add": {
    "path": "date=2017-12-10/part-000...c000.gz.parquet",
    "partitionValues": {"date": "2017-12-10"},
    "size": 841454,
    "modificationTime": 1512909768000,
    "dataChange": true,
    "baseRowId": 4071,
    "defaultRowCommitVersion": 41,
    "stats": "{\"numRecords\":1,\"minValues\":{\"val..."
  }
}
```

ACID properties

Implementing Atomicity

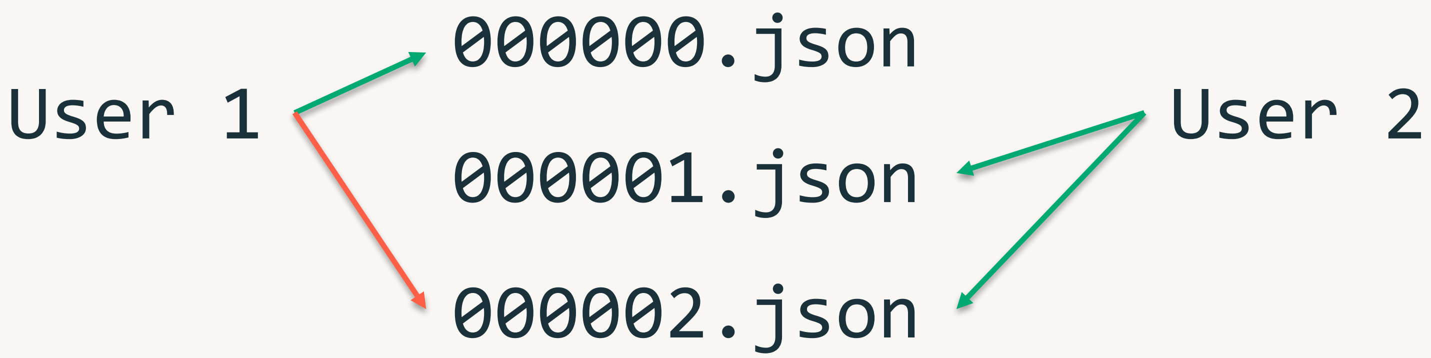
Changes to the table are stored as ordered, atomic units called **commits**

```
000000.json
  Add 1.parquet
  Add 2.parquet

000001.json
  Remove 1.parquet
  Remove 2.parquet
  Add 3.parquet
```

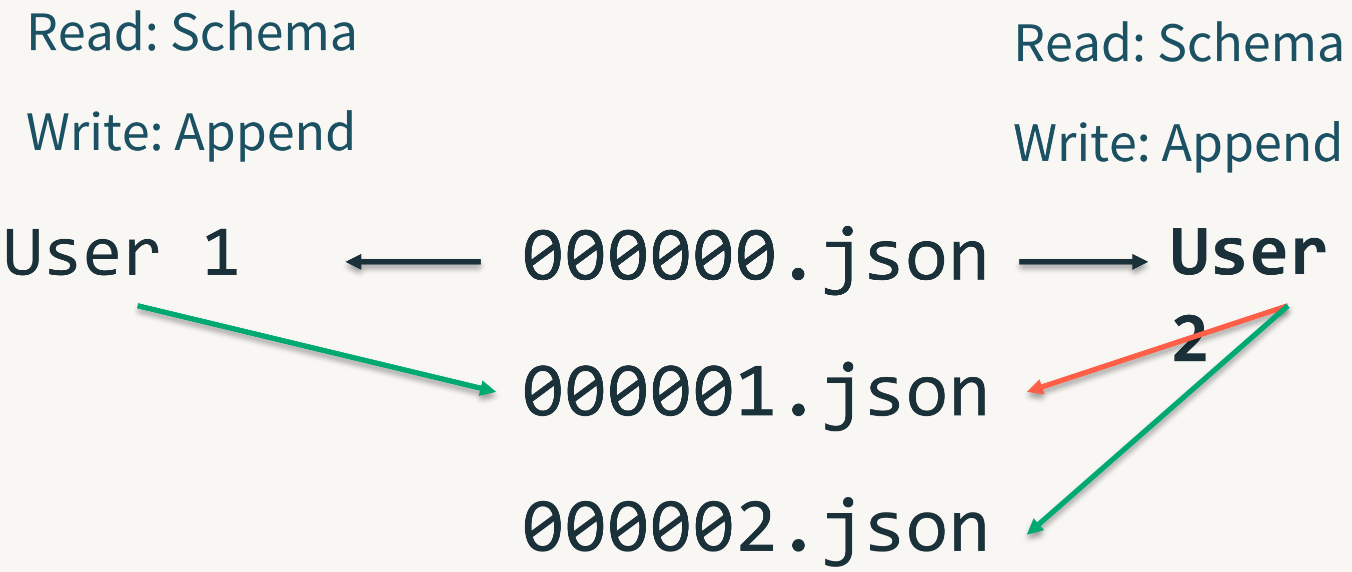
Ensuring Serializability

Need to agree on the order of changes, even when there are multiple writers.



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



Transactions and reliability are great, but what about performance?

Handling Massive Metadata

Large tables can have millions of files. Delta Lake can use a distributed engine for scaling

```
Add 1.parquet
Add 2.parquet
Remove 1.parquet
Remove 2.parquet
Add 3.parquet
```

Updating Delta Lake’s State

listFrom version 0

000000.json

000001.json

000002.json

000003.json

000004.json

000005.json

000005.json

000006.json

000007.json

Cache
version 7

Updating Delta Lake's State

000000.json

...

000007.json

000008.json

000009.json

0000010.json

0000010.checkpoint.parquet

0000011.json

0000012.json

listFrom version 7

Read the checkpoint

Cache version 12

Finding the latest metadata

The Delta transaction log can contain many (e.g. 10,000+) commits and this can take a long time to list

`_last_checkpoint` provides a pointer to near the end of the log

`listFrom` storage API provides the ability to list only from the last known checkpoint

```
/mytable/
      _delta_log/
            0000.json
            0001.json
            0002.json
            …

      0100.checkpoint.parquet
            0101.json
            …

      0200.checkpoint.parquet
            0201.json
            _last_checkpoint
      …
```

Time Travel

Time Traveling by version

```
SELECT * FROM my_table VERSION AS OF 500;

SELECT * FROM my_table@v500

spark.read.option("versionAsOf", 500).load("/some/path")

spark.read.load("/some/path@v500")
```

```
deltaLog.getSnapshotAt(500)
```

Time Traveling by timestamp

```
SELECT * FROM my_table TIMESTAMP AS OF '2019-10-16';

SELECT * FROM my_table@20191016000000000    -- yyyyMMddHHmmssSSS

spark.read.option("timestampAsOf", "2019-10-16").load("/some/path")

spark.read.load("/some/path@20191016000000000")
```

```
deltaLog.getSnapshotAt(500)
```

Time Traveling by timestamp

Commit timestamps come from storage system modification timestamps

001070.json	2019-10-16
001071.json	2021-05-24
001072.json	2022-07-20
001073.json	2022-06-30

Time Traveling by timestamp

Timestamps can be out of order. We adjust by adding 1 millisecond to the previous commit's timestamp

001070.json	2019-06-19	2019-06-19
001071.json	2021-05-24	2021-05-24
001072.json	2022-07-20	2022-07-20
001073.json	2022-06-30	2022-07-20 00:00:00.01

Time Traveling by timestamp

Price is right rules – pick the closest commit timestamp that doesn't exceed the users timestamp

001070.json	2019-06-19
001071.json	2021-05-24
	← 2022-04-13
001072.json	2022-07-20
001073.json	2022-07-20 00:00:00.01

The Single Source of Truth!

Information required to plan a query

Information	Parquet Source	Delta Lake Source
1. Schema	1. HMS or inferred from file footer	1. Transaction Log
2. Partition Columns and values	2. HMS or inferred	2. Transaction Log
3. Files to read	3. FileSystem listing	3. Transaction Log
4. File Statistics	4. NA	4. Transaction Log
5. Protocol (Delta Only)	5. NA	5. Transaction Log

Getting the schema of a Delta Lake table

Read the transaction log!

Collect all the metadata actions for your table

Merge the schema strings together

Time Travel allows you to go back before meta changes!

```
{
  "metaData":{
    "id":"af23c9d7-fff1-4a5a-a2c8-55c59bd782aa",
    "format":{"provider":"parquet","options":{}},
    "schemaString":"...",
    "partitionColumns":[],
    "configuration":{
      "appendOnly": "true"
    }
  }
}
```

Getting the partition columns

Read the transaction log!

Collect all the metadata actions for your table

Collect list of partition columns

Scales to millions of partitions

```
{
  "metaData":{
    "id":"af23c9d7-fff1-4a5a-a2c8-55c59bd782aa",
    "format":{"provider":"parquet","options":{}},
    "schemaString":"...",
    "partitionColumns":[],
    "configuration":{
      "appendOnly": "true"
    }
  }
}
```

Getting the list of files to read

Read the transaction log!

Collect all the add file actions

Apply partition and data filters

Collect list of paths

Scales to millions of files


```
{
  "add": {
    "path": "date=2017-12-10/part-000...c000.gz.parquet",
    "partitionValues": {"date": "2017-12-10"},
    "size": 841454,
    "modificationTime": 1512909768000,
    "dataChange": true,
    "stats": "{\"numRecords\":1,\"minValues\":{\"val..."
  }
}
```

Additional Features

Generated Columns

A generated column is a special column that's defined with a SQL Expression

```
CREATE TABLE events(
    eventId BIGINT,
    data STRING,
    eventType STRING,
    eventTime TIMESTAMP,
    eventDate date GENERATED ALWAYS AS (CAST(eventTime AS
    DATE))
)
  PARTITIONED BY (eventType, eventDate)
```

Generated Columns

Querying a generated column will apply partition pushdown if you use the generated column, or the column it was generated from

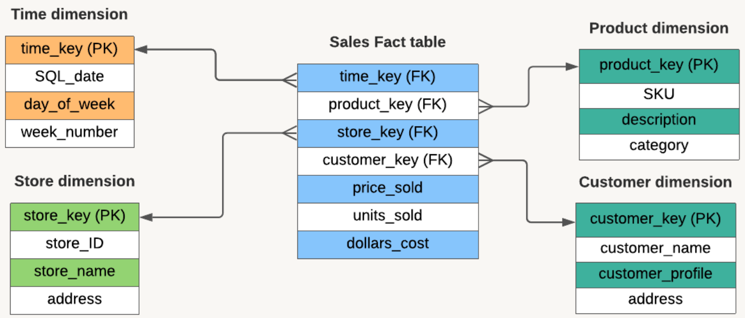
```
SELECT * From events WHERE eventTime >= "2020-10-01
00:00:00" <= "2020-10-01 12:00:00"
```

For the above query, we will only read the date 2020-10-01 even though the partition filter is not specified

Support for Identity Columns, Primary + Foreign Key Constraints

IDENTITY COLUMNS

- Define **IDENTITY** column on a table
- Delta can **automatically** generate **unique integer values** when new rows are added to the table with IDENTITY columns
- Users can also explicitly insert values for the IDENTITY columns



PRIMARY + FOREIGN KEY DECLARATIONS

- Declare unenforced Primary and Foreign keys with **ALTER TABLE**
- Visible in **INFORMATION_SCHEMA** and DESCRIBE TABLE
- Allow end users to understand **relationships** between tables

GOAL: Enable **data quality** and **easy** table **relationship discovery** for tools and users that are not familiar with the data model.

Identity Columns

Delta Lake Identity Support

```
CREATE TABLE IF NOT EXISTS dim_loan
(
 Loan_sk BIGINT GENERATED ALWAYS AS IDENTITY,
 Loan_id BIGINT,
........
)
USING DELTA
LOCATION 'abfs://<container>@<storage account>/'
```

Options

```
ALWAYS | BY DEFAULT
START WITH start
INCREMENT BY step
```

- Always option doesn't allow column override
- By Default option does allow column override but *doesn't enforce duplicates*
- Start With option allows you to start anywhere
- Increment option allows you to set the increment

POP QUIZ

Speeding up queries

Speeding up queries

Reading only the necessary rows for a query = Efficient query processing

How does the transaction log help with that?

Partitioned Tables : Partition Pruning

```
/mytable/
      part=1/part_00001.parquet
      part=1/part_00002.parquet
      part=2/part_00001.parquet
      part=2/part_00002.parquet
```

```
select * from mytable where part = 2
```

Data Skipping

Simple, well-known I/O pruning technique

- Track file-level stats like min & max
- Leverage them to avoid scanning irrelevant files

```
SELECT input_file_name() as “file_name”,
       min(col) AS “col_min”,
       max(col) AS “col_max”
FROM table
GROUP BY input_file_name()
```

file_name	col_min	col_max
1.parquet	6	8
2.parquet	3	10
3.parquet	1	4

Data Skipping

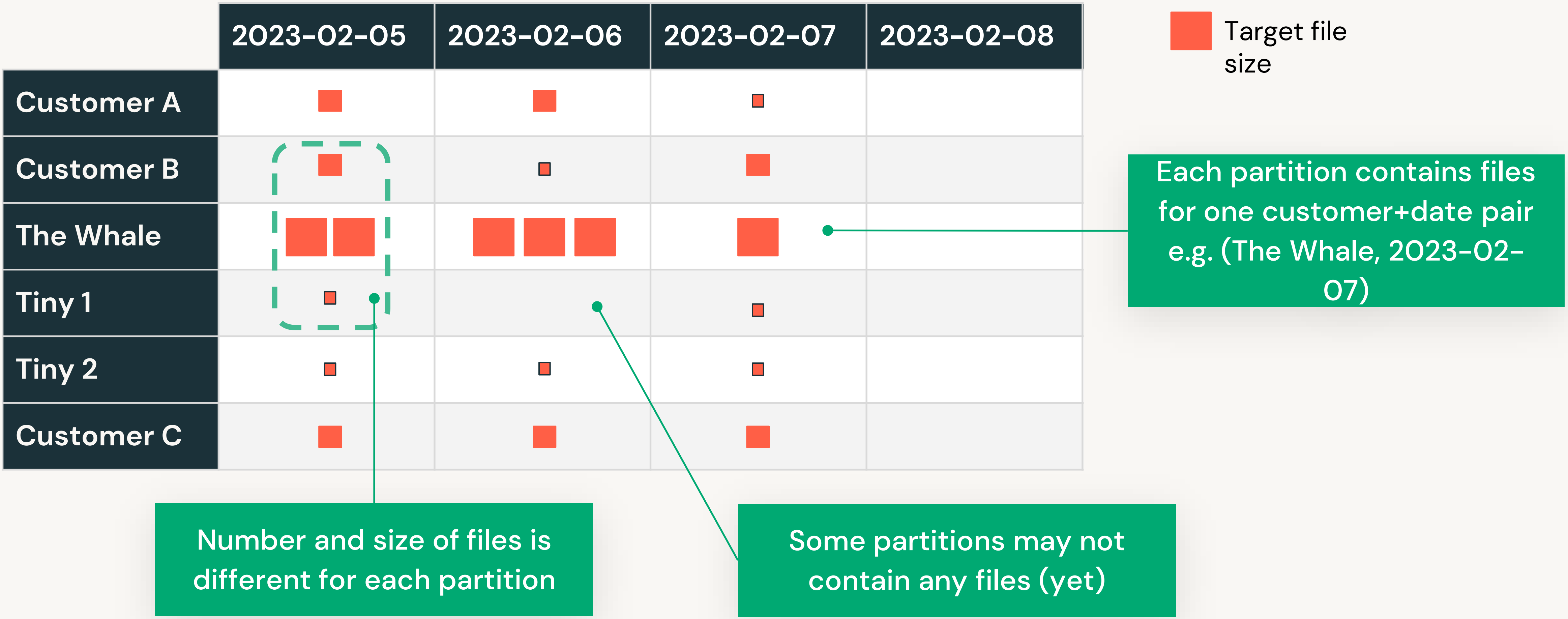
```
SELECT file_name FROM index
WHERE col_min < 5 AND col_max >= 5
```

file_name	col_min	col_max
1.parquet	6	8
2.parquet	3	10
3.parquet	1	4

Data Layout Challenges

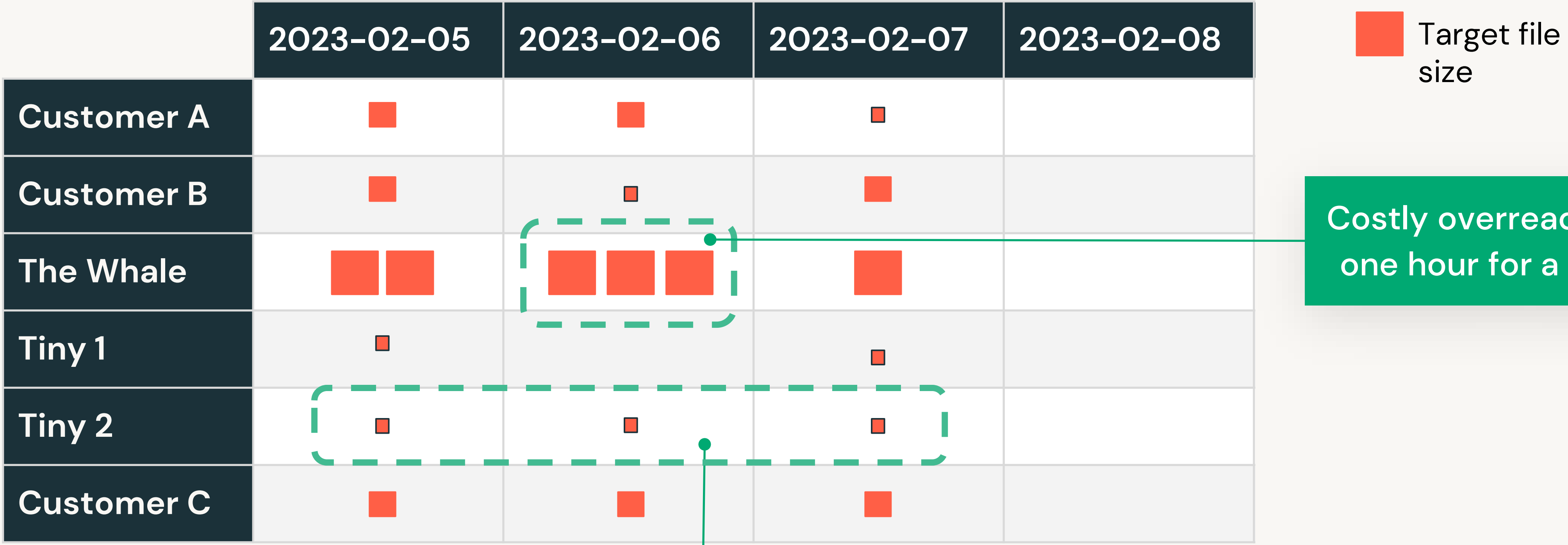
Hive-style partitioning

Working example: A table partitioned by customer ID and date



Hive-style partitioning

A table can be over- or under-partitioned — or both at the same time!



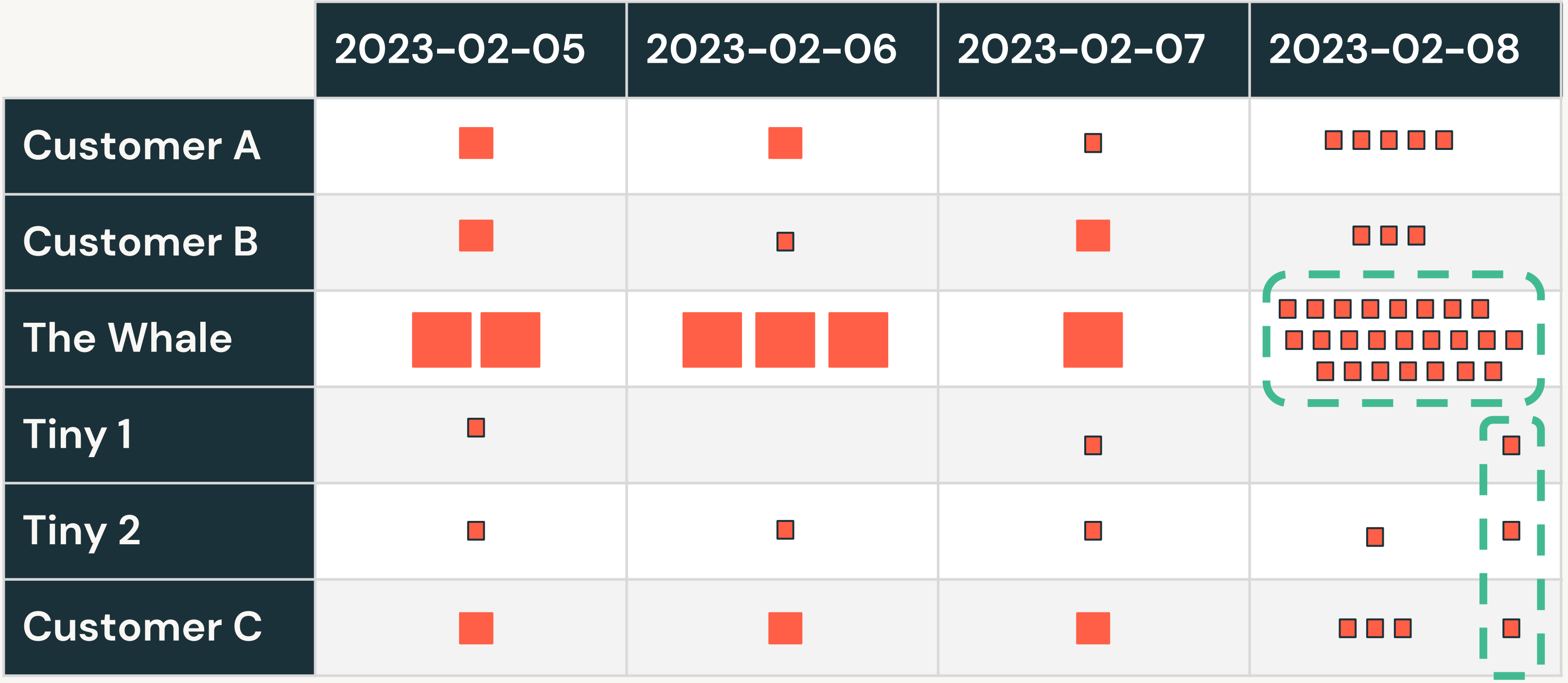
Costly overread to query just one hour for a big customer

Costly small-file explosion to query a small customer

Prefer to partition by week? By hour? Have to rewrite the whole table

Hive-style partitioning

Most ingest is small, causing small-file explosion



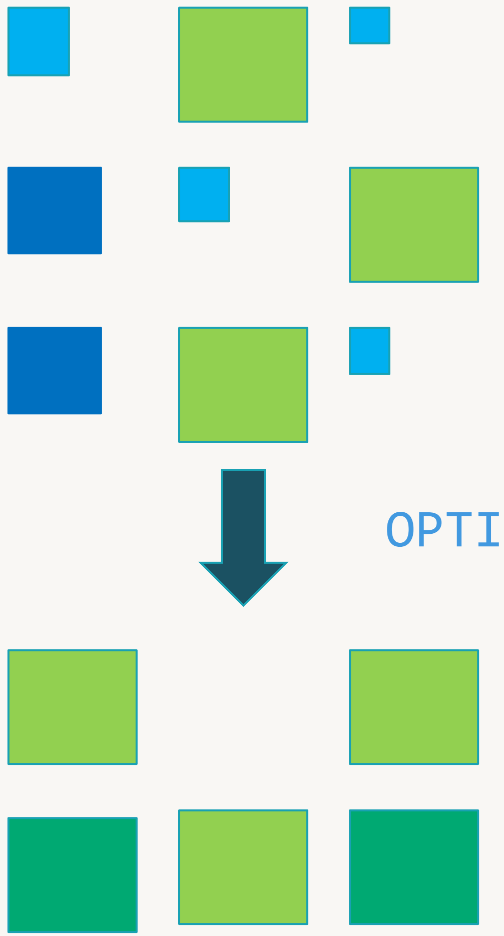
Target file size

Ingest new data every hour?
24 files per customer/day.

Ingest small data for many customers at once?
One tiny file per customer.

Frequent table maintenance needed to control file counts

OPTIMIZE your table



OPTIMIZE my_table

```
{
  "remove": {
    "path": "part-00001-9…..snappy.parquet",
    "deletionTimestamp": 1512909768000,
    "dataChange": false
         }
}
{
  "add": {
    "path": "part-0000…..gz.parquet",
    "size": 256841454,
    "modificationTime": 1512909768000,
    "dataChange": false,
    "stats":
"{
       \"numRecords\":123456789,\"minValues\":{\"val..."
  }
}
```

Z-Ordering

```
optimize my_table zorder by col
```

Old Layout

file_name	col_min	col_max
1.parquet	6	8
2.parquet	3	10
3.parquet	1	4

New Layout

file_name	col_min	col_max
1.parquet	1	3
2.parquet	4	7
3.parquet	8	10

Z-Ordering

```
select * from table where col = 7
```

Old Layout

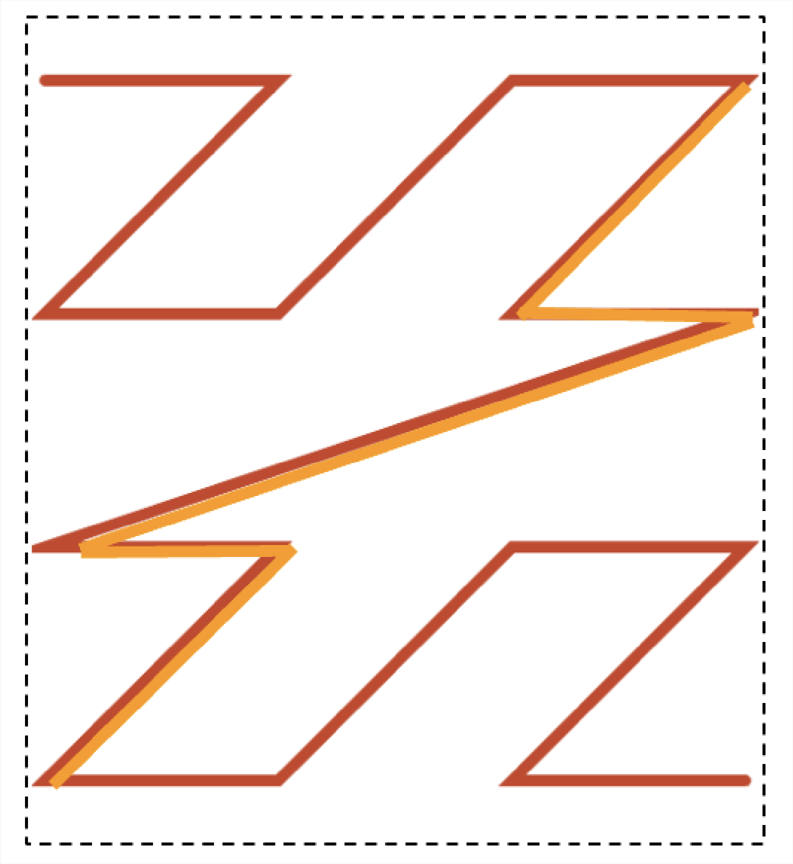
file_name	col_min	col_max
1.parquet	6	8
2.parquet	3	10
3.parquet	1	4

New Layout

file_name	col_min	col_max
1.parquet	1	3
2.parquet	4	7
3.parquet	8	10

Challenges with Z-order

- Due to the span of the Z-Curve, some files will have min/max range equal to the full range, and data skipping can't skip these files.
- Any new data ingested after the OPTIMIZE ZORDER BY run is **not automatically clustered**, and the user needs to rerun the command to cluster the new data.
- OPTIMIZE ZORDER BY reclusters already well-clustered data, resulting in **high write amplification**.
- ZORDER BY columns are not persisted and the user is required to remember the previous ZORDER BY columns, often causing user errors if different columns are used



Z-Curve

Data Layout Innovations

Liquid Clustering – No more partitions

- Fast
  - Faster writes and similar reads vs. well-tuned partitioned tables
- Self-tuning
  - Avoids over- and under-partitioning
- Incremental
  - Automatic partial clustering of new data
- Skew-resistant
  - Produces consistent file sizes and low write amplification
- Flexible
  - Want to change the clustering columns? No problem!
- Better concurrency

Liquid clustering Usage Walkthrough

Create a new Delta table with liquid clustering

```
CREATE [EXTERNAL] TABLE tbl (id INT, name STRING) CLUSTER BY(id)
```

Change Liquid Clustering keys on existing clustered table:

```
ALTER TABLE tbl CLUSTER BY (name);
```

Clustering data in a Delta table with liquid clustering:

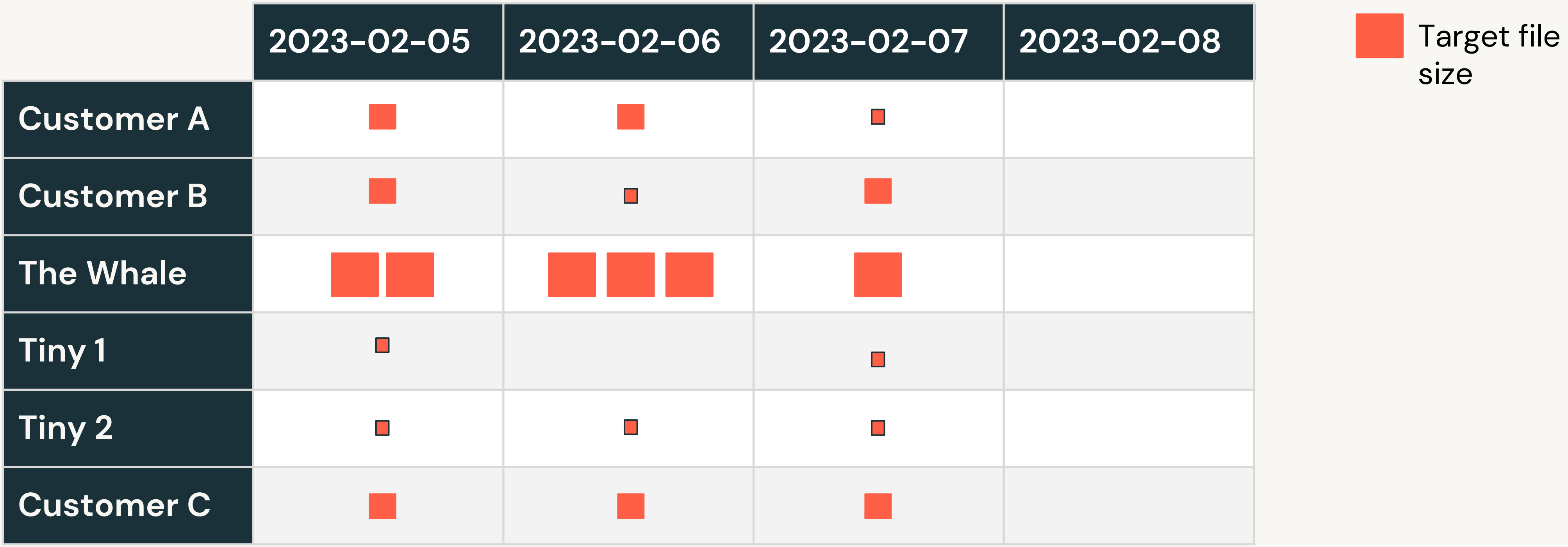
```
OPTIMIZE tbl;
```

What you don’t need to worry about:

- Optimal file sizes
- Whether a column can be used as a clustering key
- Order of clustering keys

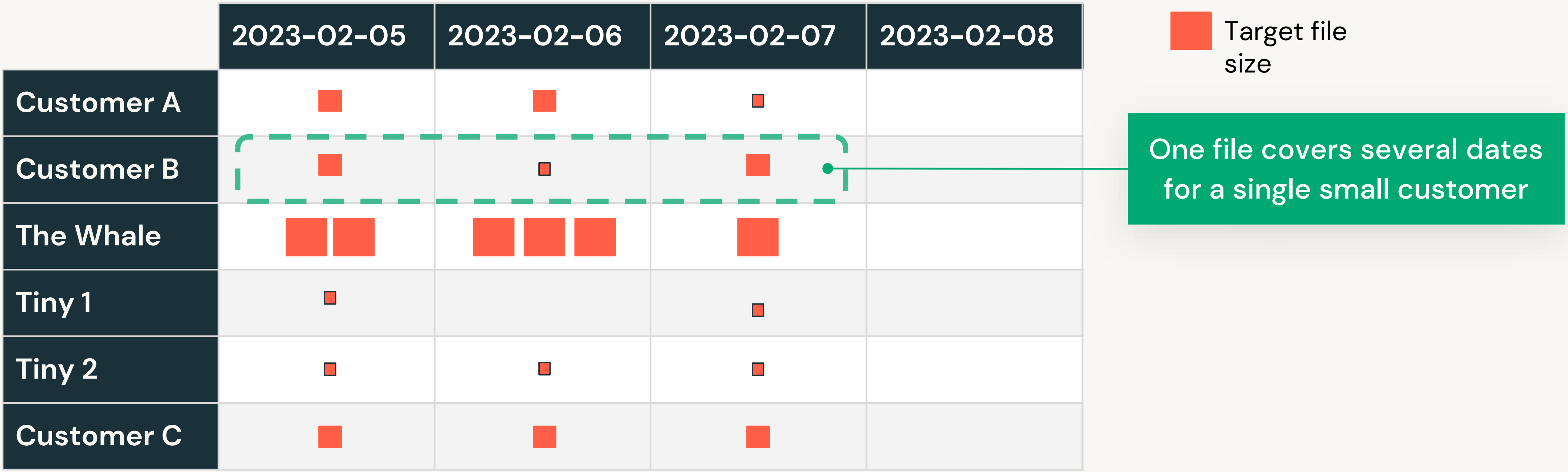
Liquid clustering in action

Efficiently balance clustering vs. file size



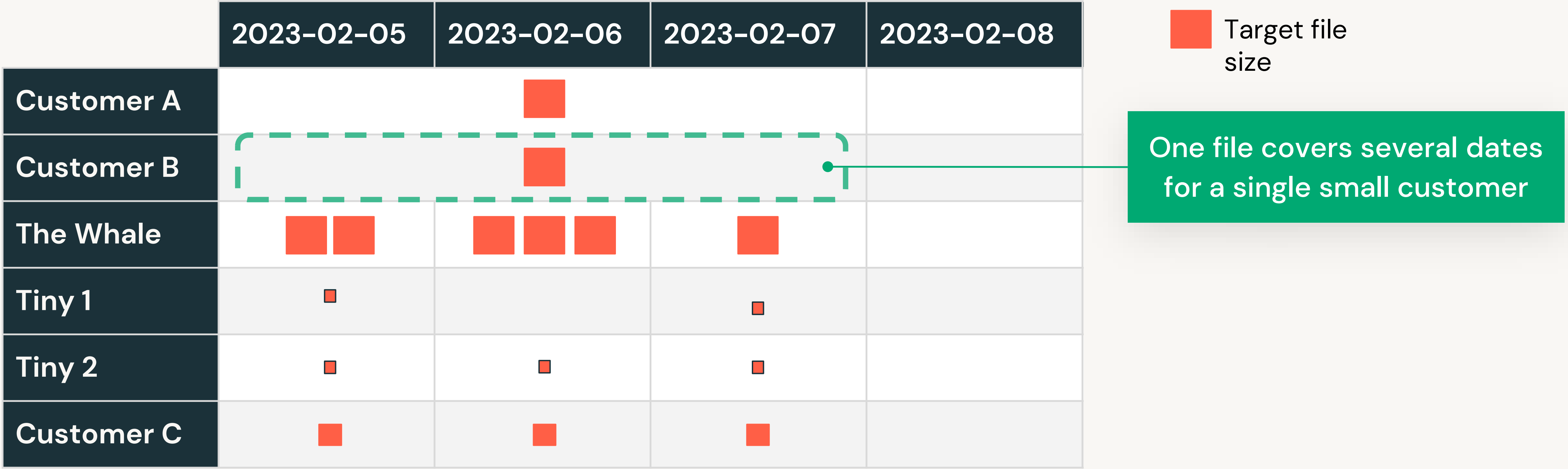
Liquid clustering in action

Efficiently balance clustering vs. file size



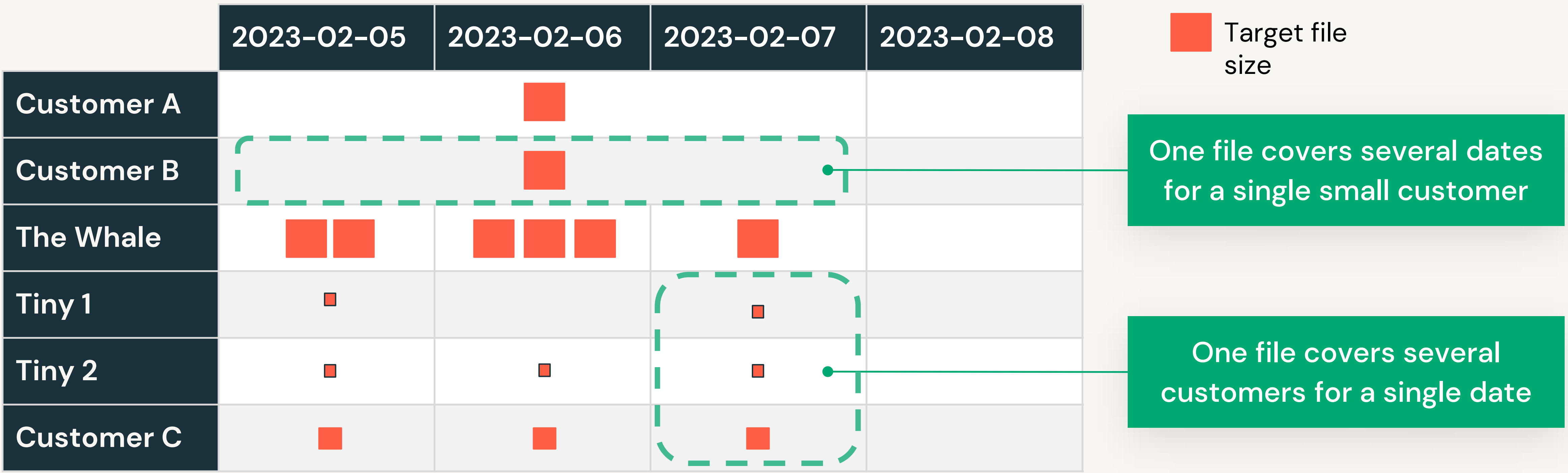
Liquid clustering in action

Efficiently balance clustering vs. file size



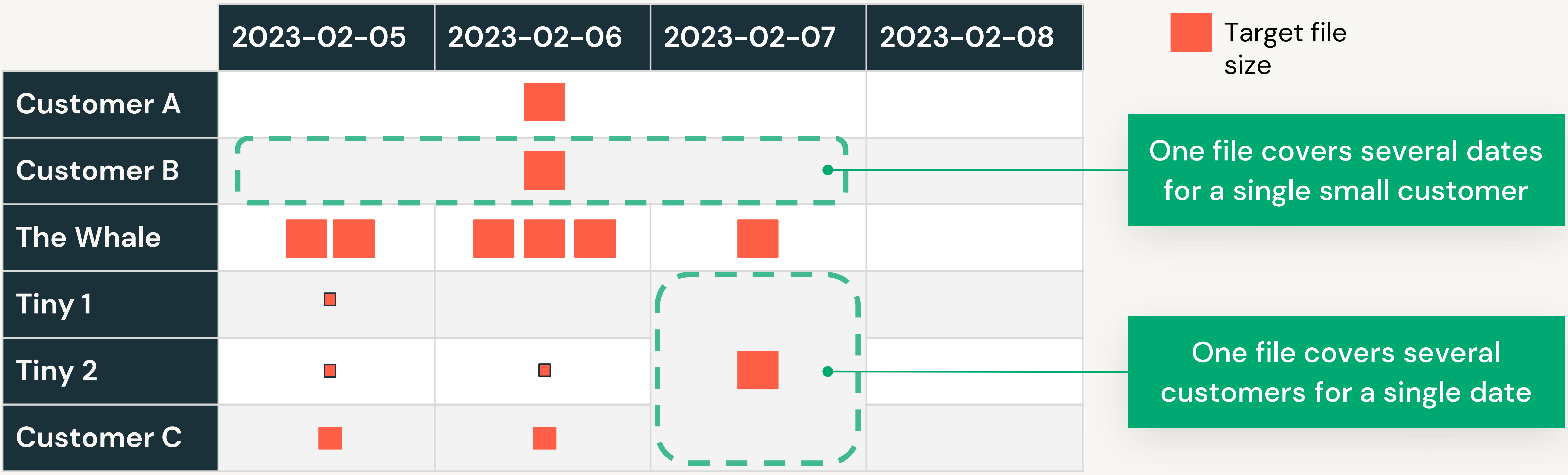
Liquid clustering in action

Efficiently balance clustering vs. file size



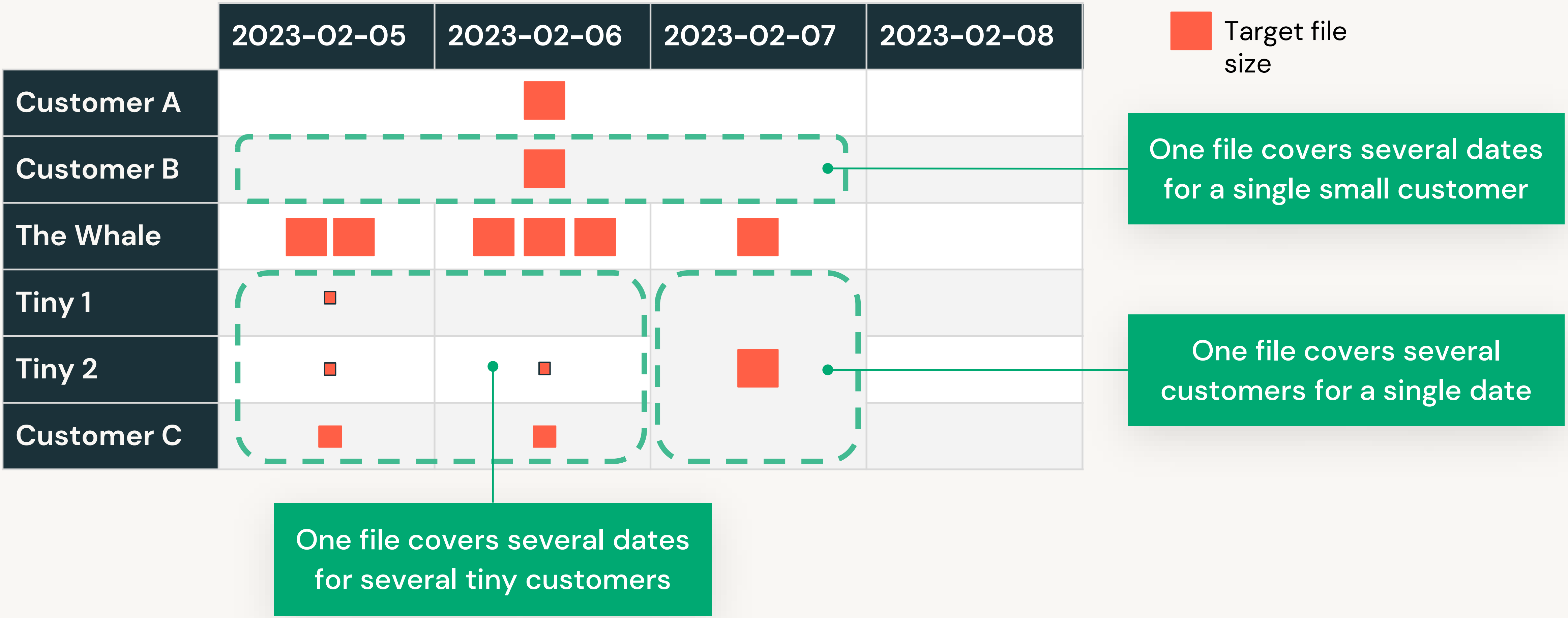
Liquid clustering in action

Efficiently balance clustering vs. file size



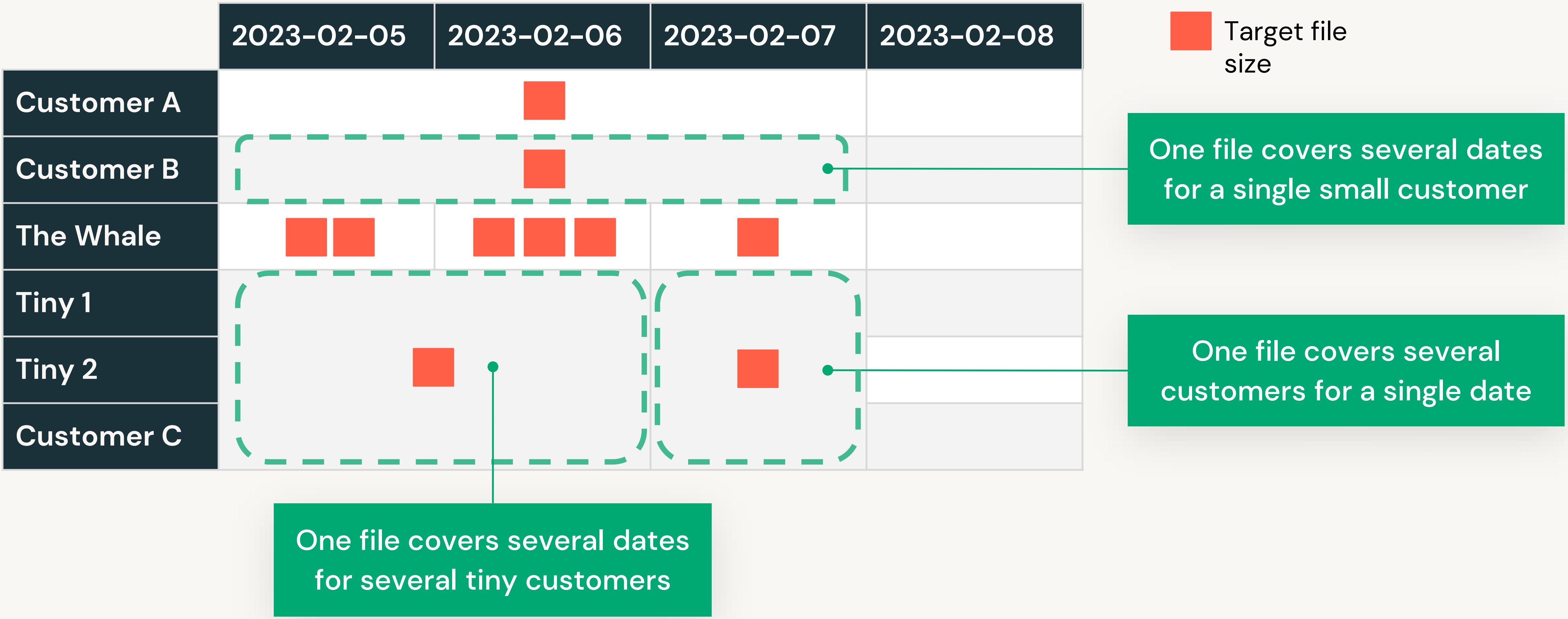
Liquid clustering in action

Efficiently balance clustering vs. file size



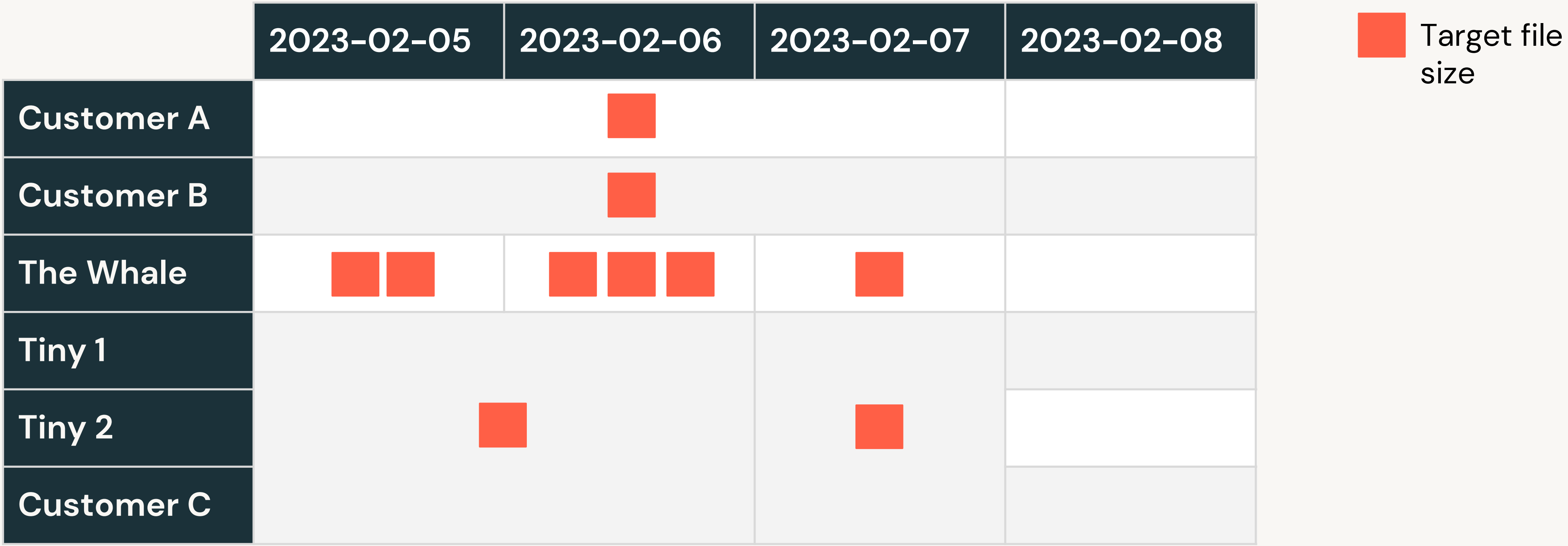
Liquid clustering in action

Efficiently balance clustering vs. file size



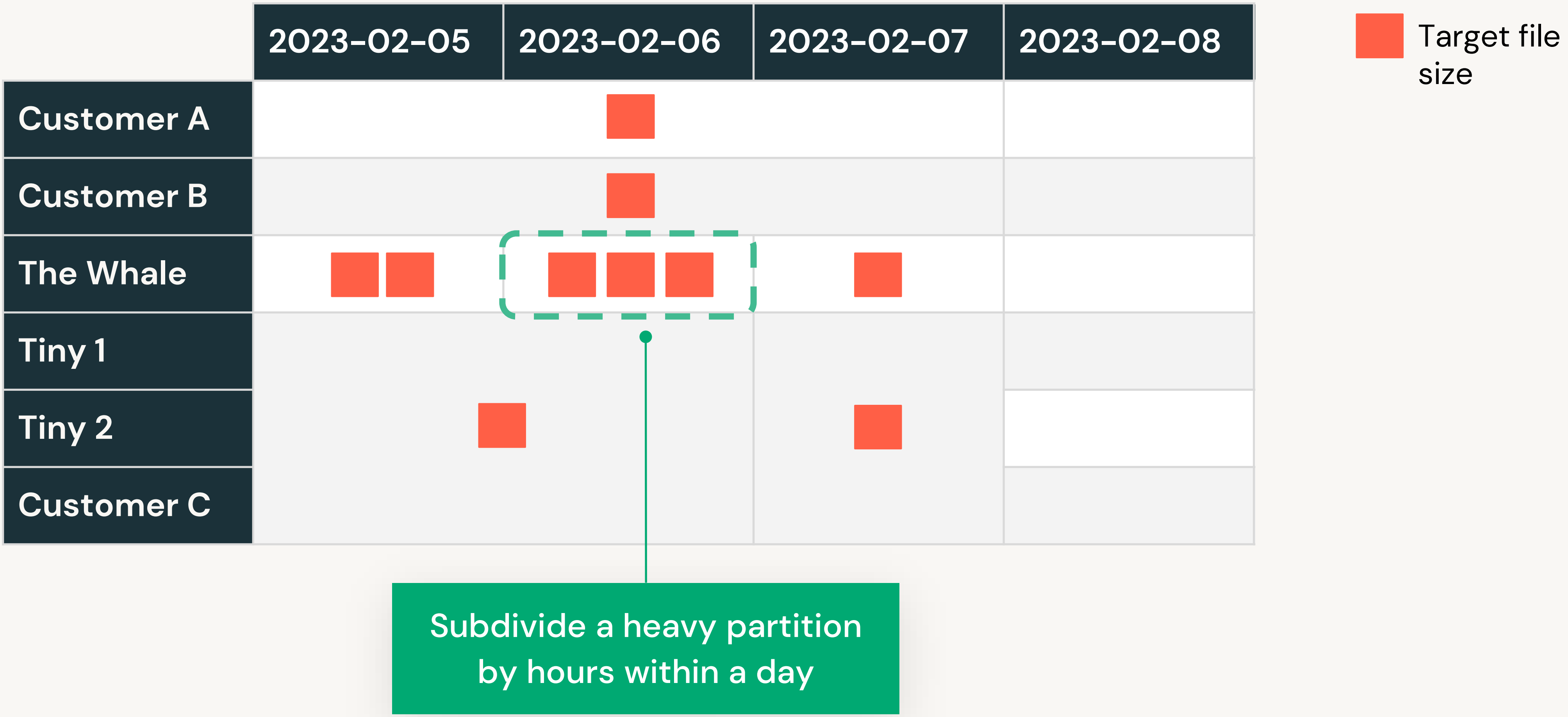
Liquid clustering in action

But wait, there’s more!



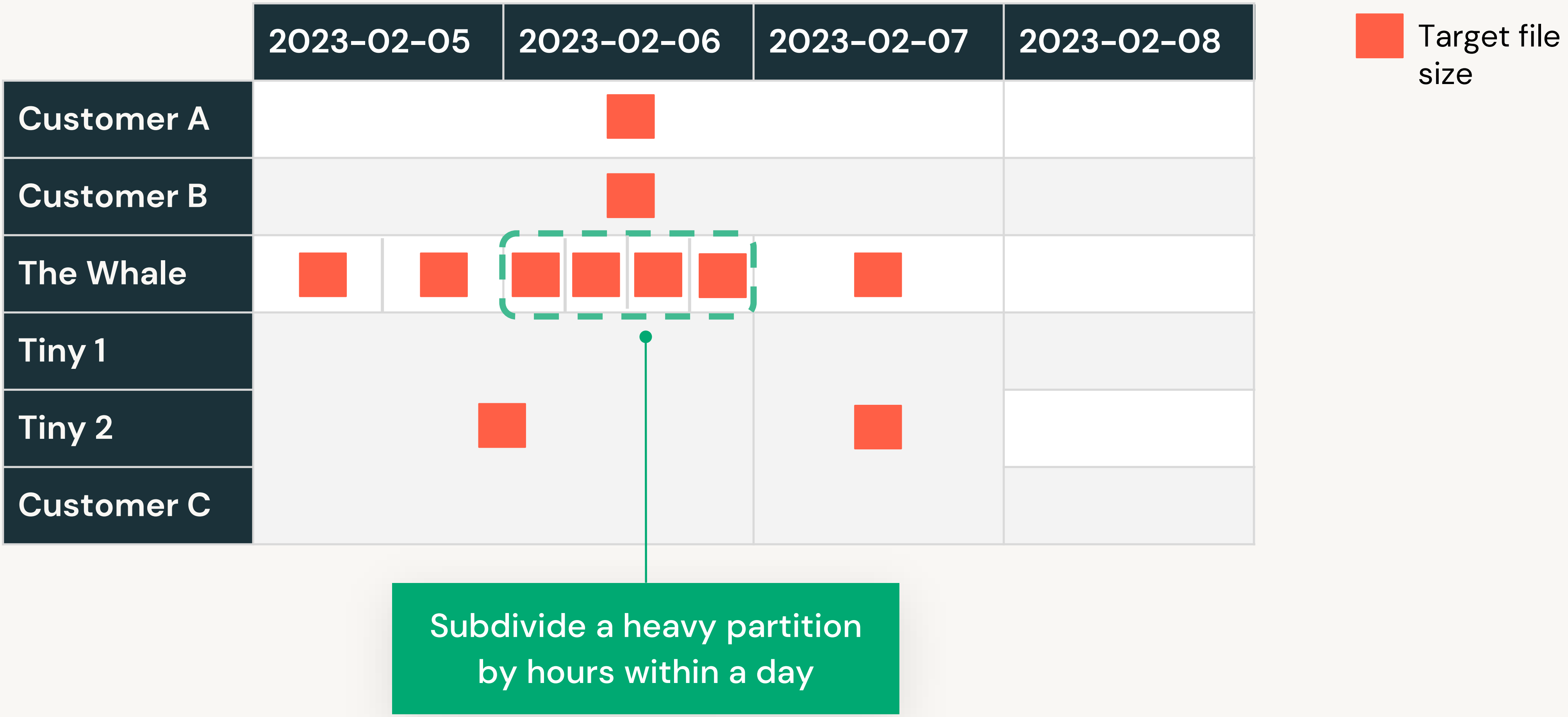
Liquid clustering in action

Automatically cluster heavy partitions more finely



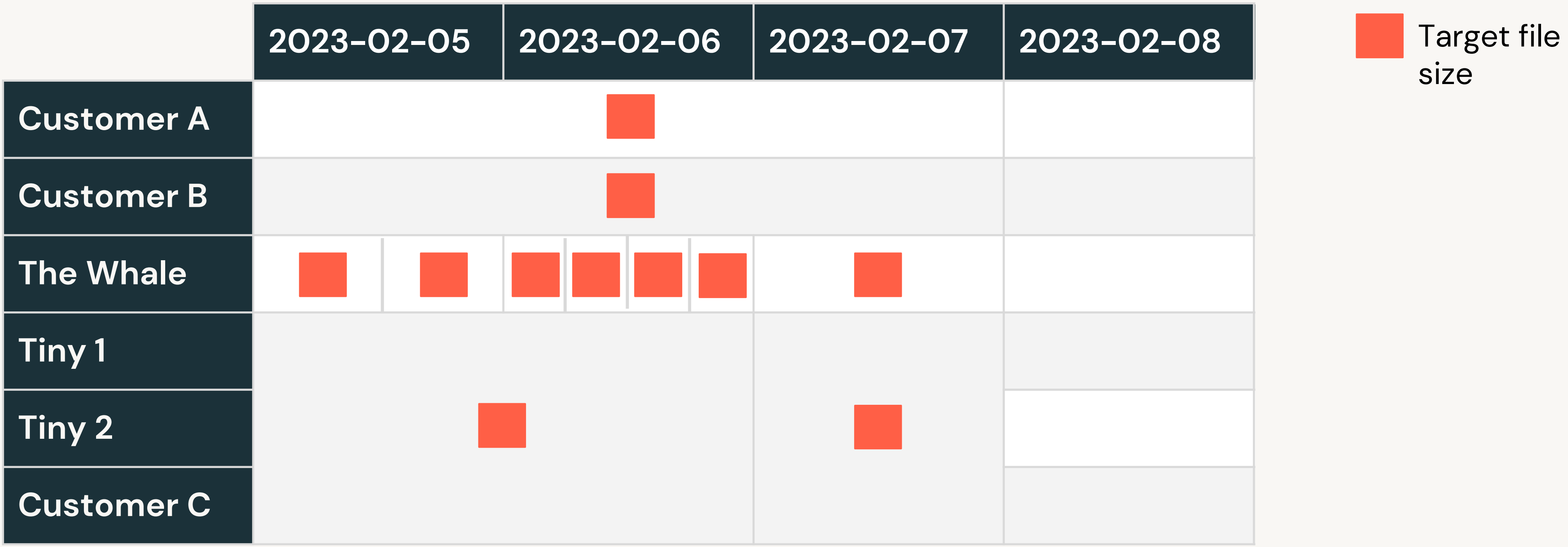
Liquid clustering in action

Automatically cluster heavy partitions more finely



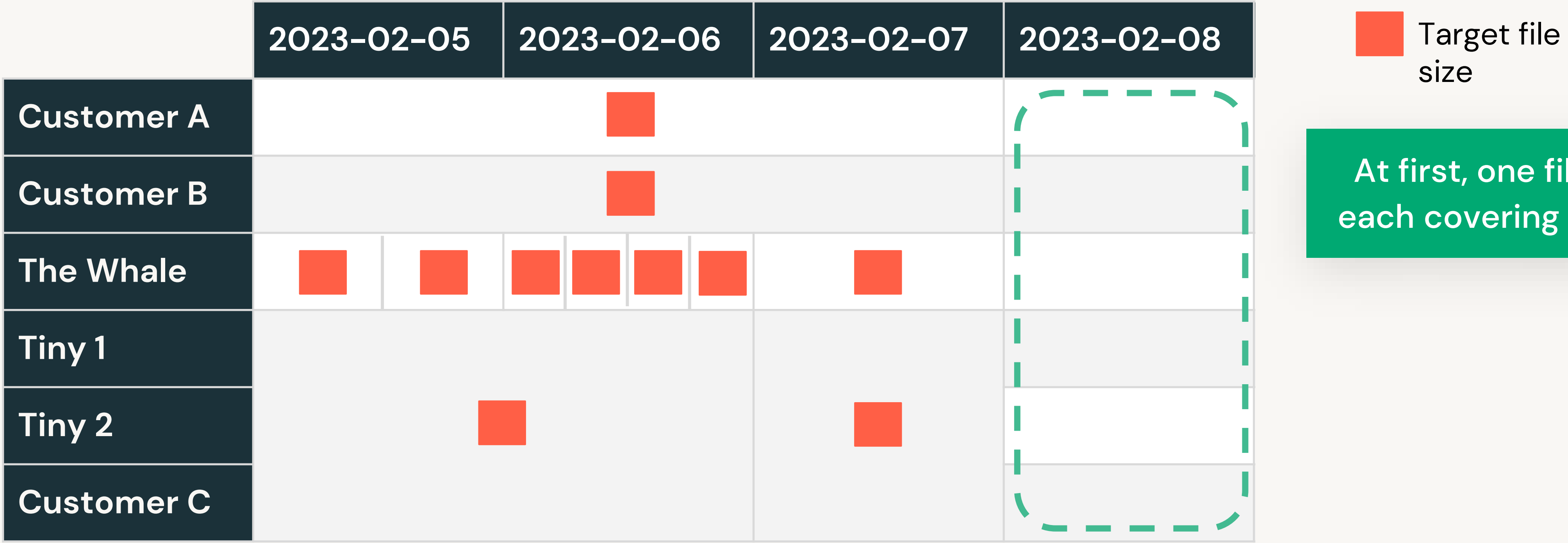
Liquid clustering in action

But wait, there's more!



Liquid clustering in action

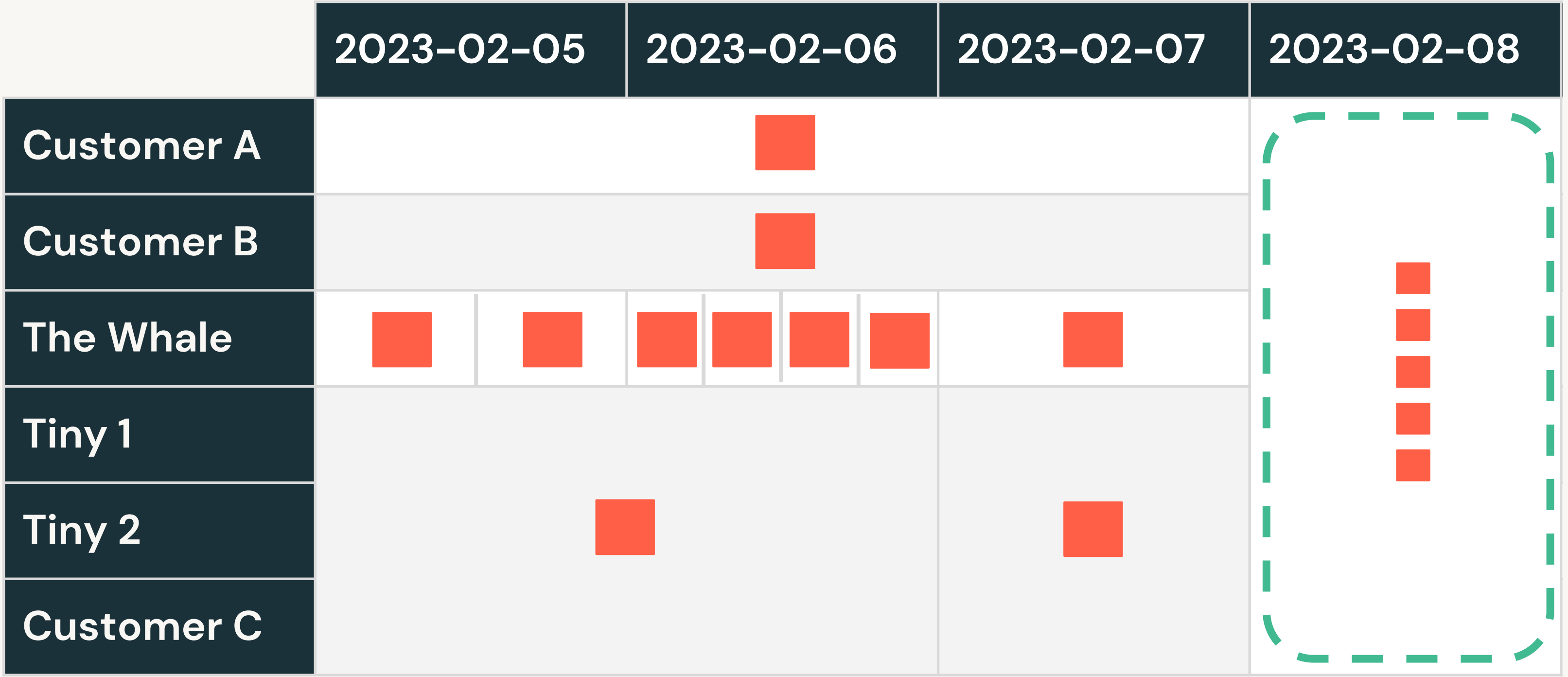
Efficient ingest with lazy/partial clustering



At first, one file per ingest, each covering all customers

Liquid clustering in action

Efficient ingest with lazy/partial clustering

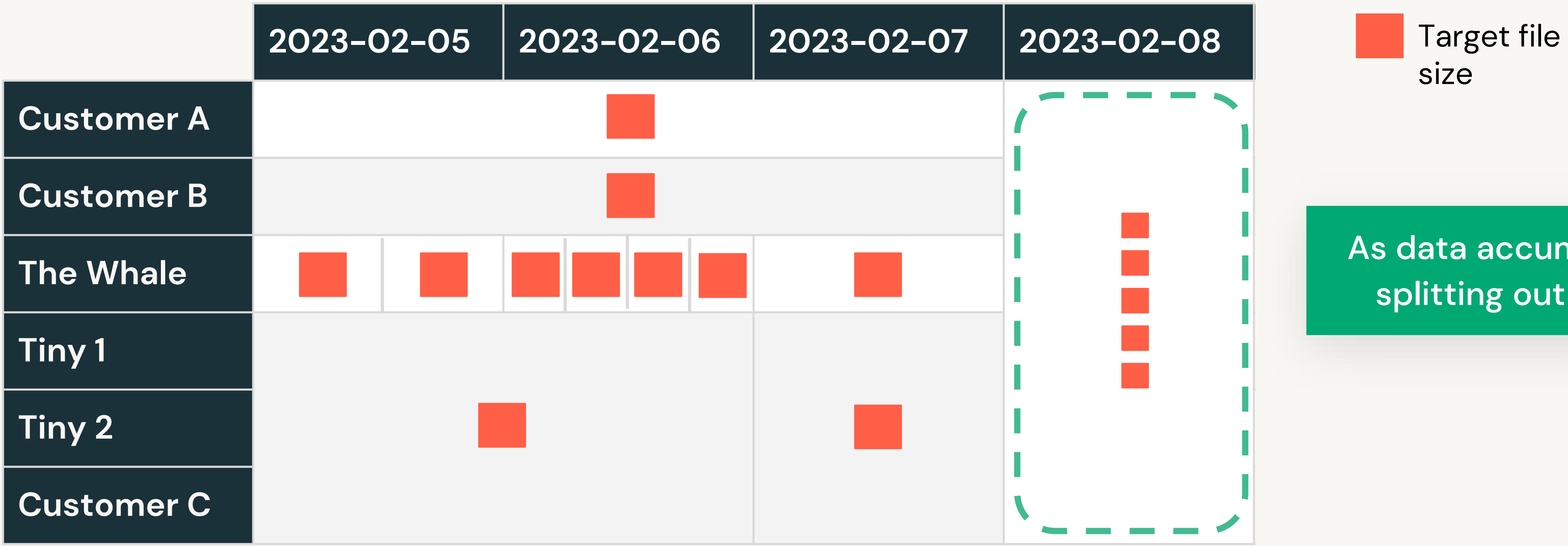


Target file size

At first, one file per ingest, each covering all customers

Liquid clustering in action

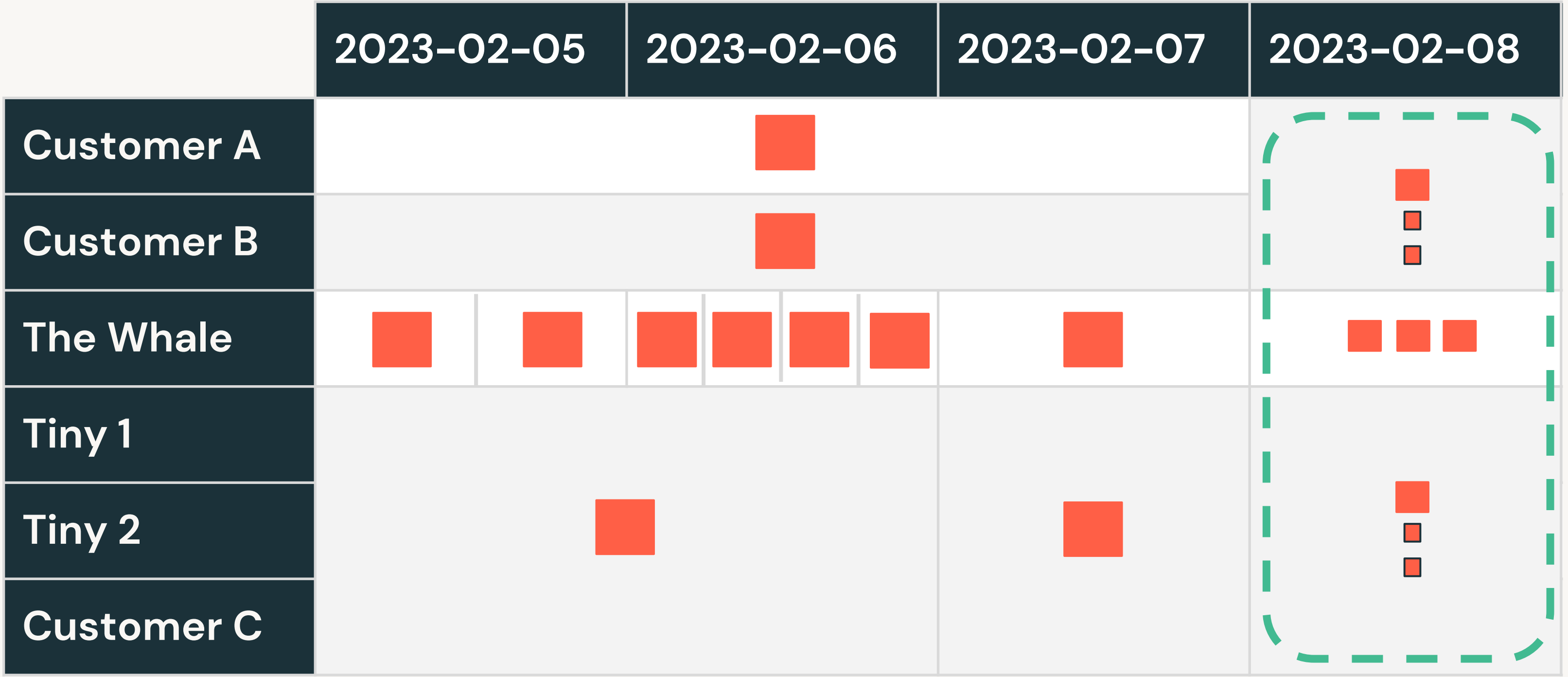
Efficient ingest with lazy/partial clustering



As data accumulates, start splitting out customers

Liquid clustering in action

Efficient ingest with lazy/partial clustering

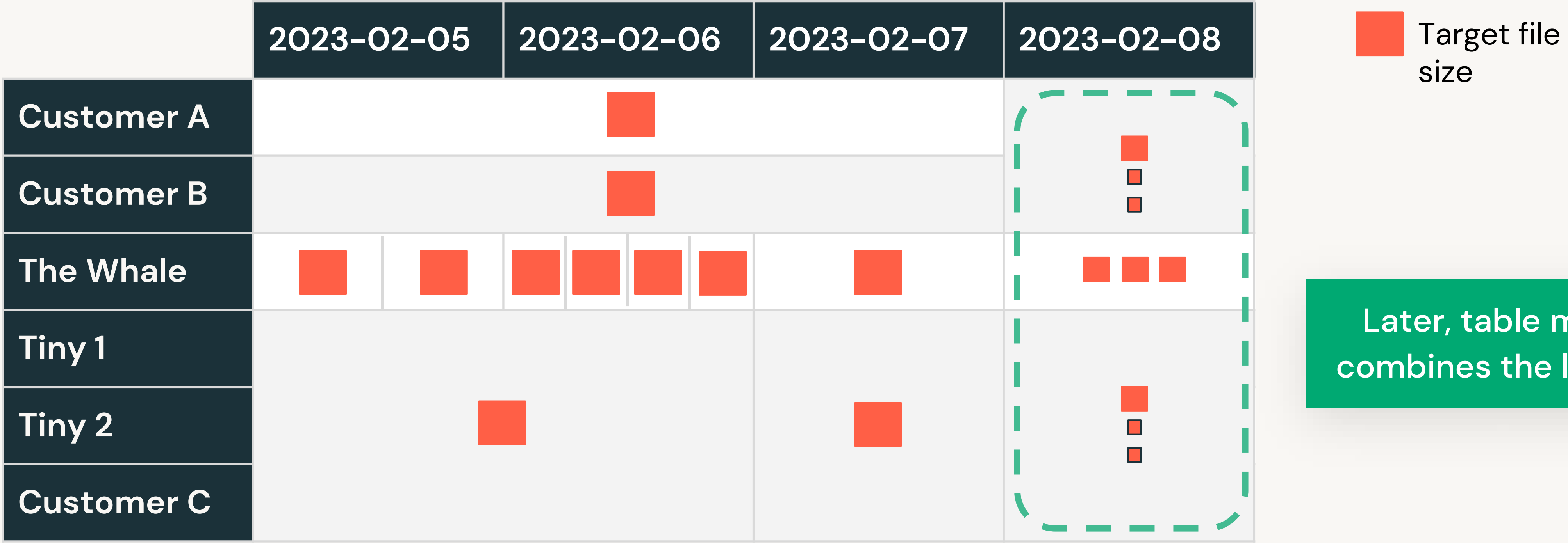


Target file size

As data accumulates, start splitting out customers

Liquid clustering in action

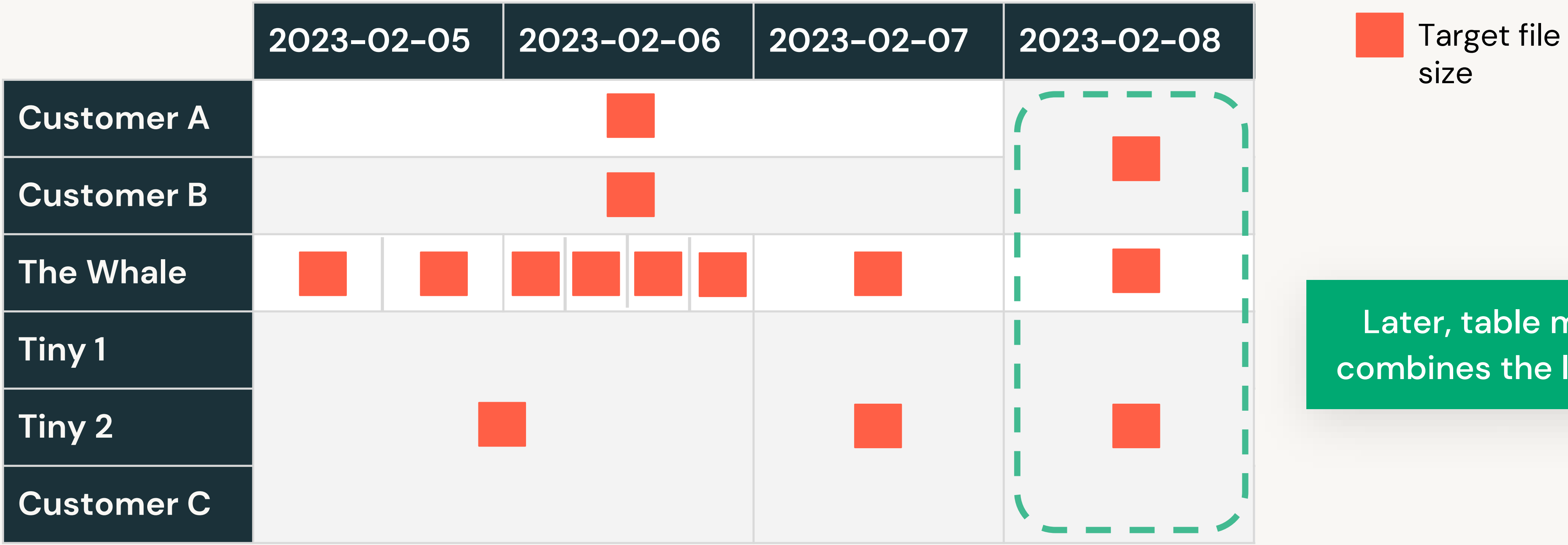
Efficient ingest with lazy/partial clustering



Later, table maintenance combines the last small files

Liquid clustering in action

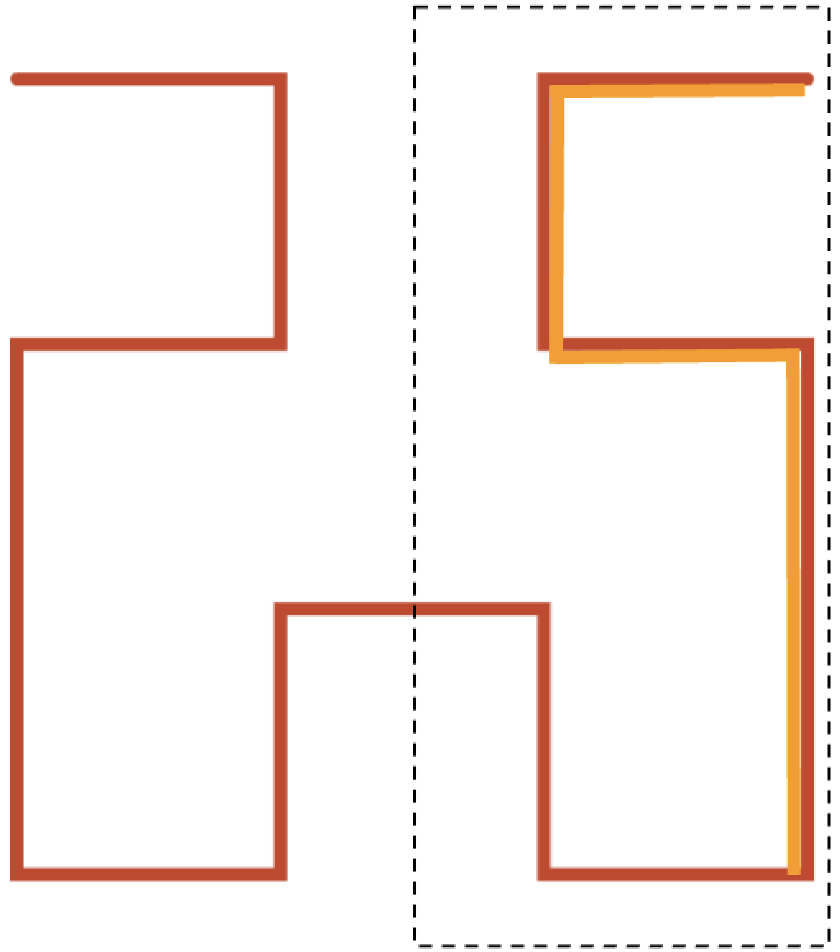
Efficient ingest with lazy/partial clustering



Later, table maintenance combines the last small files

Liquid under-the-hood

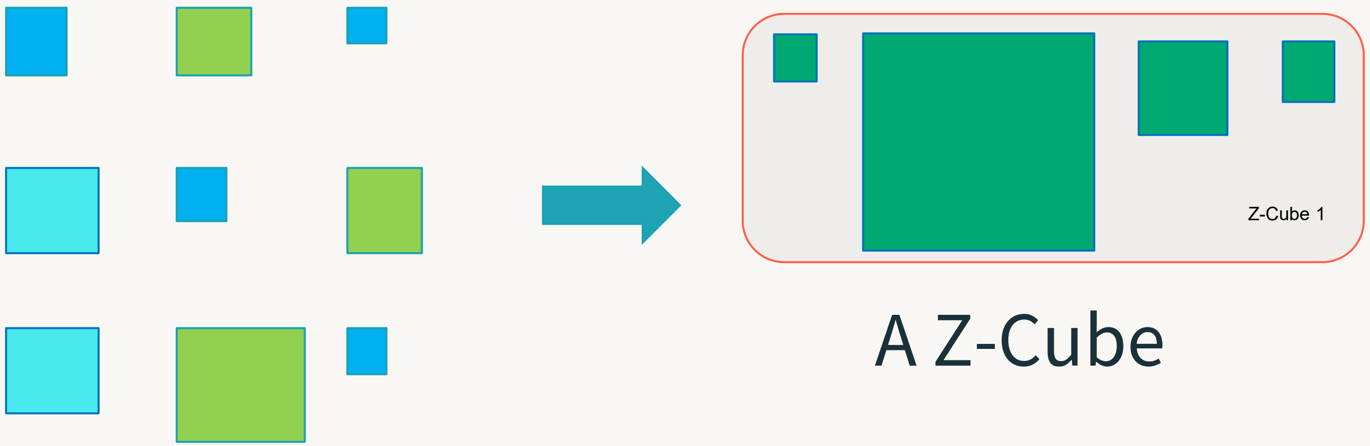
Better data-skipping due to hilbert curves



Hilbert Curve: The bounding box for the orange line spans only half the space

Liquid clustering is incremental

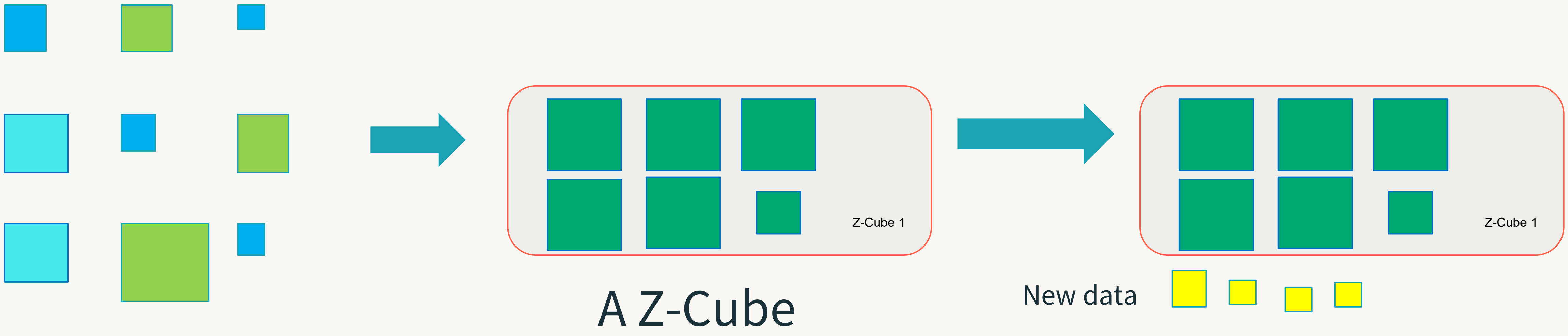
OPTIMIZE my_table

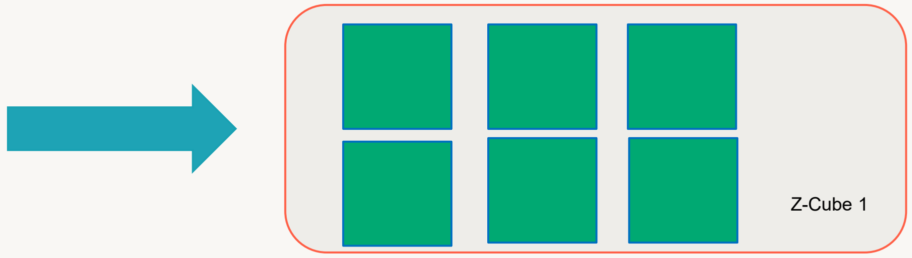


A Z-Cube

Liquid clustering is incremental

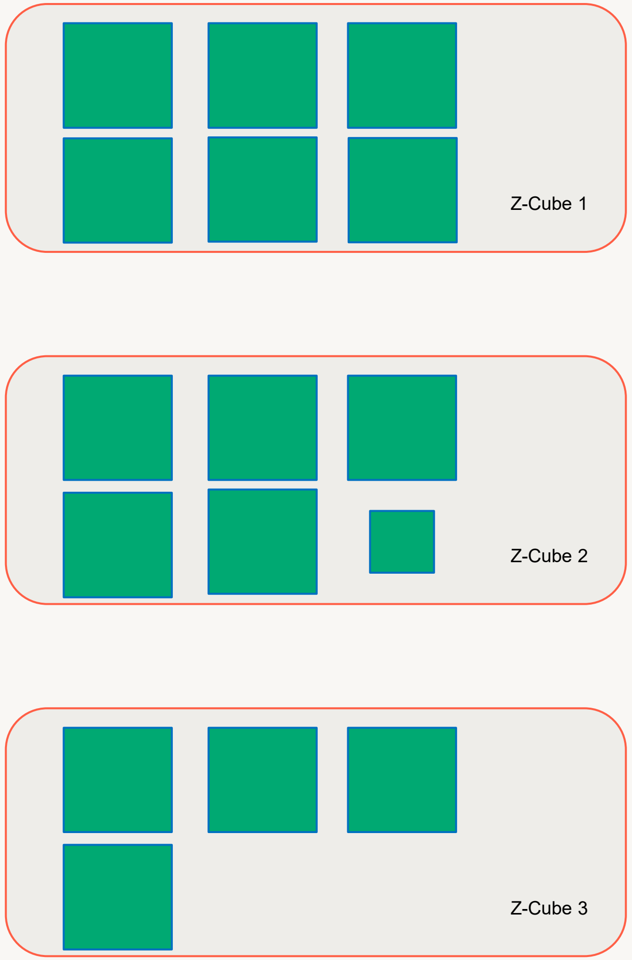
```
OPTIMIZE my_table
```





Incorporate new data into existing Z-cubes if cubes < 150 GB

Tables can have many ZCubes



- When we get to 150gb, we start a new ZCube to minimize write amplification on Zorder
- When data is removed from ZCube, possibly due to DML, once the ZCube reaches a threshold its eligible for more data to be added to it

POP QUIZ

UniForm

Choosing a data lake format?

	Apache Hudi	Delta Lake	Apache Iceberg
Metadata Used for transactional source of truth, concurrency control, etc.	Metadata	Metadata	Metadata
Data All formats use Parquet!	Parquet	Parquet	Parquet
Connector Ecosystem	trino, Starburst, DORIS, ClickHouse, StarRocks	Microsoft Fabric, Apache kafka, Athena, amazon REDSHIFT, dbt	presto, snowflake, HIVE, Google Big Query, dremio

Delta UniForm

Write Delta, read as Iceberg

Delta Lake
With UniForm

Metadata
Used for transactional source of truth, concurrency control, etc.

Metadata

Data
All formats use Parquet!

Parquet

Connector Ecosystem

trino
Starburst
Microsoft Fabric
kafka
Athena
presto
snowflake
HIVE
DORIS
ClickHouse
StarRocks
amazon REDSHIFT
dbt
Google Big Query
dremio

How Delta Lake UniForm works

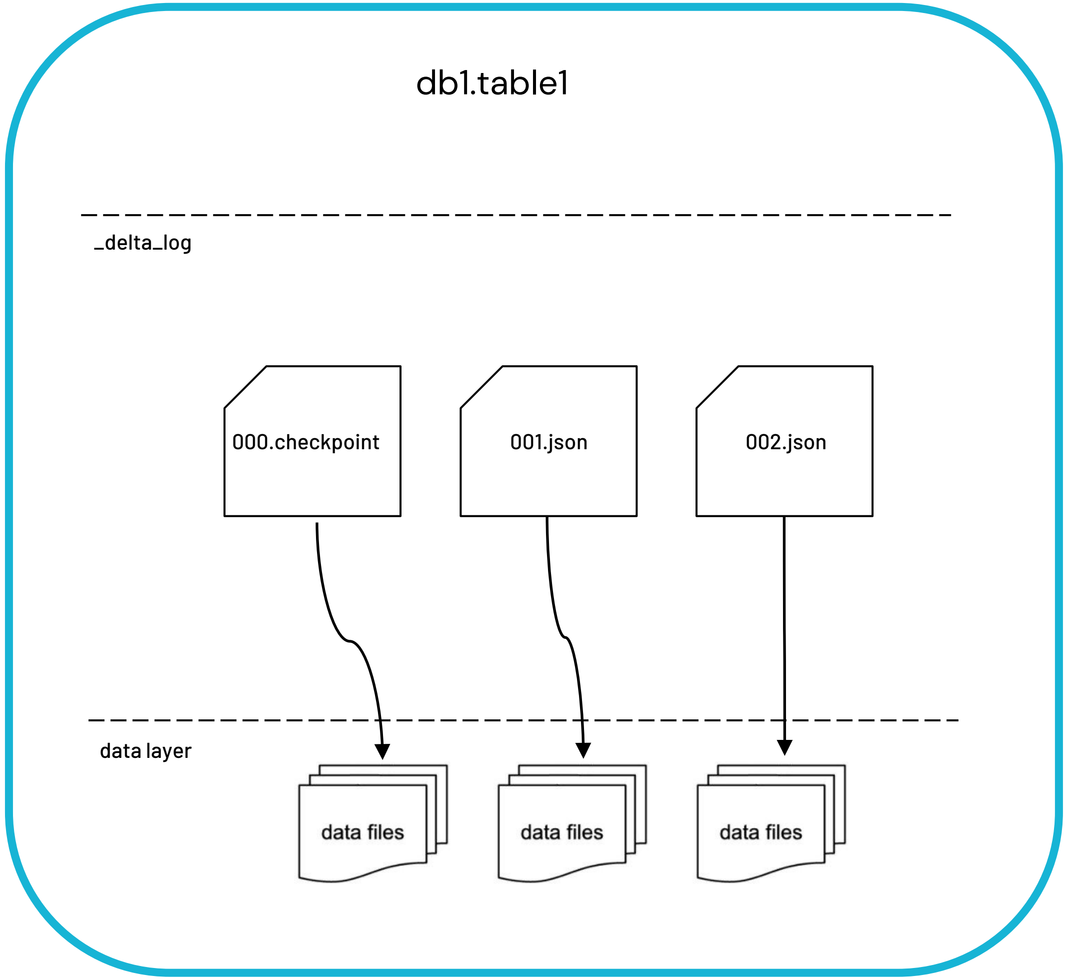
Delta Lake UniForm

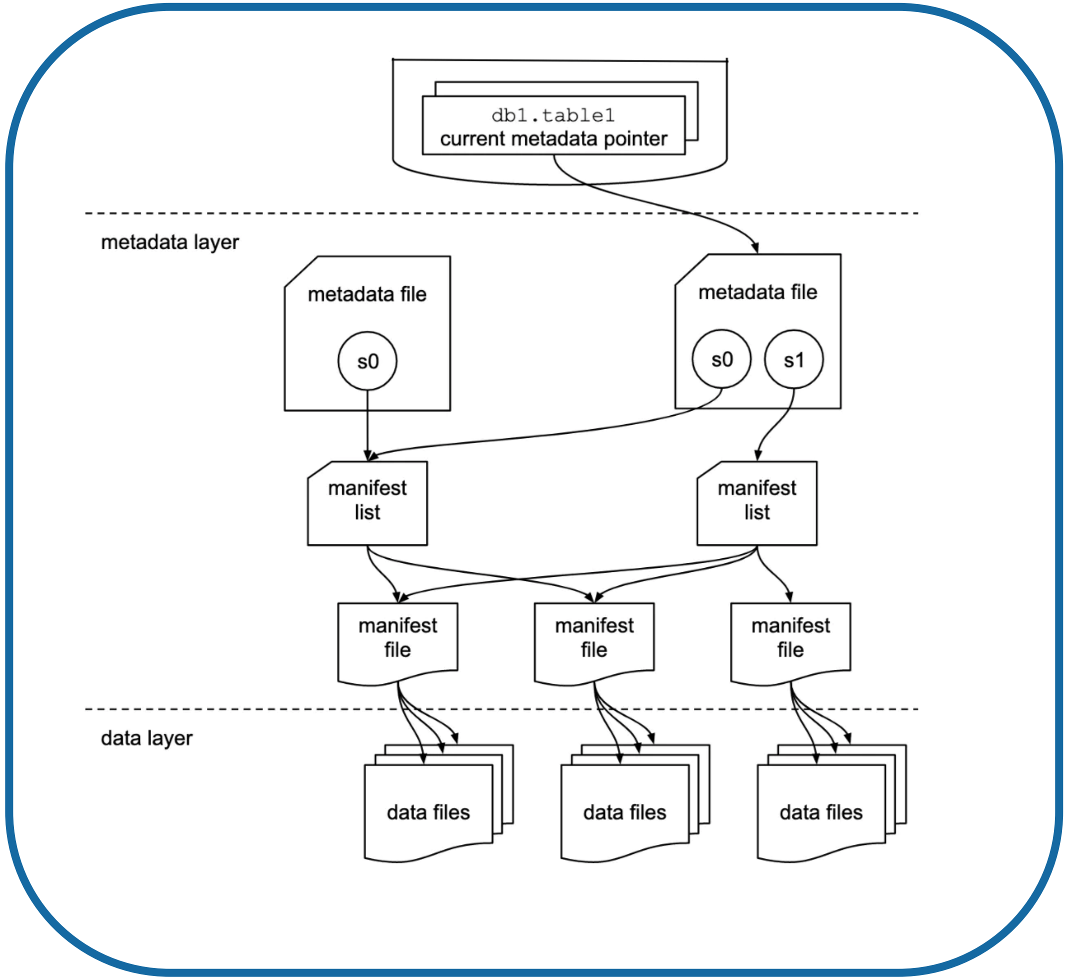
Data stored in Delta can be read
as if it were Iceberg or Hudi

- ✓ Metadata automatically generated to make Delta accessible as Iceberg/Hudi
- ✓ Parquet files remain the same
- ✓ Metadata is co-located with data

DELTA LAKE

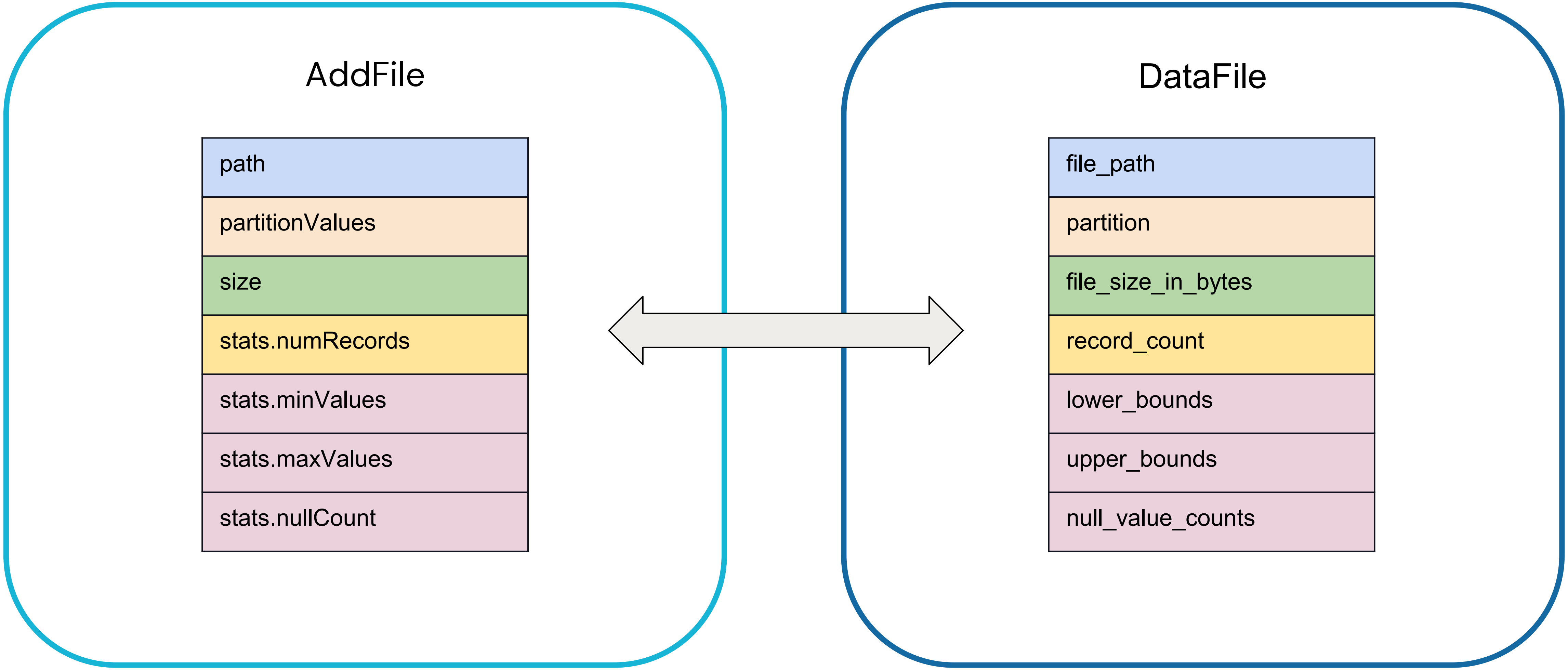
ICEBERG



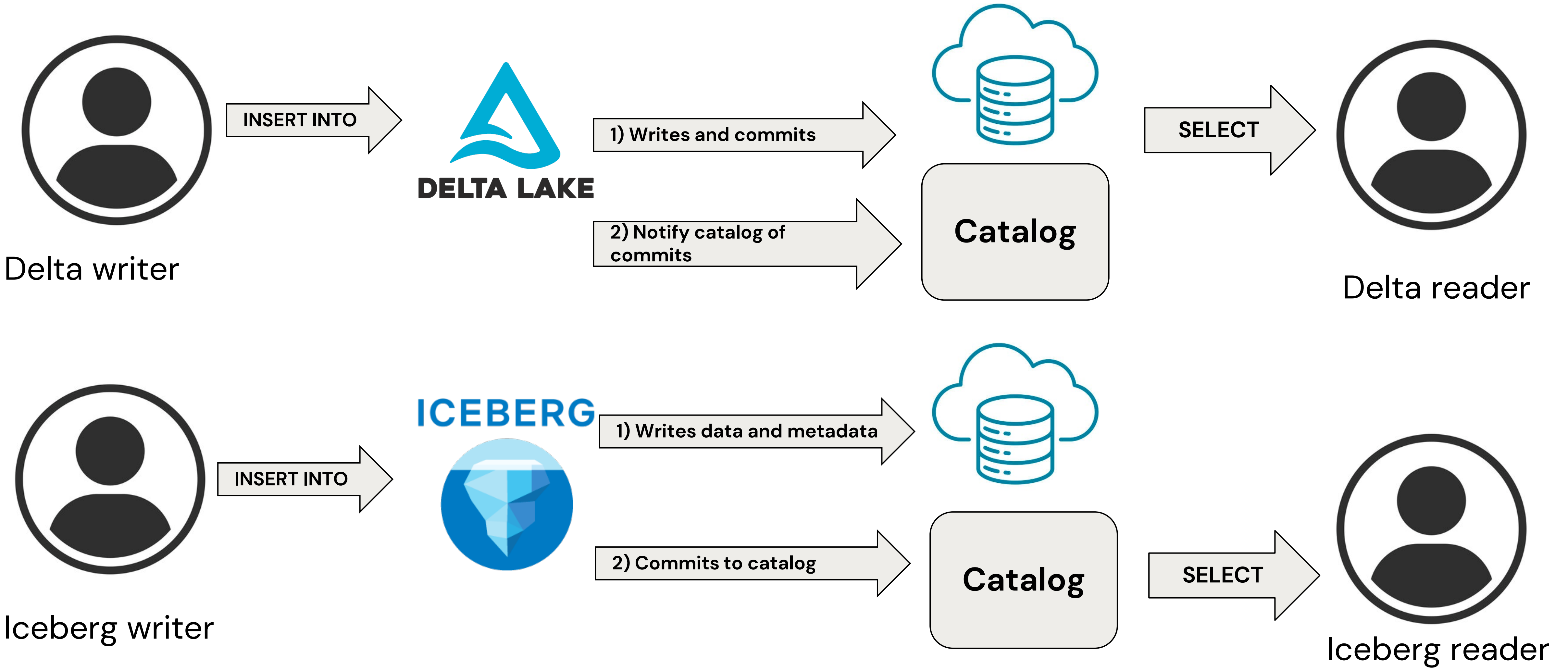


DELTA LAKE

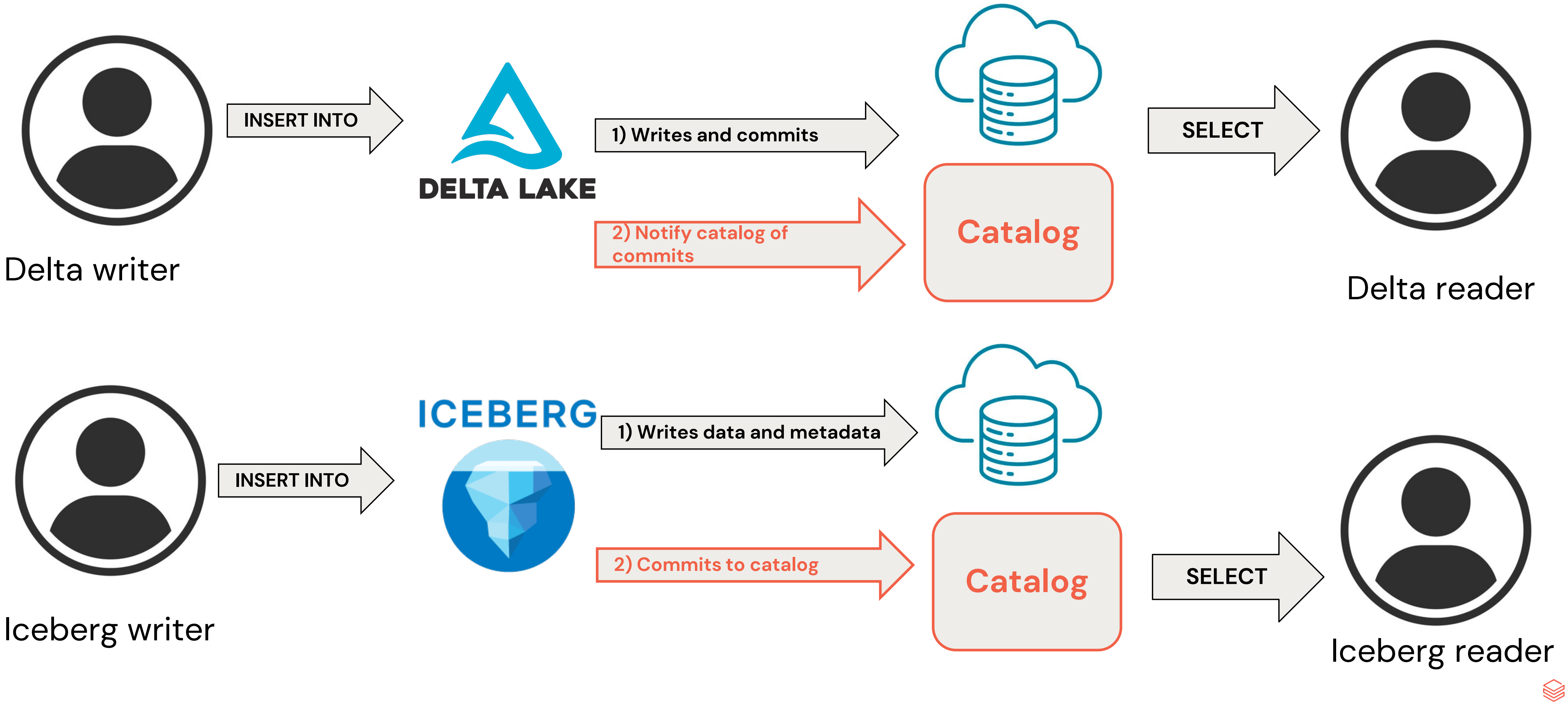
ICEBERG



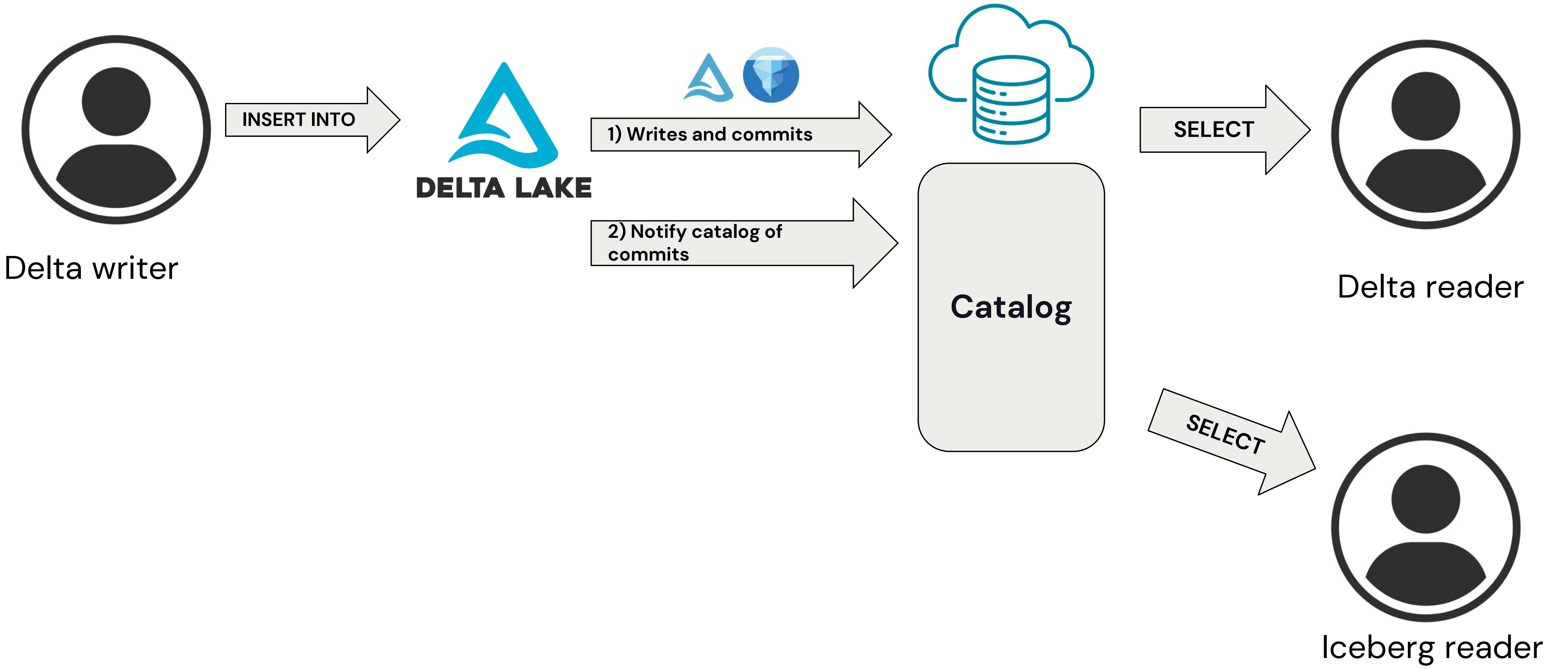
Observation: Very similar writes on both sides



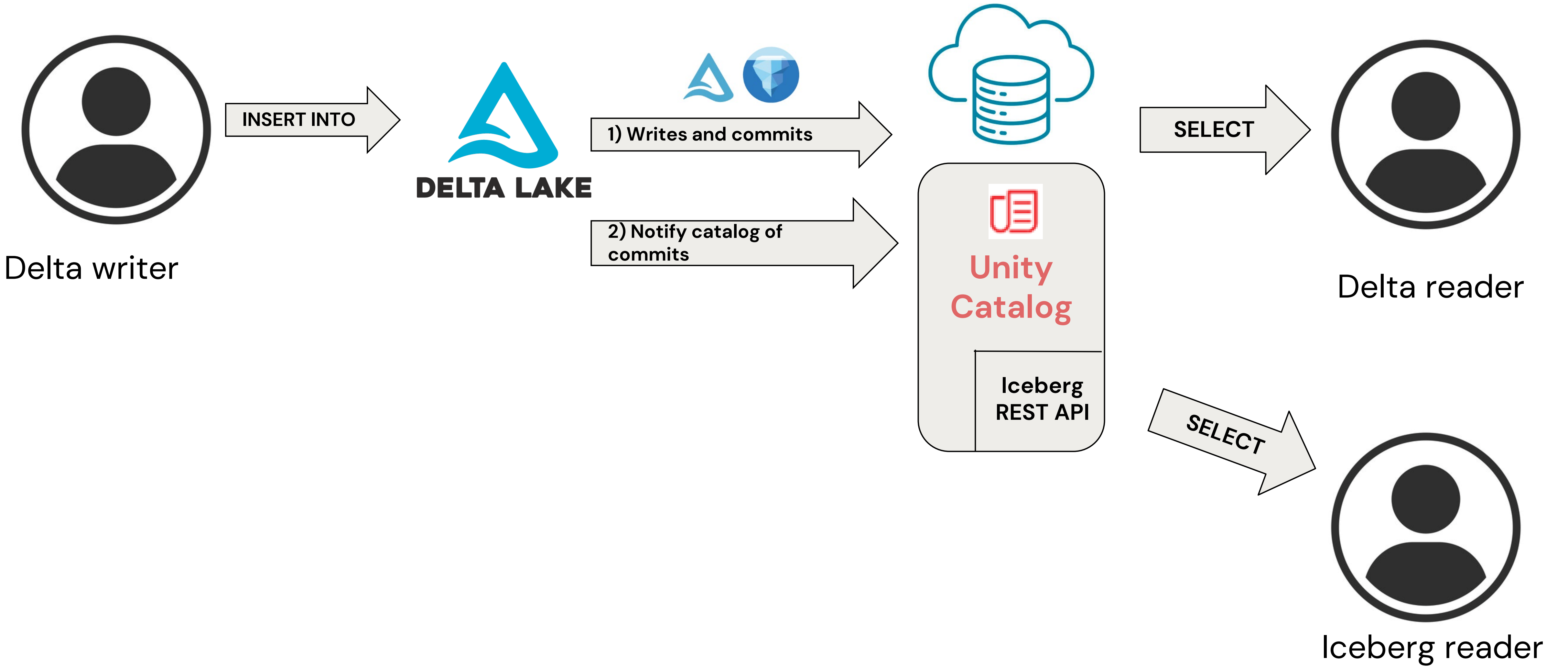
Observation: Very similar writes on both sides

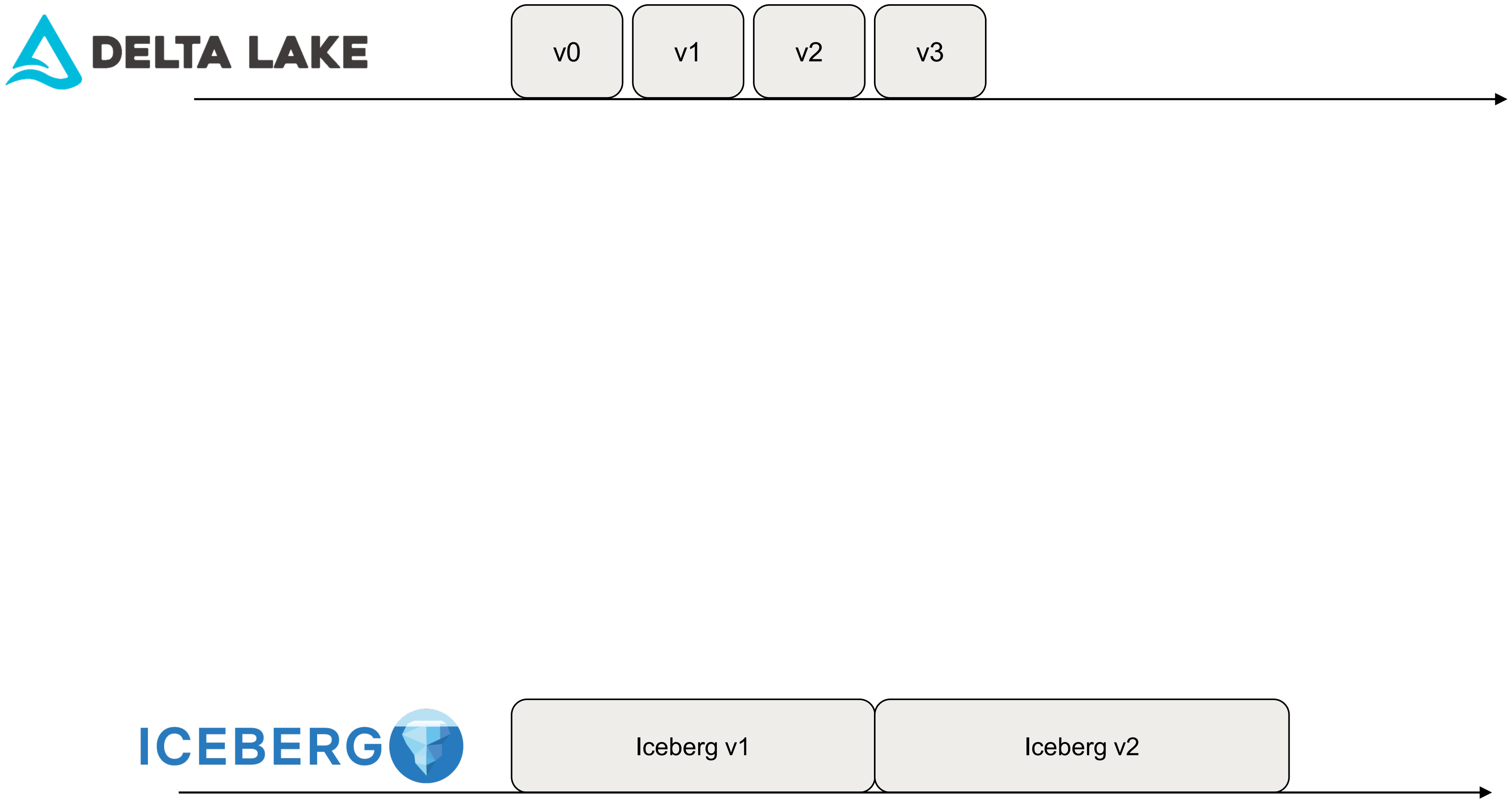


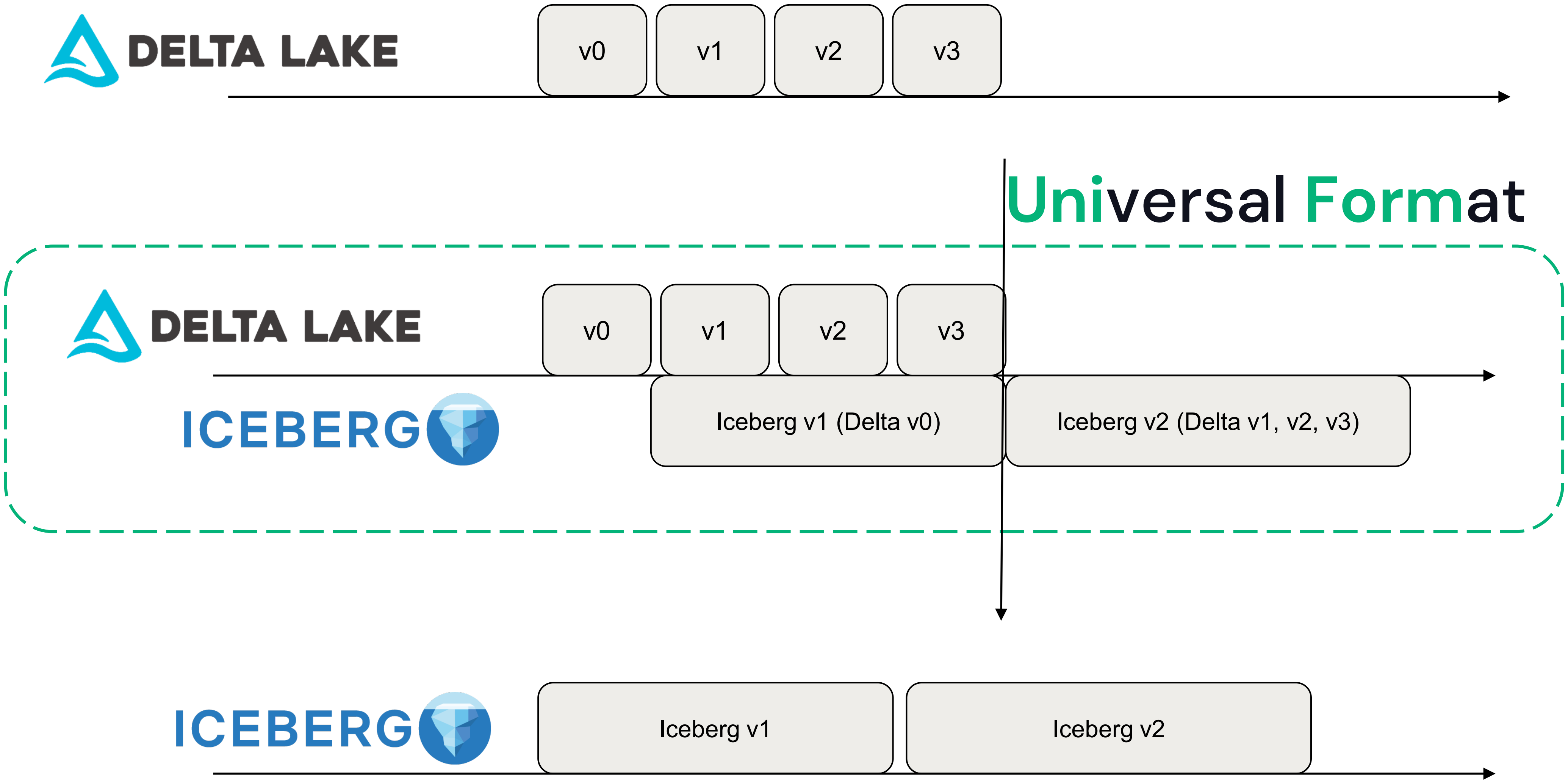
UniForm concept unifies the write path



UniForm as implemented by Databricks




DELTA LAKE
v0
v1
v2
v3
ICEBERG
Iceberg v1
Iceberg v2


DELTA LAKE
v0
v1
v2
v3
Universal Format
DELTA LAKE
v0
v1
v2
v3
ICEBERG
Iceberg v1 (Delta v0)
Iceberg v2 (Delta v1, v2, v3)
ICEBERG
Iceberg v1
Iceberg v2

POP QUIZ

Use Cases

Challenge

Leverage data across their business lines to impact sales, purchasing, supply chain, and product optimization

"Our legacy systems could take weeks to ETL data for analytics and reporting. As a result, we were unable to support a variety of use cases, impacting analyst and line-of-business satisfaction."

Lara Minor

Senior enterprise data manager

Industry:
Retail

Use cases
Advertising effectiveness, customer segmentation, product matching, recommendation engines

Solution

- With Databricks, build high-performance ETL pipelines that support batch and real-time workloads.
- The pipelines feed into Delta Lake which provides secure access to curated data

Industry:
Retail

Use cases
Advertising effectiveness, customer segmentation, product matching, recommendation engines

"Delta Lake provides ACID capabilities that simplify data pipeline operations to increase pipeline reliability and data consistency. At the same time, features like caching and auto-indexing enable efficient and performant access to the data."

Lara Minor

Senior enterprise data manager

Columbia

Industry:
Retail

Use cases
Advertising effectiveness, customer segmentation, product matching, recommendation engines

Outcome

70% reduction in ETL pipeline creation time

48x improvement in time to process ETL workloads (4 hours to 5 minutes)

"One of the benefits of this platform is how fast people can come up to speed on it. All that data is coming in, and more business units are using it across the enterprise in a self-service manner that was not possible before."

Lara Minor

Senior enterprise data manager

Challenge

Creating the most efficient transportation network in North America

- Unlock value of data stuck in legacy DW systems
- Massive data volumes from data streams from IoT sensors
- Legacy systems struggled to scale
- This made telemetry-based use cases leveraging machine learning (ML) and AI nearly impossible.

Industry:
Manufacturing and Logistics

Use cases
Demand forecasting

Industry:
Manufacturing and Logistics

Use cases
Demand forecasting

Solution

Create an open, interoperable and rapid data lakehouse.

- Delta Lake as the open storage layer brought efficiency and portability at TB-scale
- Stream data real-time to Delta Lake – high performance and reliability at any scale
- Single copy of data for easier analysis and reproducibility
- Build ML models atop single source of truth data

Outcome

99.8% Faster freight recommendations

$2.7M in IT infrastructure savings

Industry:
Manufacturing and Logistics

Use cases
Demand forecasting

Replicating application data to the Lakehouse

Current Challenges

Identifying Changes

Updates in ETL struggle to find **changes in the data** from version to version in large tables

Without information regarding the specific changes to be made, all data must be compared

Current Challenges

Identifying Changes

Updates in ETL struggle to find **changes in the data** from version to version in large tables

Without information regarding the specific changes to be made, all data must be compared

Updating BI & Analytics Data

Real-time updates to BI and analytics require additional processing as changes arrive

Recalculating full datasets causes downtime to users incompatible with real-time needs

Current Challenges

Identifying Changes

Updates in ETL struggle to find **changes in the data** from version to version in large tables

Without information regarding the specific changes to be made, all data must be compared

Updating BI & Analytics Data

Real-time updates to BI and analytics require additional processing as changes arrive

Recalculating full datasets causes downtime to users incompatible with real-time needs

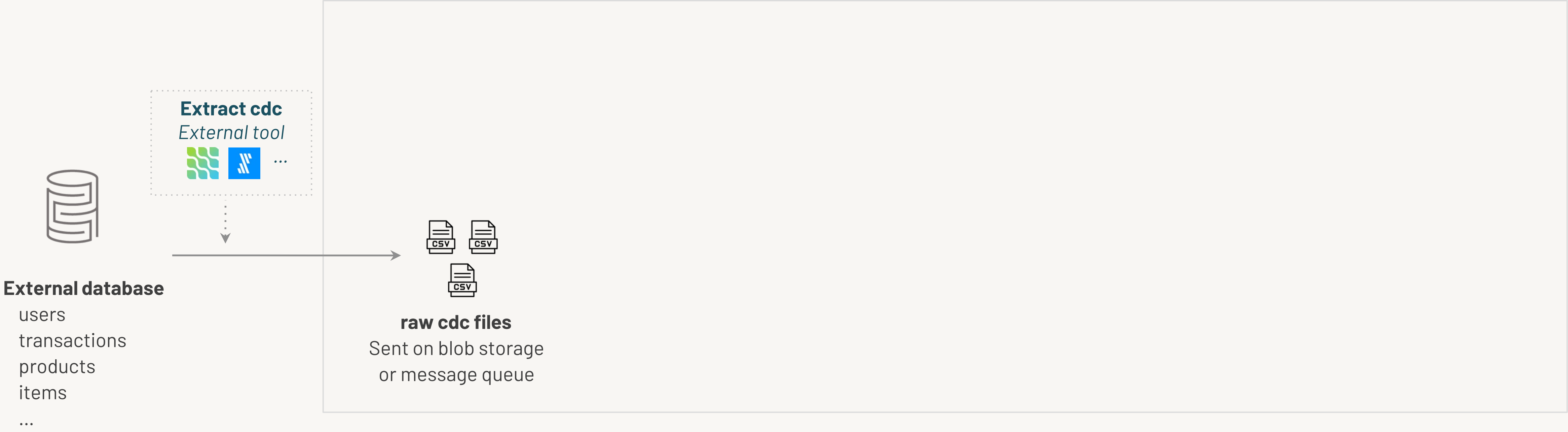
Producing an Audit Trail

Audits of records, en masse or individually, demand the ability to readily construct data as it was at any or every point in time

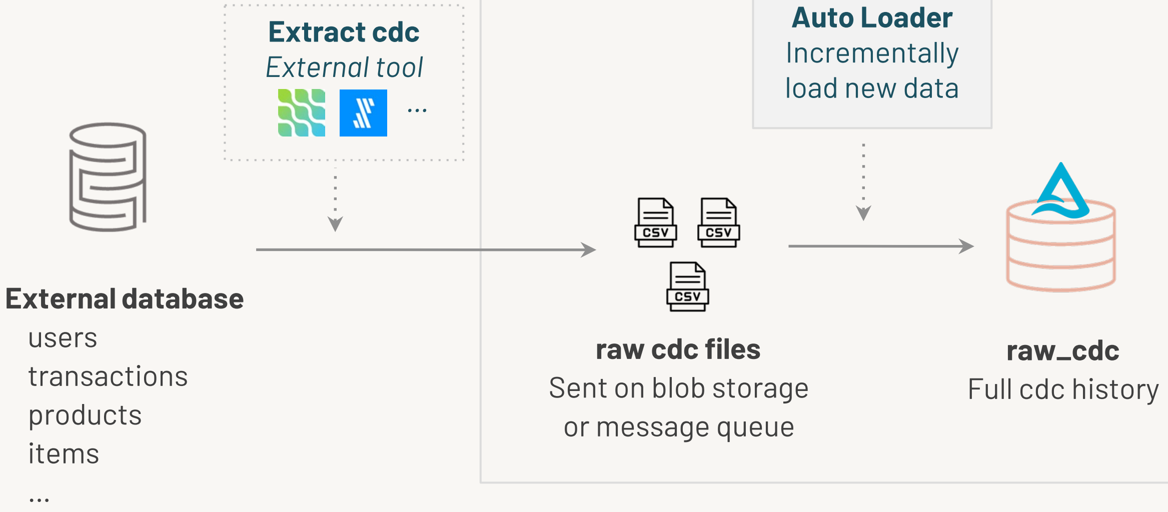
Digging through all versions is impractical yet required to meet compliance requirements

Centralizing all your data shouldn't be hard

One of the most common use cases

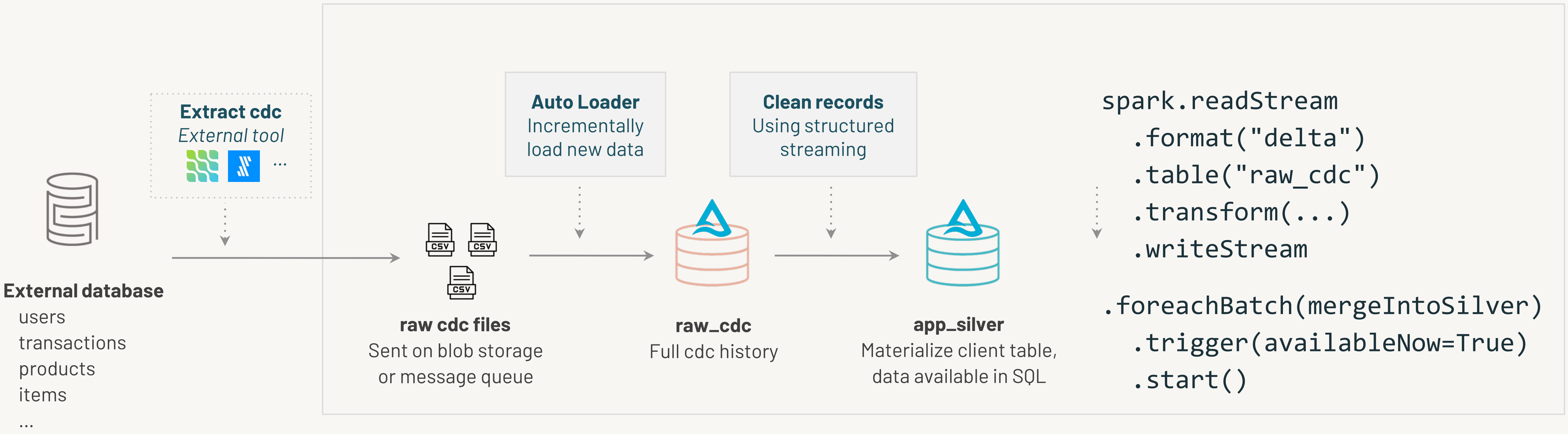


Use autoloader to incrementally ingest your raw data into Delta Lake



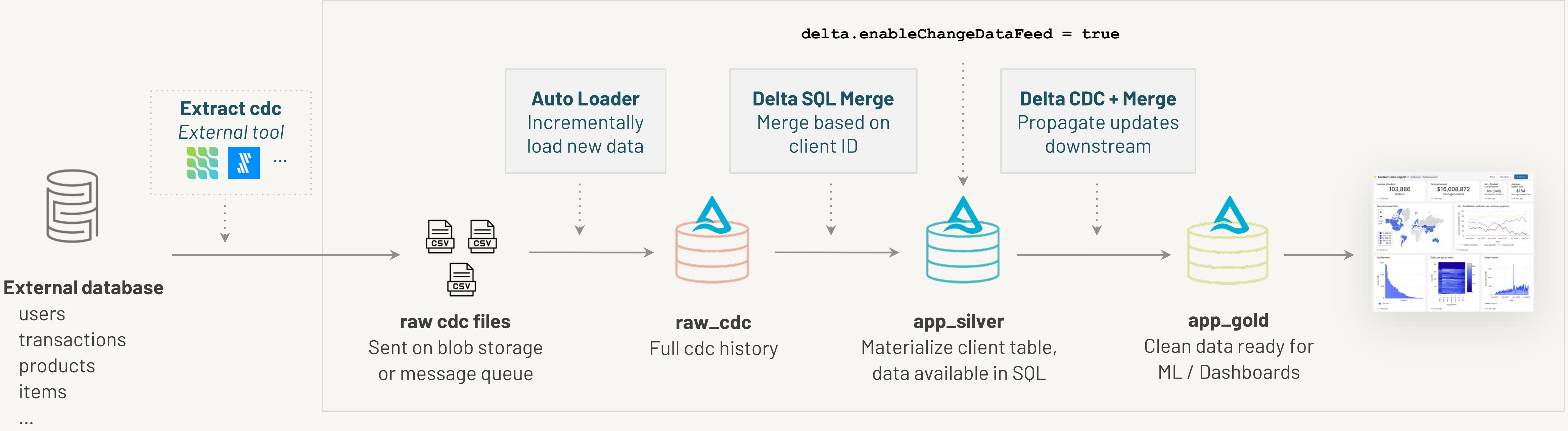

```
spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format",
"json")
  .load("/raw_cdc_files")
  .writeStream
  .trigger(availableNow=True)
  .toTable("raw_cdc"))
```

Use Structured Streaming to incrementally clean records from bronze to silver



MERGE into Gold Table

One of the most common use cases



Developing AI/ML models with Delta Lake

Reproducibility for AI/ML development

Good ML starts with high-quality data.

Model reproducibility starts with data reproducibility

Many factors affect the outcome of a model

- Adding new data sets
- Data distribution
- Sample changes

Delta Lake makes model reproducibility easy

Use cases: model retraining, comparison of different model versions, debugging

Dataset versioning

Automatic versioning for every change (insert, delete, update)

Change tracking

Maintain a detailed log of all data modifications, facilitates audits and lineage

Full history and rollback

Rollback to previous versions of the dataset as needed

Step 1: Initial model training

```
# Initialize Spark session
spark = SparkSession.builder.appName("DeltaLakeExample").getOrCreate()

# Load version 1 of the dataset
df_v1 = spark.read.format("delta").option("versionAsOf", 1).load("/path/to/delta-table")

# Preprocess data
assembler = VectorAssembler(inputCols=["feature1", "feature2"], outputCol="features")
data_v1 = assembler.transform(df_v1)

# Train initial model
lr = LinearRegression(featuresCol="features", labelCol="label")
model_v1 = lr.fit(data_v1)

# Save the model
model_v1.save("/path/to/save/model_v1")
```

Step 2: Adding new data

```
# Load new data
new_data = spark.read.format("csv").option("header", "true").load("/path/to/new-data.csv")

# Merge new data into the Delta table
new_data.write.format("delta").mode("append").save("/path/to/delta-table")
```

Step 3: Retraining the model

```
# Load version 2 of the dataset
df_v2 = spark.read.format("delta").option("versionAsOf", 2).load("/path/to/delta-table")

# Preprocess data
data_v2 = assembler.transform(df_v2)

# Retrain model
model_v2 = lr.fit(data_v2)

# Save the new model
model_v2.save("/path/to/save/model_v2")

# Compare model performance
predictions_v1 = model_v1.transform(data_v2)
predictions_v2 = model_v2.transform(data_v2)
```

Step 3: Retraining the model

```
# Load version 2 of the dataset
df_v2 = spark.read.format("delta").option("versionAsOf", 2).load("/path/to/delta-table")

# Preprocess data
data_v2 = assembler.transform(df_v2)

# Retrain model
model_v2 = lr.fit(data_v2)

# Save the new model
model_v2.save("/path/to/save/model_v2")

# Compare model performance
predictions_v1 = model_v1.transform(data_v2)
predictions_v2 = model_v2.transform(data_v2)
```

📉🔥

Step 4: Rollback and debugging

```
# Rollback to version 1 of the dataset
df_rollback = spark.read.format("delta").option("versionAsOf", 1).load("/path/to/delta-table")

# Compare version 1 and version 2 data
df_v2 = spark.read.format("delta").option("versionAsOf", 2).load("/path/to/delta-table")

df_rollback.show()
df_v2.show()
```

Step 4: Rollback and debugging

```
# Rollback to version 1 of the dataset
df_rollback = spark.read.format("delta").option("versionAsOf", 1).load("/path/to/delta-table")

# Compare version 1 and version 2 data
df_v2 = spark.read.format("delta").option("versionAsOf", 2).load("/path/to/delta-table")

df_rollback.show()
df_v2.show()
```

Rollback and history() make it easy to trace the lineage of all changes to the underlying data, ensuring that your model can be reproduced with exactly the same data it was built on.

Roadmap

Roadmap

1. Simplified user experience
2. Data integrity and reliability
3. Seamless interoperability

Roadmap

1. **Simplified user experience**
2. Data integrity and reliability
3. Seamless interoperability

Simplified data management

Convert partitioned tables to Liquid without rewrite
Upgrade tables in-place to Liquid

Simplified data management

Convert partitioned tables to Liquid without rewrite
Upgrade tables in-place to Liquid

Identity columns
Easy button for primary and foreign keys

Simplified data management

Convert partitioned tables to Liquid without rewrite
Upgrade tables in-place to Liquid

Identity columns
Easy button for primary and foreign keys

Type widening
Seamless, no-copy updates to wider data types (e.g., INT > LONG)

Simplified data management

Convert partitioned tables to Liquid without rewrite
Upgrade tables in-place to Liquid

Identity columns
Easy button for primary and foreign keys

Type widening
Seamless, no-copy updates to wider data types (e.g., INT > LONG)

VARIANT data type
Highly flexible, highly performant data type for semi-structured data

Roadmap

1. Simplified user experience
2. **Data integrity and reliability**
3. Seamless interoperability

Data consistency and availability across multiple environments

Coordinated Commits

Multi-cluster, multi-cloud writes

Data consistency and availability across multiple environments

Coordinated Commits

Multi-cluster, multi-cloud writes

Built-in cross-region disaster recovery

Ensure writes are accurately reflected in secondary region

Data consistency and availability across multiple environments

Coordinated Commits
Multi-cluster, multi-cloud writes

Built-in cross-region disaster recovery
Ensure writes are accurately reflected in secondary region

Collations
Custom sorting and comparison rules

Data consistency and availability across multiple environments

Coordinated Commits
Multi-cluster, multi-cloud writes

Built-in cross-region disaster recovery
Ensure writes are accurately reflected in secondary region

Collations
Custom sorting and comparison rules

Spark Connect support
Improved debuggability, upgradability and reliability

Data consistency and availability across multiple environments

Coordinated Commits
Multi-cluster, multi-cloud writes

Built-in cross-region disaster recovery
Ensure writes are accurately reflected in secondary region

Collations
Custom sorting and comparison rules

Spark Connect support
Improved debuggability, upgradability and reliability

Multi-statement and **Multi-table transactions**
Atomic transactions across tables

Roadmap

1. Simplified user experience
2. Data integrity and reliability
3. **Seamless interoperability**

Seamless interoperability

Delta Kernel
Integrate your client once, get the latest Delta innovations forever.

Seamless interoperability

Delta Kernel
Integrate your client once, get the latest Delta innovations forever.

Expanding connector ecosystem
Collaborating with community and partners to build connectors with Kernel

Seamless interoperability

Delta Kernel
Integrate your client once, get the latest Delta innovations forever.

Expanding connector ecosystem
Collaborating with community and partners to build connectors with Kernel

Delta UniForm
Improved interoperability with latest Delta capabilities - e.g., Deletion Vectors

POP QUIZ

Learn more at the summit!

Databricks
Events App

Tells us what you think

- We kindly request your valuable feedback on this session.
- Please take a moment to rate and share your thoughts about it.
- You can conveniently provide your feedback and rating through the **Mobile App**.

What to do next?

- Discover more related sessions in the mobile app!
- Visit the Demo Booth: Experience innovation firsthand!
- More Activities: Engage and connect further at the Databricks Zone!

Get trained and certified

- Visit the Learning Hub Experience at Moscone West, 2nd Floor!
- Take complimentary certification at the event; come by the Certified Lounge
- Visit our Databricks Learning website for more training, courses and workshops! databricks.com/learn

databricks