# Iceberg Foundations

# Data Lake vs Data Warehouse

Existing approaches require trade offs

## Data Lake Centric

Lacks intelligent storage optimization

Takes away decades of databases capabilities such as transactions and deletes/updates

## Data Warehouse Centric

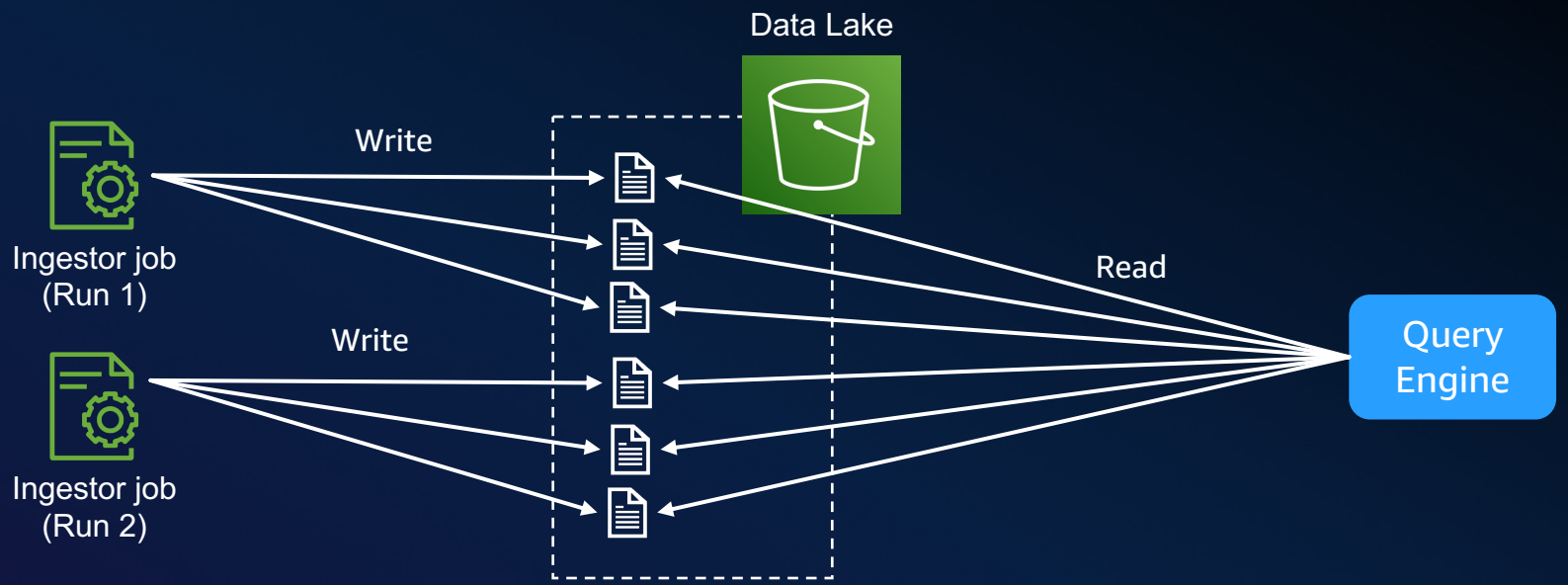
Lacks open access to data warehouse data

Data silos

**Apache Iceberg: Move from a "Data Lake" to a "Transactional Lakehouse"**

# Challenges with a traditional Datalake

# Small files impact query performance



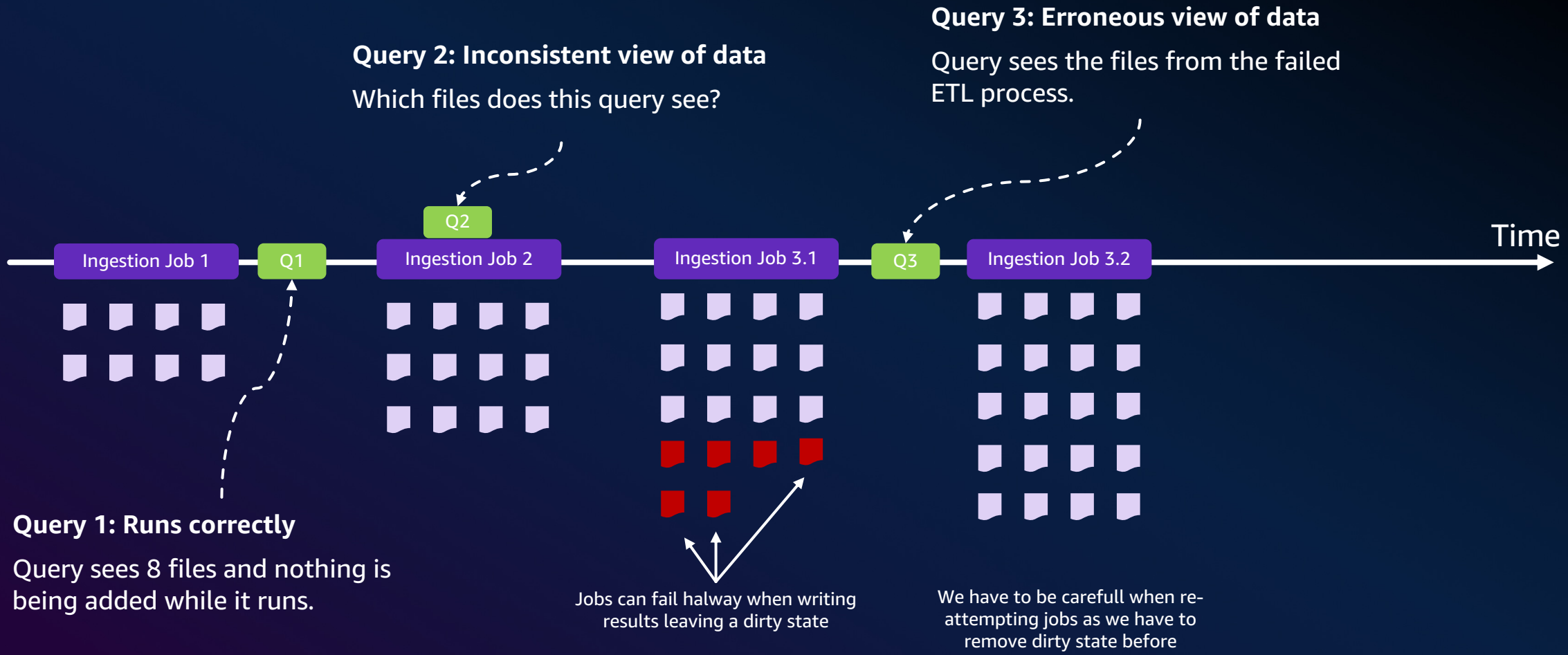

## Challenges

- Streming jobs or frequently executed ingestion jobs often generate small files
- Small files degrades performance on query/reader side (multiple files to open
- Not trivial to build compaction jobs

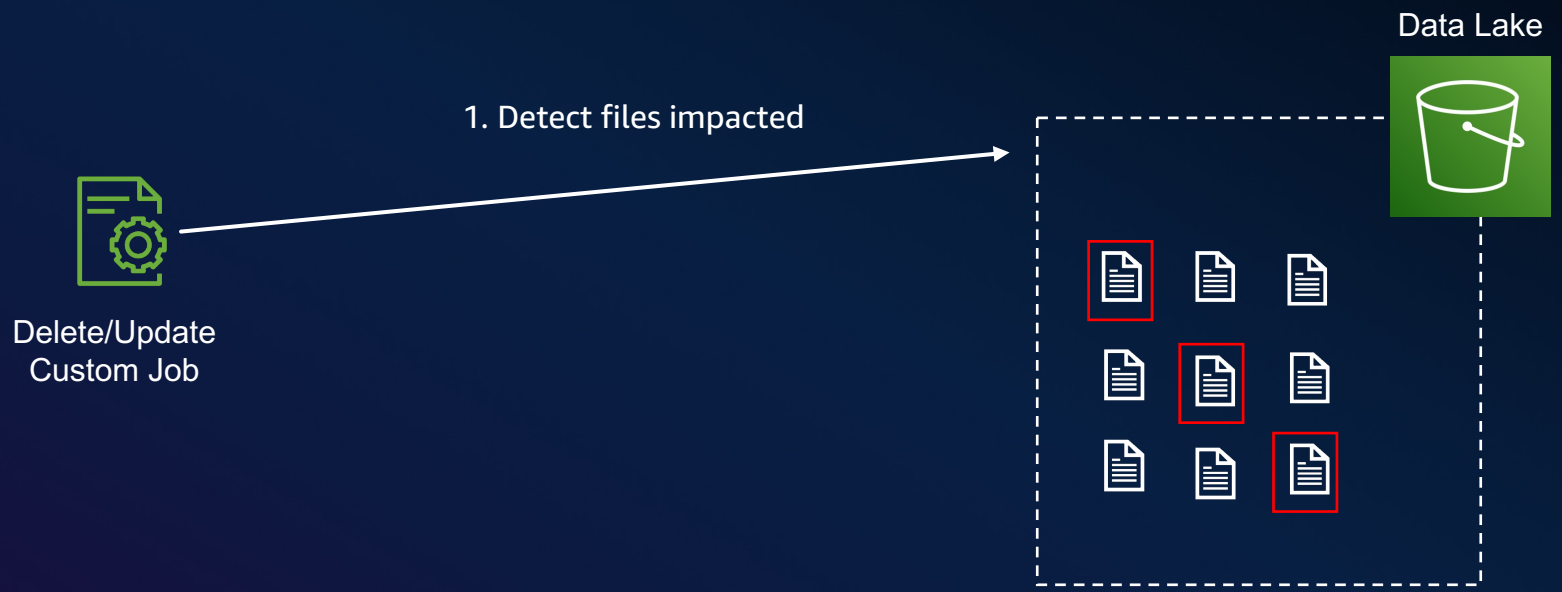
# No ACID transactions

*What happens when jobs read and write simultaneously?*

*What is the impact of an ingestion job that fails half-way?*



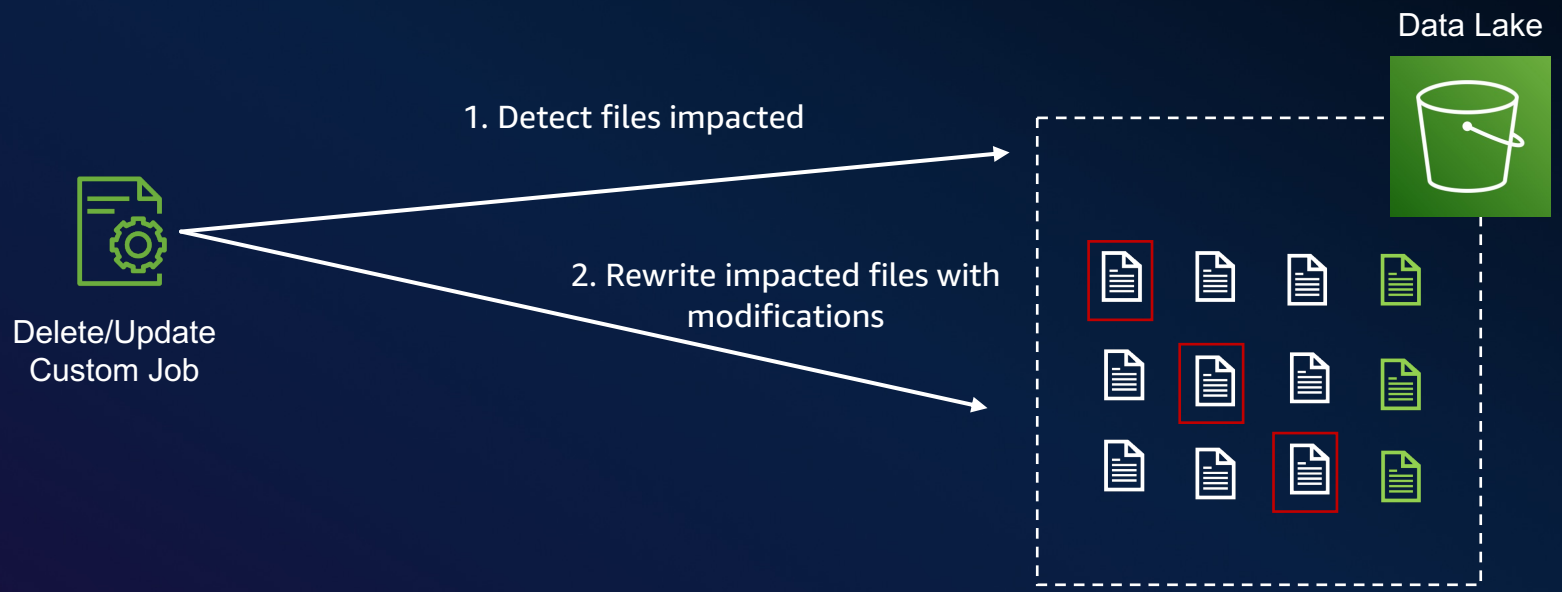
# Deletion and Updates not trival to implement




## Challenges

- **Multi-step, error-prone workflow**: Requires custom orchestration with no built-in transaction log or automatic rollback on failure
- **Lack of complete isolation**: Brief windows where readers see inconsistent state

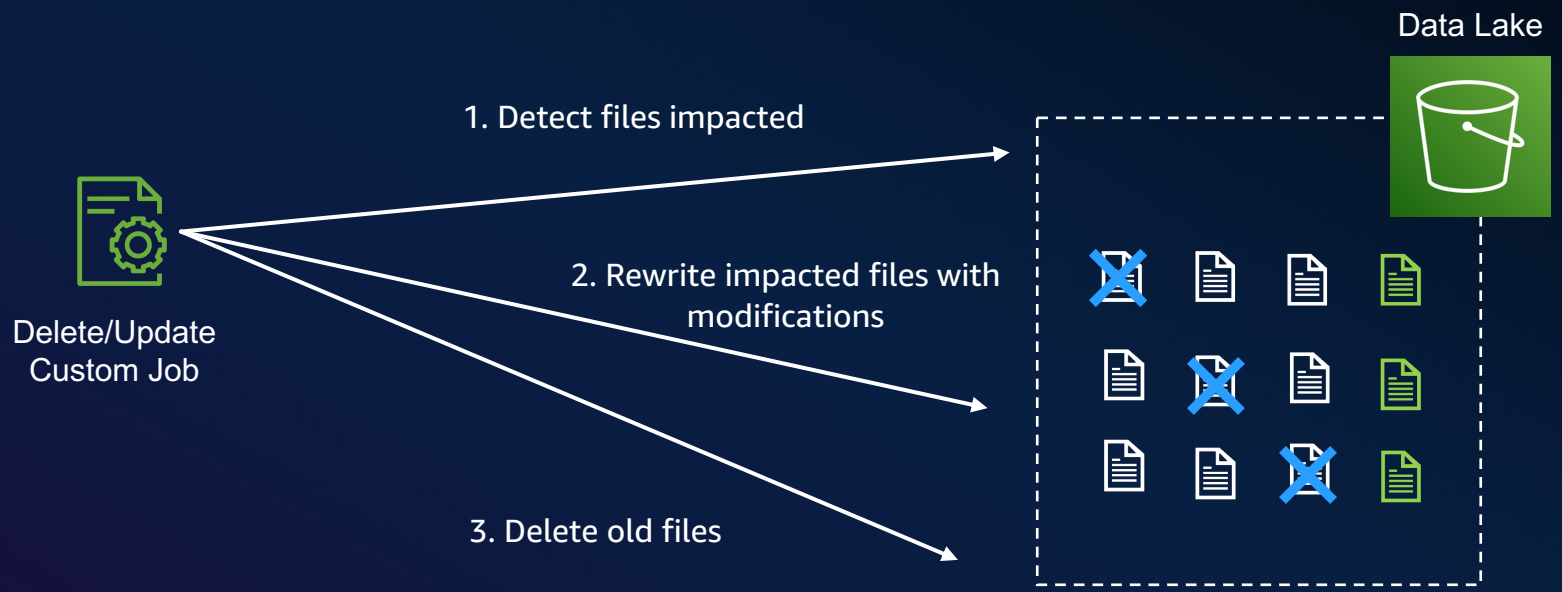
# Deletion and Updates not trival to implement




## Challenges

- **Multi-step, error-prone workflow**: Requires custom orchestration with no built-in transaction log or automatic rollback on failure
- **Lack of complete isolation**: Brief windows where readers see inconsistent state

# Deletion and Updates not trival to implement




## Challenges

- **Multi-step, error-prone workflow**: Requires custom orchestration with no built-in transaction log or automatic rollback on failure
- **Lack of complete isolation**: Brief windows where readers see inconsistent state

# Schema Evolution on datalake tables

- Schema is fixed at table creation time
- Change schema means rewrite table data in case of renaming columns

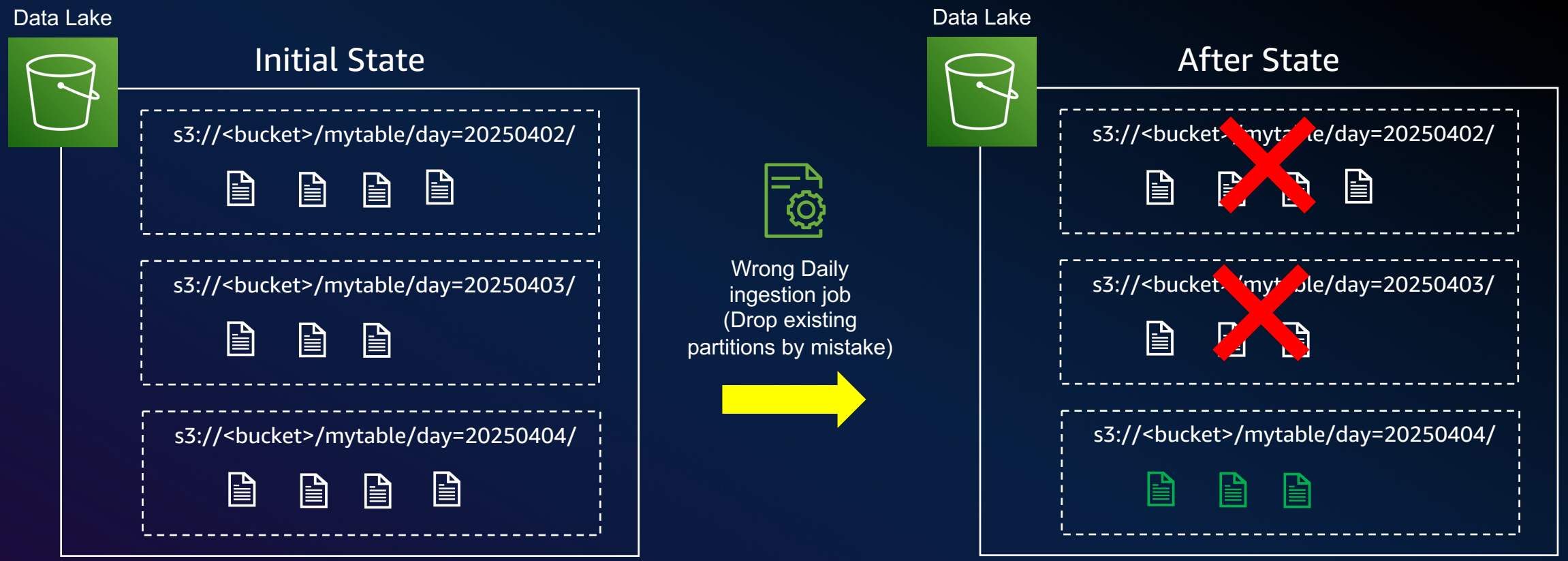
```
CREATE TABLE db.logs(
    id long,
    log_event string,
    level string,
    event_time timestamp,
    event_date string
)
USING parquet
PARTITIONED BY(event_date)
LOCATION 's3://BUCKET/db/logs/'
```

```
CREATE TABLE catalog.db.logs_v2
USING parquet
PARTITIONED BY (event_year)
LOCATION 's3://BUCKET/db/logs_v2/'
AS
SELECT
    id,
    log_event,
    level AS log_level,
    event_time,
    event_date
FROM catalog.db.logs;
```

## Challenges

- Entire table rewritten (expensive)
- Impact readers (e.g., to be redirected to a new table)
- Potentially need to re-grant access permissions to this table if permissions given at table and not database level

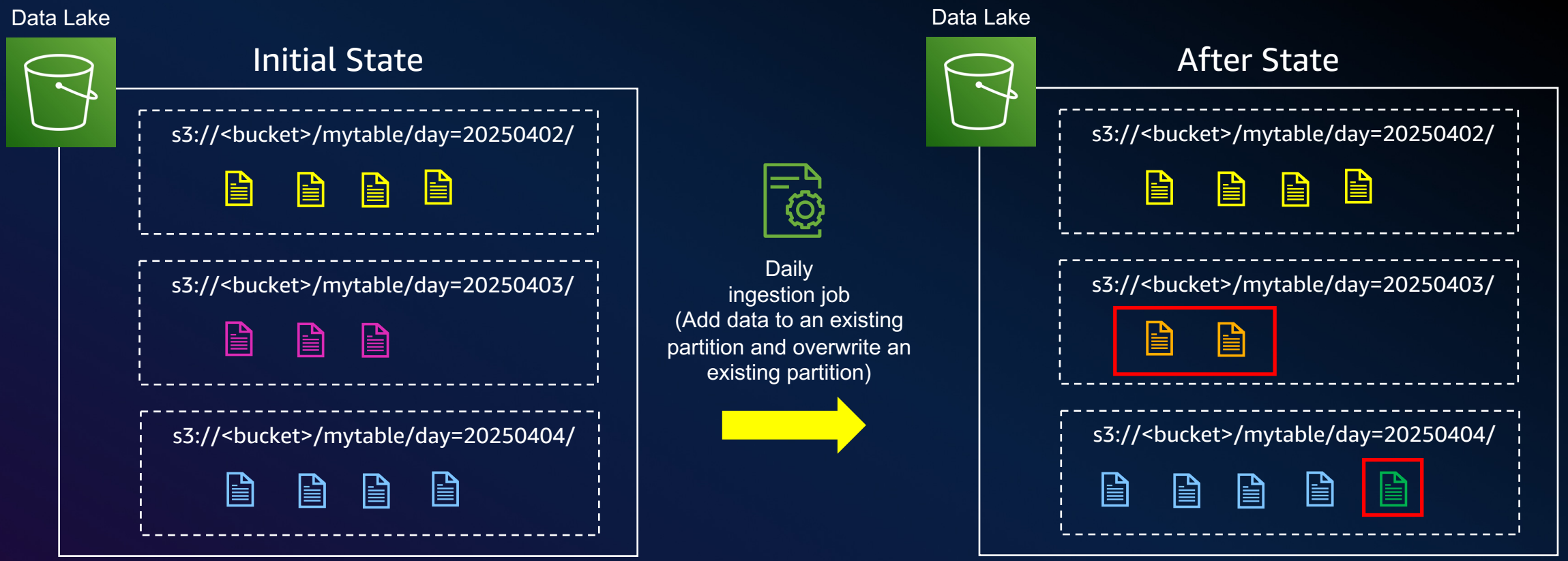
# No Rollbacks capabilities




## Challenges

- Not straightforward way to recover mistakes
- In many cases have to go back to source systems to take historical data (what if retention expired on source systems?)

# No Time Travel capabilities




## Challenges

- Not straightforward way to retrieve table state in the past (e.g., before an ingestion job executed)
- Deep dive on files timestamps for append use cases, not possible for overwrite use cases

# Partition Evolution on datalake tables

- Partitioning is fixed at table creation time
- Change partitioning means rewrite table data

```
CREATE TABLE db.logs(
    id long,
    log_event string,
    level string,
    event_time timestamp,
    event_date string
)
USING parquet
PARTITIONED BY(event_day)
LOCATION 's3://BUCKET/db/logs/'
```



```
CREATE TABLE catalog.db.logs_v2
USING parquet
PARTITIONED BY (event_year)
LOCATION 's3://BUCKET/db/logs_v2/'
AS
SELECT
    id,
    log_event,
    level,
    event_time,
    YEAR(event_time) AS event_year
FROM catalog.db.logs;
```

## Challenges

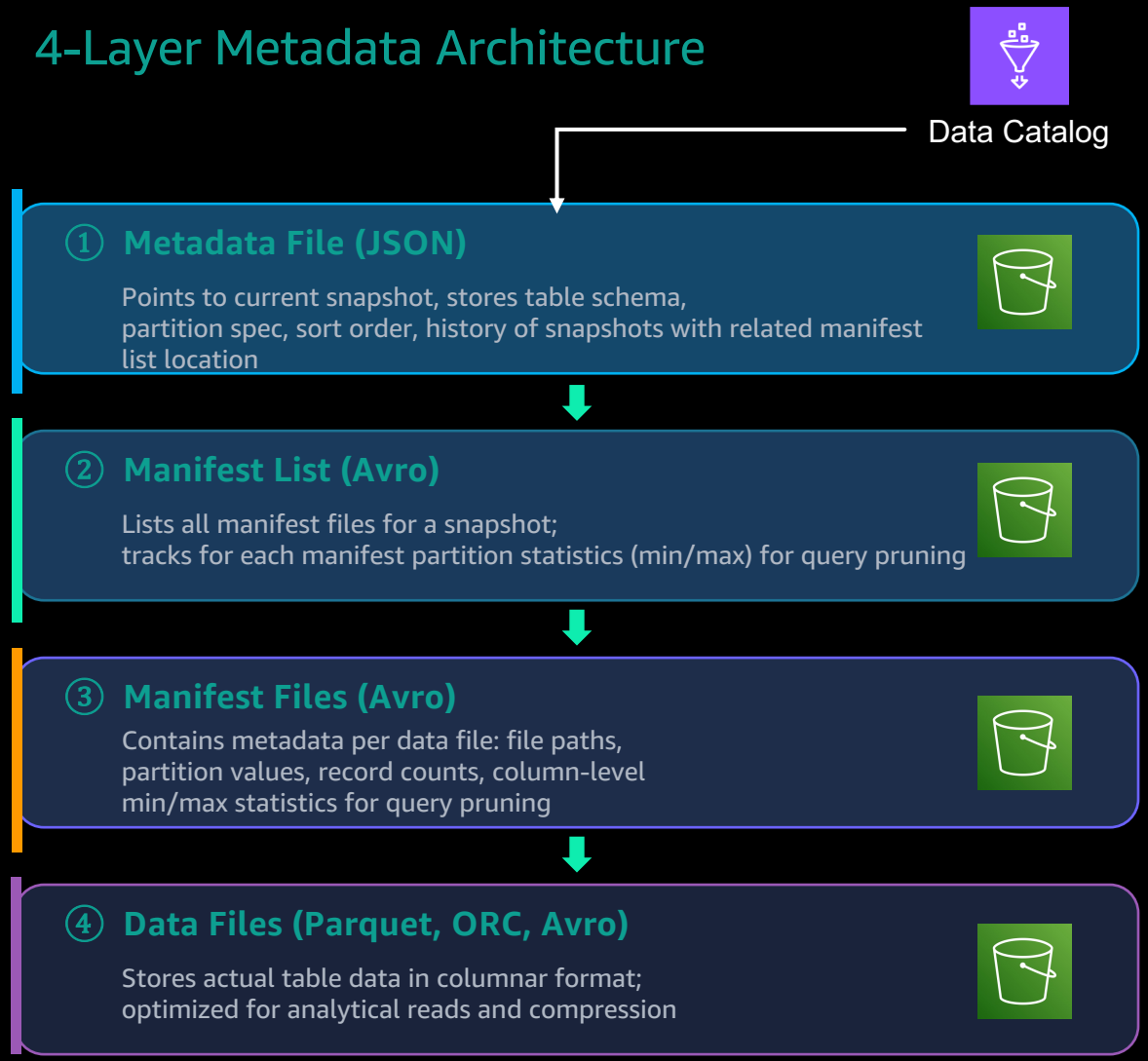
- Entire table rewritten (expensive)
- Impact readers (e.g., to be redirected to a new table)
- Potentially need to re-grant access permissions to this table if permissions given at table and not database level

# Introduction to Apache Iceberg

# Apache Iceberg: Table Format Architecture

*How Iceberg organizes metadata and data files to enable ACID transactions, time travel, and efficient query planning*

## 4-Layer Metadata Architecture




## Key Capabilities

### ACID Transactions

Atomic commits via metadata file swap ensure readers never see partial writes. Optimistic concurrency handles conflicts.

### Time Travel & Rollbacks

Each commit creates an immutable snapshot. Query any historical state by snapshot ID or timestamp. Rollback to a previous state.

### Metadata-driven performance

Manifests files contain files locations, avoid S3 list operations.
Manifest files contain min/max statistics to skip irrelevant data files at plan time, avoid getting files footers to get those statistics.

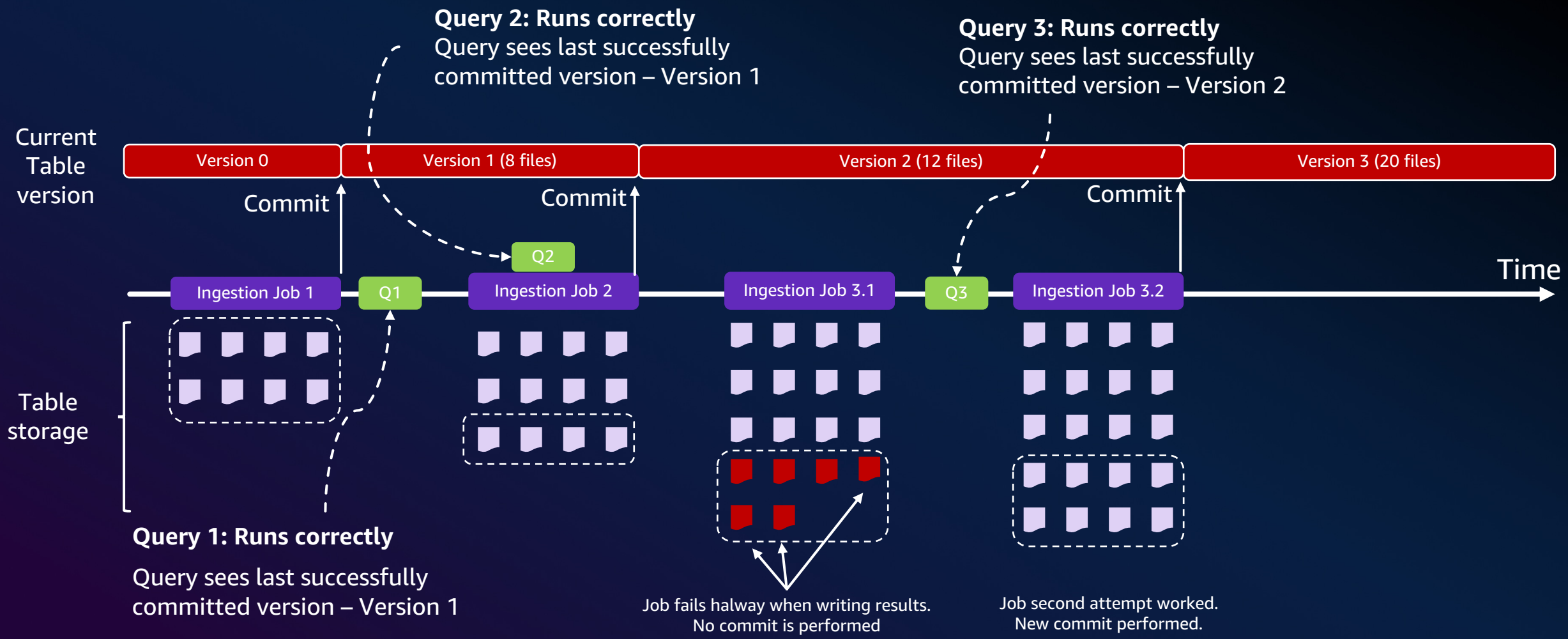
### Schema & Partition Evolution

Schema changes and partition layout updates are metadata-only operations. No data rewrite required.

### Native UPDATE, DELETE, MERGE support

UPDATE, DELETE and MERGE statements are natively supported with ACID transactions guarantees

# Iceberg Tables – ACID Transactions

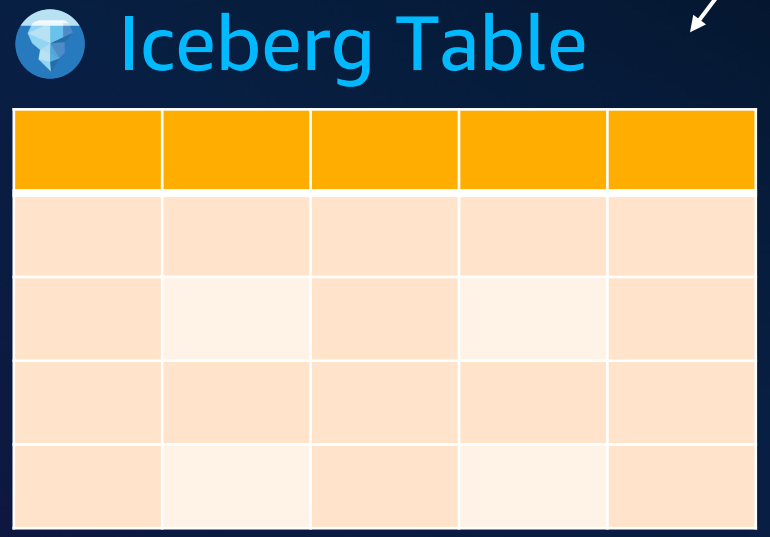


# Iceberg natively supports INSERT, UPDATE, DELETE, MERGE

```
INSERT INTO sample_events
VALUES (1, 'alpha', 'A');
```

```
UPDATE sample_events
SET value = 'ALPHA_UPDATED'
WHERE id = 1;
```

```
DELETE FROM sample_events
WHERE category = 'C';
```




```
MERGE INTO sample_events t
USING staging_events s
ON t.id = s.id
WHEN MATCHED THEN UPDATE SET *WHEN
NOT MATCHED THEN INSERT *;
```

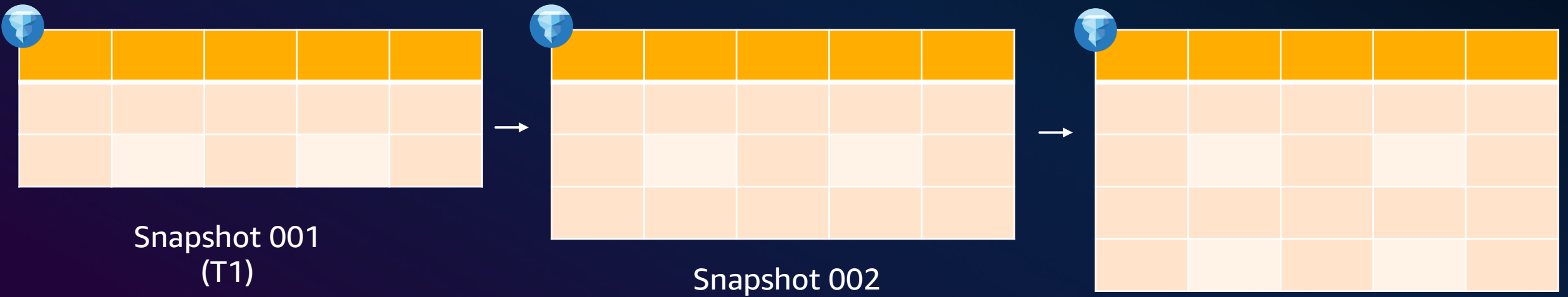
```
MERGE INTO sample_events t
USING staging_events s
ON t.id = s.id
WHEN MATCHED AND s.op = 'D' THEN DELETE
WHEN MATCHED AND s.op = 'U' THEN
  UPDATE SET t.value = s.value, t.category =
s.category
WHEN NOT MATCHED THEN
  INSERT (id, value, category) VALUES (s.id,
s.value, s.category);
```

# Iceberg natively supports Time Travel

```
SELECT *
FROM tablename
TIMESTAMP AS OF T1
```

```
SELECT *
FROM tablename
VERSION AS OF 002
```

```
SELECT *
FROM tablename
```

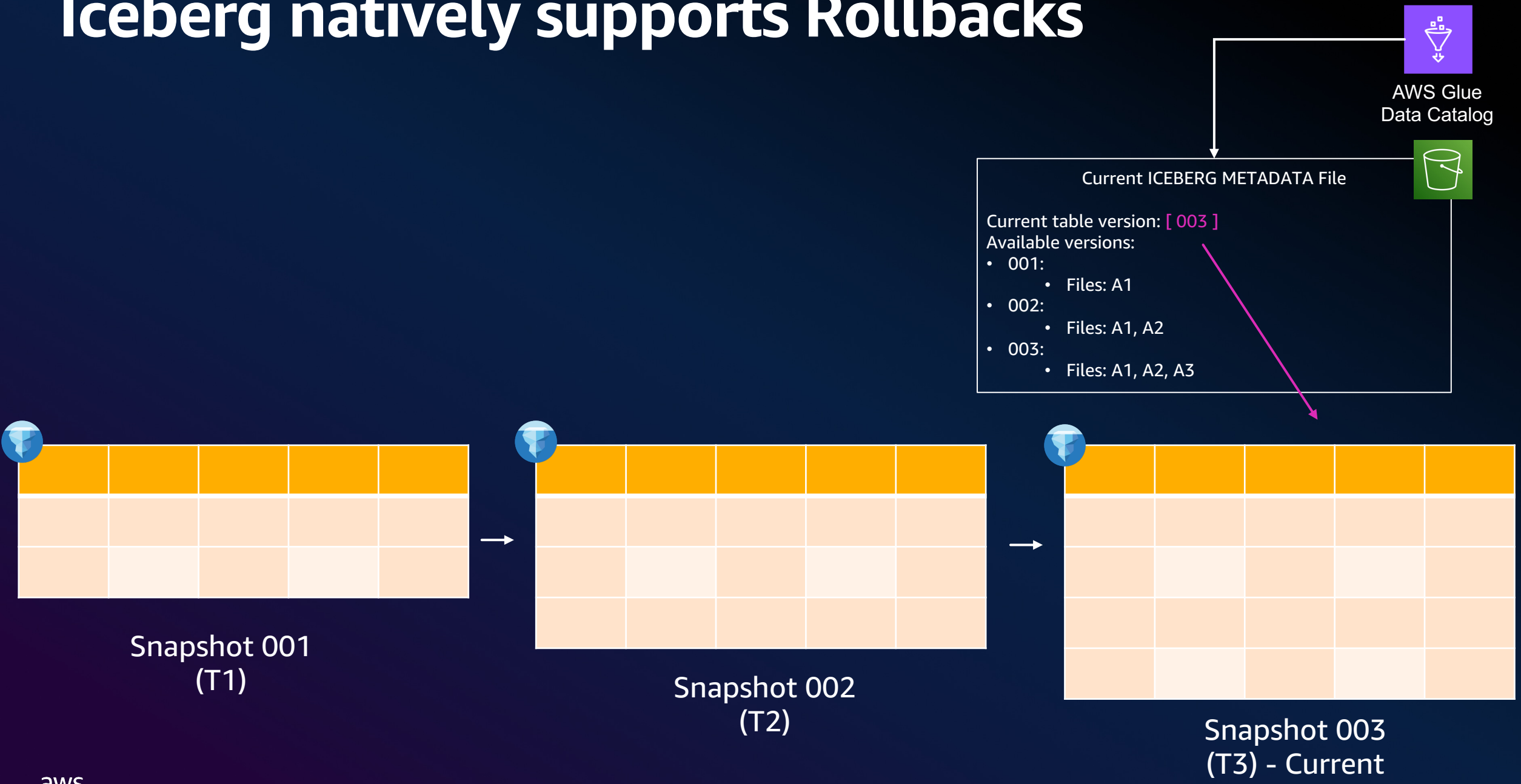


Snapshot 001
(T1)

Snapshot 002
(T2)

Snapshot 003
(T3) - Current

# Iceberg natively supports Rollbacks



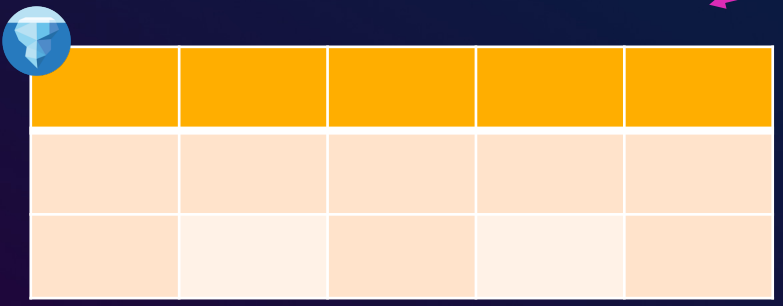
# Iceberg natively supports Rollbacks

```
CALL system.rollback_to_snapshot(
    'tablename',
    001
)
```

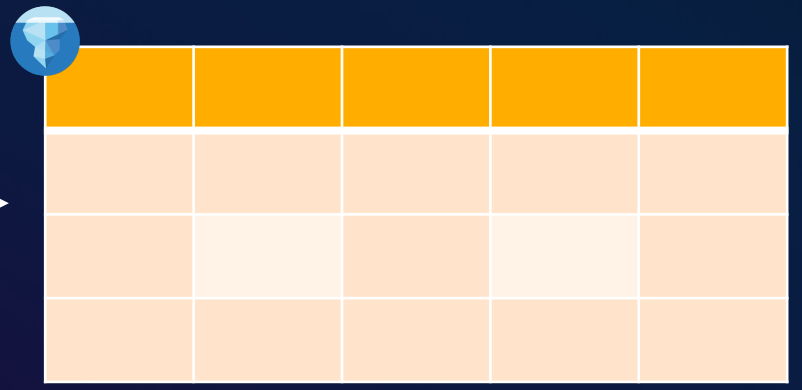

Current ICEBERG METADATA File

Current table version: [ 001 ]
Available versions:

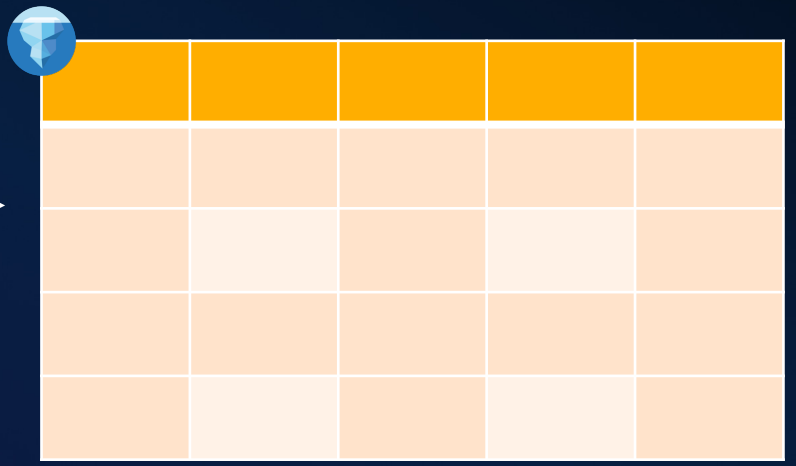
- 001:
  - Files: A1
- 002:
  - Files: A1, A2
- 003:
  - Files: A1, A2, A3



Snapshot 001
(T1)



Snapshot 002
(T2)



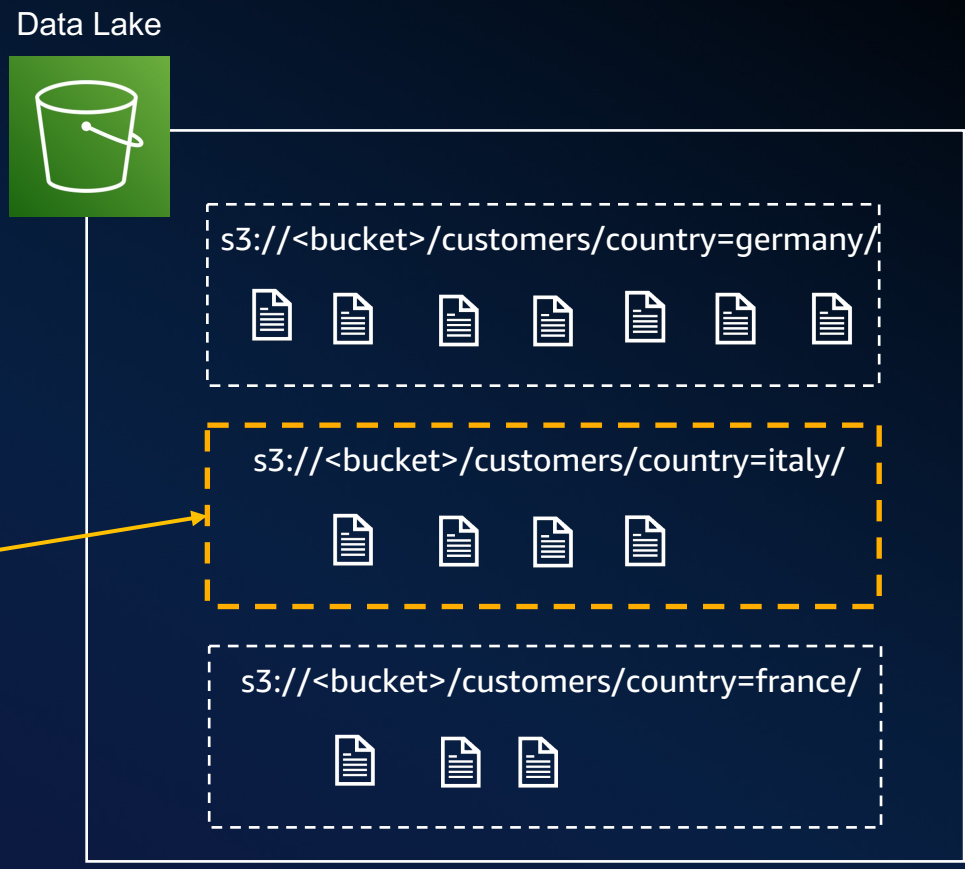
Snapshot 003
(T3) - Current

# Partitioning as a common technique to improve performance on data lakes tables

```
CREATE TABLE customers(
    id string,
    born_year int,
    …
    country string,
)
PARTITIONED BY(country)
USING parquet;
```

```
SELECT *
FROM   customers
WHERE  country = 'italy'
       AND born_year > 1990
```

Only inspect data files in the partition folder




Partition tables on columns most used on query filters!

# Standard Data Lake tables partitioning

## Challenges at write time

```
CREATE TABLE catalog.db.logs(
    id long,
    log_event string,
    level string,
    event_time timestamp,
    event_date date
)
PARTITIONED BY (event_date)
USING parquet;
```

How to insert data

Source data lacks the `event_date` partition column

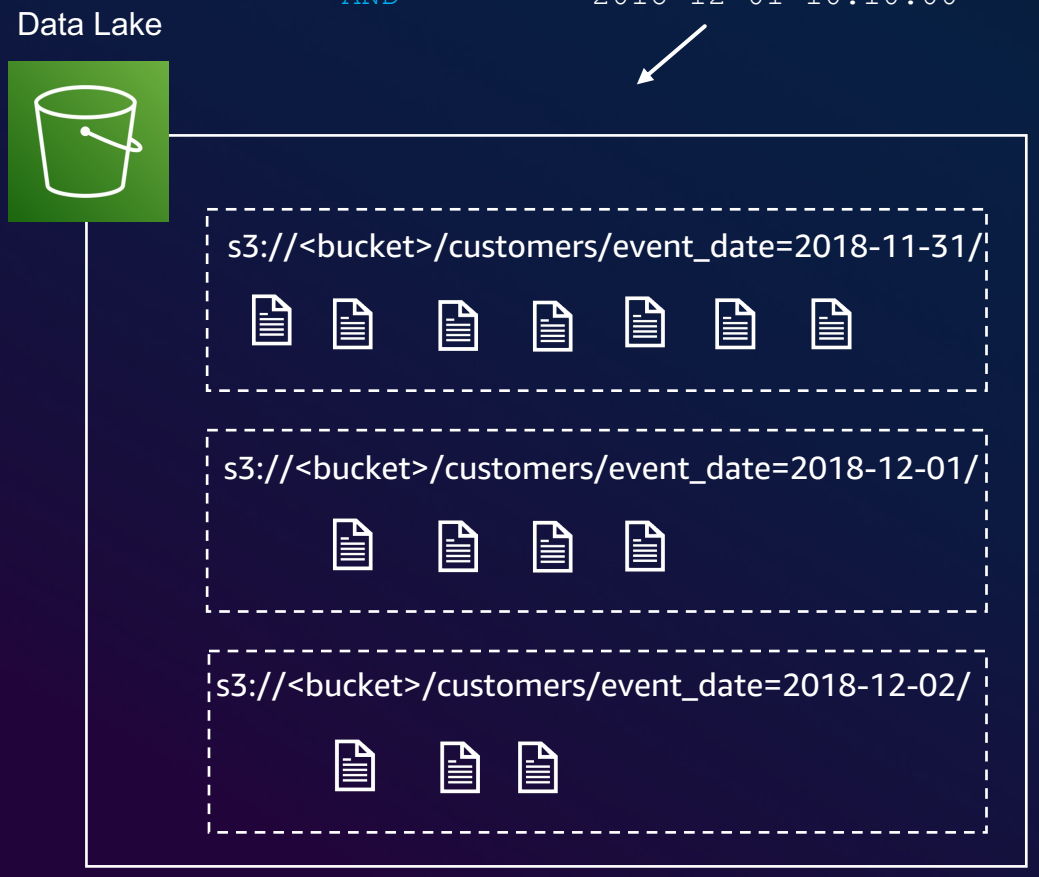


The ingestion process extracts `event_date` from the `event_time` timestamp for each record, adds the computed column on-the-fly and perform the INSERT operation

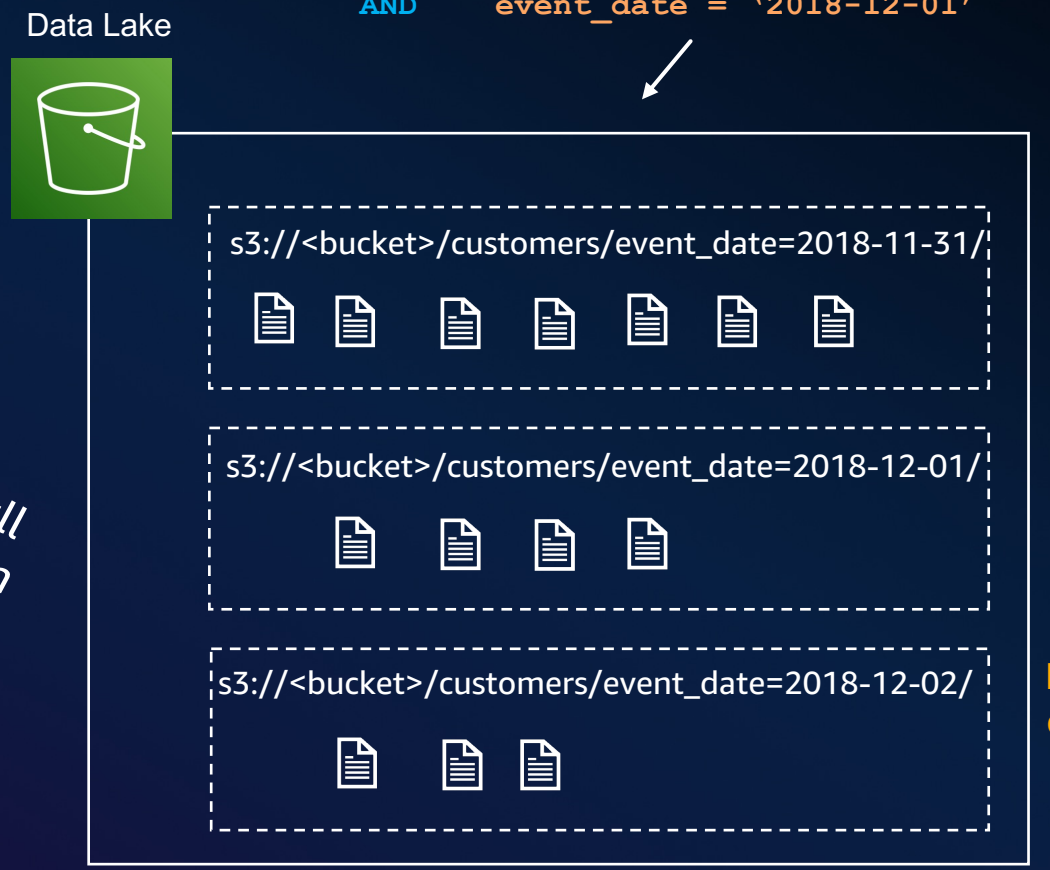
# Standard Data Lake tables partitioning

## Challenges at read time

```
SELECT level, Count(1) as count
FROM   logs
WHERE  event_time BETWEEN
            '2018-12-01 10:00:00'
AND         '2018-12-01 10:10:00'
```




```
SELECT level, Count(1) as count
FROM   logs
WHERE  event_time BETWEEN
            '2018-12-01 10:00:00'
AND       '2018-12-01 10:10:00'
AND    event_date = '2018-12-01'
```




Partition pruning only if specifying a filter on the partition column

# Iceberg Hidden Partitioning

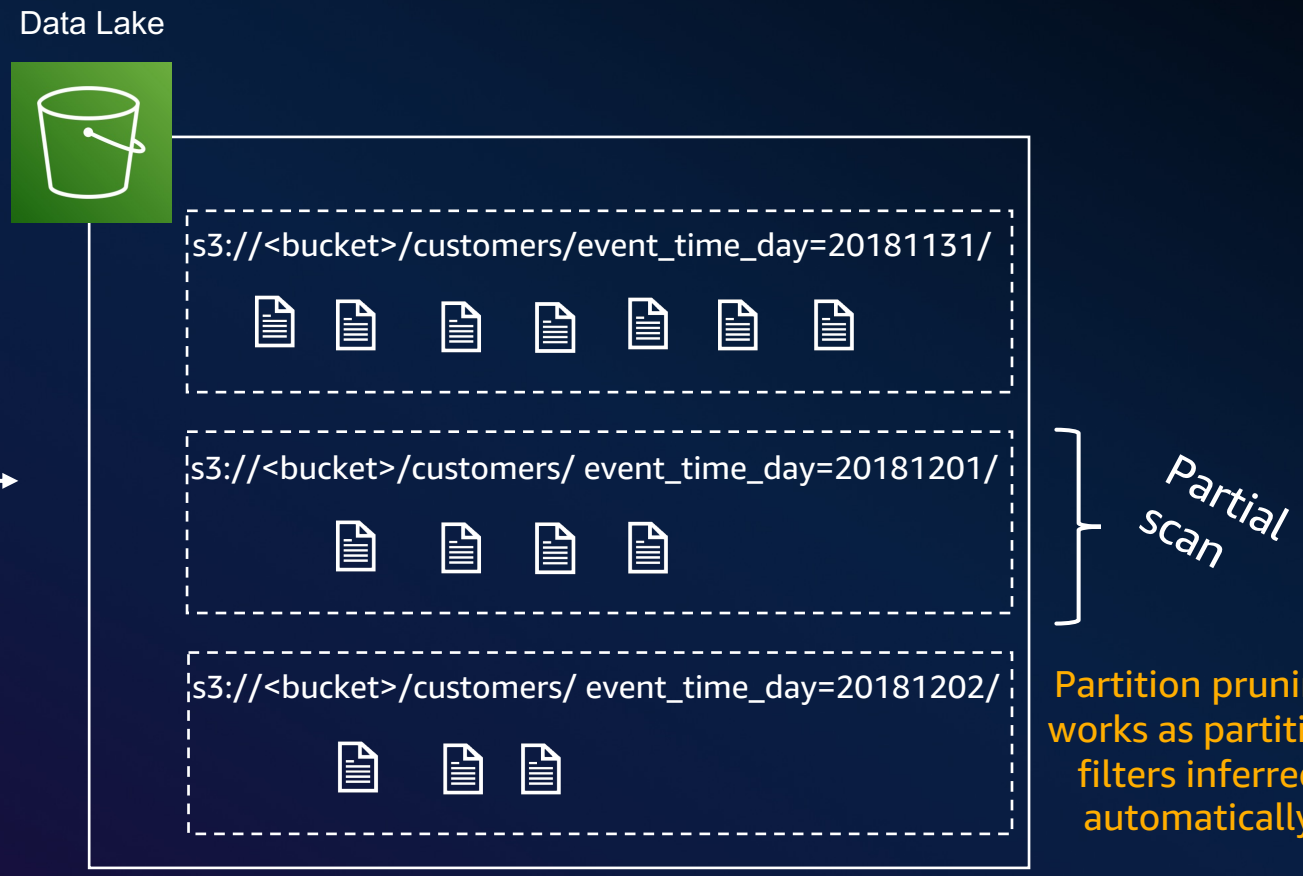
## Tables are created using “Partition Transforms”

```
CREATE TABLE catalog.db.logs(
    id long,
    log_event string,
    level string,
    event_time timestamp
) PARTITIONED BY (day(event_time)) USING iceberg;
```

No additional column for partitioning simplifies ingestion (no need to infer additional columns at ingestion time)

## Example of an Iceberg query

```
SELECT level, Count(1) as count
FROM   logs
WHERE  event_time BETWEEN
            '2018-12-01 10:00:00'
AND       '2018-12-01 10:10:00'
```




Partition pruning works as partition filters inferred automatically

# Iceberg Schema Evolution

Schema evolution changes are **independent and free of side-effects** without rewriting files (metadata only operations)

Supported schema evolution changes (even in nested structures)

- Add column
- Drop column
- Rename column
- Reorder column
- Update column

```
ALTER TABLE prod.db.sample ADD COLUMN
new_column string;
```

```
ALTER TABLE prod.db.sample DROP COLUMN id;
```

```
ALTER TABLE prod.db.sample RENAME COLUMN data
TO payload;
```

```
ALTER TABLE prod.db.sample ALTER COLUMN
category AFTER id;
```

```
ALTER TABLE prod.db.sample ALTER COLUMN
measurement TYPE double;
```

Iceberg allows updating column types if the update is safe.
Safe updates are:

- `int` to `bigint`
- `float` to `double`
- `decimal(P,S)` to `decimal(P2,S)` when P2 > P (scale cannot change)

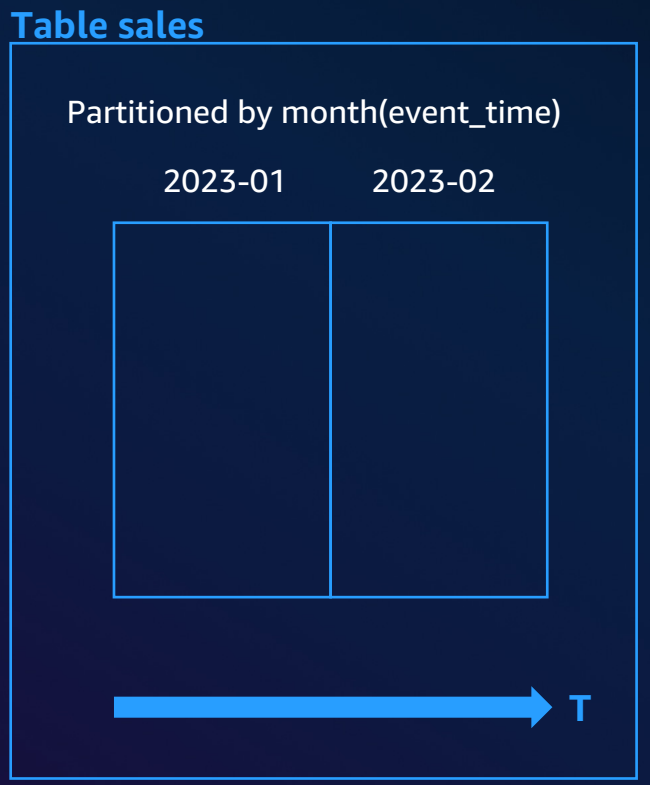
# Iceberg Partition Evolution

## Iceberg supports changing partition layout

```
CREATE TABLE catalog.db.sales(
    id long,
    product string,
    amount double,
    event_time timestamp
) PARTITIONED BY (month(event_time))
USING iceberg;
```

Table Location
s3://datalake/sales_table/

Partitions
s3://datalake/sales_table/event_time_month=2023-01/
s3://datalake/sales_table/event_time_month=2023-02/




← Insert data January and February

# Iceberg Partition Evolution

## Just a metadata operation

```
ALTER TABLE sales REPLACE FIELD
months(event_time) WITH days(event_time)
```

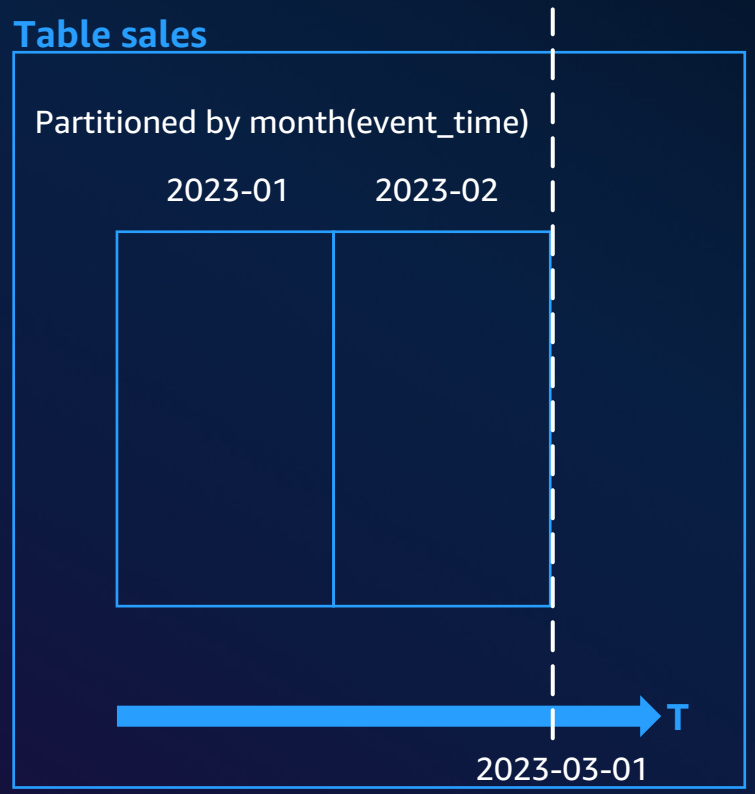


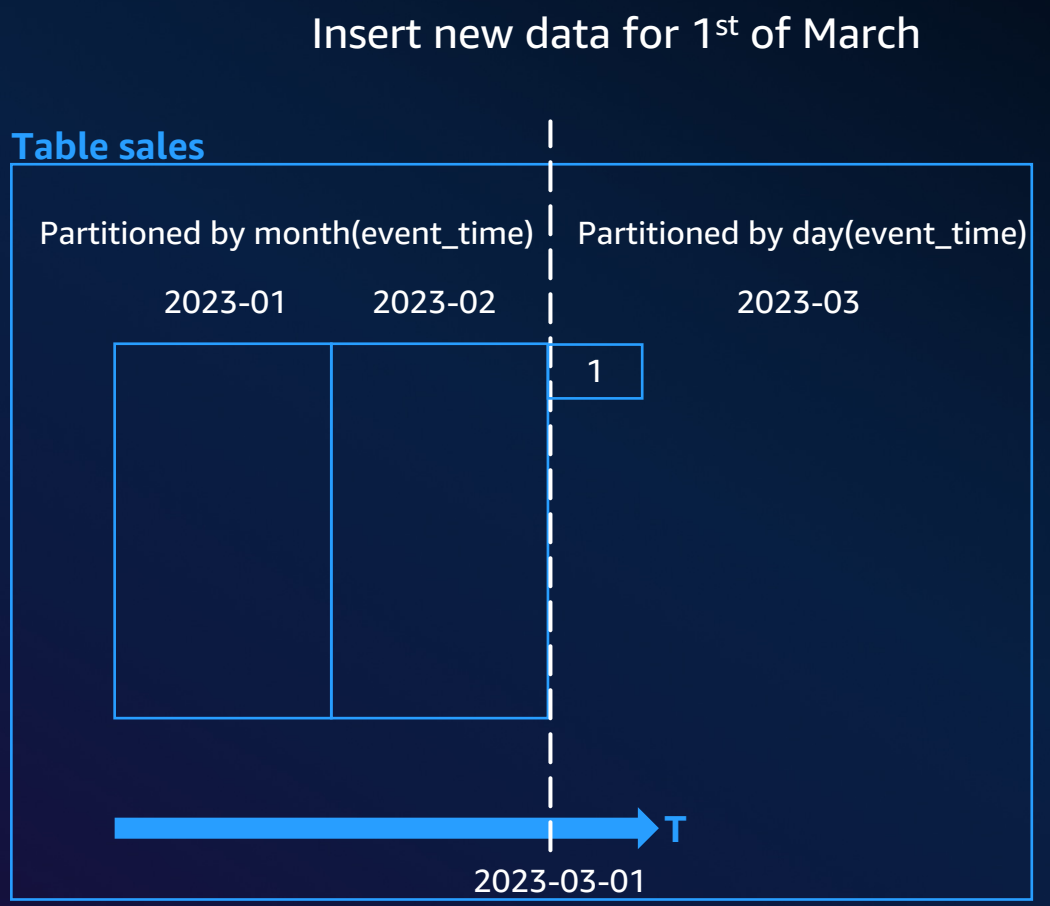

Table Location
s3://datalake/sales_table/

Partitions
s3://datalake/sales_table/event_time_month=2023-01/
s3://datalake/sales_table/event_time_month=2023-02/

# Iceberg Partition Evolution

Table Location
s3://datalake/sales_table/

Partitions
s3://datalake/sales_table/event_time_month=2023-01/
s3://datalake/sales_table/event_time_month=2023-02/
s3://datalake/sales_table/event_time_day=2023-03-01/



# Iceberg Partition Evolution

There are no additional costs such as moving data, migrating tables, or rewriting queries

Table Location
s3://datalake/sales_table/

Partitions
s3://datalake/sales_table/event_time_month=2023-01/
s3://datalake/sales_table/event_time_month=2023-02/
s3://datalake/sales_table/event_time_day=2023-03-01/
s3://datalake/sales_table/event_time_day=2023-03-02/
…
s3://datalake/sales_table/event_time_day=2023-03-31/

# Iceberg Partition Evolution

## Query example (partitions inspected)

```
SELECT * FROM sales
WHERE
   event_time > 2023-02-11 AND
 event_time < 2023-03-18
```

Table Location
s3://datalake/sales_table/

Partitions
s3://datalake/sales_table/event_time_month=2023-01/
s3://datalake/sales_table/event_time_month=2023-02/
s3://datalake/sales_table/event_time_day=2023-03-01/
s3://datalake/sales_table/event_time_day=2023-03-02/
s3://datalake/sales_table/event_time_day=2023-03-03/
s3://datalake/sales_table/event_time_day=2023-03-04/
…
s3://datalake/sales_table/event_time_day=2023-03-17/
s3://datalake/sales_table/event_time_day=2023-03-18/
s3://datalake/sales_table/event_time_day=2023-03-19/
…
s3://datalake/sales_table/event_time_day=2023-03-28/
s3://datalake/sales_table/event_time_day=2023-03-29/
s3://datalake/sales_table/event_time_day=2023-03-31/

**Table sales**

Partitioned by month(event_time) Partitioned by day(event_time)

| 2023-01 | 2023-02 | 2023-03 | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 |
| | | 6 | 7 | 8 | 9 | 10 |
| | | 11 | 12 | 13 | 14 | 15 |
| | | 16 | 17 | 18 | 19 | 20 |
| | | 21 | 22 | 23 | 24 | 25 |
| | | 26 | 27 | 28 | 29 | 30 |
| | | 31 | | | | |

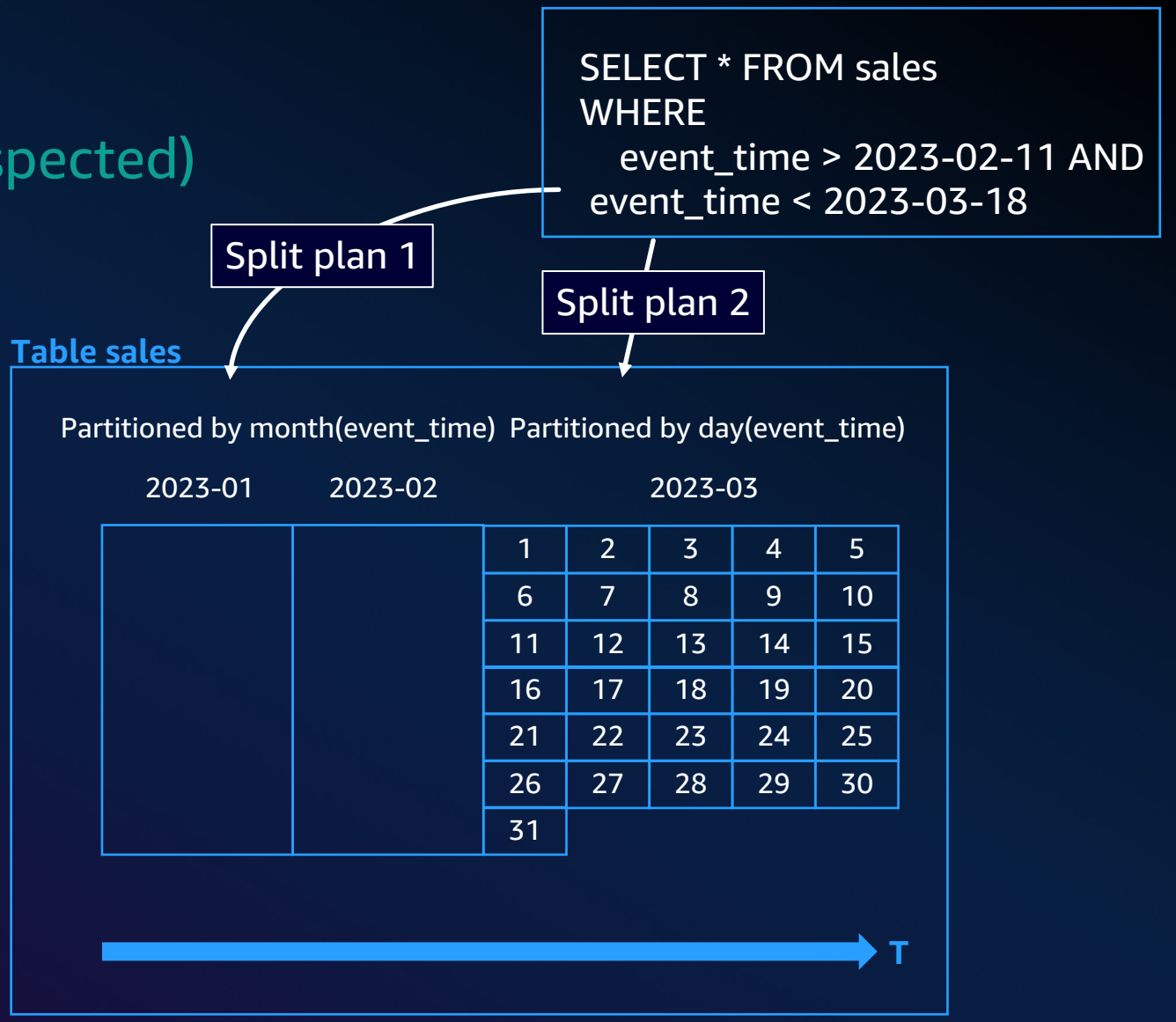
T

# Iceberg Partition Evolution

## Query example (partitions inspected)

Table Location
s3://datalake/sales_table/

Partitions
s3://datalake/sales_table/event_time_month=2023-01/
s3://datalake/sales_table/event_time_month=2023-02/
s3://datalake/sales_table/event_time_day=2023-03-01/
s3://datalake/sales_table/event_time_day=2023-03-02/
s3://datalake/sales_table/event_time_day=2023-03-03/
s3://datalake/sales_table/event_time_day=2023-03-04/
…
s3://datalake/sales_table/event_time_day=2023-03-17/
s3://datalake/sales_table/event_time_day=2023-03-18/
s3://datalake/sales_table/event_time_day=2023-03-19/
…
s3://datalake/sales_table/event_time_day=2023-03-28/
s3://datalake/sales_table/event_time_day=2023-03-29/
s3://datalake/sales_table/event_time_day=2023-03-31/

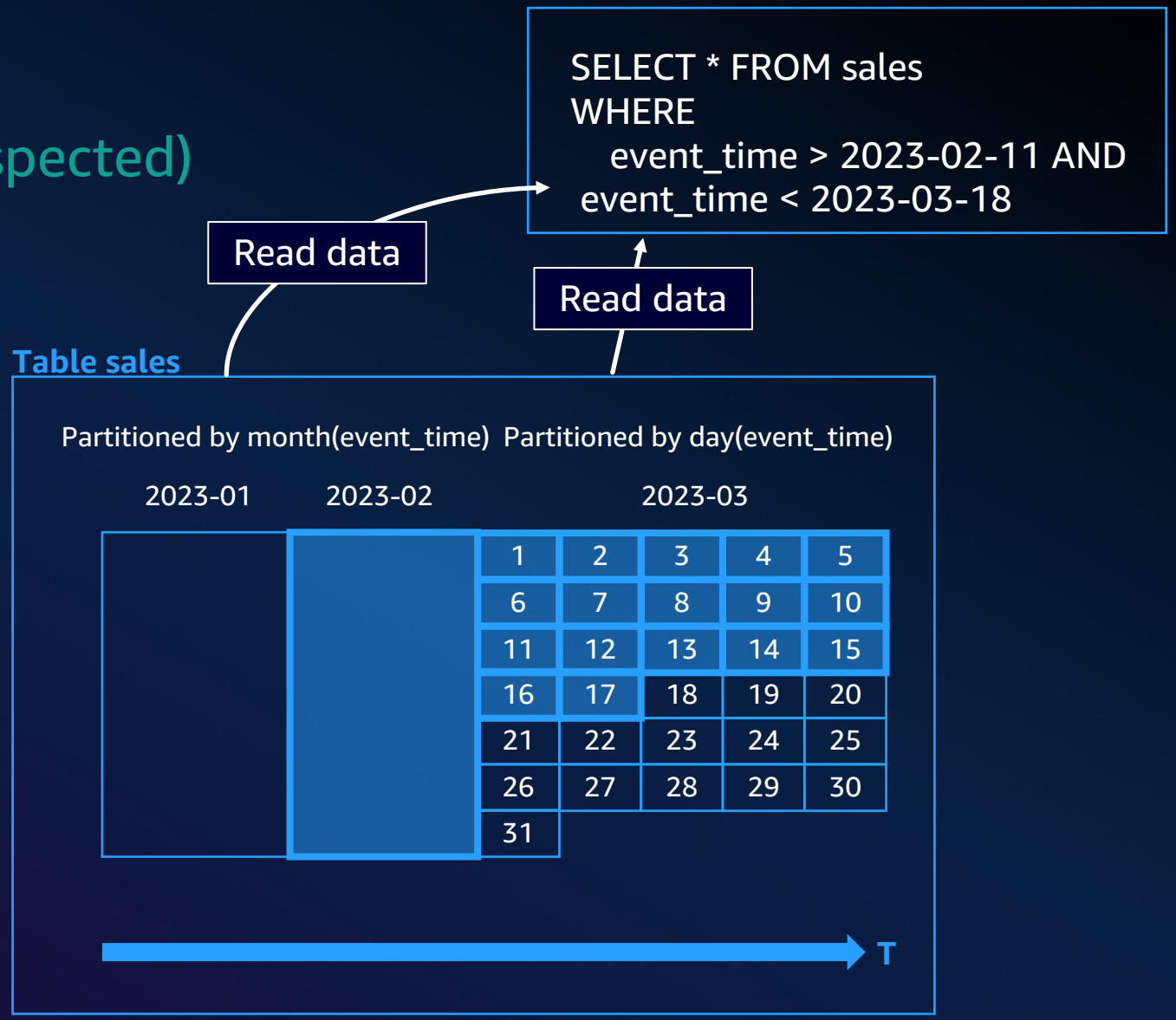


# Iceberg Partition Evolution

## Query example (partitions inspected)

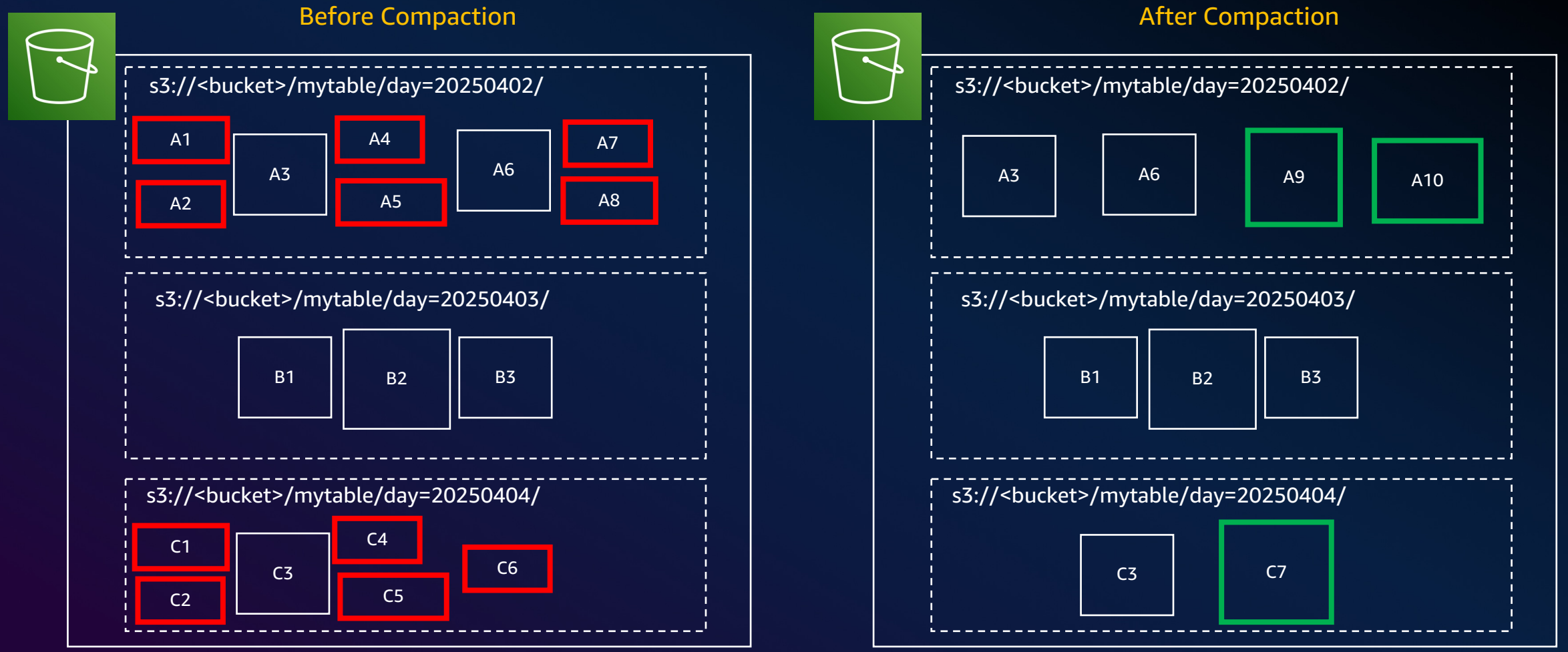
Table Location
s3://datalake/sales_table/

Partitions
s3://datalake/sales_table/event_time_month=2023-01/
s3://datalake/sales_table/event_time_month=2023-02/
s3://datalake/sales_table/event_time_day=2023-03-01/
s3://datalake/sales_table/event_time_day=2023-03-02/
s3://datalake/sales_table/event_time_day=2023-03-03/
s3://datalake/sales_table/event_time_day=2023-03-04/
…
s3://datalake/sales_table/event_time_day=2023-03-17/
s3://datalake/sales_table/event_time_day=2023-03-18/
s3://datalake/sales_table/event_time_day=2023-03-19/
…
s3://datalake/sales_table/event_time_day=2023-03-28/
s3://datalake/sales_table/event_time_day=2023-03-29/
s3://datalake/sales_table/event_time_day=2023-03-31/



# Iceberg Compaction

Iceberg detects small files inside partitions and compact them



# Compaction without Sorting

| order_id | product_id | units_sold |
|---|---|---|
| 9 | P01 | 80 |
| 5 | P09 | 60 |
| 1 | P50 | 40 |

| order_id | product_id | units_sold |
|---|---|---|
| 8 | P11 | 10 |
| 4 | P90 | 1000 |
| 2 | P16 | 50 |

| order_id | product_id | units_sold |
|---|---|---|
| 3 | P89 | 15 |
| 7 | P67 | 600 |
| 6 | P03 | 90 |

Spark SQL

```
spark.sql(f"""
   CALL system.rewrite_data_files(
      table => 'iceberg_db.orders_table’
   )
""")
```

# Compaction without Sorting

| order_id | product_id | units_sold |
|---|---|---|
| 9 | P01 | 80 |
| 5 | P09 | 60 |
| 1 | P50 | 40 |

| order_id | product_id | units_sold |
|---|---|---|
| 8 | P11 | 10 |
| 4 | P90 | 1000 |
| 2 | P16 | 50 |

| order_id | product_id | units_sold |
|---|---|---|
| 3 | P89 | 15 |
| 7 | P67 | 600 |
| 6 | P03 | 90 |



Compaction without sorting

| order_id | product_id | units_sold |
|---|---|---|
| 9 | P01 | 80 |
| 5 | P09 | 60 |
| 1 | P50 | 40 |
| 8 | P11 | 10 |
| 4 | P90 | 1000 |

| order_id | product_id | units_sold |
|---|---|---|
| 2 | P16 | 50 |
| 3 | P89 | 15 |
| 7 | P67 | 600 |
| 6 | P03 | 90 |

# Compaction with Sorting

| order_id | product_id | units_sold |
|---|---|---|
| 9 | P01 | 80 |
| 5 | P09 | 60 |
| 1 | P50 | 40 |

| order_id | product_id | units_sold |
|---|---|---|
| 8 | P11 | 10 |
| 4 | P90 | 1000 |
| 2 | P16 | 50 |

| order_id | product_id | units_sold |
|---|---|---|
| 3 | P89 | 15 |
| 7 | P67 | 600 |
| 6 | P03 | 90 |

Spark SQL

```
spark.sql(f"""
   CALL system.rewrite_data_files(
      table => 'iceberg_db.orders_table’,
      strategy => 'sort’,
      sort_order => 'order_id ASC NULLS LAST’
   )
""")
```

# Compaction with Sorting

| order_id | product_id | units_sold |
|---|---|---|
| 9 | P01 | 80 |
| 5 | P09 | 60 |
| 1 | P50 | 40 |

| order_id | product_id | units_sold |
|---|---|---|
| 8 | P11 | 10 |
| 4 | P90 | 1000 |
| 2 | P16 | 50 |

| order_id | product_id | units_sold |
|---|---|---|
| 3 | P89 | 15 |
| 7 | P67 | 600 |
| 6 | P03 | 90 |

compaction with sorting (by order_id)

During query plans we can skip more efficiently data files if we have a filter on order_id using column stats (eg., MAX/MIN)

File 1

| order_id | product_id | units_sold |
|---|---|---|
| 1 | P50 | 40 |
| 2 | P16 | 50 |
| 3 | P89 | 15 |
| 4 | P90 | 1000 |
| 5 | P09 | 60 |

File 2

| order_id | product_id | units_sold |
|---|---|---|
| 6 | P03 | 90 |
| 7 | P67 | 600 |
| 8 | P11 | 10 |
| 9 | P01 | 80 |

Iceberg Metadata

File1:
- MIN order_id: 1
- MAX order_id : 5

File2:
- MIN order_id :6
- MAX order_id :9

# Compaction with Sorting

| order_id | product_id | units_sold |
|---|---|---|
| 9 | P01 | 80 |
| 5 | P09 | 60 |
| 1 | P50 | 40 |

| order_id | product_id | units_sold |
|---|---|---|
| 8 | P11 | 10 |
| 4 | P90 | 1000 |
| 2 | P16 | 50 |

| order_id | product_id | units_sold |
|---|---|---|
| 3 | P89 | 15 |
| 7 | P67 | 600 |
| 6 | P03 | 90 |

compaction with sorting (by order_id)

```
SELECT *
FROM
iceberg_db.orders_table
WHERE order_id = 4
```

File 1

| order_id | product_id | units_sold |
|---|---|---|
| 1 | P50 | 40 |
| 2 | P16 | 50 |
| 3 | P89 | 15 |
| 4 | P90 | 1000 |
| 5 | P09 | 60 |

File 2

| order_id | product_id | units_sold |
|---|---|---|
| 6 | P03 | 90 |
| 7 | P67 | 600 |
| 8 | P11 | 10 |
| 9 | P01 | 80 |

Iceberg Metadata

File1:
- MIN order_id: 1
- MAX order_id : 5

File2:
- MIN order_id :6
- MAX order_id :9

# Compaction with Sorting

| order_id | product_id | units_sold |
|---|---|---|
| 9 | P01 | 80 |
| 5 | P09 | 60 |
| 1 | P50 | 40 |

| order_id | product_id | units_sold |
|---|---|---|
| 8 | P11 | 10 |
| 4 | P90 | 1000 |
| 2 | P16 | 50 |

| order_id | product_id | units_sold |
|---|---|---|
| 3 | P89 | 15 |
| 7 | P67 | 600 |
| 6 | P03 | 90 |

compaction with sorting (by order_id)

```
SELECT *
FROM
iceberg_db.orders_table
WHERE order_id = 4
```

File 1

| order_id | product_id | units_sold |
|---|---|---|
| 1 | P50 | 40 |
| 2 | P16 | 50 |
| 3 | P89 | 15 |
| 4 | P90 | 1000 |
| 5 | P09 | 60 |

File 2

| order_id | product_id | units_sold |
|---|---|---|
| 6 | P03 | 90 |
| 7 | P67 | 600 |
| 8 | P11 | 10 |
| 9 | P01 | 80 |

Iceberg Metadata

File1:
- MIN order_id: 1
- MAX order_id : 5

File2:
- MIN order_id :6
- MAX order_id :9



Only File 1 is inspected (4 is outside MIN-MAX of FILE 2)

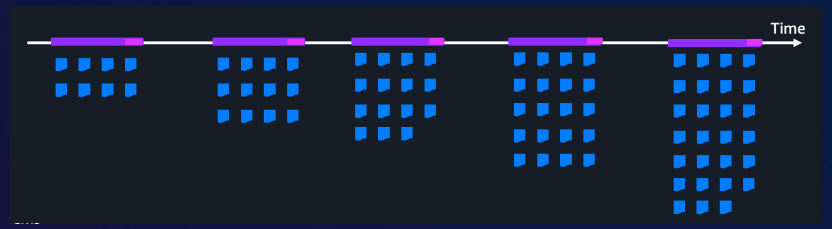
# Iceberg tables need maintenance

**Every write creates a new snapshot chain:**


*Snapshots accumulate rapidly in high-throughput environments consuming storage (impact on cost) and increasing Iceberg metadata (impact on performance)*

→ Expire Snapshots

**Write operations often generate small files:**




*Small files increase both S3 calls to open them as well as increase Iceberg metadata (impact on performance)*

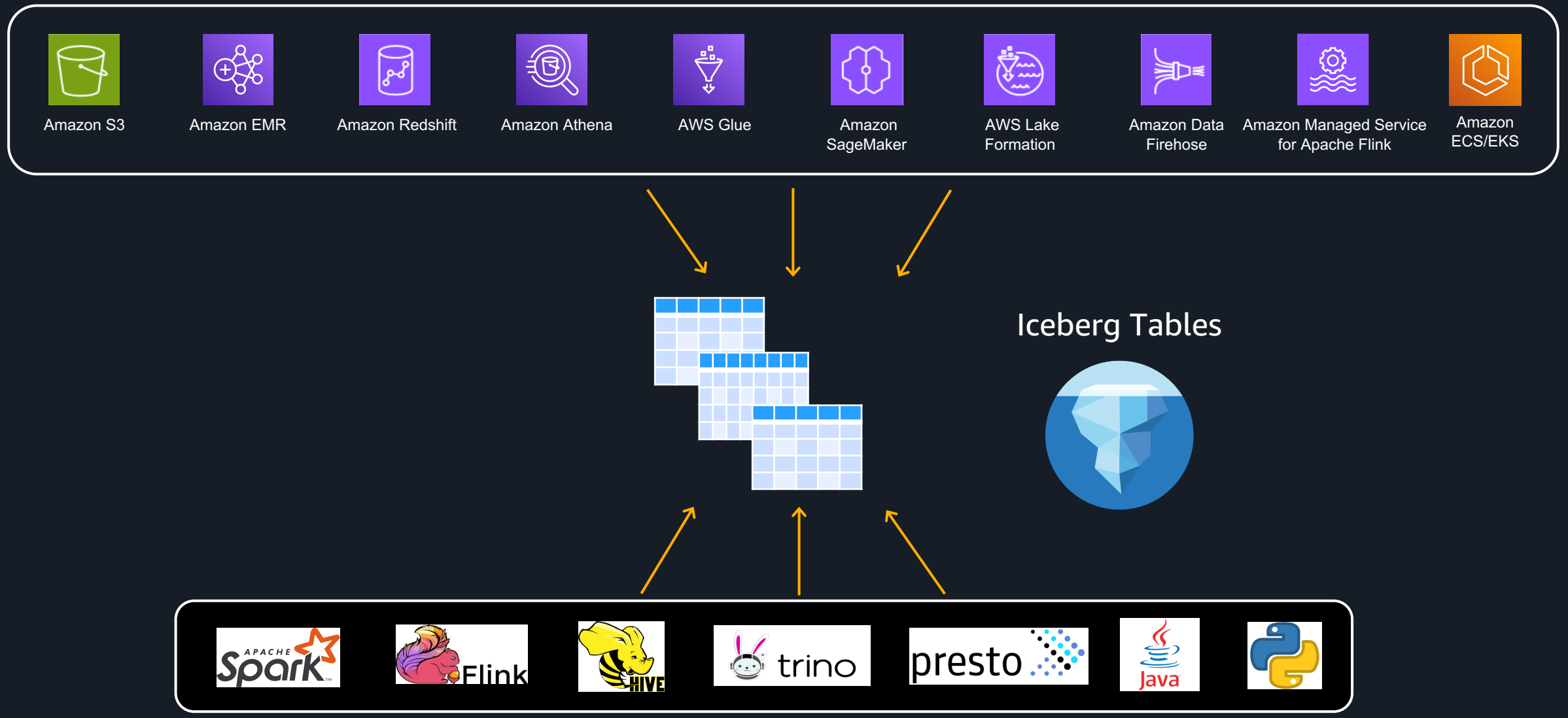
→ Compaction

**Failed jobs may leave unreferenced files:**


*Unreferenced files are unused files on storage (impact on costs)*

→ Remove Unreferenced Files

# One Iceberg table, many engines



# Key Takeaways

Eight ideas to carry forward.

1 🔒 **ACID Compliance**

Atomic metadata commits ensure readers always see complete, valid snapshots. No dirty reads, no partial writes

2 📂 **Iceberg Metadata Architecture**

A layered metadata tree with two levels of indexing enables efficient query planning and minimal data scanning

3 ✏️ **CRUD Operations**

Native SQL operations rewrite only affected files. No partition rewrites, full reader/writer isolation

4 ⏪ **Time Travel & Rollbacks**

Snapshot history enables querying any past version and reverting table state with a single metadata pointer swap

5 📦 **Compaction**

A single command merges small files. With optional sorting to produce tight column statistics for better pruning

6 🪄 **Hidden Partitioning**

Partition transforms automate partition routing at write time and partition pruning at read time. No extra columns or filters

7 🔄 **Schema & Partition Evolution**

Column and partition changes are metadata-only operations. No data rewrites, no consumer disruption

8 ✅ **Multi-Engine Interoperability**

One Iceberg table, many engines — Glue, Athena, and Amazon Quick all reading from the same source of truth

# Building a fully managed data lakehouse architecture with AWS

# Agenda

- **Building the data lakehouse** – Production-ready architecture patterns
- **Demo**

# From self managed to AWS managed

## Self-managed Apache Iceberg Lakehouse

- High operational overhead
- Manual performance optimization
- Manual & complex cost optimization
- S3 throttling errors at large scale
- Complex custom multi-region replication

## AWS Managed Apache Iceberg Lakehouse

**Amazon S3 tables**

- Zero operational overhead
- 3X faster queries with auto optimizations
- 80% In-built cost-optimization
- 10x higher transactions per second
- Native multi region replication

# We take care of all of that for you…

Amazon S3 Tables


EASY ADOPTION

AUTOMATIC STORAGE OPTIMIZATION

EASY DATA RETENTION

- Fully managed storage for Apache Iceberg Data Lake
- 10x Transactions per second compared to standard Amazon S3 buckets
- APIs to manage the tables
- Managed Compaction – Binpack, sort, z-order
- Data Governance and fine grained access control(FGAC)
- Iceberg Rest Catalog Endpoint (IRC)
- 3rd party Integrations
- S3 Table MCP Servers
- Variant data type support (Iceberg V3 spec)

# Building the data lakehouse

# 3 Main Steps

## Ingestion

Data ingestion from a variety of data sources into the data lake

## Governance

Define access controls, protect data from un-authorized access while ensuring data is accessible easily for authorized personnel

## Consumption

Enable data driven decision making using the data from data lakes/ Use a variety of consumption tools for various use-cases

# Ingestion


## Zero ETL Integrations

Low/No code solutions. Easy to setup and manage. Minimizes the need to build ETL data pipelines

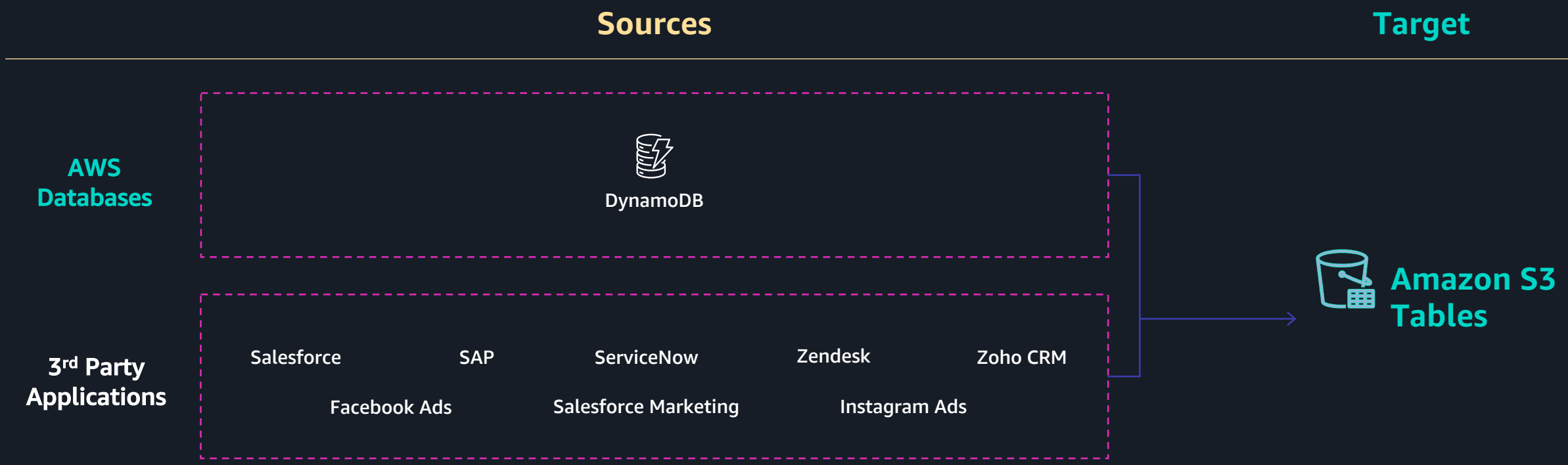
## Streaming Ingestion

Reliable ingestion of high-speed data at scale

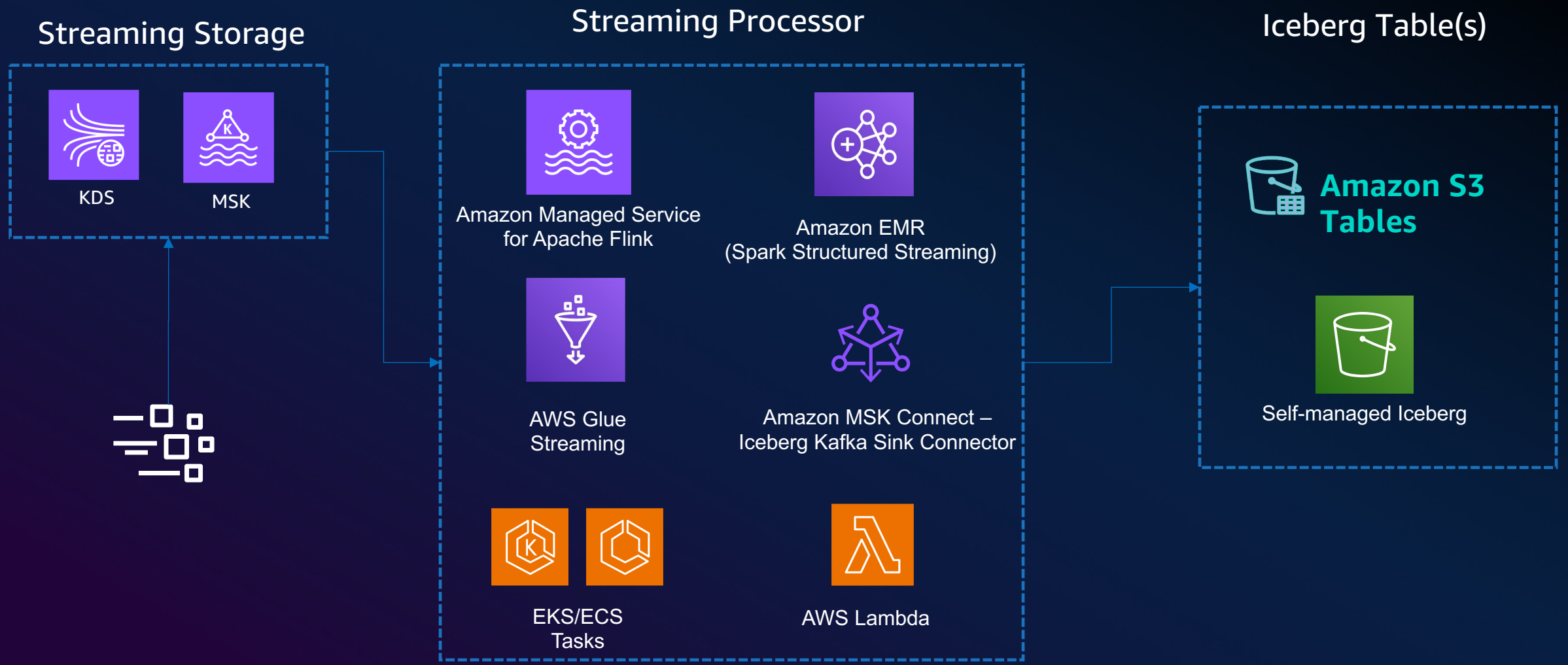
## Batch Ingestion

Large datasets from a wide variety of sources

# Zero-ETL supports ingestion to S3 Tables

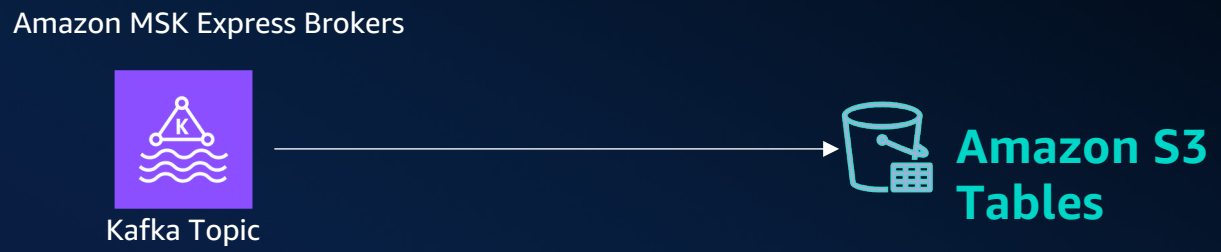


# Stream data into Iceberg tables

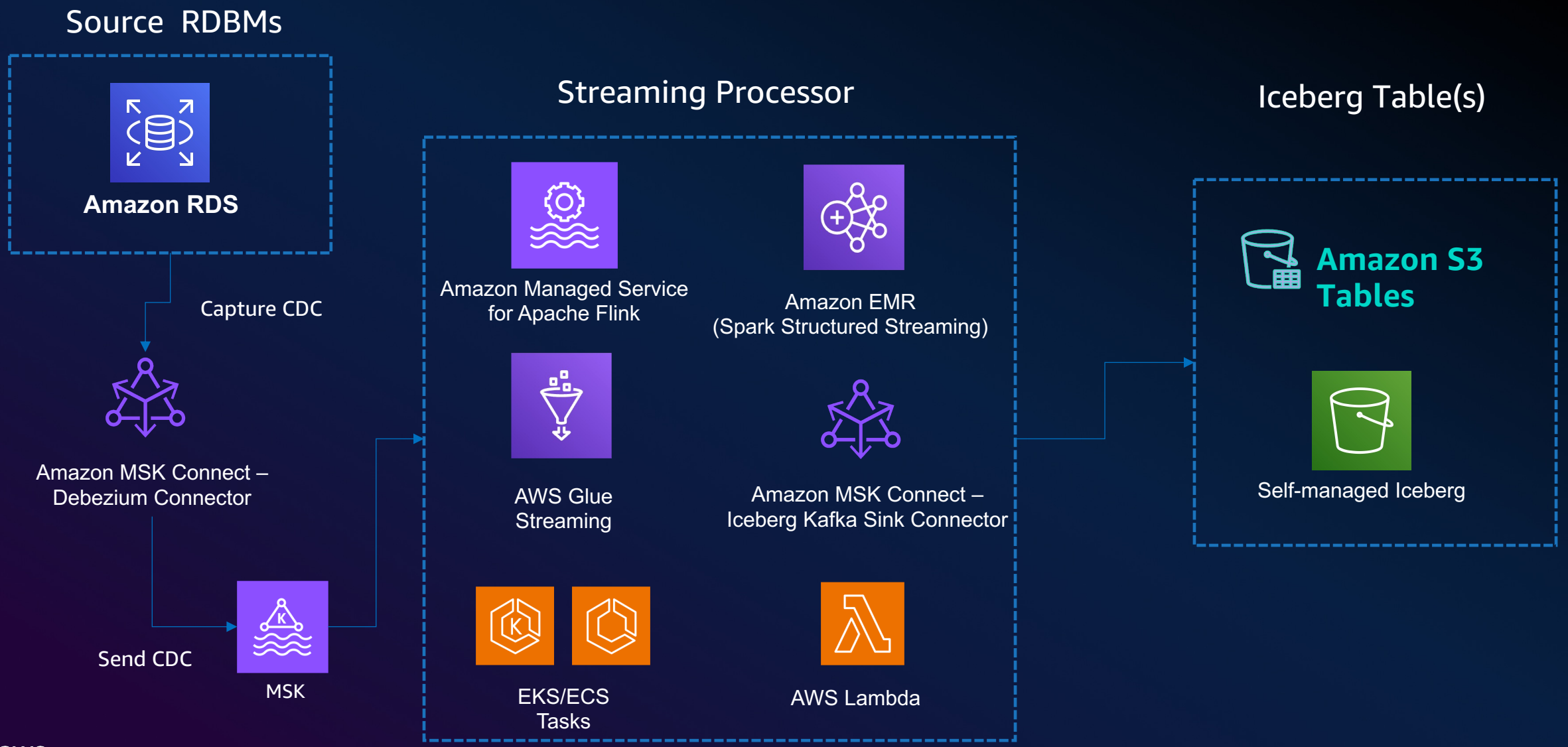


# New - Amazon MSK Express – streaming tables for Apache Iceberg

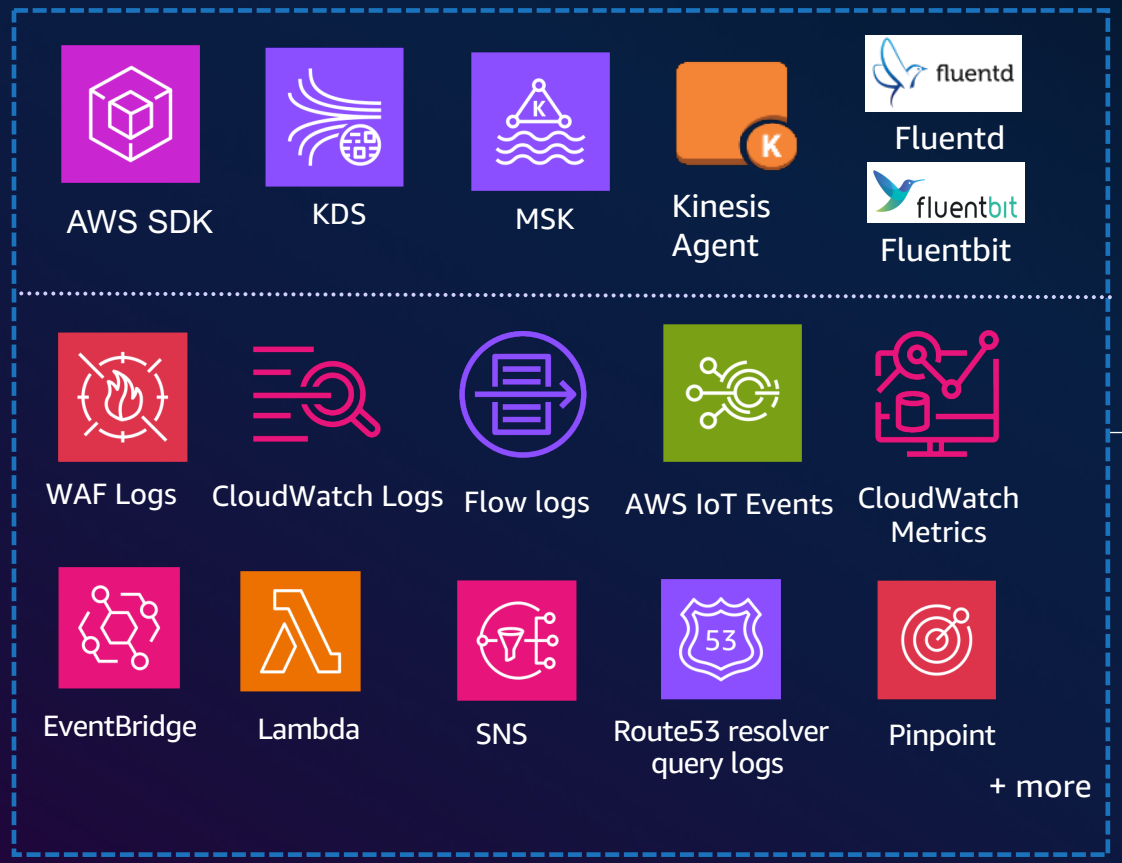
- Continuously materializes Kafka topics as Iceberg tables on S3 Tables - fully managed
- No connectors, Flink jobs, or custom consumers needed
- **30% lower** downstream query costs via optimized file sizing
- Auto-discoverable in Glue Data Catalog; enrichable with business context

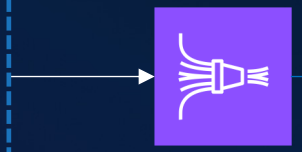


# Stream CDC into Iceberg tables



# Stream data into S3 tables using Firehose

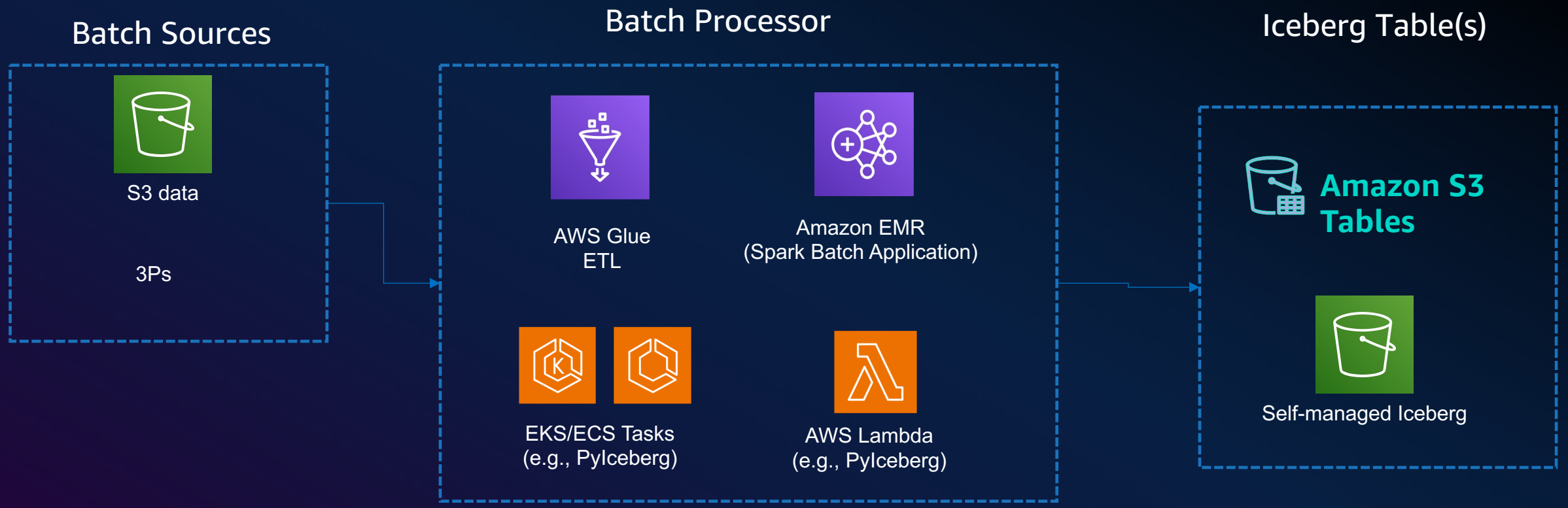



**Amazon S3 Tables**

- Route one stream to many S3 tables
- Automatic insert, update & deletes
- Exactly-once delivery semantics
- Delivers data in seconds

# Batch insert data into Iceberg tables



# Integrate using Iceberg REST endpoint

DuckDB Labs
snowflake
dremio
HighByte
trino
ICEBERG
Starburst
daft
StreamNative
Apache Spark
Flink
PuppyGraph

Glue Iceberg REST endpoint

S3 Tables Iceberg REST endpoint

# Discover, govern, share, and collaborate on data and AI securely


Data classification | GenAI-powered business MD curation | Data quality | Data lineage | Publishing and subscriptions workflows | API driven | Use-case based projects | Open standards and formats

# Building a scalable foundation

ORGANIZE DATA, PEOPLE, AND TOOLS. DEFINE YOUR BUSINESS LANGUAGE.

Enterprise-wide business data catalog

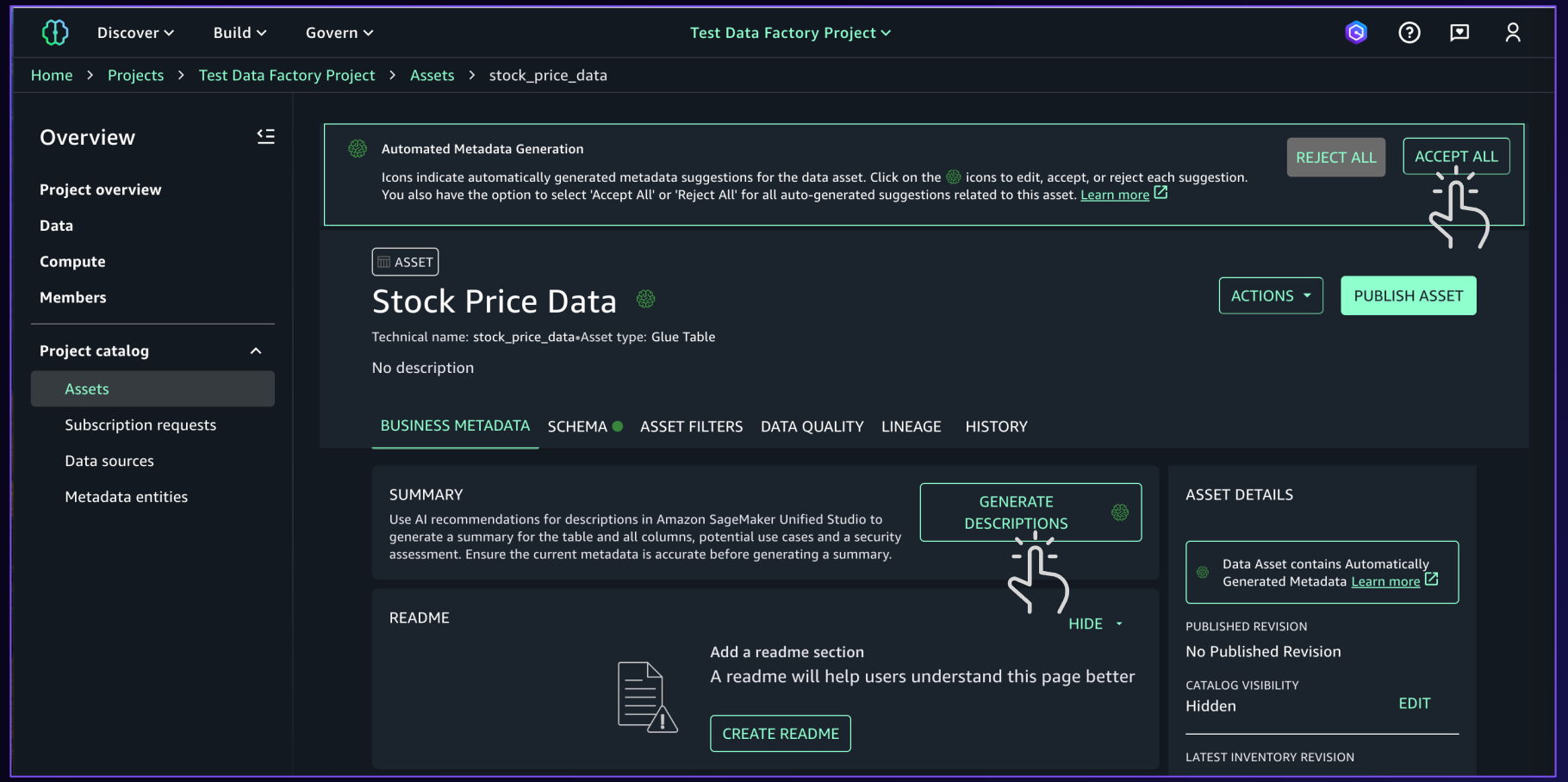
Organizational domains

Business glossary

Metadata curation

Build a common language to streamline communication and collaboration

# Scale your catalog with GenAI powered business metadata generation



# Capture and visualize data lineage

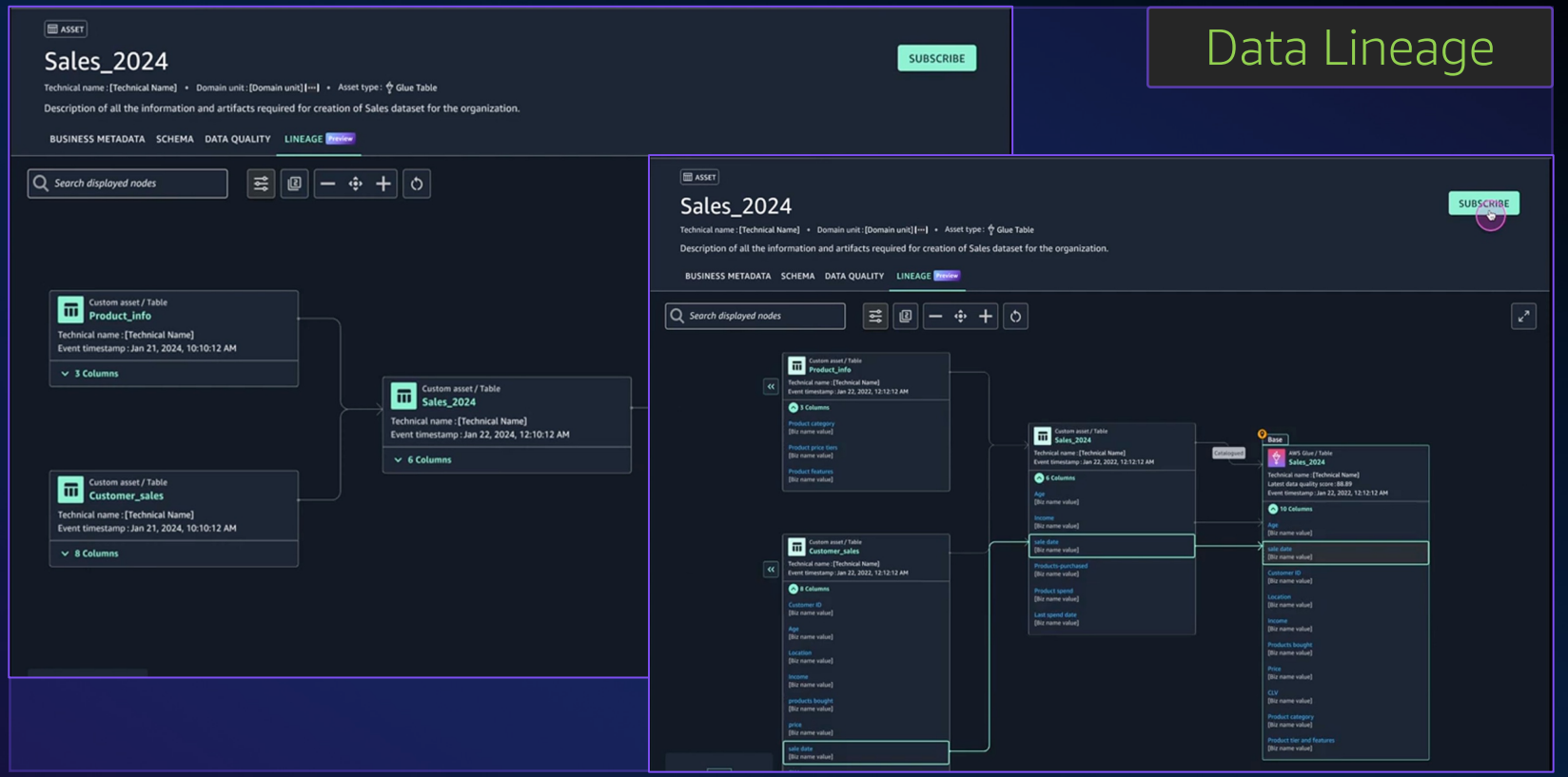
## Help ensure GDPR compliance

### Highlights

- API driven
- OpenLineage compliant
- Interactive visualization
- Asset & column lineage w/ versioning

### Common use cases:

- Track provenance, storage, processing, and uses of data
- Impact analysis
- Troubleshooting
- Help identify and fix non-compliance



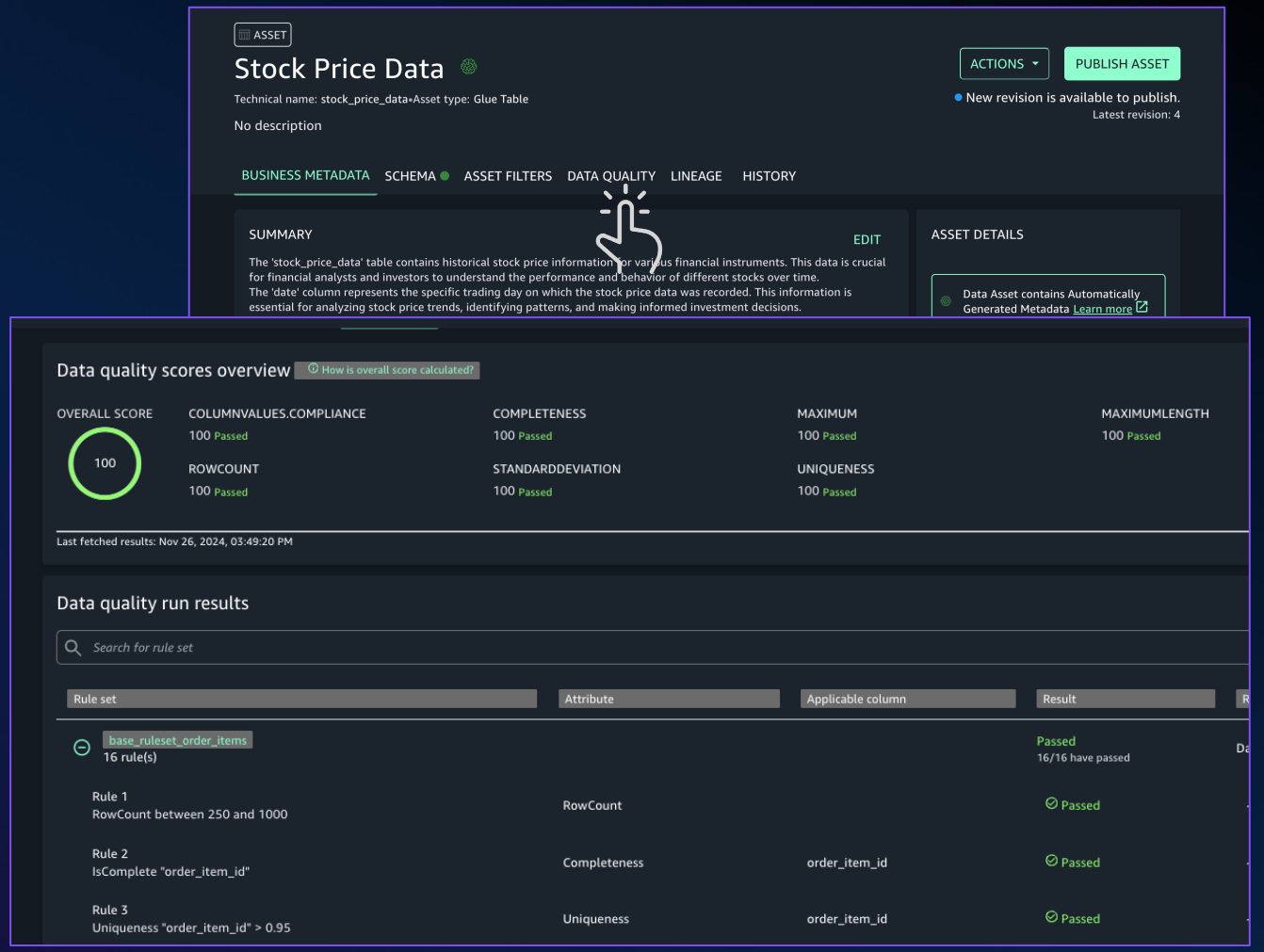
# Drive better insights and AI outcomes with data quality.

## Highlights:

- Built on top of open-source DeeQu framework
- APIs available for 3P and programmatic entry
- Asset and column view of data quality scores
- Leverage ML to detect anomalies

## Use cases in focus:

- Technical data validation
- Business-specific requirements and regulatory compliance
- Minimize potential for bias, hallucinations, and inaccurate results



# Consumption

AWS Analytics services

Amazon SageMaker Unified Studio

Amazon QuickSuite

Any Apache Iceberg compatible engine

Machine learning (ML)

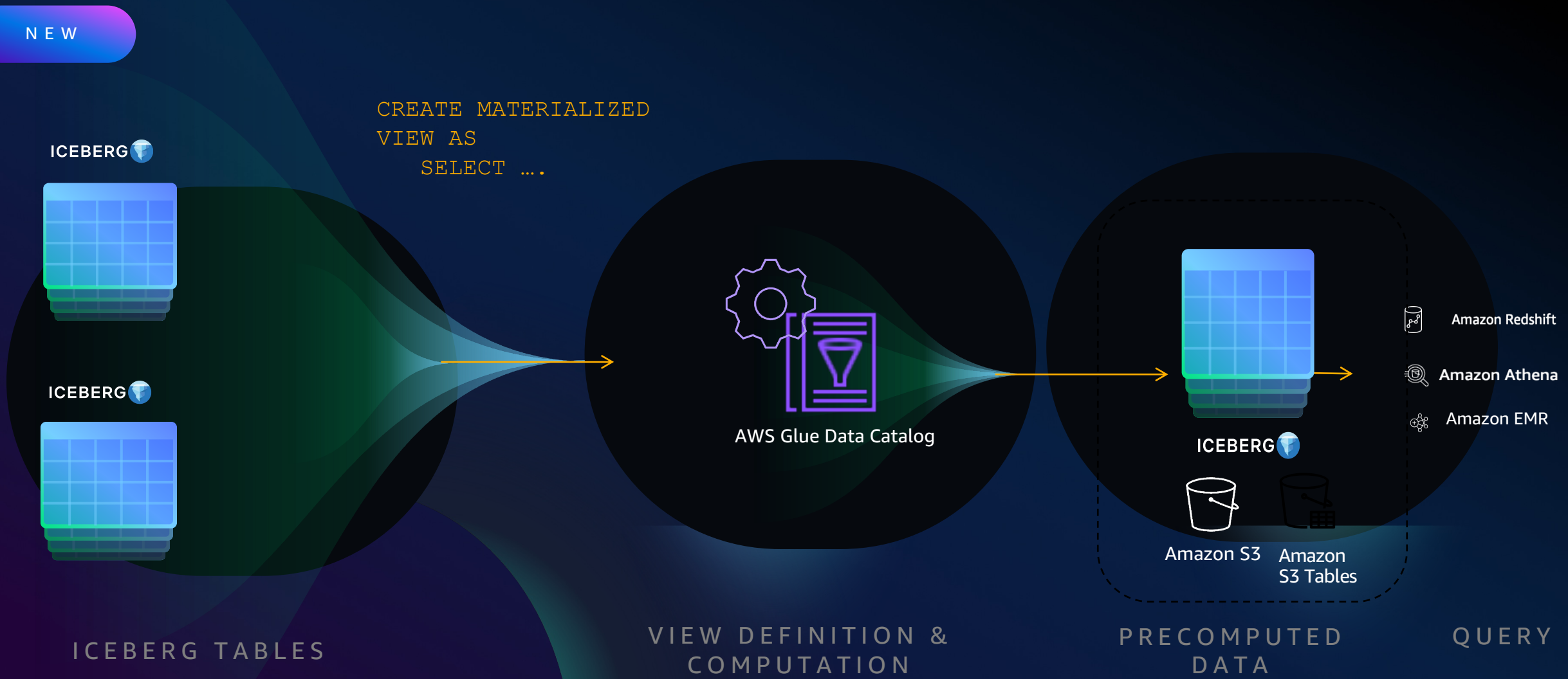
Generative AI

MCP Server

# S3 Tables integration with AWS analytics

# Materialized Views in AWS Glue

NEW



# Real time Analytics on AWS With Apache Iceberg

# Agenda

Core Concepts: CDC & Ingestion Patterns

Ingestion Patterns – Append & Upsert

Streaming Ingestion to Iceberg Tables with Streaming Frameworks

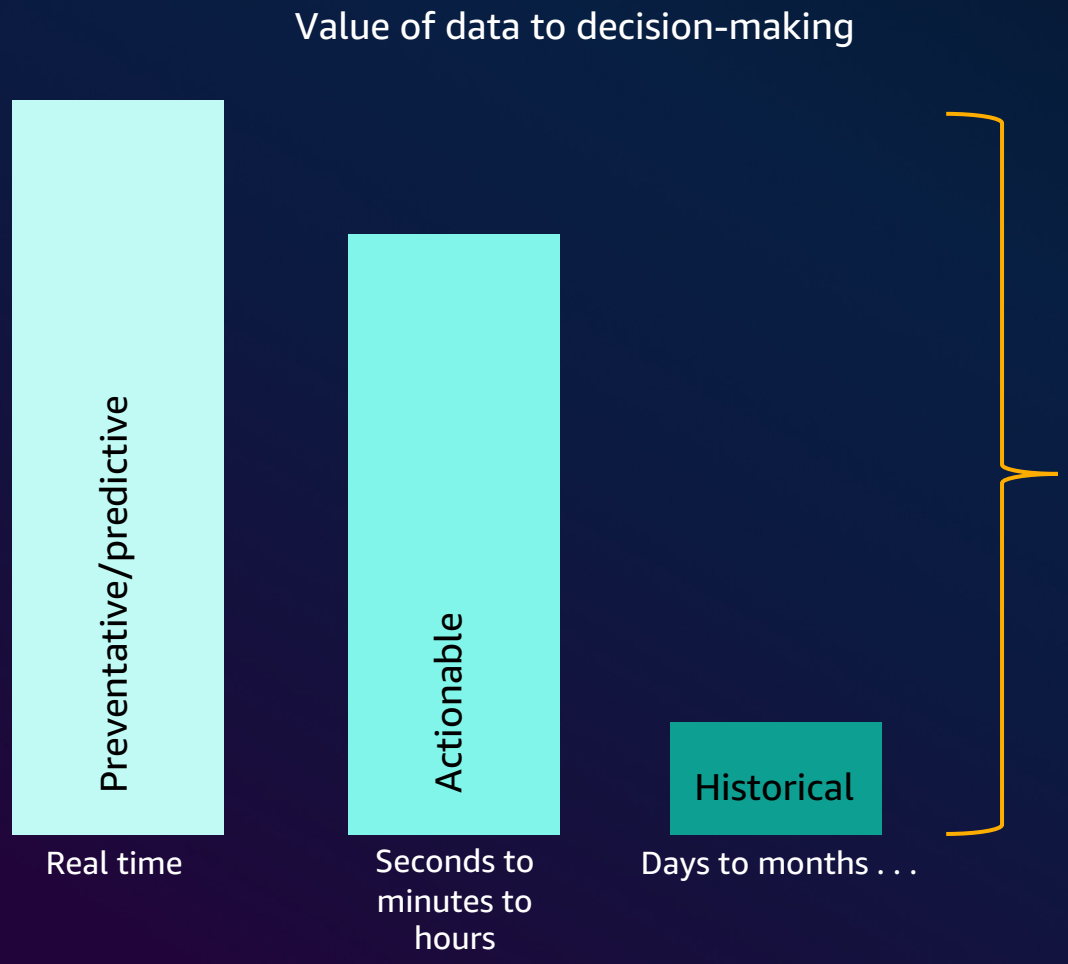
Low Code / No Code Streaming Ingestion to Iceberg Tables

Customer Use Cases

Resources

# The value of data diminishes over time

Data has a short shelf life of actionability; real-time data lets you act **as fast as the market dictates**




**75%** of enterprise organizations surveyed believe untimely data inhibited business opportunities[1]

The value of the data for insights reduces by **50%** in 8 hours[2]

[1] InterSystems - Data management challenges survey
[2] Real-time analytics

# Streaming Architecture Basics

# Real-World Use Cases for Streaming Analytics

Real-Time Personalization

Fraud Detection

Predictive Maintenance

Customer 360° Analytics

Supply Chain Optimization

# Streaming Into a Data Lake

- High-throughput, near real-time ingestion
- Query your data instantly upon arrival
- Streaming read – incremental data processing

# Streaming With Apache Iceberg

- High-throughput, near real-time ingestion
- Query your data instantly upon arrival
- Streaming read – incremental data processing

**Scalability and Performance**

**Upsert and record-level modifications**

**Schema Evolution**

**Snapshot Isolation**

**Time Travel**

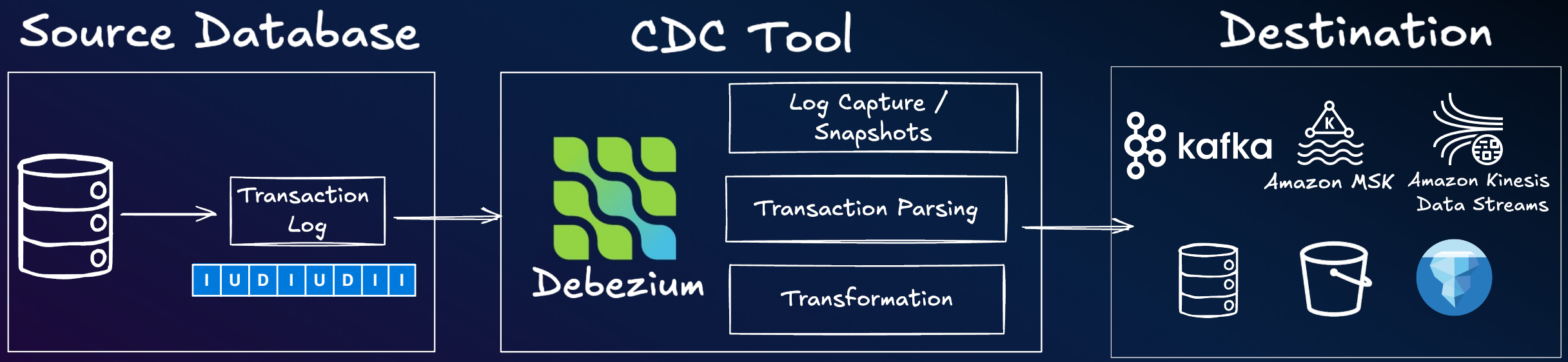
# Streaming With Apache Iceberg

- CDC – Streaming data changes in a database
- Analytics with Stream processing frameworks
- Unifying Batch and Stream Processing

# Core Concepts: CDC & Ingestion Patterns

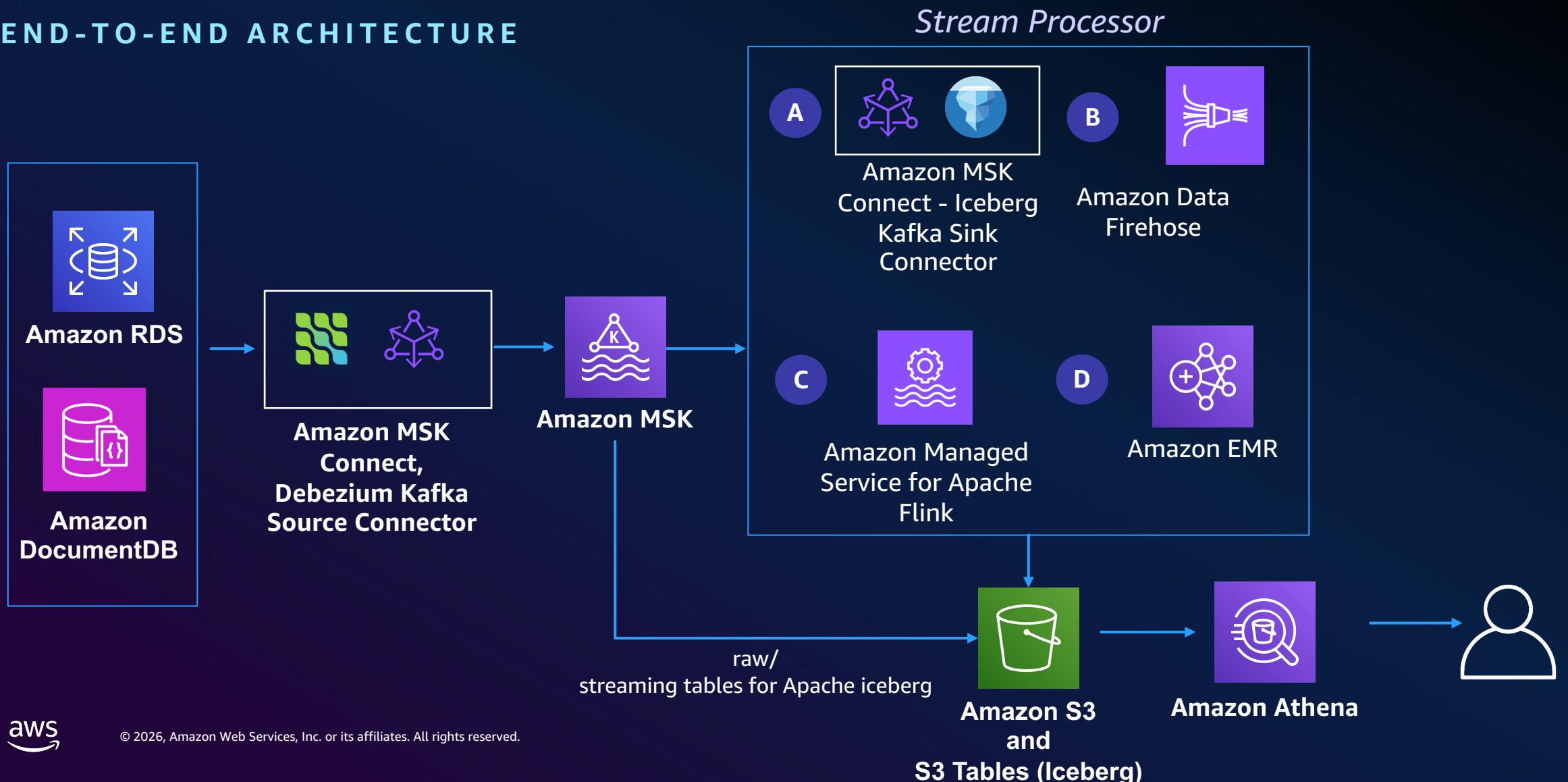
# Streaming Data Changes in a Database

## LOG-BASED CDC WITH DEBEZIUM



# Example: CDC to Iceberg with Debezium Kafka Source Connector

END-TO-END ARCHITECTURE



# Ingestion Patterns – Append & Upsert

# Writing stream events to the target

```
CREATE TABLE
iceberg_catalog.db_name.user_clicks (
    user STRING,
    cnt BIGINT,
    last_updated TIMESTAMP
)
USING iceberg
PARTITIONED BY (bucket(128, user))
COMMENT 'Stores the number of clicks
per user with a timestamp'
```

# Writing stream events to the target

### Append Write

```
INSERT INTO user_clicks
 (user, cnt, last_updated)
VALUES ('user123', 5, '2024-09-08
12:34:56');
```

### UPSERT Write

```
MERGE INTO user_clicks uc
USING changes c
ON uc.user = c.user
WHEN MATCHED AND c.operation = 'D' THEN
        DELETE
WHEN MATCHED THEN
        UPDATE SET
          uc.cnt = c.cnt,
          uc.last_updated = c.last_updated
WHEN NOT MATCHED AND c.operation != 'D'
        THEN INSERT (user, cnt, last_updated)
        VALUES (c.user, c.cnt, c.last_updated);
```

# Writing stream events to the target

**Append Write**

| user | cnt | op |
|---|---|---|
| | | |
| | | |
| | | |

**UPSERT Write**

| user | cnt | op |
|---|---|---|
| | | |
| | | |
| | | |

# Writing stream events to the target


**Append Write**

| user | cnt | op |
|---|---|---|
| Alex | 1 | I |
| | | |
| | | |

**UPSERT Write**

| user | cnt | op |
|---|---|---|
| Alex | 1 | I |
| | | |
| | | |

# Writing stream events to the target

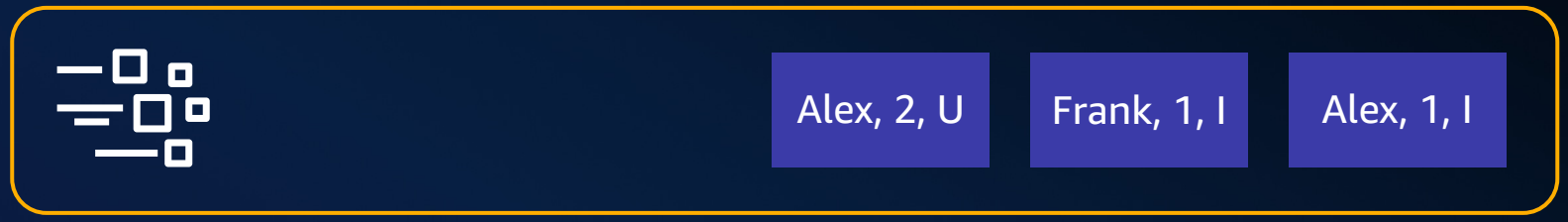

**Append Write**

| user | cnt | op |
|---|---|---|
| Alex | 1 | I |
| Frank | 1 | I |
| | | |
| | | |

**UPSERT Write**

| user | cnt | op |
|---|---|---|
| Alex | 1 | I |
| Frank | 1 | I |
| | | |
| | | |

# Writing stream events to the target



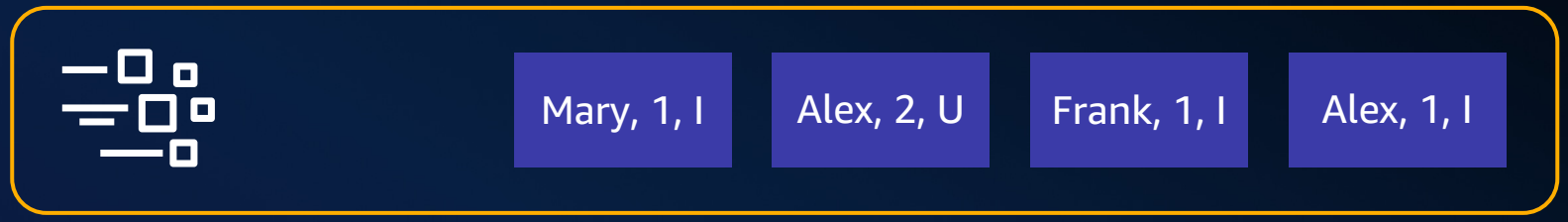

**Append Write**

| user | cnt | op |
|---|---|---|
| Alex | 1 | I |
| Frank | 1 | I |
| Alex | 2 | U |
| | | |

**UPSERT Write**

| user | cnt | op |
|---|---|---|
| Alex | 2 | U |
| Frank | 1 | I |
| | | |

# Writing stream events to the target



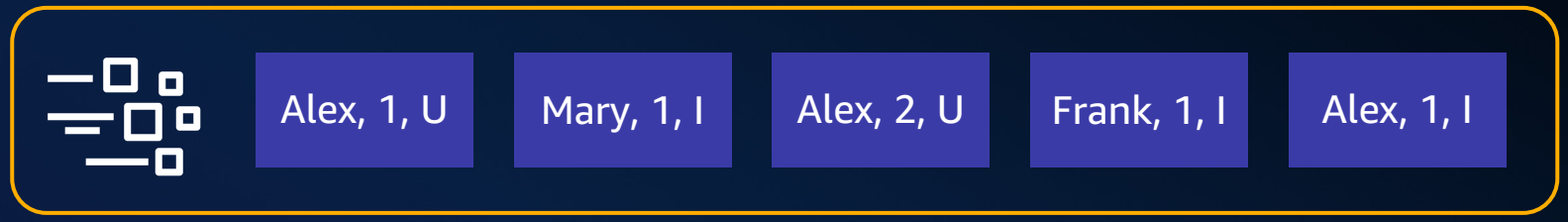

**Append Write**

| user | cnt | op |
|---|---|---|
| Alex | 1 | I |
| Frank | 1 | I |
| Alex | 2 | U |
| Mary | 1 | I |

**UPSERT Write**

| user | cnt | op |
|---|---|---|
| Alex | 2 | U |
| Frank | 1 | I |
| Mary | 1 | I |

# Writing stream events to the target




**Append Write**

| user | cnt | op |
|---|---|---|
| Alex | 1 | I |
| Frank | 1 | I |
| Alex | 2 | U |
| Mary | 1 | I |
| Alex | 1 | U |

**UPSERT Write**

| user | cnt | op |
|---|---|---|
| Alex | 1 | U |
| Frank | 1 | I |
| Mary | 1 | I |

# Append Only vs UPSERT - Trade-Offs

## Append Only

| user | cnt | op |
|---|---|---|
| Alex | 1 | I |
| Frank | 1 | I |
| Alex | 2 | U |
| Mary | 1 | I |
| Alex | 1 | U |

- Captures all data changes
- Accurate History
- Simple write logic
- Requires additional "Merge" step
- Complicated read logic

## UPSERT

| user | cnt | op |
|---|---|---|
| Alex | 1 | U |
| Frank | 1 | I |
| Mary | 1 | I |

- Change history is lost
- Complicated write logic – double updates problem
- Efficiency vs latency dilema
- Write amplification
- Simple to query

# Streaming Ingestion to Iceberg Tables with Streaming Frameworks

# Streaming Frameworks

Flink


Event Aggregation

Data Routing

Windowing

Filtering

Complex Event Processing (CEP)

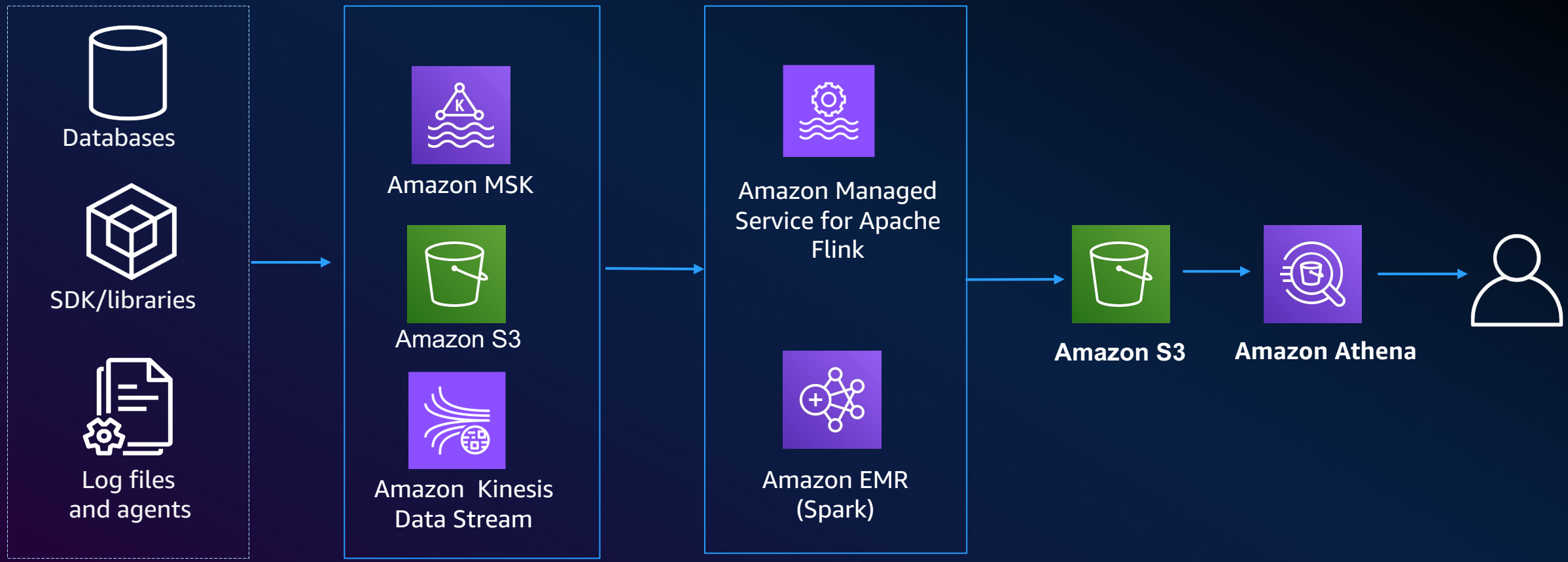
Enrichment

Streaming Joins

Rule Engine

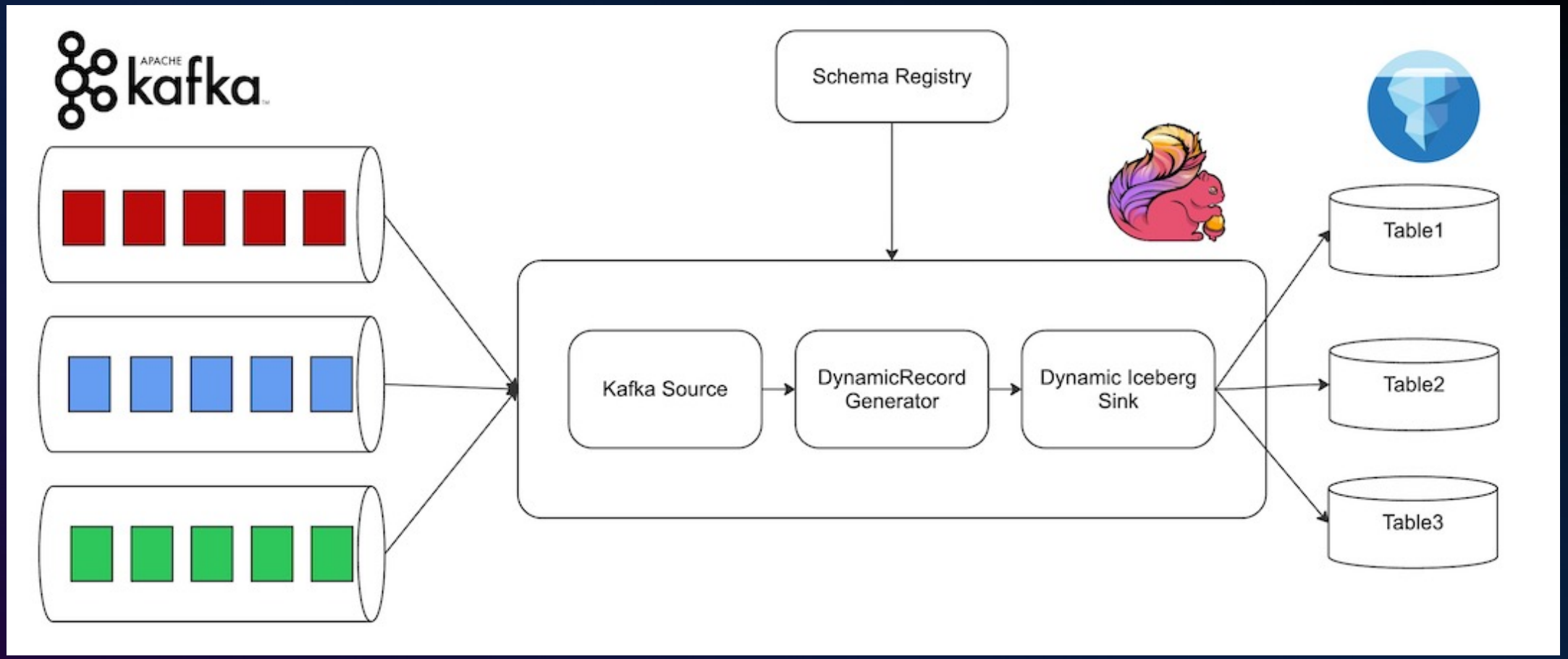
Event Correlation

# Streaming from/to Iceberg Tables with Streaming Frameworks



# Streaming with Apache Flink

# Writing to Iceberg using Flink



# Apache Iceberg Connector - Features

- Available for DataStream & SQL API
- Batch and Stream Processing
- Leverages Flink Checkpoints for Exactly Once Delivery
- Source / Sink Connectors (Read & Write)
- Streaming Queries for append only tables
- Append / Overwrite / Upsert Write Operations
- Branch Writes
- Embedded Maintenance Operations
- Schema Evolution

# Streaming with Apache Spark Structured Streaming

# Apache Iceberg with Spark Structured Streaming

*Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine*

Iceberg Integration:

- Uses Apache Spark's DataSourceV2 API
- Reads: Supports reading data incrementally from an append only snapshots
- Writes: Uses DataStreamWriter to stream the writes to Iceberg tables
- Spark processes data streams in **micro-batches**, with Iceberg tables committed after each trigger interval

| | Apache Spark | Flink |
|---|---|---|
| **DDL** | yes | Limited ALTER TABLE |
| **Snapshot Management (rollback, set current snapshot and more)** | yes | no |
| **Metadata Management ( snapshot expiration, remove orphan files , rewrite manifest and more)** | yes | yes |
| **Compact Data Files** | yes | yes |
| **Table Migration** | yes | no |

# Streaming to Iceberg - Caveats

# #1 UPSERTS - Double Updates Problem

```
MERGE INTO user_clicks uc
USING changes c
ON uc.user = c.user
WHEN MATCHED AND c.operation = 'D' THEN
        DELETE
WHEN MATCHED THEN
        UPDATE SET
          uc.cnt = c.cnt,
          uc.last_updated = c.last_updated
WHEN NOT MATCHED AND c.operation != 'D'
        THEN INSERT (user, cnt, last_updated)
        VALUES (c.user, c.cnt, c.last_updated);
```

NON DETERMINISTICS UPDATES


All updates within a batch need to be consolidated into a single modification per row

- deduplicate rows
- take the latest update

# #2 Small Files

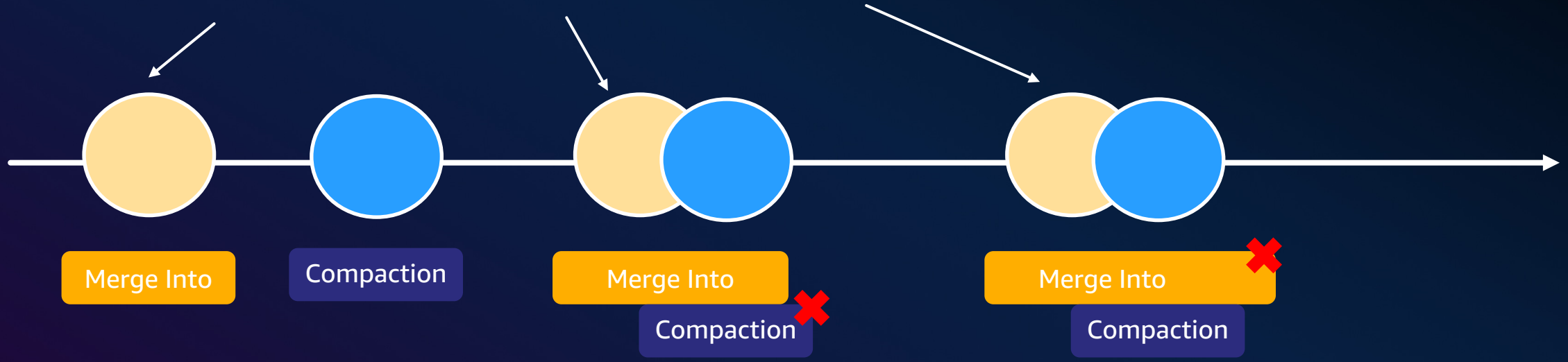
- High rates of commits
- Merge-on-Read Update
- No shuffle or sort before writing

**Small Files Problem**

# #3 Running table maintenance

## Naively scheduling Data Compaction

always running Spark or Flink application for Streaming with Iceberg




N ephemeral applications for data compaction

# #3 Running table maintenance

## Workaround to schedule Data Compaction in Spark

Answer: sequential scheduling


Spark Structured Streaming with:

- Micro batch set to 200M records
- Enough resources to process the micro batch asap
- Compaction limited to the last N partitions

# #3 Running table maintenance

## Data Compaction in Flink

Answer: New Iceberg 1.10 Sink Connector for Apache Flink Supports Embedded Table Maintenance Operations

### TableMaintenance

- Expire Snapshots
- Rewrite Data Files
- Delete Orphan Files

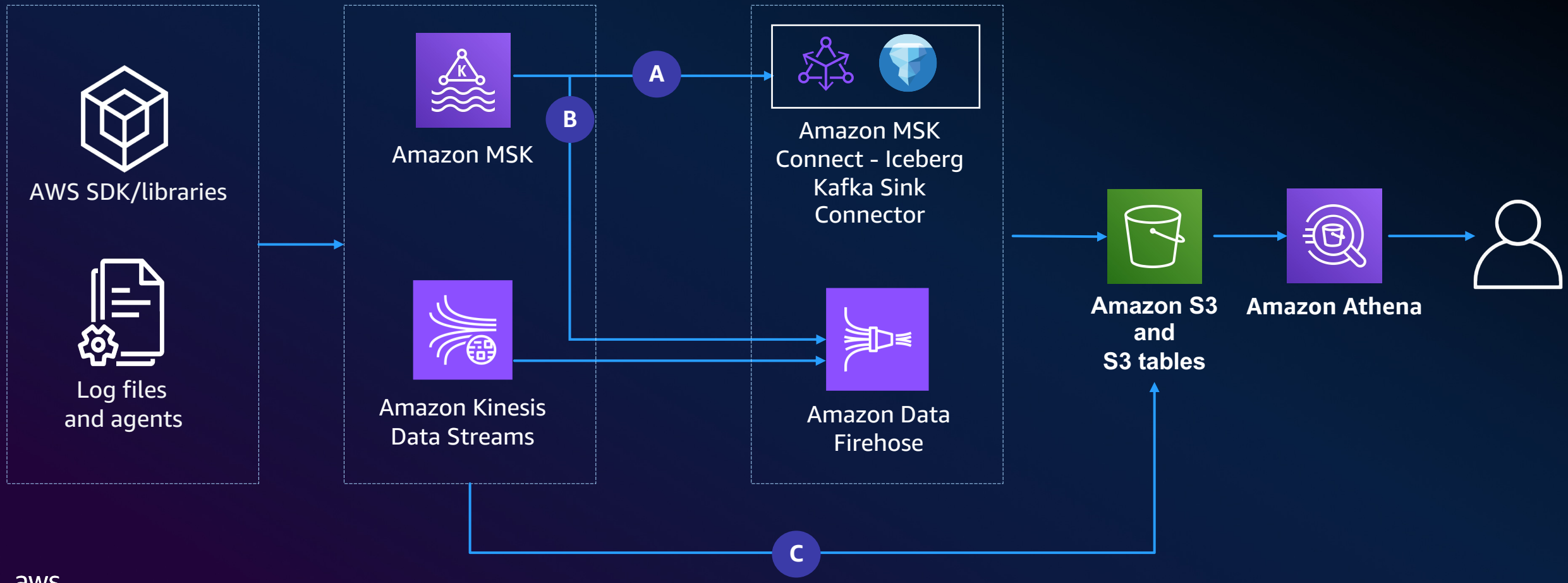
### Lock Management

- Concurrent Access
- Data Consistency
- Resource Management

# Low Code / No Code Streaming Ingestion to Iceberg Tables

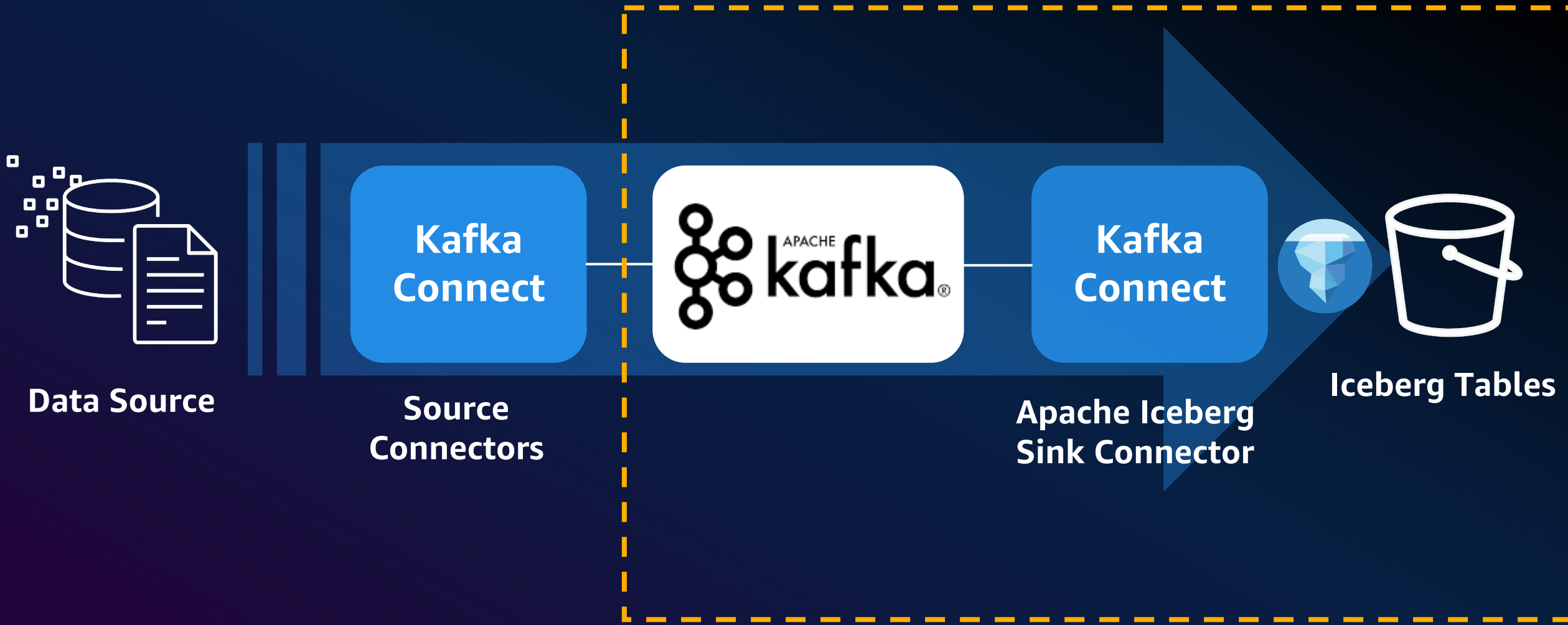
# Low Code Streaming Ingestion to Iceberg Tables

## Other Data Sources:



# Kafka Iceberg Sink Connector

# Apache Kafka Connect



# Ingesting with Kafka Connect and Apache Iceberg Sink Connector

https://iceberg.apache.org/docs/nightly/kafka-connect/

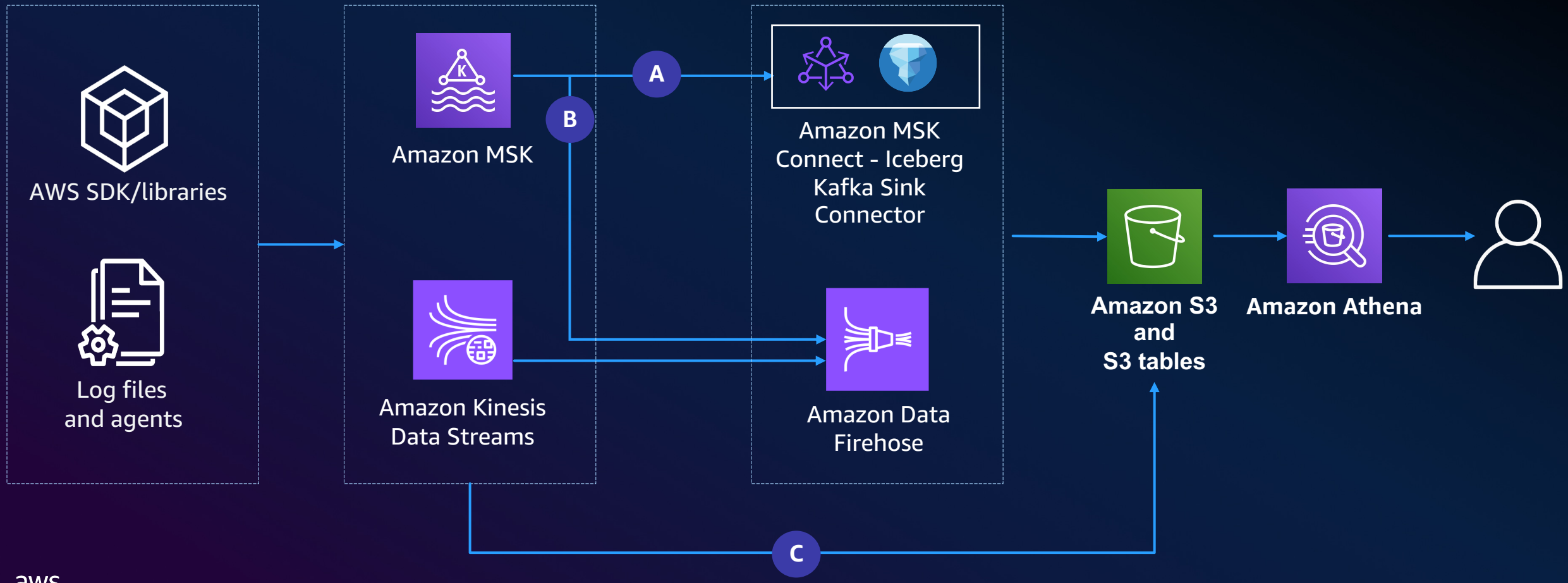
# Ingesting with Kafka Connect and Apache Iceberg Sink Connector

- Commit coordination and special commit topic for centralized Iceberg commits
- Exactly Once Delivery Semantics
- Multi-table fan-out
- Row mutations (update/delete) and upsert mode
- Automatic table creation and schema evolution

# Low Code Ingestion to Iceberg Tables with Amazon Data Firehose

# Low Code Streaming Ingestion to Iceberg Tables

## Other Data Sources:



# What Amazon Data Firehose does for you

## Capture
From 20+ Sources

- 20+ AWS Service Logs
  WAF, IoT, CloudWatch, SNS, etc.
- Amazon Kinesis Data Streams
- AWS SDK
  Put, PutBatch API
- Client Agents
  FluentD, FluentBit. Kinesis Agent
- Amazon MSK

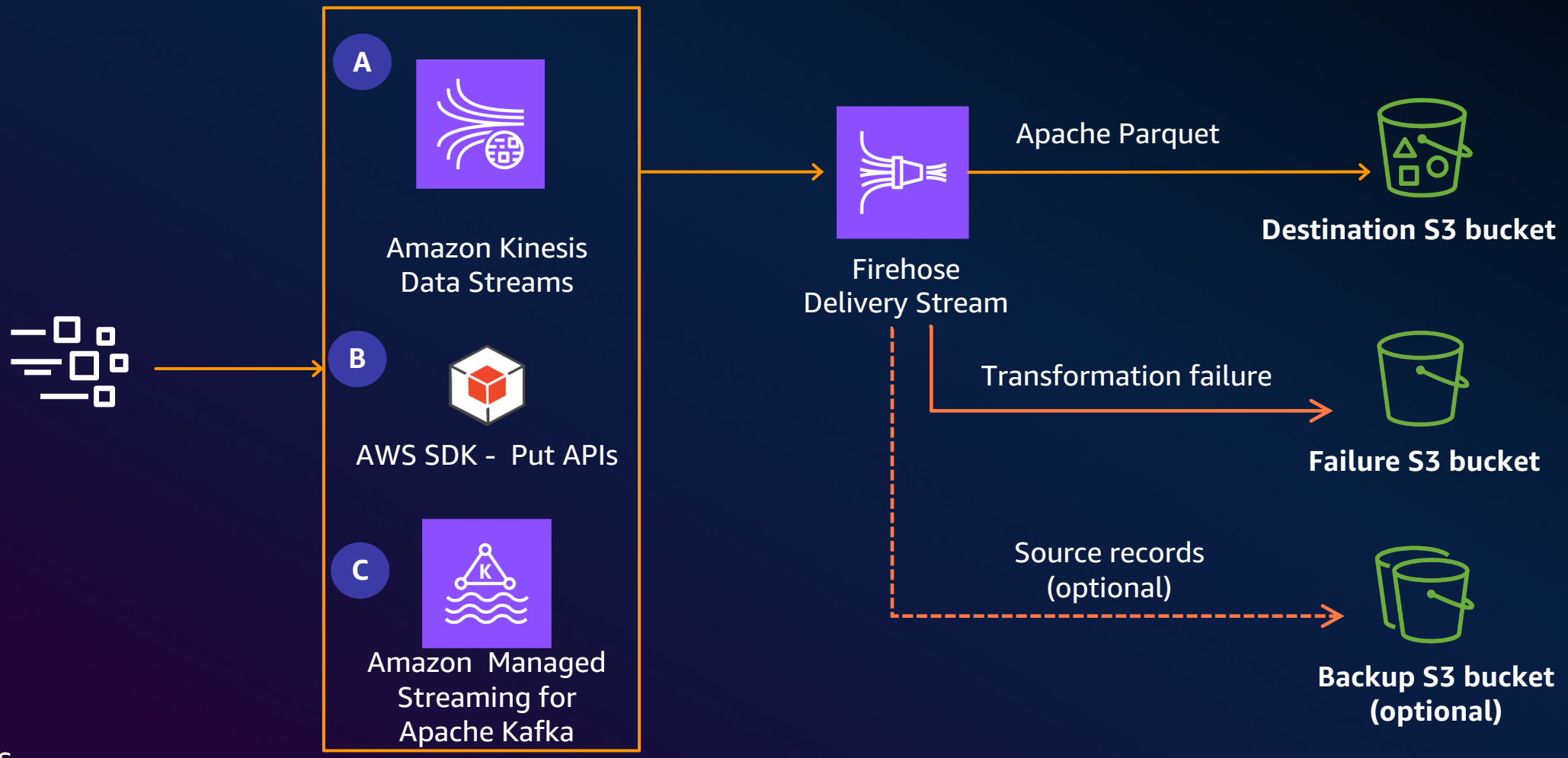
## Transform

- Convert to Parquet
- Compress and decompress Output
- Add Delimiters
  E.g, new line
- Batching
  For optimal output file size
- Create Dynamic Partitions for Amazon S3 buckets
- Perform custom transformations using AWS Lambda

## Deliver
To 15+ Destinations

- Amazon S3 / S3 Tables
- Amazon Redshift / Redshift Serverless
- Amazon OpenSearch / OpenSearch Serverless
- Splunk
- HTTP Endpoint
  (9+ partners such as Datadog, etc…)
- Snowflake
- Iceberg

# Ingesting with Amazon Data Firehose

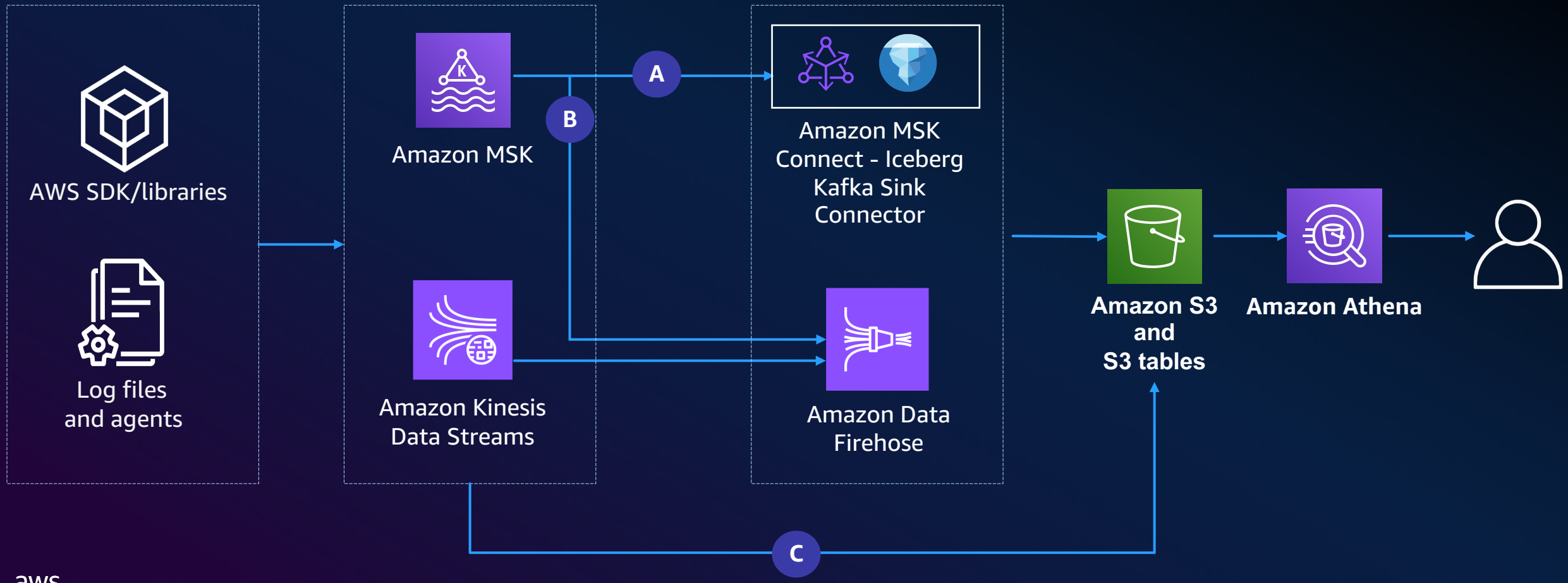


# streaming tables for Apache Iceberg on Amazon S3 Tables

A *native, fully managed* integration from *Amazon MSK Express brokers and Amazon Kinesis On-Demand and On-Demand Advantage* directly to *Apache Iceberg on Amazon S3 Tables*

# Low Code Streaming Ingestion to Iceberg Tables

## Other Data Sources:



# Amazon S3 Tables

Fully managed Apache Iceberg tables in S3

# Two Delivery Paths for the same streaming data

New

## streaming tables for Apache Iceberg on Amazon S3 Tables

*Near Real-Time Analytics*

- Building Lakehouse
- Dashboards & BI — Query within minutes
- ML inference & model training
- Data analysis
- Security analytics & threat reports

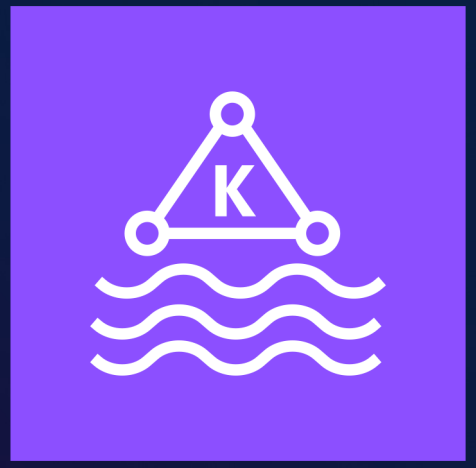
## Data delivery to general purpose Amazon S3 buckets

*Archive & Long-term Storage*

- Data archival & retention
- Backup & disaster recovery
- Audit & compliance (immutable)
- Replay & reprocessing

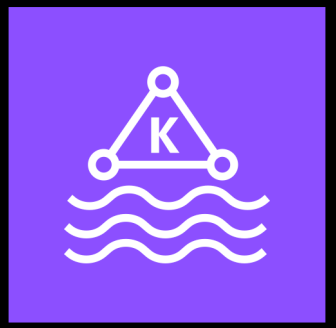
**Ingest once, deliver everywhere**

# Supported for both -



Amazon MSK Express Brokers

Amazon Kinesis – On-Demand and On-Demand Advantage



# Amazon MSK Express brokers

**3x** More throughout per broker

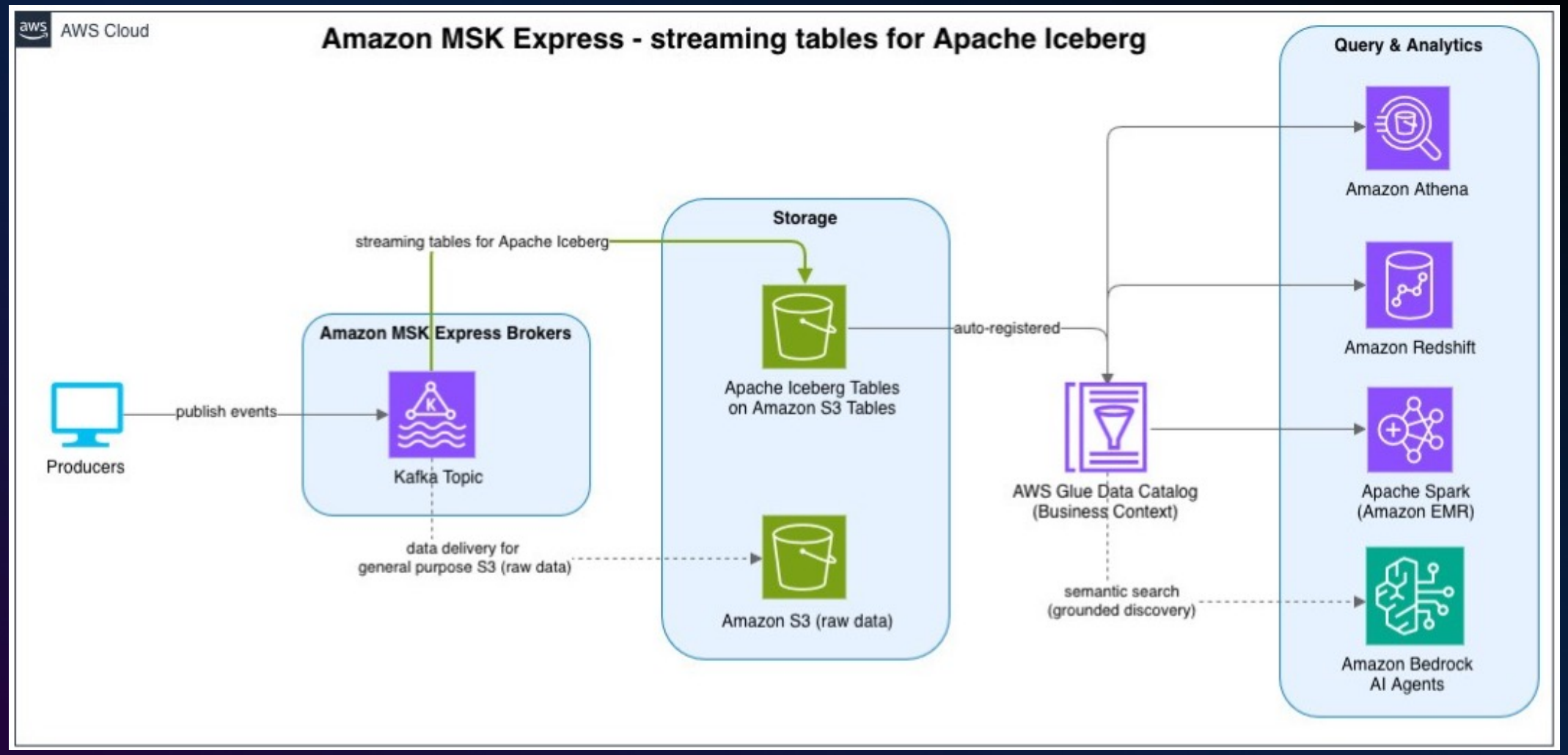
**20x** Faster scaling

**90%** Faster recovery

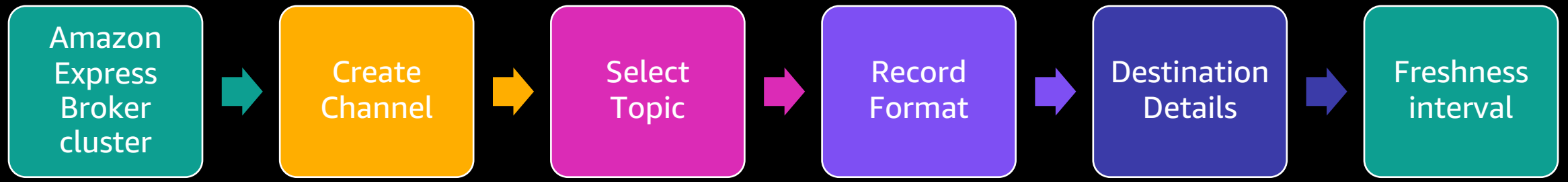
**180x** Faster Partition Rebalancing

**5x** Partition count per broker

# Streaming tables for Apache Iceberg
## with Amazon MSK Express



# Amazon MSK - Getting Started in Minutes




**Supported at Launch**

- ✓ Destinations: S3 Tables & S3 Buckets
- ✓ Input format: JSON
- ✓ Partitioning: Time-based
- ✓ Data freshness: 5-15 minutes
- ✓ Catalog: AWS Glue Data Catalog
- ✓ Mode: Append-only (new rows only)
- ✓ Monitoring: CloudWatch
- ✓ Interfaces: Console, SDK, API/CLI, CDK

# Amazon Kinesis

## *On-Demand*

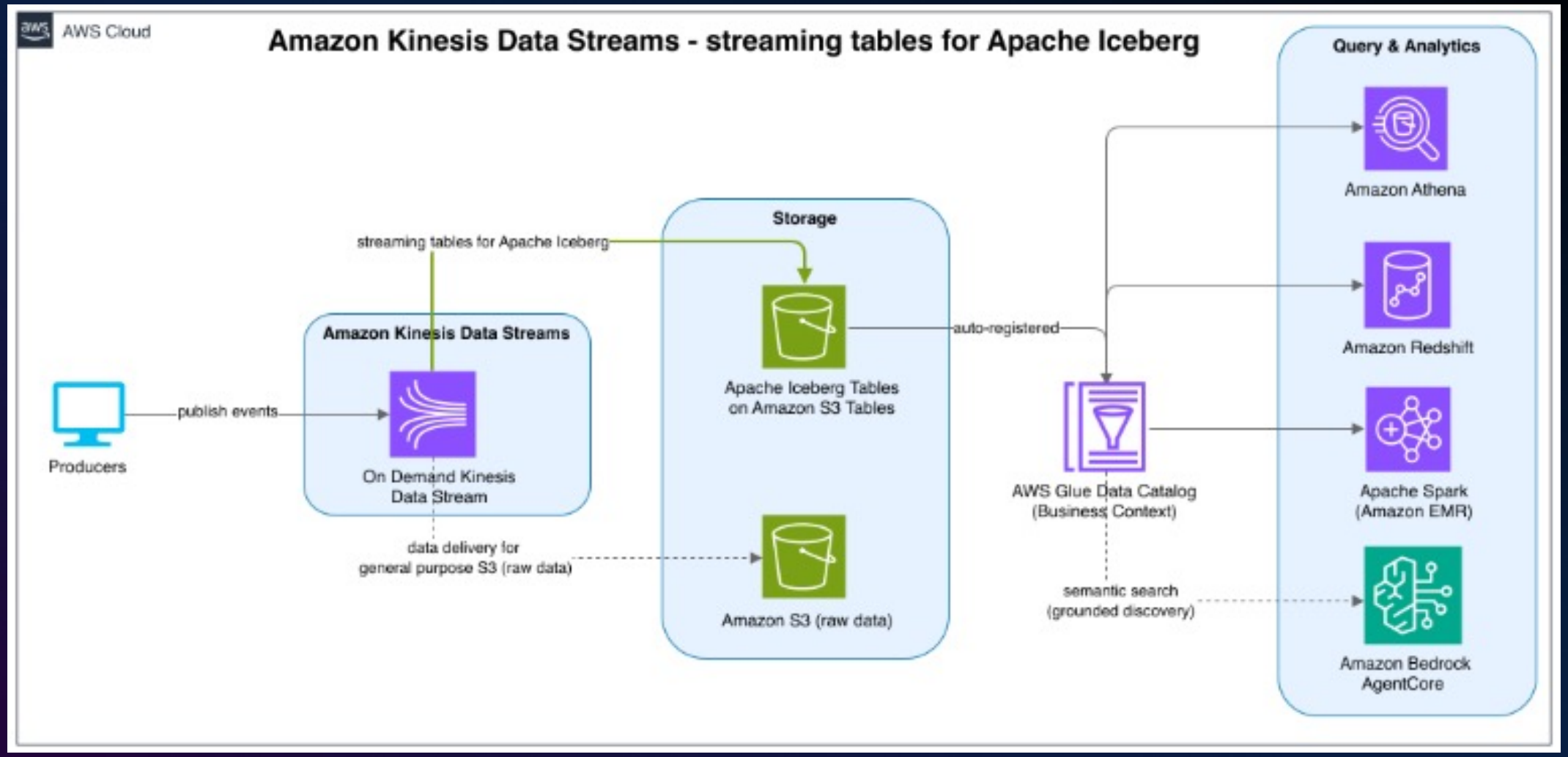
- **10 GB/s** of write and **20 GB/s** of read throughput
- Record sizes up to **10 MiB**
- Auto-scales with demand, no shard management

**On-Demand Advantage Mode**

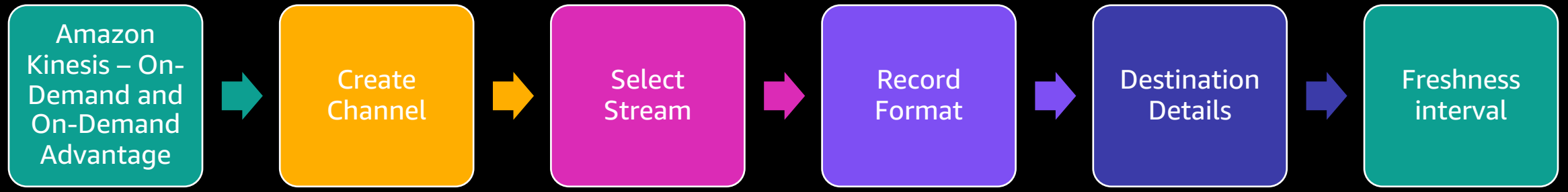
- Set *warm throughput* for On-Demand streams
- Simpler, lower pricing (**60%** savings) , no stream-hour charge
- No premium for Enhanced Fan-Out or Extended Retention
- **50** EFO consumers/ stream

# Streaming tables for Apache Iceberg

## with Amazon Kinesis Data Streams



# Amazon Kinesis - Getting Started in Minutes




**Supported at Launch**

- ✓ Destinations: S3 Tables & S3 Buckets
- ✓ Input format: JSON
- ✓ Partitioning: Time-based
- ✓ Data freshness: 5 -15 minutes
- ✓ Catalog: AWS Glue Data Catalog
- ✓ Mode: Append-only (new rows only)
- ✓ Monitoring: CloudWatch
- ✓ Interfaces: Console, SDK, API/CLI, CDK

# Price-Performance

**60%**

Lower ingestion cost vs self-managed connectors

**30%**

Lower downstream query cost

**$0**

Egress cost

# Customer Use Cases

# Medidata Achieves 99% Performance Improvement with Real-Time  Lakehouse on AWS

## Challenge

Medidata's legacy batch ETL architecture created significant latency—hours to days between data ingestion and processing. Multiple fragmented pipelines increased maintenance burden exponentially as data volume grew, with complex dependencies creating numerous failure points and requiring coordinated scaling across the entire infrastructure stack.

## Solution

Medidata built a unified real-time streaming Lakehouse using Apache Flink on Amazon EKS, Apache Kafka on Amazon MSK, and Apache Iceberg tables backed by AWS Glue Data Catalog—eliminating fragmented pipelines and creating a single source of truth accessible to all consumers without additional downstream processing.

## Result

- Reduced pipeline latency from hours/days to minutes—achieving 99% performance improvement from data ingestion to analytics layers
- Decreased operational maintenance footprint by 5x through unified pipelines and consolidated observability
- Enabled real-time delivery of billions of clinical events to downstream consumers, applications, and AI agents

## Benefits

"By modernizing our clinical data platform with Apache Iceberg on AWS, we transformed how we deliver data to thousands of clinical trials worldwide, reducing pipeline latency from hours to minutes while cutting operational maintenance by five times."

— Mike Araujo, Principal Engineer, Medidata Solutions



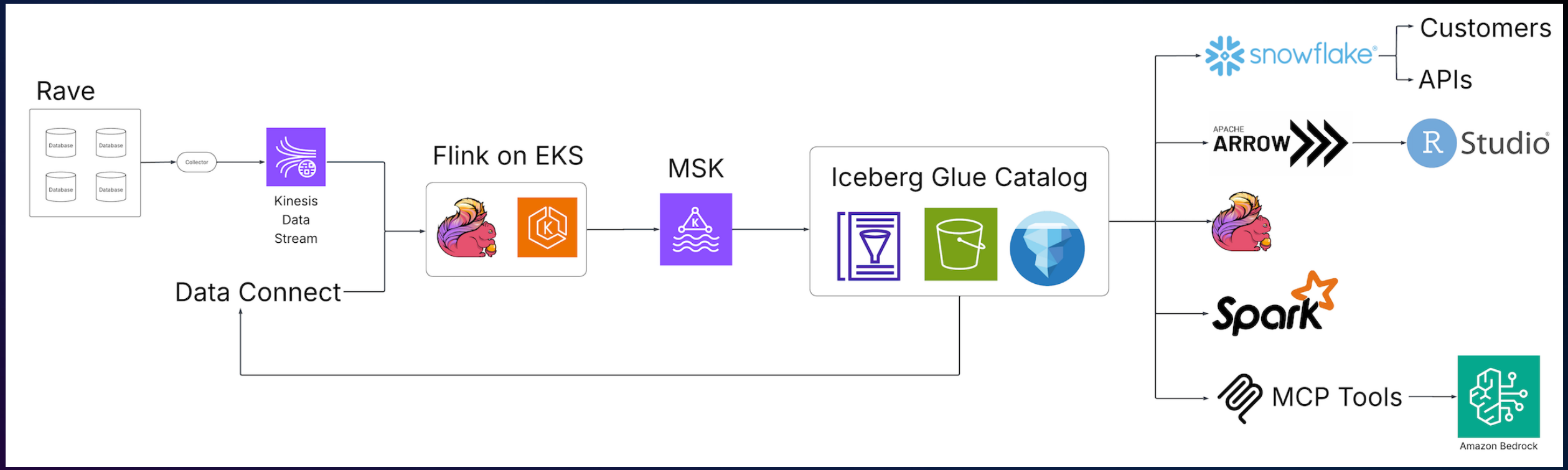
Read Blog Here

Amazon MSK

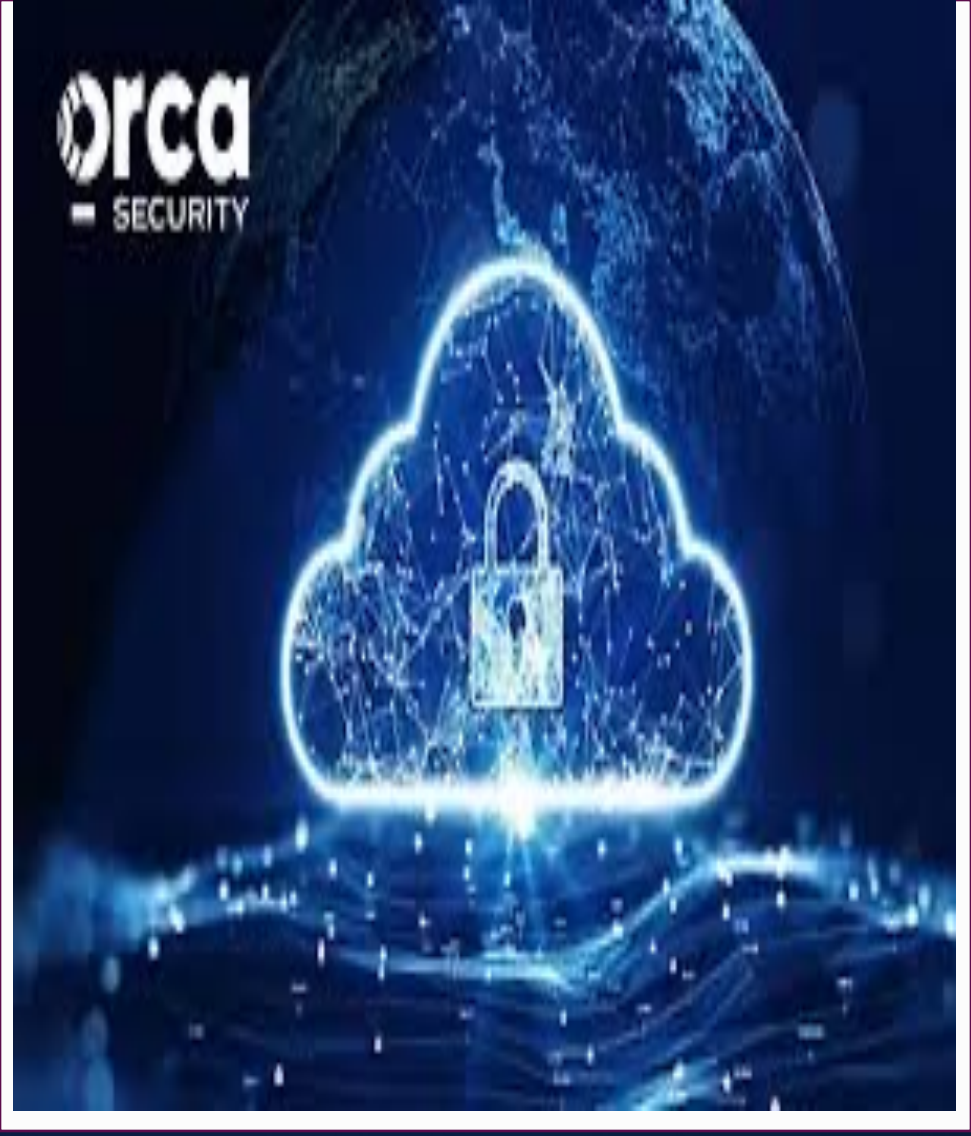
Amazon S3

Amazon Redshift

AWS Glue



# Orca Security Scales Cloud Security Platform with Apache Iceberg on AWS, Eliminating Partition Limits




## Challenge

Orca Security faced critical scaling constraints with partition limits across all tables. With 10,000+ customer accounts and daily partitions, the partition explosion problem threatened their ability to onboard new customers and scale their SaaS platform, directly impacting revenue growth and customer experience.

## Solution

Orca Security built a modern data lake architecture on AWS centered on Apache Iceberg. The solution leverages Amazon S3 for scalable storage, Amazon MSK for real-time data ingestion, Apache Spark on Amazon EMR for stream processing, and Amazon Athena Version 3 for interactive queries. Iceberg's hidden partitioning eliminated per-customer, per-day partitioning schemes while its metadata layer automatically tracks file-level statistics, removing dependency on Glue Catalog partition limits.

## Result

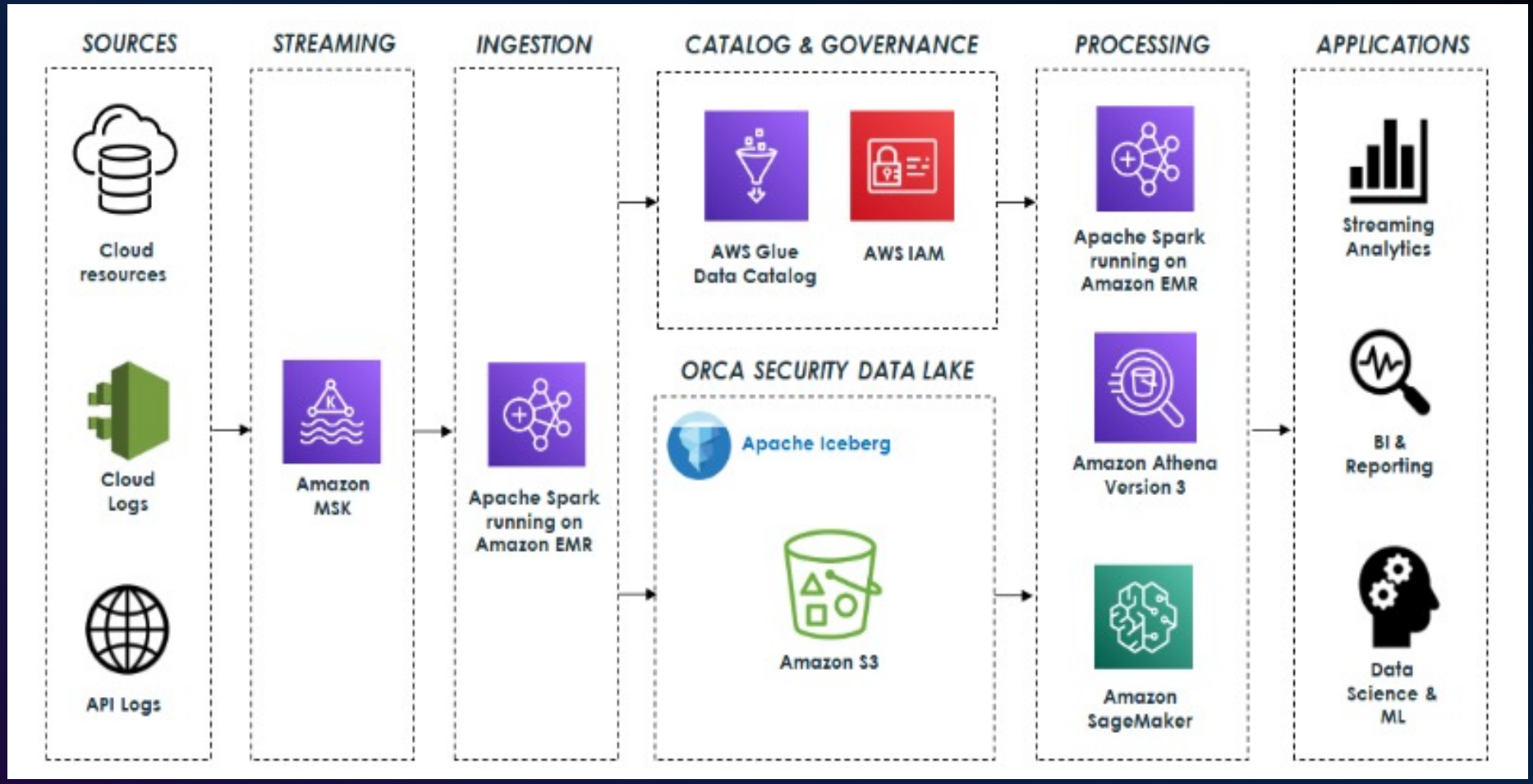
- Eliminated partition ceiling enabling unlimited customer onboarding without infrastructure constraints
- Dramatically reduced query execution time through intelligent file pruning and metadata-driven optimization
- Simplified operations with automated metadata management and self-optimizing storage

## Benefits

"Apache Iceberg on AWS transformed our data infrastructure by eliminating the hard scaling limits that threatened our business growth. The combination of Iceberg's intelligent metadata management with AWS managed services gave us unlimited scalability, dramatically improved query performance, and simplified operations—enabling us to focus on delivering security insights to our customers rather than managing data infrastructure."
— Miri Gorenshtein, Big Data Group Manager, Orca Security

 Read Blog Here

Amazon MSK | Amazon EMR | Amazon S3 | Amazon SageMaker | AWS Glue Catalog



# Natural Intelligence Migrates from Apache Hive to Iceberg on AWS, Achieving Zero-Touch Schema Evolution

## Challenge

Natural Intelligence, a leading comparison shopping platform processing billions of user interactions, faced critical scaling challenges with their Apache Hive-based data lake. Schema evolution required rewriting entire tables causing significant downtime, managing thousands of Hive partitions became increasingly complex and error-prone, and Hive's lack of ACID transactions created data quality problems during concurrent writes.

## Solution

Natural Intelligence migrated from Apache Hive to Apache Iceberg on AWS, building an event-driven architecture with Amazon S3 for storage, Apache Spark on Amazon EMR for ETL processing, and AWS Glue Data Catalog for metadata management. The solution leverages Amazon EventBridge to monitor schema changes and AWS Lambda to automatically propagate updates to downstream systems including Snowflake, Tableau, and Amazon Athena. Iceberg's hidden partitioning eliminated manual partition management while enabling seamless schema evolution without table rewrites.

## Result

- Achieved zero downtime for schema changes versus hours of table rewrites with Hive
- Automated 100% of schema propagation to downstream systems eliminating manual intervention
- Enabled near real-time CDC updates through automated snapshot-based change tracking

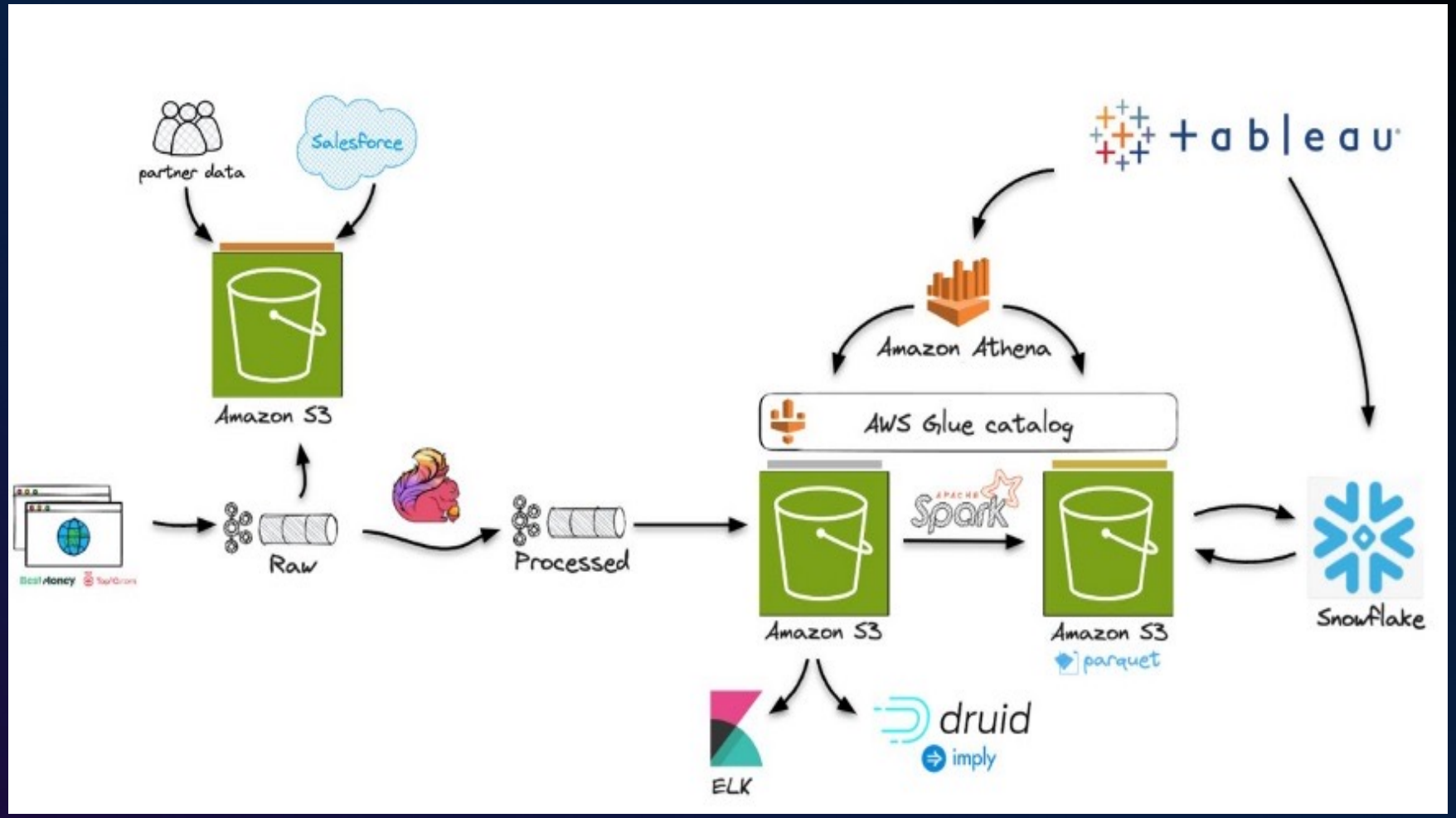
## Benefits

Migrating to Apache Iceberg on AWS transformed our data operations by eliminating the operational overhead of schema management and partition maintenance. The event-driven architecture with EventBridge and Lambda automated what used to be manual, error-prone processes, allowing our engineering team to focus on delivering insights to millions of consumers making purchase decisions rather than managing data infrastructure."
— Engineering Leadership, Natural Intelligence


Amazon S3 Amazon Athena AWS Glue Catalog



Read Blog Here



# Resources

https://www.youtube.com/@ApacheIceberg

https://aws.amazon.com/blogs/big-data/deliver-apache-kafka-data-to-streaming-tables-for-apache-iceberg-with-amazon-msk-express-brokers/

https://aws.amazon.com/blogs/big-data/deliver-real-time-data-to-streaming-tables-for-apache-iceberg-with-amazon-kinesis-data-streams/

# From warehouse to Lakehouse: Redshift Iceberg Integration

# Agenda

- Amazon Redshift overview
- Why Lakehouse?
- Redshift Iceberg integration & Reference architecture
- Performance enhancements
- Data sharing for data lake tables
- SageMaker Lakehouse

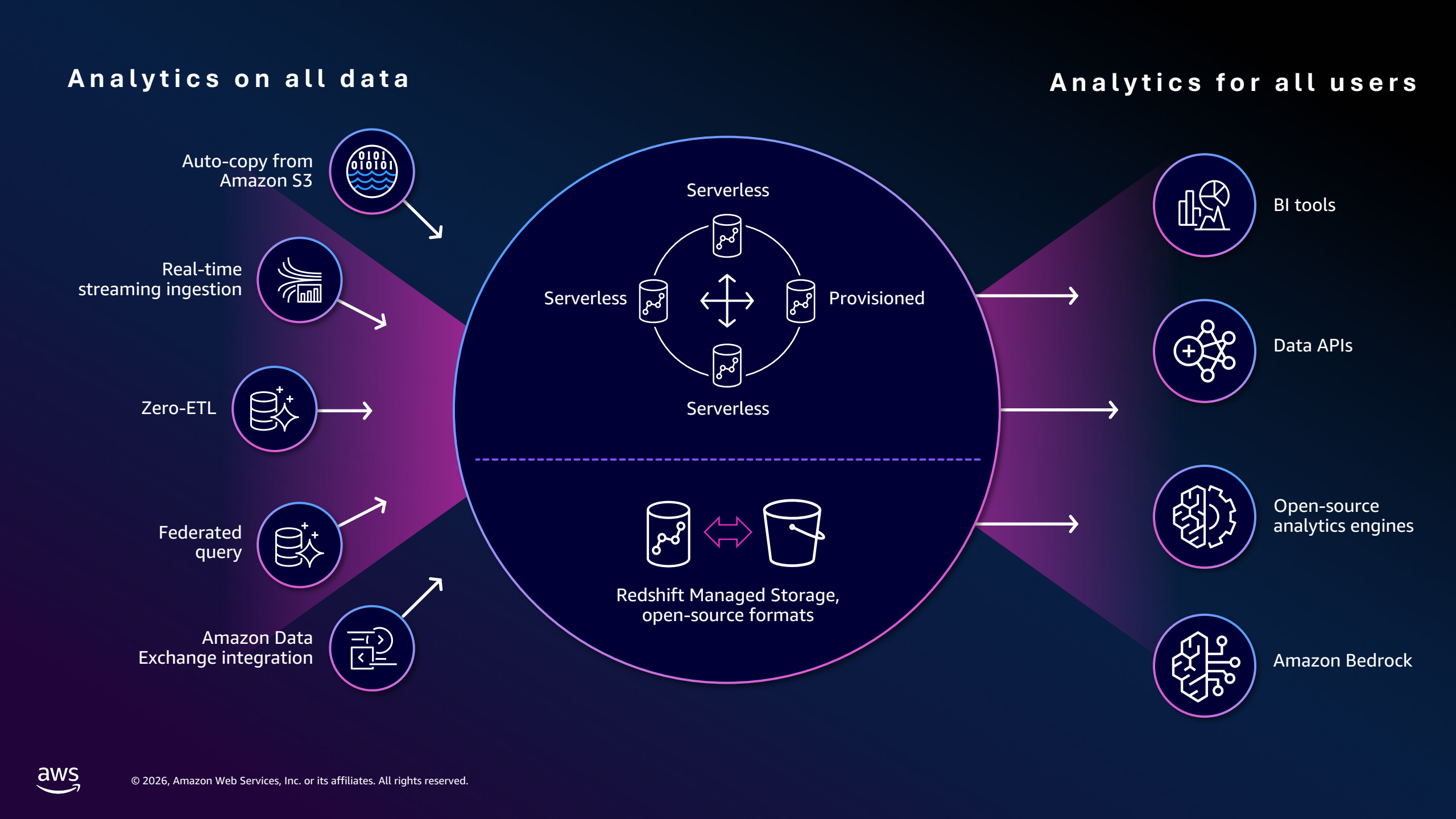
# Amazon Redshift

**Fully managed** provisioned and serverless offerings, allowing you to pay-as-you-go and optimise workloads for cost

**Scale to any sized workload** seamlessly, providing best in class price performance for all analytics workloads

**Low touch integration** with other AWS services, making it easier than ever to integrate all of your data

**Secure by design** with the ability to implement role based access controls, data masking, and cell level access



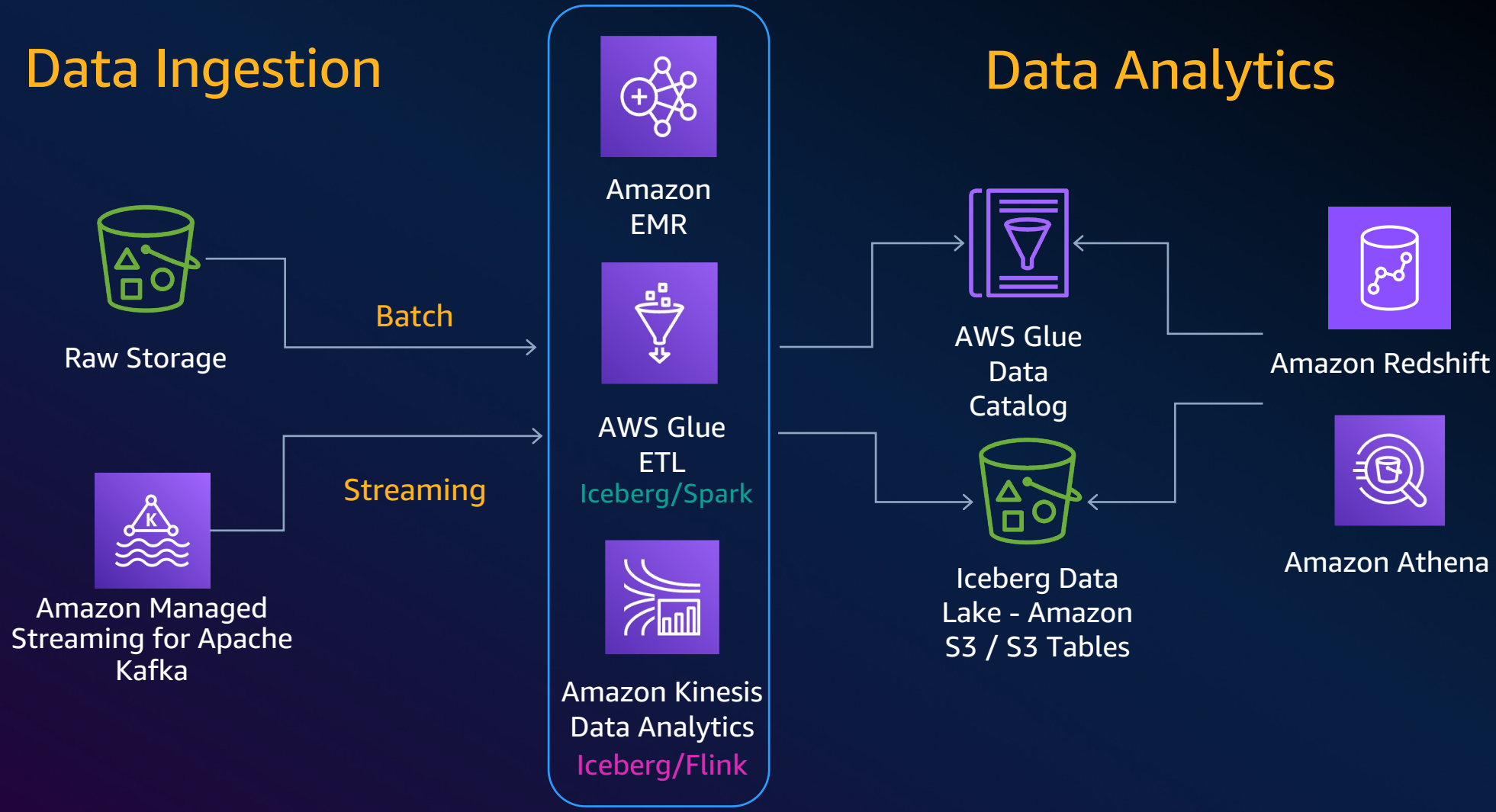
# Apache Iceberg and Amazon Redshift Integration

# High-performance SQL for Lakehouse

Query and write to the Apache Iceberg open table format from Amazon Redshift

- Access to Apache Iceberg tables in AWS Glue Data Catalog
- Query and write to the data lake with transactional consistency
- Join data warehouse data and data lake data easily
- Powerful materialized view support

# Reference Architecture



# Redshift Iceberg Integration

- **Version Support**
  - V1: Manages large analytic tables with immutable data files
  - V2: Row-level updates/deletes (copy-on-write & merge-on-read)
- **Read and Write Access**
  - Transactional consistent select queries
  - Schema management via AWS Glue/Amazon EMR/Amazon Athena
  - INSERT INTO, CREATE TABLE AS (CTAS)
  - Write directly via awsdatacatalog mount — no external schema needed (NEW - May 2026)
- **Partition Management**
  - Automatic partition detection
  - No manual partition updates needed

# Redshift Iceberg Integration

- **Data Ingestion**
  - INSERT INTO, CREATE TABLE AS
  - Writes via awsdatacatalog mount (NEW - May 2026)
  - Lake Formation permissions supported for write operations
- **Materialized Views**
  - Incremental
  - Auto-refresh
- **Performance**
  - Glue column statistics required

# Redshift Iceberg Integration

- **Security & Control**
  - AWS Lake Formation fine-grained access control
  - Lake Formation permissions for Iceberg write operations
  - Tables modified by Redshift remain compatible with EMR, Athena (cross-engine)
  - User-defined data handling parameters
    - invalid_char_handling, surplus_char_handling, numeric_overflow_handling
- **Pricing Models**
  - Cluster access: Redshift Spectrum pricing
  - Workgroup access: Redshift Serverless pricing

# Latest data lake performance enhancements

DELIVERING improved performance for data lake queries in Amazon redshift

**Amazon Redshift Serverless data lake performance**
Benchmark derived from TPC-DS 3TB

**Total TPC-DS runtime in seconds (Lower is better)**


Up to **3x** performance improvement for data lake queries YoY

Optimizations include AWS Glue column statistics, improved data and metadata filtering, dynamic partition elimination, faster/parallel processing of Iceberg manifest files, and scanner improvements

**NEW!** **GA**

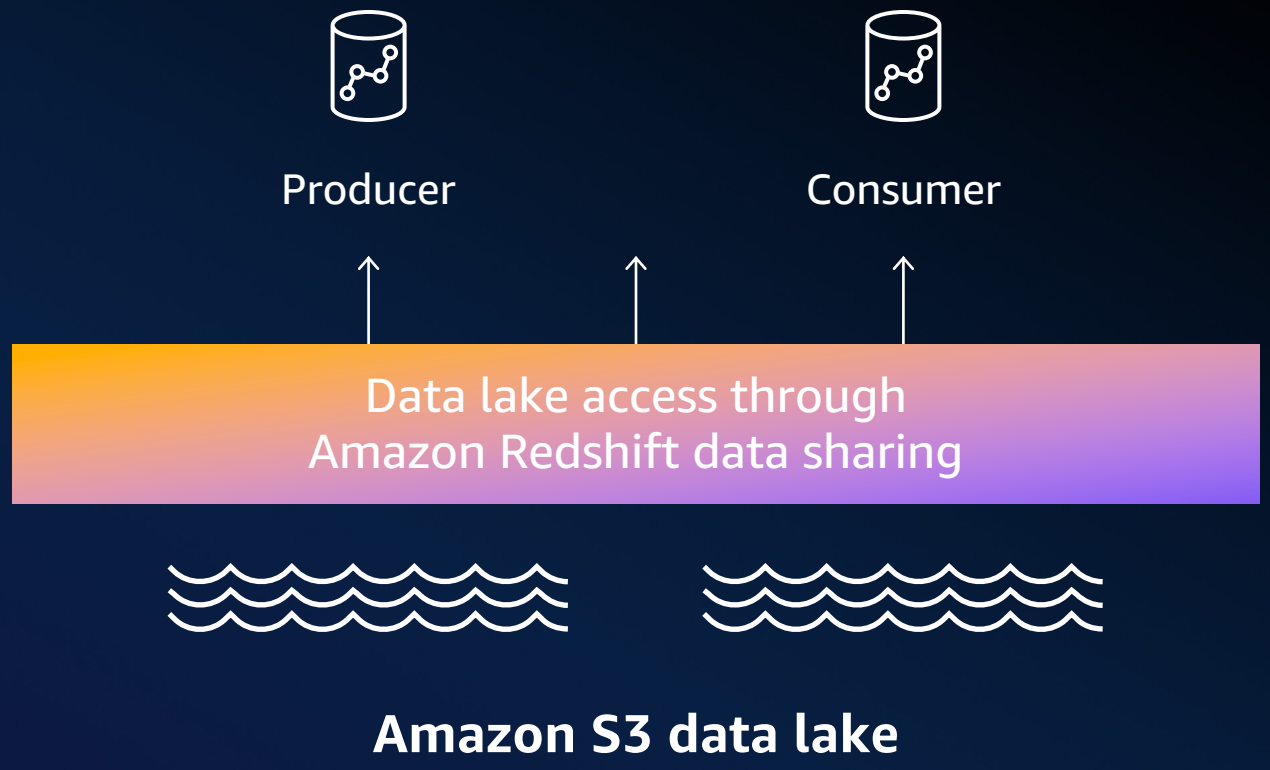
Incremental refresh support for materialized views on data lake tables to eliminate the need for rescanning already materialized data

NEW! GA

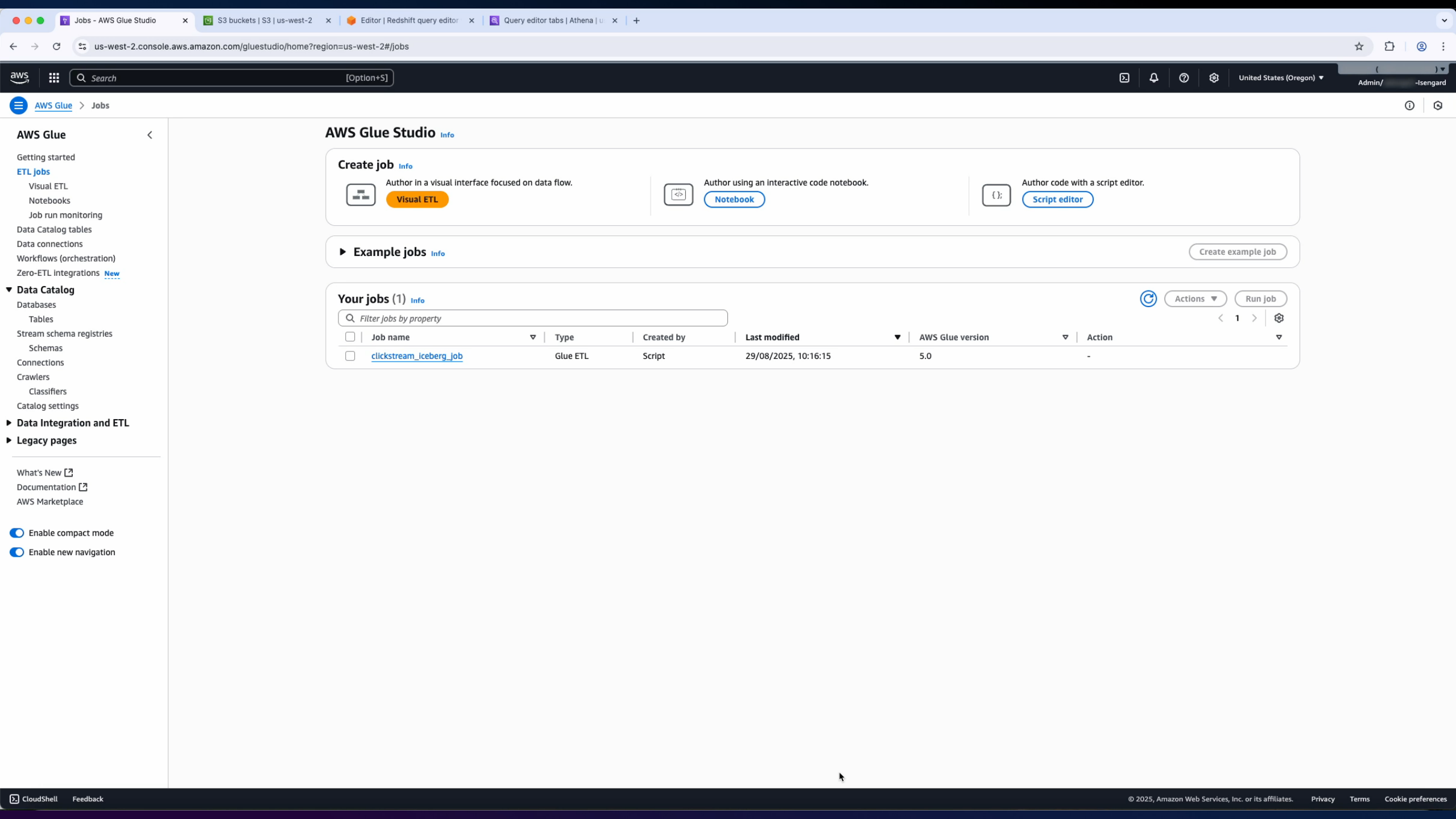
# Data sharing support for data lake tables

## simplify data lake access using data sharing in redshift

- Eliminate setup for data lake access in consumer DWs using data sharing
- Authorization using producer identity or federated user identity
- Support for sharing late binding views on data lake tables



# Demo


Jobs - AWS Glue Studio
S3 buckets | S3 | us-west-2
Editor | Redshift query editor
Query editor tabs | Athena
us-west-2.console.aws.amazon.com/gluestudio/home?region=us-west-2#/jobs
Search
[Option+S]
United States (Oregon)
Admin/ -Isengard
AWS Glue
Jobs
AWS Glue
Getting started
ETL jobs
Visual ETL
Notebooks
Job run monitoring
Data Catalog tables
Data connections
Workflows (orchestration)
Zero-ETL integrations New
Data Catalog
Databases
Tables
Stream schema registries
Schemas
Connections
Crawlers
Classifiers
Catalog settings
Data Integration and ETL
Legacy pages
What's New
Documentation
AWS Marketplace
Enable compact mode
Enable new navigation
AWS Glue Studio Info
Create job Info
Author in a visual interface focused on data flow.
Visual ETL
Author using an interactive code notebook.
Notebook
Author code with a script editor.
Script editor
Example jobs Info
Create example job
Your jobs (1) Info
Actions
Run job
Filter jobs by property
1
Job name
Type
Created by
Last modified
AWS Glue version
Action
clickstream_iceberg_job
Glue ETL
Script
29/08/2025, 10:16:15
5.0
-
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

# Further Information

Using Apache Iceberg tables with Amazon Redshift Documentation

Query your Iceberg tables in data lake using Amazon Redshift Blog

Using Amazon S3 Tables with Amazon Redshift to query Apache Iceberg tables Blog

Use Redshift Spectrum to Query Iceberg tables
Workshop

# Apache Iceberg Migration Strategies & Guidance

In-place & full migration | Iceberg on Amazon S3 & Amazon S3 Tables | production patterns

AGENDA

# Five chapters, thirty minutes

**01** **Why migrate to Iceberg?**
Current challenges and Iceberg advantages

**02** **Typical migration paths & solutions**
In-place, full migration, and Amazon S3 Tables

**03** **How to approach the migration**
Cutover strategies and phased transitions

**04** **Ongoing maintenance**
Keeping Iceberg tables performant

**05** **Recap and what to do next**
The four takeaways, and the ask before you cut over

# 01

# Why Apache Iceberg?

Understanding the need for a transactional lakehouse

WHY APACHE ICEBERG

# A transactional lakehouse foundation on open data

## ACID (atomicity, consistency, isolation, durability) transactions

**No dirty reads while concurrent writes commit.**

A write publishes a **new snapshot**, then the catalog pointer flips in one atomic step. Readers see the table before or after, never mid-write. A failed job leaves nothing to clean up.

## Time travel

**Query any snapshot; roll back instantly.**

Old snapshots aren't deleted, so you can read the table **as of a timestamp or snapshot ID**, and undo a bad write by re-pointing at the previous one instead of restoring a backup.

## Scalable metadata

**Petabyte scale without slow file listings.**

Manifests record every file with its **value ranges**, so the planner skips files up front instead of listing Amazon S3 prefixes. Planning time stops tracking table size.

## Schema evolution

**Add, drop, rename columns, with no data rewrite.**

Columns are tracked by a **unique field ID**, not by position, so inserting or renaming one can't silently shift values into the wrong field.

## Row-level operations

**MERGE / UPDATE / DELETE for GDPR and CDC (change data capture).**

Delete one customer's rows without rewriting the partition. **Copy-on-write** for read speed, **merge-on-read** for write speed.

## Engine agnostic

**One table, many engines, no lock-in.**

Amazon Athena, Amazon EMR, Amazon Redshift, Apache Spark, AWS Glue and Amazon SageMaker read *and write* the same table, with **no copies**, no per-engine format.

**One table, many engines**

Amazon Athena · Amazon EMR · Amazon Redshift · AWS Glue · Amazon SageMaker · Amazon Data Firehose · Amazon Quick

**... and any Iceberg-compatible engine via the Apache Iceberg REST endpoint** Spark · trino · DuckDB · snowflake · databricks · **PyIceberg**

**Support differs by engine, by Iceberg format version, and by read vs write — confirm current capability for your own stack before you commit.**

# 02

# Migration Paths & Solutions

MIGRATION / MODERNIZATION PATTERNS

# Every path lands on an Iceberg lakehouse

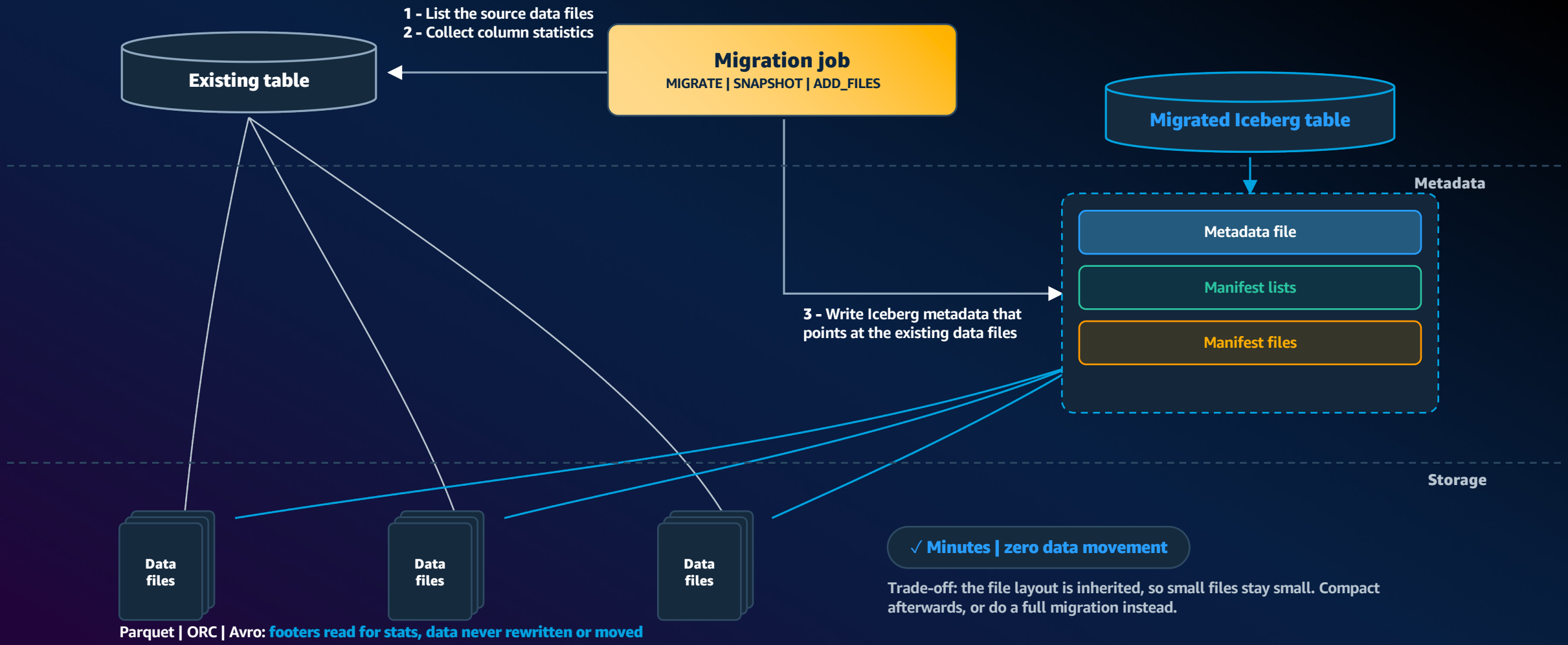
And the lakehouse is two deployment choices: self-managed Iceberg on Amazon S3, or fully managed Amazon S3 Tables

| MIGRATE FROM | | MIGRATE TO |
|---|---|---|
| Parquet | **Data lake**<br>Apache Hive \| Parquet, ORC, Avro | → **Lakehouse**<br>Amazon S3 Tables or self-managed Iceberg on S3 |
| DELTA LAKE, Apache hudi | **Other open table formats**<br>Hudi \| Delta | → **Lakehouse**<br>Amazon S3 Tables or self-managed Iceberg on S3 |
| ICEBERG | **Self-managed Iceberg**<br>Iceberg on S3 general-purpose buckets | → **Lakehouse**<br>Amazon S3 Tables: full migration, data moves into the table bucket |
| snowflake | **Data warehouse**<br>Snowflake \| Amazon Redshift \| etc. | ↔ **Lakehouse**<br>Amazon S3 Tables or self-managed Iceberg on S3 |

**Note:** the data warehouse and the lakehouse are **complementary layers**, and the arrow runs both ways. Treat this one as **modernization, not migration.**

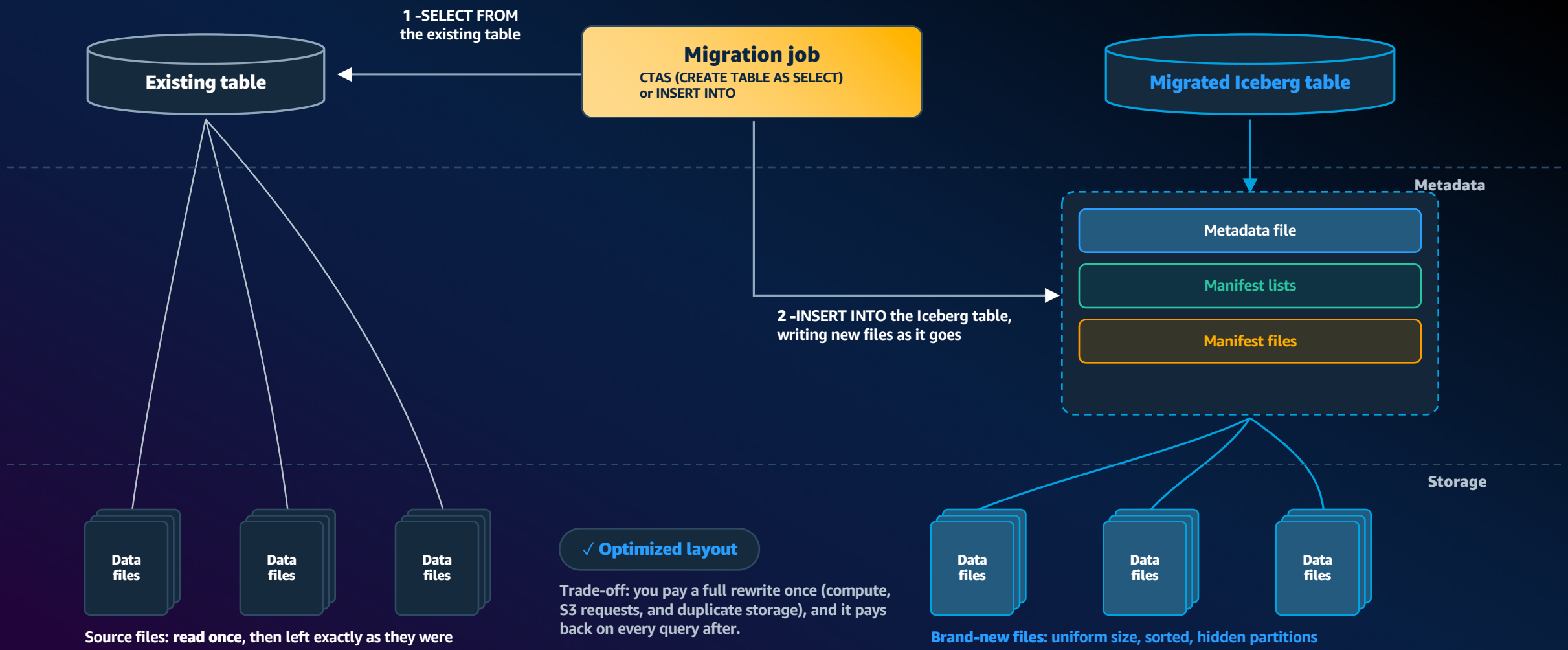
IN-PLACE MIGRATION

# New metadata over the files you already have



FULL DATA MIGRATION

# Read the source, write a brand-new optimized table



CHOOSING BETWEEN THEM

# Same destination, very different bill and payoff

| | In-place \| metadata only | Full migration \| rewrite |
|---|---|---|
| **Speed** | **Minutes**<br>Scales with the **number of files**, not their size. | **Hours to days**<br>Scales with the **volume of data** you rewrite. |
| **Cost** | **Lower, no data copy**<br>No rewrite: lists files in Amazon S3 and reads Parquet **footers** for stats (column sizes, row counts).<br>Tiered files (Glacier / IA) still get promoted (consider retrieval costs) | **Higher: compute + S3 requests**<br>A full read and a full write of the dataset. |
| **Storage & layout** | **Same files, new metadata**<br>Zero data movement, and the **old layout is inherited**. | **New files, optimized layout**<br>Right-sized, sorted, hidden partitioning from day one. |
| **Reach for it when…** | • TB–PB where rewriting is cost-prohibitive<br>• You need minimal downtime<br>• Data is already Parquet/ORC/Avro<br>• Current partitioning and file sizes are acceptable | **Required**<br>• Source format unsupported (e.g. JSON)<br>• Moving to Amazon S3 Tables<br>**Recommended**<br>• Re-partitioning needed (hidden partitioning)<br>• File optimization needed<br>**Preferred where feasible: it fixes the layout.** |

# 02

# Migration Paths & Solutions

**Four paths. Here is where we are.**

- **In-place migration**
  Metadata only, nothing is rewritten

  Hive, Delta and Hudi → Iceberg lakehouse

- **Full data migration**
  Rewrite into a new, optimized table

  Hive, Delta and Hudi → Iceberg lakehouse

- **Self-managed Iceberg to Amazon S3 Tables**
  Full migration, because a table bucket owns its storage

  Iceberg on general purpose S3 → Amazon S3 Tables

- **Data warehouse to lakehouse**
  Modernization, not migration

  Amazon Redshift, Snowflake → Iceberg lakehouse

1 of 4

IN-PLACE MIGRATION

# Which sources qualify, and what you use for each

**These are the common sources that support in-place, metadata-only migration.**

| Hive-style tables<br>Parquet \| ORC \| Avro | Delta Lake<br>transaction log | Apache Hudi<br>timeline metadata |
|---|---|---|
| **Spark Iceberg migration procedures**<br>MIGRATE \| SNAPSHOT \| ADD_FILES | **Delta UniForm** or **Apache XTable**<br>metadata translation | **Apache XTable**<br>metadata translation |

**While you run side by side, never delete data files**

No expire_snapshots, no remove_orphan_files, no DROP TABLE … PURGE, and do **not enable the AWS Glue Data Catalog table optimizer** on the Iceberg table. Both tables point at the **same files**, so any of these corrupts the source you kept as your way back.

SNAPSHOT auto-sets gc.enabled=false to block exactly these operations. Flip it back to true only when you promote the Iceberg table to primary.

**OSS Delta UniForm writes no column statistics**

It does not populate column stats in the Iceberg manifests, so Iceberg readers cannot do data skipping or min/max pruning, so queries scan far more than they need to (delta-io #5469). **Databricks Delta Lake is not affected.**

IN-PLACE MIGRATION · HIVE-STYLE TABLES

# Three Spark procedures: none of them rewrite a file

**Spark SQL procedures that build Iceberg metadata over the data you already have.**

## MIGRATE

Replaces the source table with an Iceberg table and renames the original to <table>_backup_.

Complete cutover from Hive to Iceberg.

Hive metastore only, **not supported on the AWS Glue Data Catalog**

```
spark.sql("""
 CALL system.migrate(
   table => 'mydb.my_table'
 )
""")
```

## SNAPSHOT

Creates a new Iceberg table pointing at the **same data files**. The source table is left untouched.

Side-by-side testing and validation before you commit.

Can reach the MIGRATE outcome with a few extra steps

```
spark.sql("""
 CALL system.snapshot(
   source_table => 'mydb.my_table',
   table       => 'mydb.my_table_iceberg',
   location    => 's3://my_lake/mydb/my_table_iceberg/'
 )
""")
```

## ADD_FILES

Adds existing Parquet/ORC/Avro files from specific source partitions into an **existing** Iceberg table.

Incremental sync of new partitions, or partial migration.

The one you schedule for external-sync cutovers

```
spark.sql("""
 CALL system.add_files(
   source_table    => 'mydb.my_table',
   table           => 'mydb.my_iceberg_table',
   partition_filter => map('day', '20250904')
 )
""")
```

### UNDER THE HOOD, ALL THREE DO THE SAME THREE THINGS

1. List the source table's data files
2. Collect column statistics for those files
3. Write the Iceberg metadata that points at them

No file is read for its contents and no file is rewritten, which is why this takes minutes, not hours.

IN-PLACE MIGRATION · DELTA LAKE & HUDI

# Two tools translate the metadata: neither rewrites data

**Your Delta or Hudi files stay exactly where they are. Only Iceberg metadata gets written over them.**

## Apache XTable

Hudi and Delta Lake

```
java -jar xtable-utilities_XXX-bundled.jar
```

**First run** does a full sync. **Every run after** uses incremental mode, so each new commit is translated into an Iceberg snapshot.

## Delta Lake UniForm

Delta Lake only

```
REORG TABLE <table>
APPLY (UPGRADE UNIFORM(ICEBERG_COMPAT_VERSION=2))
```

Then sync is **automatic:** full table sync the first time, and after that every write is translated to Iceberg as it lands.

Read files + column stats from the Delta/Hudi metadata → Write Iceberg metadata → Data files untouched

### Check the limitations before you promise a date

**Hudi merge-on-read and Delta deletion vectors**
At Iceberg V2 both tools translate **base files only**, so Hudi MoR log files and Delta deletion vectors are not carried across.

**Iceberg V3 as a target**
Apache XTable does not target V3. Delta UniForm reaches V3 only on Databricks, not in OSS.

**While your run Side by side:** both formats point at the same files, so no `expire_snapshots` / `remove_orphan_files` / `PURGE`, and do not enable the AWS Glue table optimizer.

# 02

# Migration Paths & Solutions

**Four paths. Here is where we are.**

### In-place migration
Metadata only, nothing is rewritten

Hive, Delta and Hudi → Iceberg lakehouse

### Full data migration
Rewrite into a new, optimized table

Hive, Delta and Hudi → Iceberg lakehouse

### Self-managed Iceberg to Amazon S3 Tables
Full migration, because a table bucket owns its storage

Iceberg on general purpose S3 → Amazon S3 Tables

### Data warehouse to lakehouse
Modernization, not migration

Amazon Redshift, Snowflake → Iceberg lakehouse

FULL MIGRATION

# Two ways to write the new table: pick by how you query it

**Independent of the target: either approach writes to either destination, and only the catalog name changes.**

## Option 1 — CTAS (CREATE TABLE AS SELECT)

Amazon Athena or Apache Spark | one statement

```
CREATE TABLE <catalog>.db.target
USING ICEBERG
PARTITIONED BY (days(event_ts))
AS SELECT * FROM source_db.source
```

Fastest path. But CTAS **cannot apply a sort order**. An ORDER BY in the SELECT is ignored, so you get the default layout. Fine when queries are mostly time-ranged.

## Option 2 — Create + sort + insert

Apache Spark only | explicit write order

```
CREATE TABLE <catalog>.db.target (...)
USING ICEBERG
PARTITIONED BY (days(event_ts));

ALTER TABLE ... WRITE ORDERED BY customer_id;

INSERT INTO <catalog>.db.target
SELECT * FROM source_db.source;
```

Choose this when queries filter on a high-cardinality column, because sorted files prune far better. WRITE ORDERED BY is an Apache Spark feature, so this path is Spark-only, Amazon Athena cannot set it.

**Both options above target either destination. The SQL is the same, you just point at a different catalog**

**Iceberg on Amazon S3 — self-managed**

glue_catalog.db.target

**Amazon S3 Tables — managed** (catalog name differs by engine)

Apache Spark `<your-catalog>.db.target` (set to the table-bucket ARN)

Amazon Athena `"s3tablescatalog/<table-bucket>".db.target`

**Very large tables:** seed with CTAS, then load the remainder in batches (e.g. one year per INSERT) to bound job size, cost, and blast radius.

# 02

# Migration Paths & Solutions

**Four paths. Here is where we are.**

### In-place migration
Metadata only, nothing is rewritten

Hive, Delta and Hudi → Iceberg lakehouse

### Full data migration
Rewrite into a new, optimized table

Hive, Delta and Hudi → Iceberg lakehouse

### Self-managed Iceberg to Amazon S3 Tables
Full migration, because a table bucket owns its storage

Iceberg on general purpose S3 → Amazon S3 Tables

### Data warehouse to lakehouse
Modernization, not migration

Amazon Redshift, Snowflake → Iceberg lakehouse

SELF-MANAGED ICEBERG → AMAZON S3 TABLES

# Two utilities will do this move for you

**You do not have to hand-roll the move.**

### 1 Clumio (Commvault)

Automates migration of Iceberg tables registered in the **AWS Glue Data Catalog** into Amazon S3 Tables.

Enables **long-term protection** for those lakehouse assets at the same time: migration and backup in one pass.

### 2 AWS Solution Library utility

Moves **Iceberg and Hive** tables from general-purpose Amazon S3 buckets into S3 table buckets.

AWS Step Functions | Amazon EMR with Apache Spark

**Clumio walkthrough**

Commvault's write-up on simplifying the move to Amazon S3 Tables.

**AWS Solution Library guidance**

Migrating tabular data from Amazon S3 to Amazon S3 Tables: the reference implementation.

# 02

# Migration Paths & Solutions

**Four paths. Here is where we are.**

### In-place migration
Metadata only, nothing is rewritten

Hive, Delta and Hudi → Iceberg lakehouse

### Full data migration
Rewrite into a new, optimized table

Hive, Delta and Hudi → Iceberg lakehouse

### Self-managed Iceberg to Amazon S3 Tables
Full migration, because a table bucket owns its storage

Iceberg on general purpose S3 → Amazon S3 Tables

### Data warehouse to lakehouse
Modernization, not migration

Amazon Redshift, Snowflake → Iceberg lakehouse

DATA WAREHOUSE → LAKEHOUSE

# Iceberg extends the warehouse rather than replacing it

**This is a modernization, not a migration.** Iceberg extends the architecture: lower storage cost, multi-engine interoperability, better data accessibility, while the warehouse keeps doing what it is best at.

## Keep the warehouse as the serving layer

Amazon Redshift

snowflake

- Fast real-time analytics
- Reporting & dashboards

Warehouse-grade latency stays where it belongs; Iceberg becomes the open, shared copy underneath.

## Two ways to land it in Iceberg

**1 - The warehouse can write Iceberg**

CTAS (CREATE TABLE AS SELECT) straight out of the warehouse, which is preferred. Amazon Redshift writes an RMS table to Iceberg directly.

**2 - It cannot write Iceberg**

Export to Amazon S3 as Parquet/CSV via UNLOAD, external tables or a native connector, then convert to Iceberg.

### Amazon Redshift → Iceberg on Amazon S3 Tables

```
CREATE TABLE "<table_bucket_name>@s3tablescatalog"."mydb"."iceberg_events_table"
USING ICEBERG PARTITIONED BY (day(event_timestamp))
AS SELECT * FROM dev.public.rms_events_table;
```

# 03

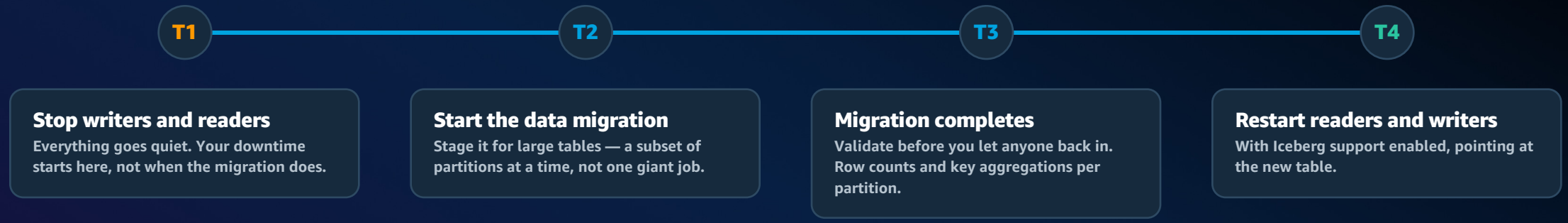
# How to Approach the Migration

Direct cutover, phased transition, validation, and cutover strategies

CUTOVER · OPTION 1 OF 2

# Direct cutover: one coordinated event

Migrate every component in a single window. Simplest to reason about. You just have to be able to afford the window.




**Maintenance window: T1 to T4, readers and writers are down for all of it**

**Best for**

Simpler estates with limited downstream dependencies, and only **after** you have rehearsed the whole sequence end-to-end in a lower environment. If you cannot take the window, the next two slides are your options.

CUTOVER · OPTION 2 OF 2 · PATTERN 1

# Phased dual writes: no downtime, but at a price

**Keep both tables live so consumers can move across incrementally, on their own schedule. Two tables means two physical copies for as long as the transition runs.**

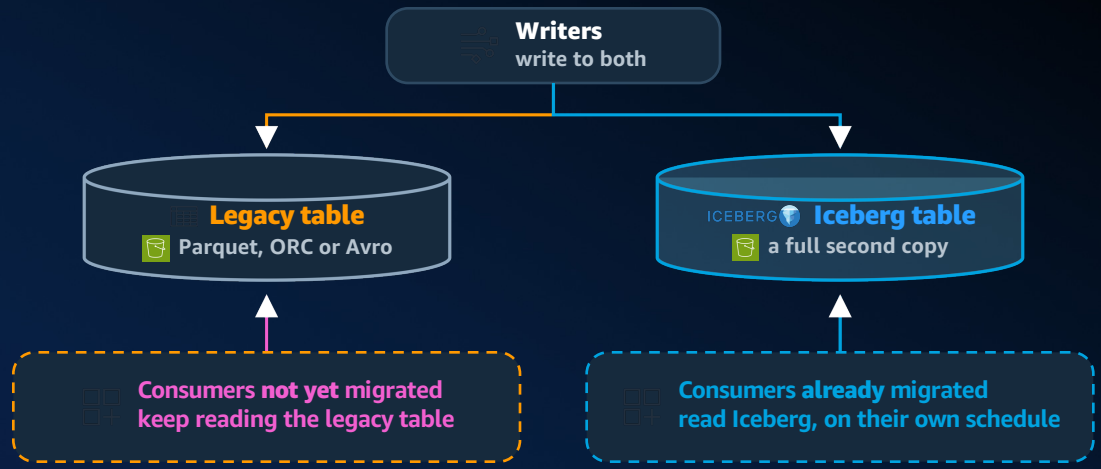
### How to

1. Stop writers
2. Initial migration to create the Iceberg table

(in stages for large tables: a subset of partitions at a time)

3. Modify ingestion to write to **both** source and Iceberg
4. Restart writers

**When to use** gradual migration where downstream dependencies cannot all move at the same time.




### Read these before you commit

Needs **error tracking and retry logic** to stay consistent. A write that lands in one table but not the other is a silent divergence.

Adds **latency to every write.** Often not feasible for streaming workloads.

**Delta Lake UniForm gives you this for free:** downstreams read the same table as Delta or Iceberg, no dual-write plumbing to own.

CUTOVER · OPTION 2 OF 2 · PATTERN 2

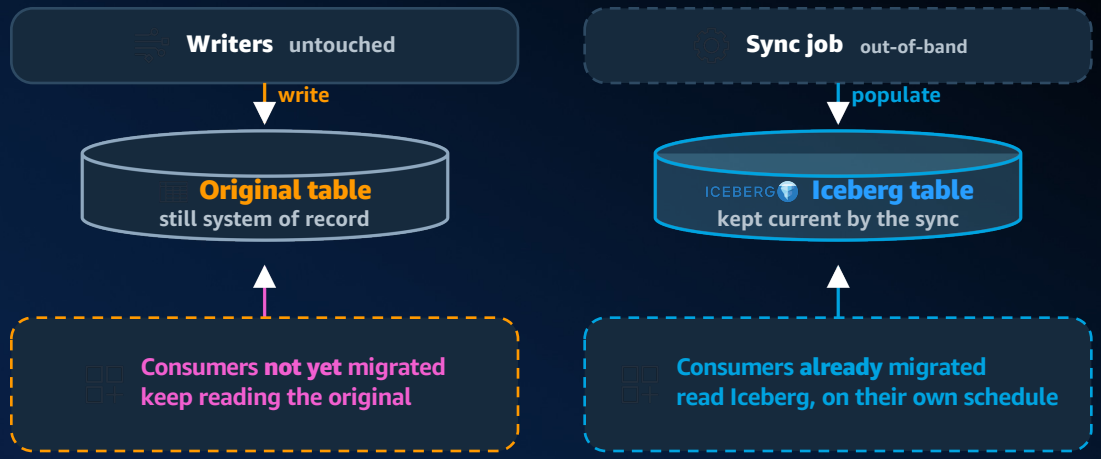
# Phased external sync: leave the pipeline alone

**The ingestion pipeline never changes. A scheduled job keeps the Iceberg copy current alongside it.**

### How to

1. Keep current ingestion unchanged
2. Initial migration to create the Iceberg table
3. Schedule sync jobs to keep the Iceberg table current

**When to use** when you cannot modify ingestion, because of a latency budget or resistance to changing a working process.




## The sync pattern follows from how you migrated

**Hudi / Delta: in-place via XTable**

Re-run XTable periodically in **incremental mode.**

**Hive-style: in-place via SNAPSHOT**

Periodically add_files for the new partitions.

**Any source: full migration**

Compute the delta since the last run, then MERGE INTO.

While both sides are live, do not evolve the Iceberg partitioning or schema. You need the two tables to stay comparable.

BEFORE YOU FLIP THE SWITCH

# Validate in four passes, then keep a way back

**The migration is the easy part. Proving it is right, and being able to walk it back, is what makes it safe.**

1

### Data correctness

Row counts match, and key aggregations agree to the cent. Compare per partition, not just the total.

2

### Performance

Benchmark your real queries before and after. Confirm partition and file pruning actually kicks in.

3

### Phased readers

Point dashboards and business intelligence (BI) at Iceberg first. Move batch and downstream jobs after they look clean.

4

### Rollback window

Keep the legacy table readable until you have run a full cycle, month-end included.

### Iceberg gives you a safety net the old world didn't

If a bad write lands, **roll the table back to the previous snapshot** instead of restoring from backup. `CALL system.rollback_to_snapshot(...)`. Validate against a snapshot ID so readers see a stable view while you check.

# 04

# Ongoing Maintenance

Keeping Iceberg tables performant over time

DAY 2 · MAINTENANCE

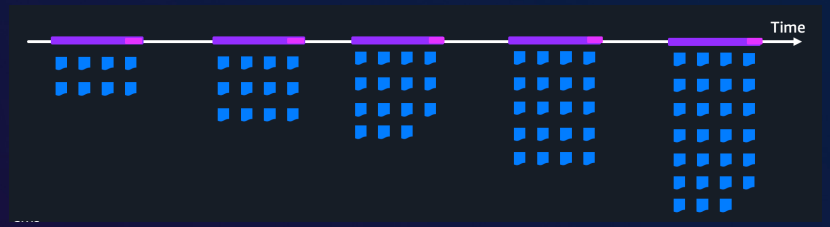
# Maintenance is three operations – plan for it from Day1

**Every write creates a new snapshot chain:**


*Snapshots accumulate rapidly in high-throughput environments consuming storage (impact on cost) and increasing Iceberg metadata (impact on performance)*

→ Expire Snapshots

**Write operations often generate small files:**




*Small files increase both S3 calls to open them as well as increase Iceberg metadata (impact on performance)*

→ Compaction

**Failed jobs may leave unreferenced files:**


*Unreferenced files are unused files on storage (impact on costs)*

→ Remove Unreferenced Files

DAY 2 · LET AWS RUN IT

# You do not have to schedule any of this yourself

**Managed maintenance exists for both deployment choices, not just Amazon S3 Tables.**

**Run for you:** **Expire snapshots** | **Compaction** (auto | binpack | sort | z-order) | **Remove unreferenced files**

## Amazon S3 Tables

Maintenance is a property of the table bucket, so there is nothing to deploy.

- Up to **10× higher transactions per second**
- Native **multi-Region replication**, read-only replicas
- **Intelligent-Tiering**, compaction skips colder tiers

## AWS Glue Iceberg table optimizer

For self-managed Iceberg in **general-purpose Amazon S3**, registered in the AWS Glue Data Catalog.

- Runs the same three operations for you
- Enable per table, no Spark job to own
- So **self-managed storage ≠ manual maintenance**

# 05

# Recap and what to do next

RECAP

# Four things to take away

If you remember nothing else from the last half hour, remember these.

1

### Pick the method by what you need to change

In-place writes metadata only and lands in minutes, but you inherit the old layout. Full migration rewrites the data and fixes the layout. Size is not the deciding factor.

2

### The target is a lakehouse with two deployment choices

Self-managed Iceberg on Amazon S3, or fully managed Amazon S3 Tables. Moving to Amazon S3 Tables is a **full migration**, because a table bucket owns its storage.

3

### Plan the migration to your platform's readiness

Match the method and the pace to how ready your engines and consumers are. When they cannot all move at once, **phase the migration** and cut consumers over on their own schedule.

4

### Maintenance is not yours to schedule, but the knobs are

Amazon S3 Tables and the AWS Glue table optimizer both run expire, compact and orphan removal. You still set retention, and your **time-travel window is your snapshot retention.**

**Before you cut over: rehearse the whole sequence in a lower environment, and keep the legacy table readable through a full cycle including month-end.**