databricks

Simplify your Streaming: Delta Live Tables

Jerrold Law

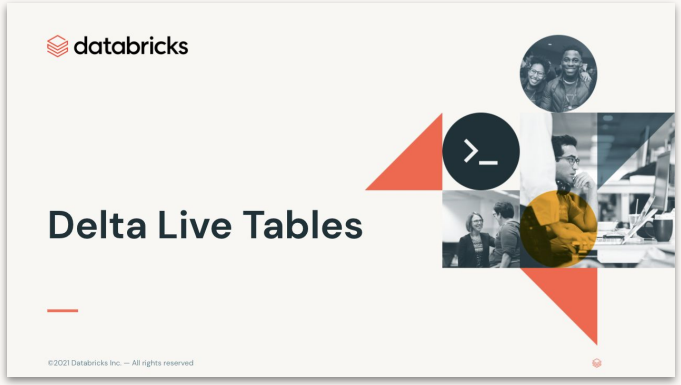
Sr. Specialist Solutions Architect

Housekeeping

- Your connection will be muted
- We will share recording with all attendees after the session
- Submit questions in the Q&A panel
- If we do not answer you question during the event, we will follow-up with you to get you the information you need!

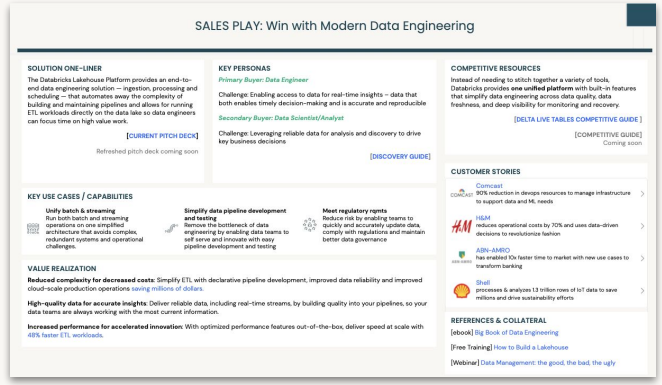
Where to learn more: go/dlt

Customer Facing Slides



This User Guide

Sales Play: Win with Modern Data Engineering

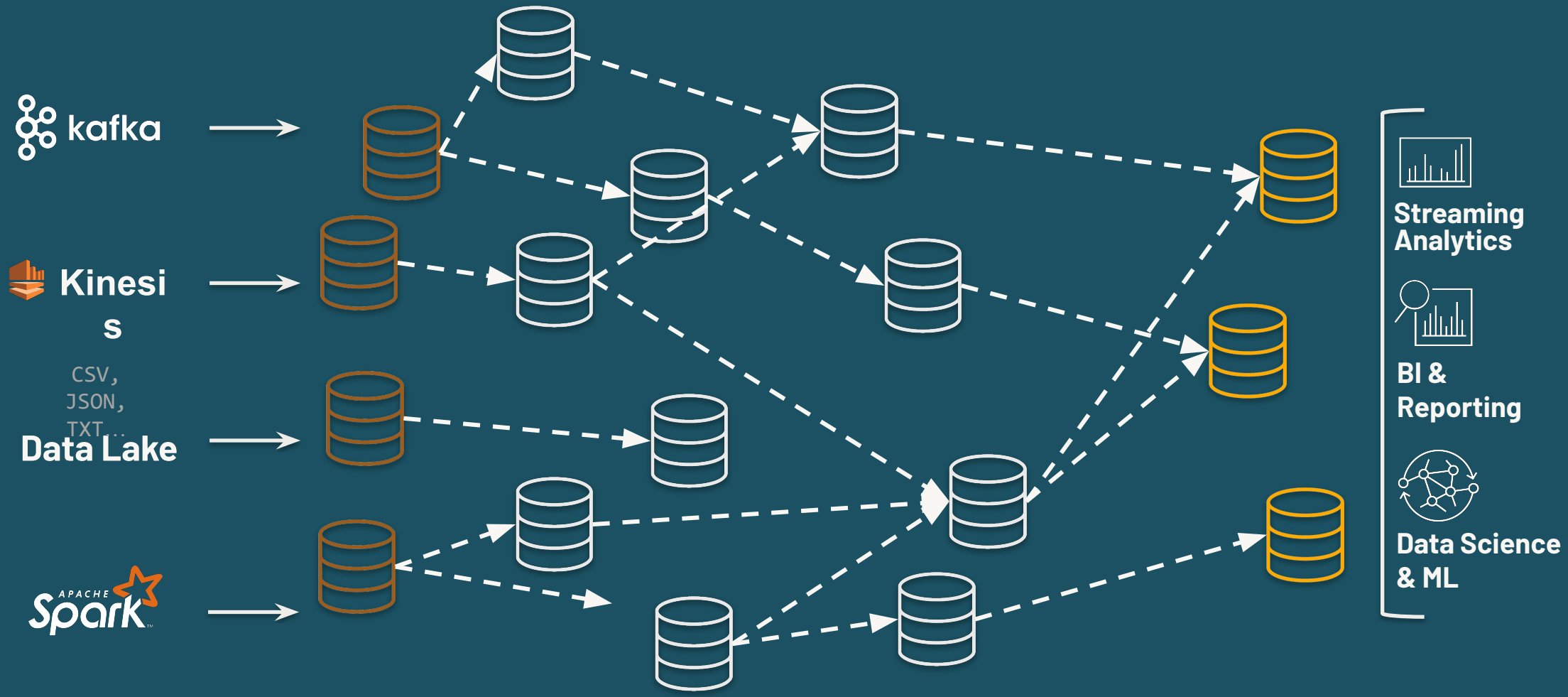


Good data is the foundation of a Lakehouse

All data professionals need clean, fresh and reliable data.

But the reality is not so simple

Maintaining data quality and reliability at scale is often complex and brittle



Life as a data professional…

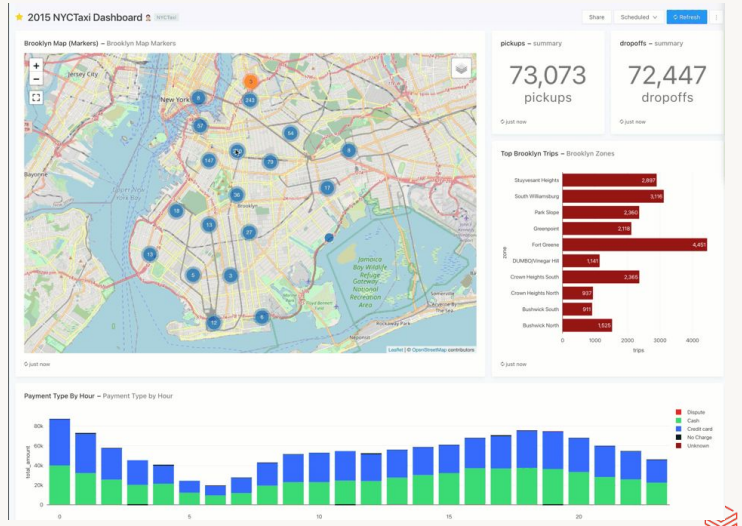
From: The CEO <ali@databricks.com>
Subject: Need an analysis ASAP!
To: Michael Armbrust
<michael@databrick.com>

Hey Michael, I need a quick analysis of our net customer retention and how it has changed over the past few quarters. Raw data can be found at s3://our-data-bucket/raw_data/...

```
1 %fs ls /data/sensors
```

	path
1	dbfs:/data/sensors/_SUCCESS
2	dbfs:/data/sensors/_committed_3908896360792309052
3	dbfs:/data/sensors/_started_3908896360792309052
4	dbfs:/data/sensors/part-00000-tid-3908896360792309052-adec30c0-9ba8-4344-a36c-7ec3
5	dbfs:/data/sensors/part-00001-tid-3908896360792309052-adec30c0-9ba8-4344-a36c-7ec3
6	dbfs:/data/sensors/part-00002-tid-3908896360792309052-adec30c0-9ba8-4344-a36c-7ec3
7	dbfs:/data/sensors/part-00003-tid-3908896360792309052-adec30c0-9ba8-4344-a36c-7ec3

???



Going from query to production

The tedious work required to turn SQL queries into reliable ETL Pipelines

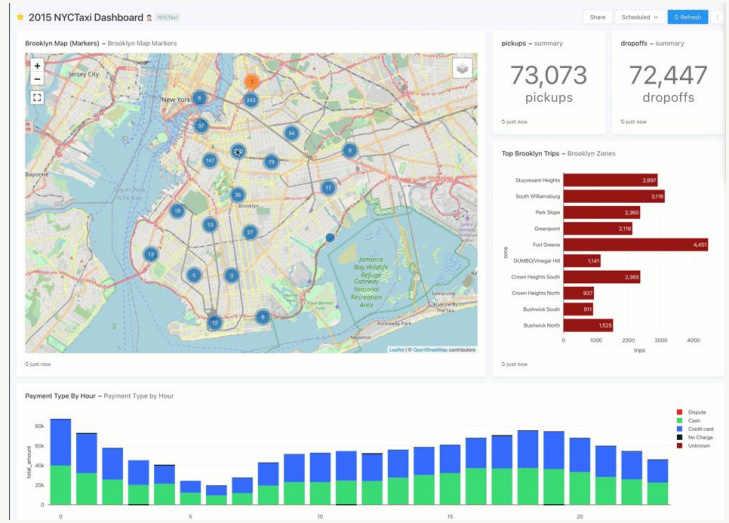
From: The CEO <ali@databricks.com>
Subject: Need an analysis ASAP!
To: Michael Armbrust <michael@databrick.com>

Great report! Can you update it ~~everyday~~ every minute?

```
CREATE TABLE raw_data as SELECT * FROM
json. `...`
CREATE TABLE clean_data as SELECT ... FROM raw_data
```

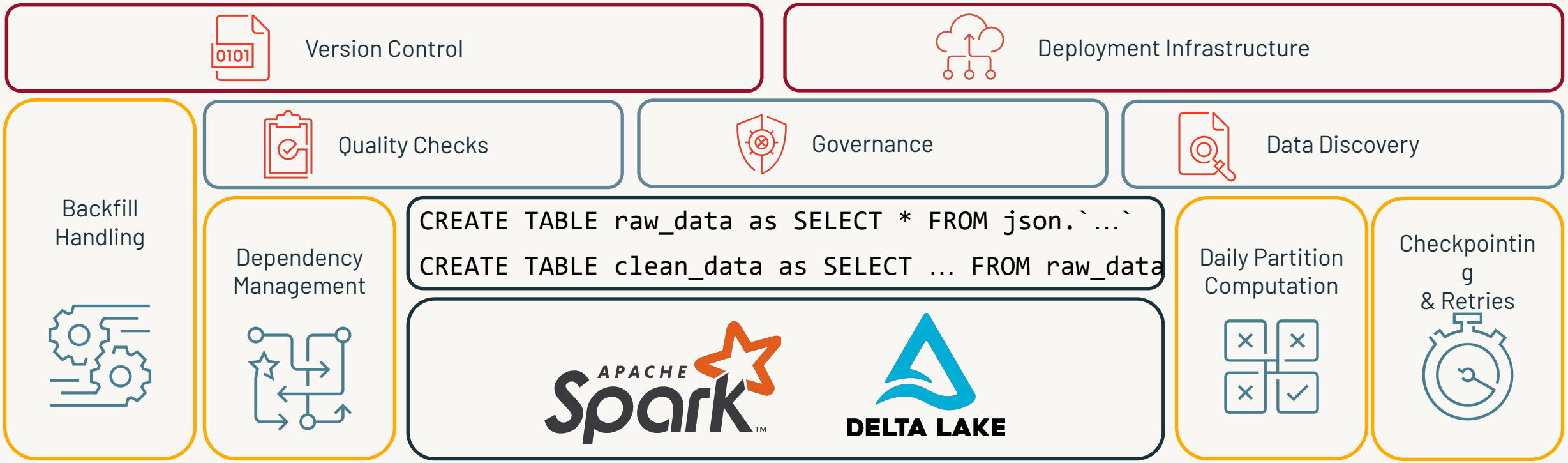
APACHE Spark™

DELTA LAKE



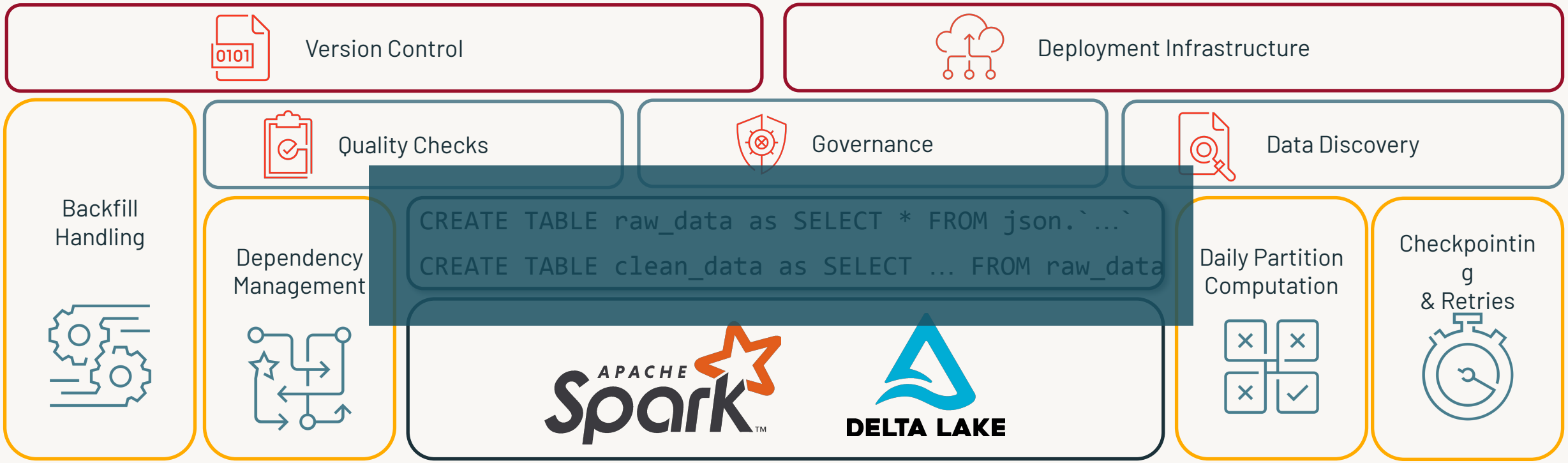
The slog from query to production

The tedious work required to turn SQL queries into reliable ETL Pipelines



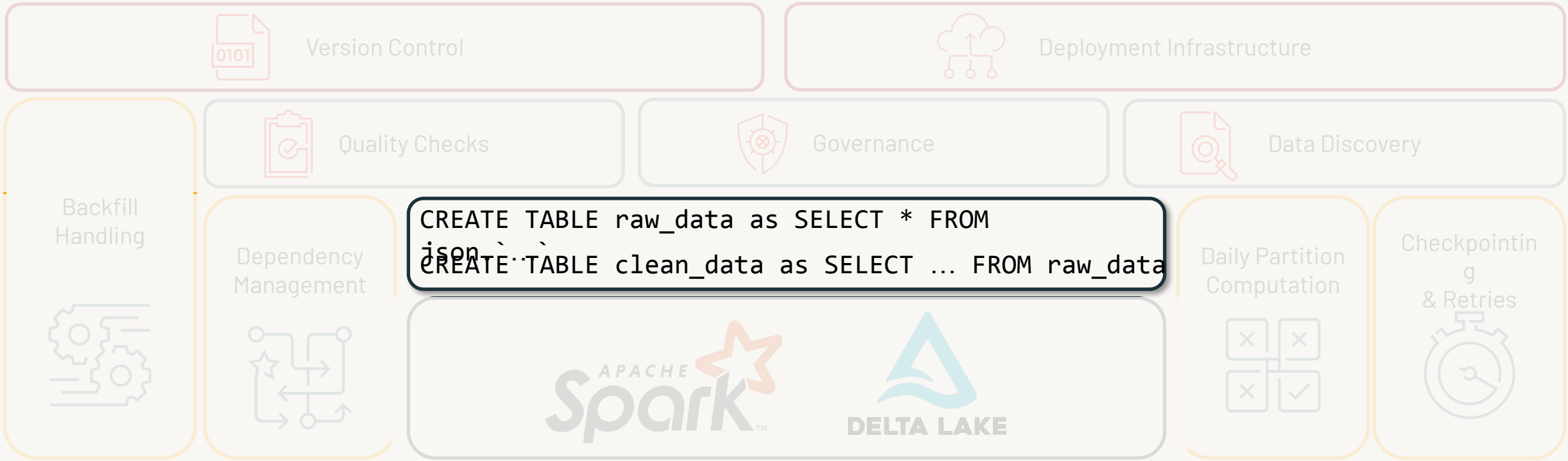
Operational complexity dominates

Time is spent on ***tooling*** instead of on ***transforming***



Where you should focus your time

Getting value from data



Introducing Delta Live Tables

From query to production pipeline just by adding LIVE.

```
CREATE STREAMING TABLE raw_data as SELECT * FROM cloud_files(…)
CREATE MATERIALIZED VIEW clean_data as SELECT … FROM LIVE.raw_data
```

Repos

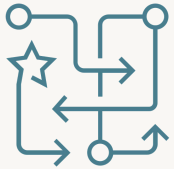
Unity Catalog

Databricks Workflows

Delta Live Tables

Full Refresh

Dependency Management



Expectations

APACHE Spark™

DELTA LAKE

Incremental Computation

Checkpointing & Retries

Introduction to Declarative Programming

Declarative programs say what should be done, not how to do it

Imperative program

```
numbers = […]
sum = 0
for n in numbers:
  sum += n
print(n)
```

Declarative Program

```
SELECT sum(n) FROM numbers
```

The query optimizer figures out the best way to calculate the sum

Data Independence
No need to write the query when physical things change (partitioning, storage location, indexes, etc)

Declarative Programming with DLT

Declarative programs say what should be done, not how to do it

Spark imperative program

```
date = current_date()
spark.read.table("orders")
  .where(s"date = $date")
  .select("sum(sales)")
  .write
  .mode("overwrite")
  .replaceWhere(s"date = $date")
  .table("sales")
```

DLT Declarative Program

```
CREATE MATERIALZED VIEW sales
AS SELECT date, sum(sales)
FROM orders
GROUP BY date
```

The DLT runtime figures out the best way to create or update this table

Workflows Or DLT?

Often Both: Workflows can orchestrate anything, including DLT

Use Workflows to run any task

- At some schedule
- After other tasks have completed
- When a file arrives
- When another table is updated

Use DLT for managing dataflow

- Creating/updating delta tables
- Running Structured Streaming

Why declarative programming?

Understanding your objective allows DLT to take care of the boring stuff

- Eliminate Boilerplate
  - Fault-tolerance
  - State management
  - Object lifecycle
  - Scheduling, dependencies, parallelism
  - Common data problems (cdc, schema evolution)
- Optimization
  - Choose the best strategy for execution dynamically, or across multiple queries
- Operations
  - HMR – detect and mitigate system regressions and cloud issues automatically

The core abstractions of DLT

You define datasets, and DLT automatically keeps them up to date

Streaming Tables

A delta table with stream(s) writing to it.

Used for:
- Ingestion
- Low latency transformations
- Huge scale

Materialized View

The result of a query, stored in a delta table.

Used for:
- Transforming data
- Building aggregate tables
- Speeding up BI queries and reports

What is a Streaming Table?

A delta table that has structured streaming writing to it.

```
CREATE STREAMING TABLE report
AS SELECT *
FROM cloud_files("/mydata/", "json")
```

- Streaming tables read from append-only data sources such as Kafka, Kinesis, or Auto Loader (files on cloud storage)
- Streaming tables allow you to reduce costs and latency by reading each input record only once.
- Streaming tables support DML (UPDATE, DELETE, MERGE) for ad-hoc data manipulation (i.e. GDPR, etc)

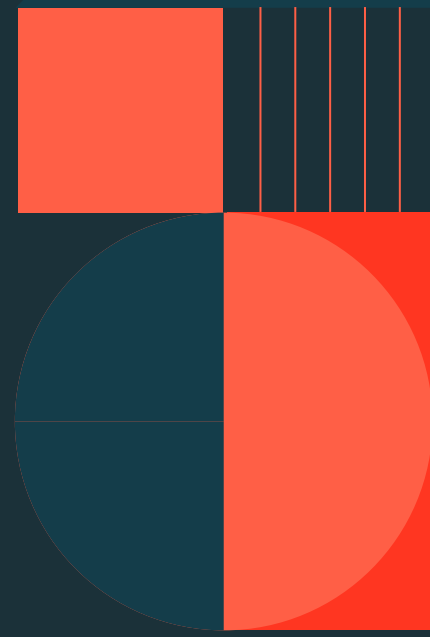
What is a Materialized View?

The result of a query, precomputed and stored in Delta

```
CREATE MATERIALIZED VIEW report
AS SELECT sum(profit)
FROM prod.sales
GROUP BY date
```

- A materialized view will always return the result of the the defining query, at the moment it was last updated (i.e. a snapshot)
- You cannot modify the data in a materialized view, you can change its query.

Data quality using Expectations



Ensure correctness with Expectations

Expectations are tests that ensure data quality in production

```
CONSTRAINT valid_timestamp
EXPECT (timestamp > '2012-01-01')
ON VIOLATION DROP
```

```
@dlt.expect_or_drop(
  "valid_timestamp",
  col("timestamp") > '2012-01-01')
```

Expectations are true/false expressions that are used to validate each row during processing.

DLT offers flexible policies on how to handle records that violate expectations:

- Track number of bad records
- Drop bad records
- Abort processing for a single bad record

Expectations using the power of SQL

Use SQL aggregates and joins to perform complex validations

```
-- Make sure a primary key is always unique.
CREATE MATERIALIZED VIEW report_pk_tests(
  CONSTRAINT unique_pk EXPECT (num_entries = 1)
)
AS SELECT pk, count(*) as num_entries
FROM LIVE.report
GROUP BY pk
```

Expectations using the power of SQL

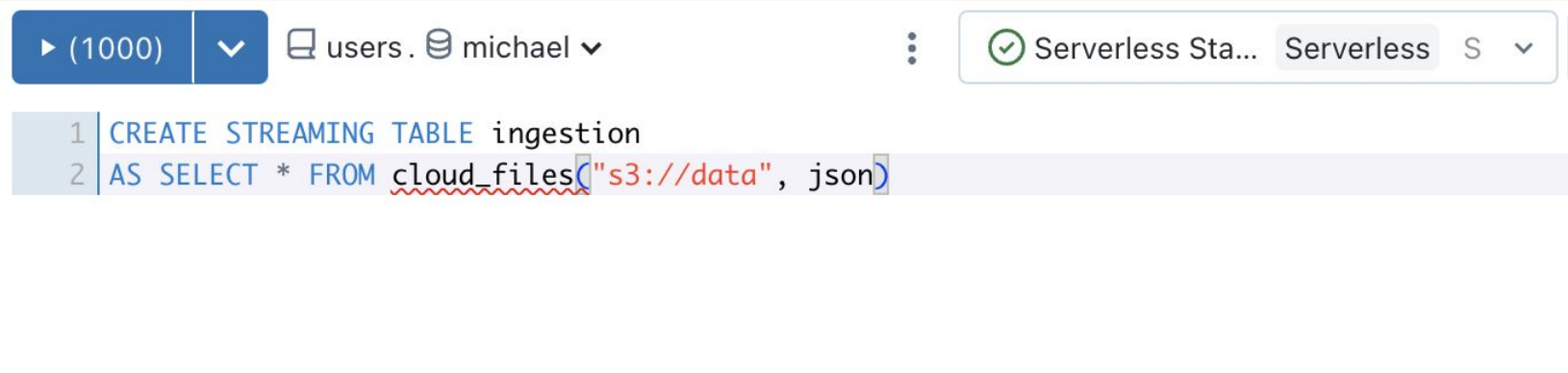
Use SQL aggregates and joins to perform complex validations

```
-- Compare records between two tables,
-- or validate foreign key constraints.
CREATE MATERIALIZED report_compare_tests(
  CONSTRAINT no_missing EXPECT (r.key IS NOT NULL)
)
AS SELECT * FROM LIVE.validation_copy v
LEFT OUTER JOIN LIVE.report r ON v.key = r.key
```

Announcing: DLT in DBSQL

PREVIEW

DLT powers streaming tables and materialized views created in DBSQL



A robust, scalable, incremental ingestion pipeline in one SQL command

Quick Note: Evolving terminology

Existing users of DLT will notice that we've evolved the names

- STREAMING LIVE TABLE ➡ STREAMING TABLE
- LIVE TABLE ➡ MATERIALIZED VIEW

The semantics remain the same, and we'll support the old syntax for compatibility. Our goal is to simplify the syntax and match other systems.

Deep dive into Streaming

Built on Spark™ Structured Streaming

Streaming tables combine spark structured streaming with a delta table.

Streaming Computation Model: Input is a growing append-only table

- Files uploaded to cloud storage
- Message busses like kafka, kinesis, or eventhub
- Delta tables with delta.appendOnly=true
- Transaction logs of other databases

Rather than wait until all data has arrived, structured streaming can produce results on demand.

- Lower latency by processing less data each update
- Lower costs by avoiding redundant work

Streaming does not always mean expensive

Delta live tables lets you choose how often to update the results.

Triggered: Manually

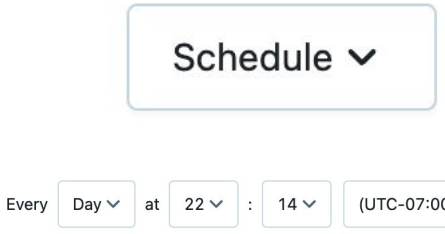
Costs: lowest

Latency: highest

Triggered: On a schedule using Databricks Jobs

Costs: depends on frequency

Latency: 10 minutes to months

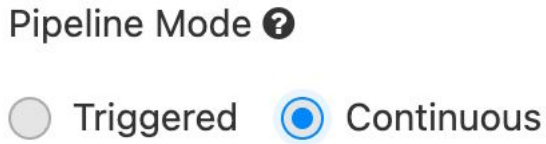


Continually

Costs: highest

Latency: minutes to seconds

(for some workloads)



Using Streaming Tables for ingestion

Easily ingest files from cloud storage as they are uploaded

```
CREATE STREAMING TABLE raw_data
AS SELECT *
FROM cloud_files("/data", "json")
```

This example creates a table with all the json data stored in "/data":

- `cloud_files` keeps track of which files have been read to avoid duplication and wasted work
- Supports both listing and notifications for arbitrary scale
- Configurable schema inference and schema evolution

Using Spark™ Structured Streaming for ingestion

Easily ingest records from message buses

```
@dlt.table
def kafka_data():
  return spark.readStream \
    .option("format", "kafka") \
    .option("subscribe", "events") \
    .load()
```

This example creates a table with all the records published to the Kafka topic "event".

- The Kafka source + DLT automatically track which partitions / offsets have already been read.
- Any structured streaming source included in DBR can be used with DLT
- Message buses provide the lowest latency for ingesting data

Using the SQL STREAM() function

Stream data from any Delta table

```
CREATE STREAMING TABLE mystream
  AS SELECT …
  FROM STREAM(prod.events)
```

```
CREATE STREAMING TABLE mystream
  AS SELECT …
  FROM STREAM(delta.`s3:/…`)
```

- STREAM(…) reads newly inserted records into the table as they arrive.
- Only works for append-only delta tables

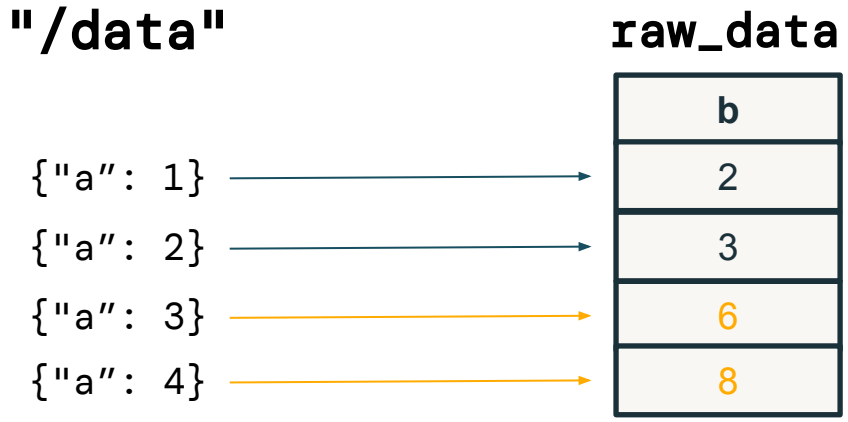
Understanding streaming state

Streaming tables are stateful

Each input row is processed only once, even if the query changes.

A change to a streaming live table's definition does not reread already processed data:

```
CREATE STREAMING TABLE raw_data
AS SELECT a + 1 AS a  a * 2 AS a
FROM cloud_files("/data", "json")
```

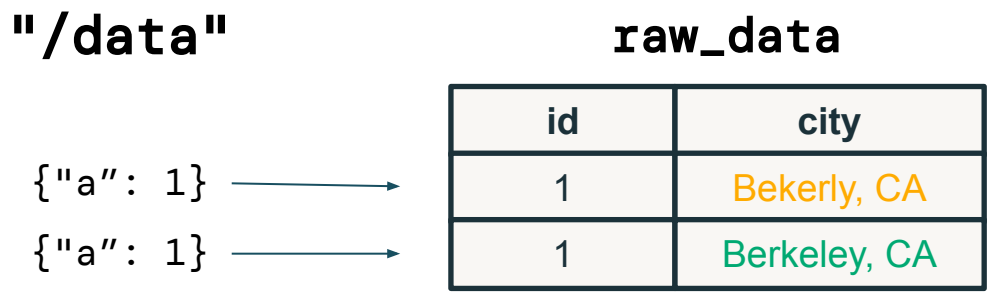


Streaming joins are stateful

Enrich data by joining with an up-to date-snapshot stored in delta

- Snapshot is automatically updated each microbatch if it changes
- A change to joined table snapshot does not recompute results:

```
CREATE STREAMING TABLE raw_data
AS SELECT *
FROM cloud_files("/data", "json") f
JOIN prod.cities c USING id
```



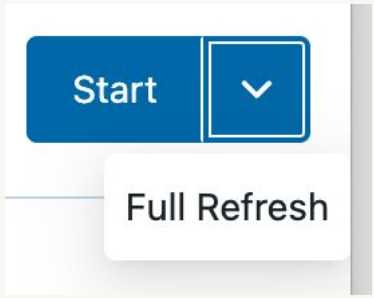
Reprocess data using full refresh

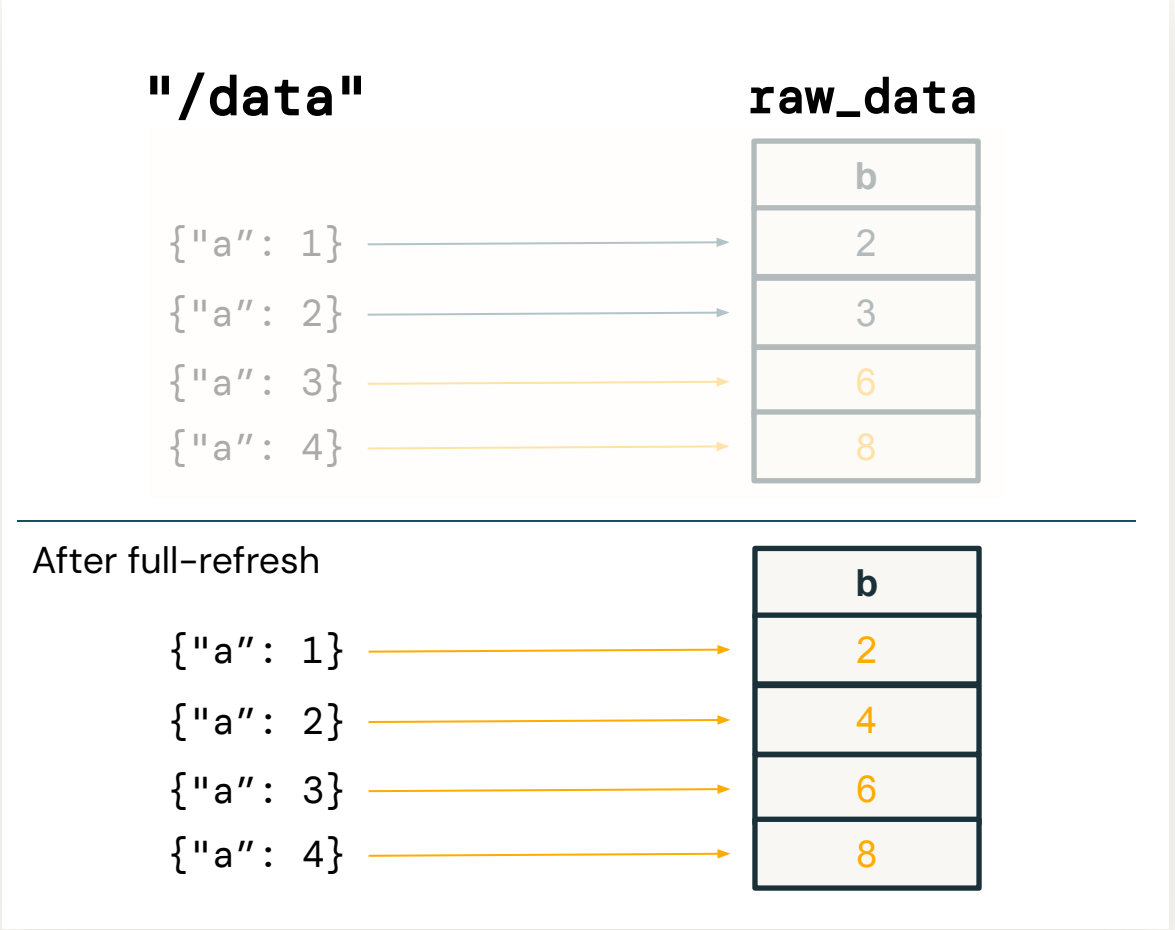
Automatically perform backfills after critical changes using full refresh

Full-refresh clears the table's data and the queries state, reprocessing all the data.

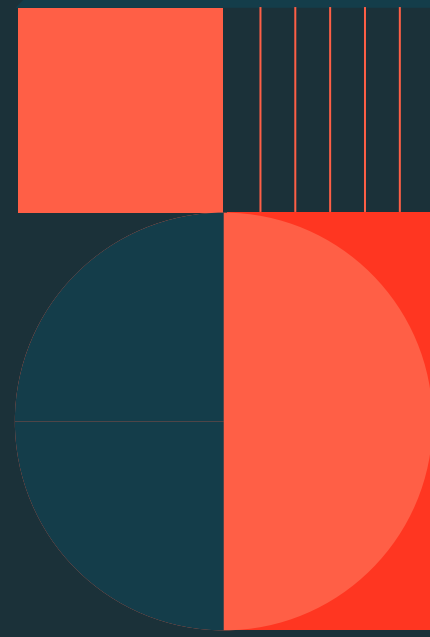
```
CREATE STREAMING TABLE raw_data
AS SELECT a * 2 AS a
FROM cloud_files("/data", "json")
```

```
REFRESH raw_data FULL
```





Using stream-stream joins



Joining facts with stream-stream joins

Stream-stream joins let you combine facts that arrive near each other

```
@dlt.table()
def joined():
  impressions = read_stream("impressions") \
    .withWatermark("impressionTime", "10 minutes")

  clicks = clicks \
    .withWatermark("clickTime", "20 minutes")

  return impressions.join(clicks,
    expr("""clickAdId = impressionAdId AND
            clickTime >= impressionTime AND
            clickTime <= impressionTime + interval 5 minutes"""))
```

This example, based on ad-tech, joins a stream of clicks with details about an ad impression that caused them to occur.

Unlike in snapshot-stream join, a stream-stream join will buffer records until a match appears.

Joining facts with stream-stream joins

Stream-stream joins let you combine facts that arrive near each other

```
@dlt.table()
def joined():
  impressions = read_stream("impressions") \
    .withWatermark("impressionTime", "10 minutes")

  clicks = clicks \
    .withWatermark("clickTime", "20 minutes")

  return impressions.join(clicks,
    expr("""clickAdId = impressionAdId AND
            clickTime >= impressionTime AND
            clickTime <= impressionTime + interval 5 minutes"""))
```

Two streams of facts, each with a watermark

The join condition

A time bound on how far apart each fact can arrive

Pitfall: If the watermark or timebound are missing, the join will buffer all data forever.

Adding and removing streaming sources

COMING SOON

Streaming Tables and Flows

In DLT a flow is the object that updates a table with new data

The simple syntax we've shown so far is a syntactic sugar for creating a table and a flow with a single DDL command.

```
CREATE STREAMING TABLE raw_data
AS SELECT * FROM kafka(…)
```

⟺

```
CREATE STREAMING TABLE raw_data

CREATE FLOW raw_data
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)
```

Checkpoints are identified by the name of the flow

`checkpoints/raw_data`

Kafka Offsets

```
{
  0: 123354,
  2: 9573943,
  3: 83275092…
}
```

COMING SOON

Multi-flow Tables

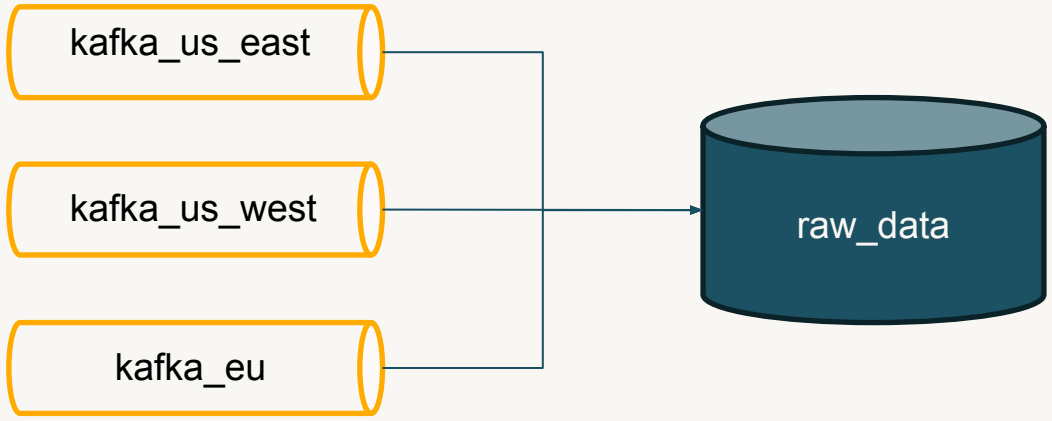
Evolve your streaming sources, with differently named flows

```
CREATE STREAMING TABLE raw_data


CREATE FLOW kafka_us_east
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)


CREATE FLOW kafka_us_west
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)


CREATE FLOW kafka_eu
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)
```



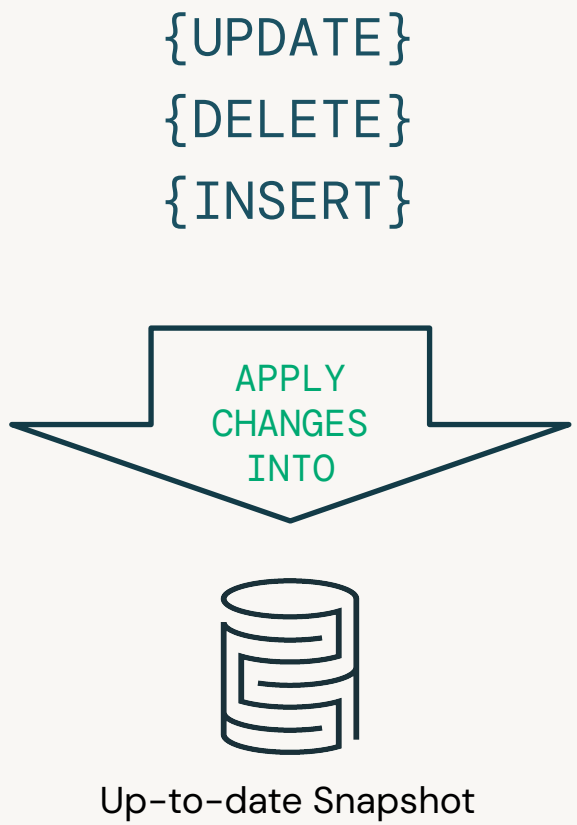
Streaming change data capture (CDC)

APPLY CHANGES INTO for CDC

Maintain an up-to-date replica of a table stored elsewhere

```
CREATE STREAMING TABLE cities

APPLY CHANGES INTO LIVE.cities
FROM STREAM(city_updates)
KEYS (id)
SEQUENCE BY ts
```



APPLY CHANGES INTO for CDC

Maintain an up-to-date replica of a table stored elsewhere

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(city_updates)
KEYS (id)
SEQUENCE BY ts
```

A source of changes, currently this has to be a stream.

city_updates

```
{"id": 1, "ts": 100, "city": "Bekerly, CA"}
```

APPLY CHANGES INTO for CDC

Maintain an up-to-date replica of a table stored elsewhere

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(city_updates)
KEYS (id)
SEQUENCE BY ts
```

A target for the changes to be applied to.

city_updates

```
{"id": 1, "ts": 100, "city": "Bekerly, CA"}
```

cities

APPLY CHANGES INTO for CDC

Maintain an up-to-date replica of a table stored elsewhere

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(city_updates)
KEYS (id)
SEQUENCE BY ts
```

A unique key that can be used to identify a given row.

city_updates

```
{"id": 1, "ts": 100, "city": "Bekerly, CA"}
```

cities

APPLY CHANGES INTO for CDC

Maintain an up-to-date replica of a table stored elsewhere

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(city_updates)
KEYS (id)
SEQUENCE BY ts
```

A sequence that can be used to order changes:

- Log sequence number (lsn)
- Timestamp
- Ingestion time

city_updates

```
{"id": 1, "ts": 100, "city": "Bekerly, CA"}
```

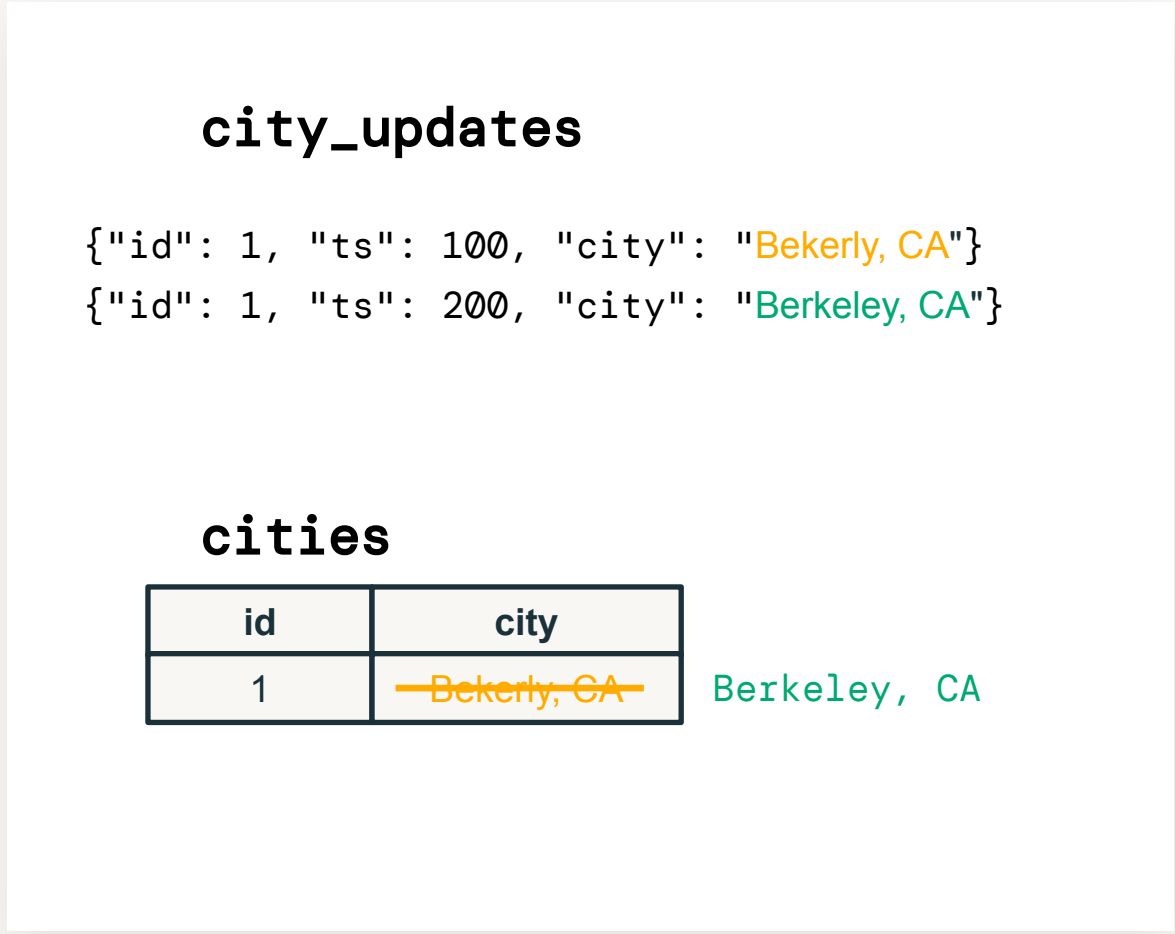
cities

id	city

APPLY CHANGES INTO for CDC

Maintain an up-to-date replica of a table stored elsewhere

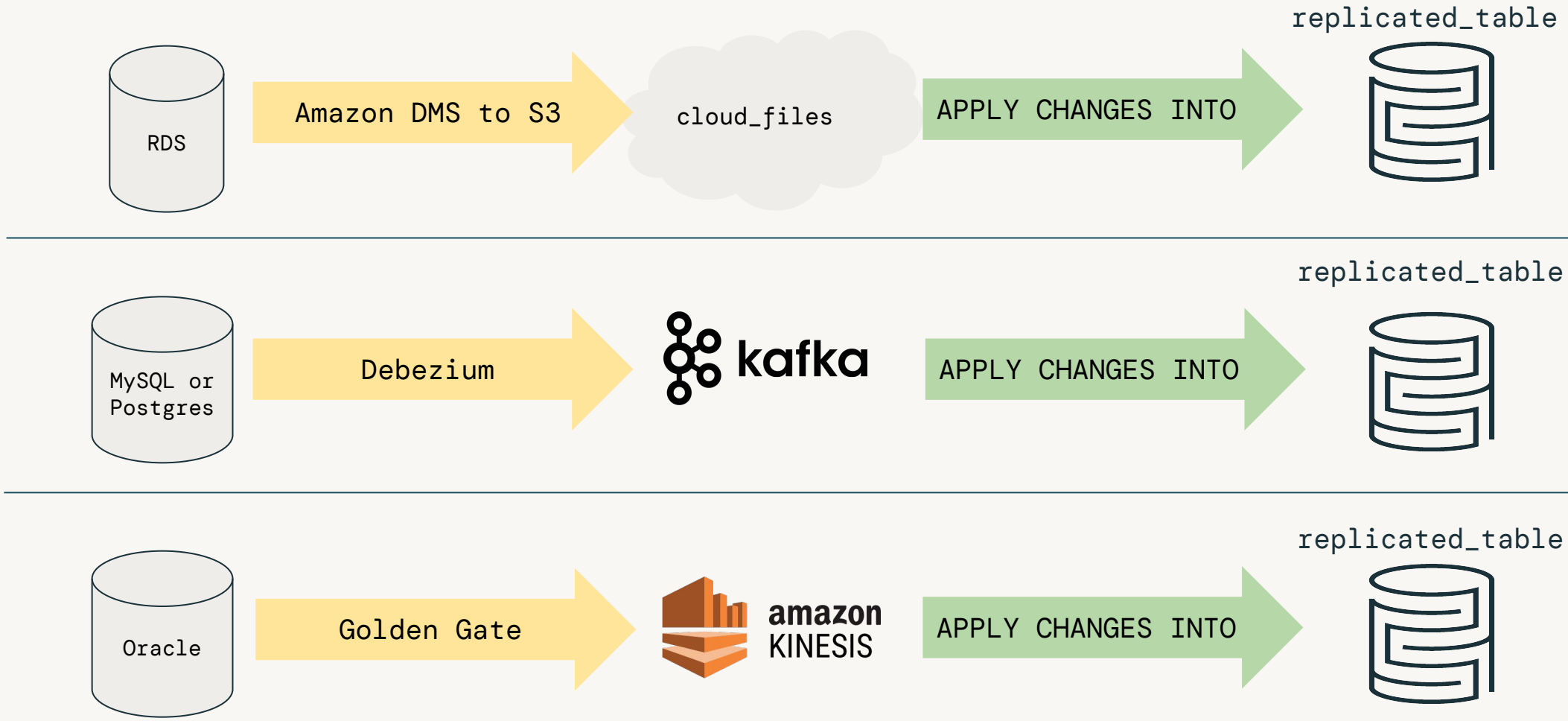
```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(LIVE.city_updates)
KEYS (id)
SEQUENCE BY ts
```



REFERENCE ARCHITECTURE

Change Data Capture (CDC) from RDBMS

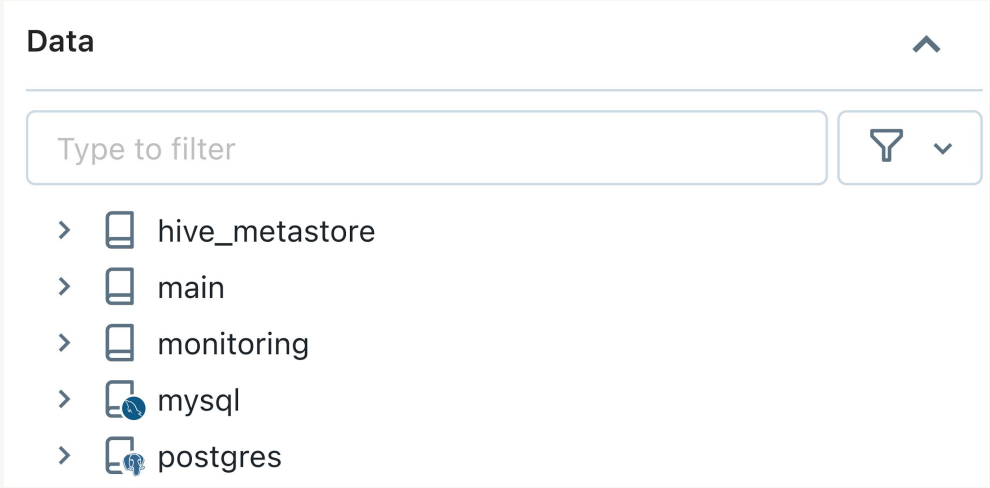
A variety of 3rd party tools can provide a streaming change feed

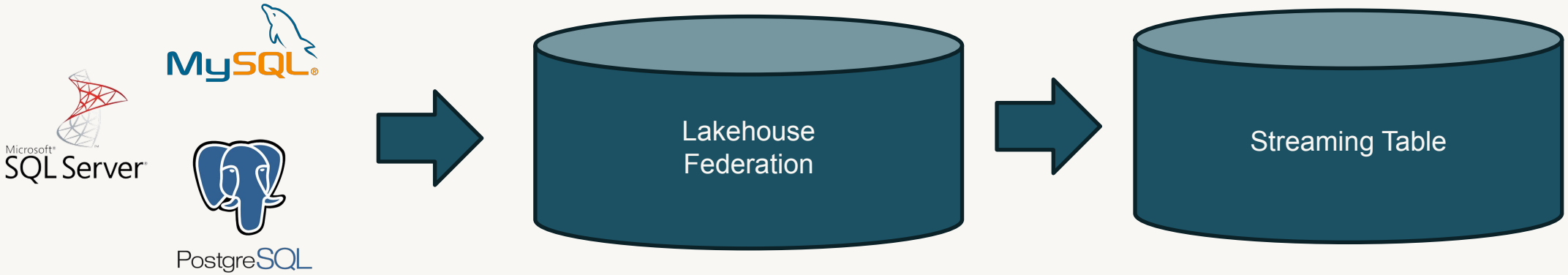


Federated CDC with UC

Seamless replication into the lakehouse

Govern access to CDC sources using Unity Catalog.





CDC: How does it work?

Reconciliation allows us to transparently handle out-of-order data

Arrival Time

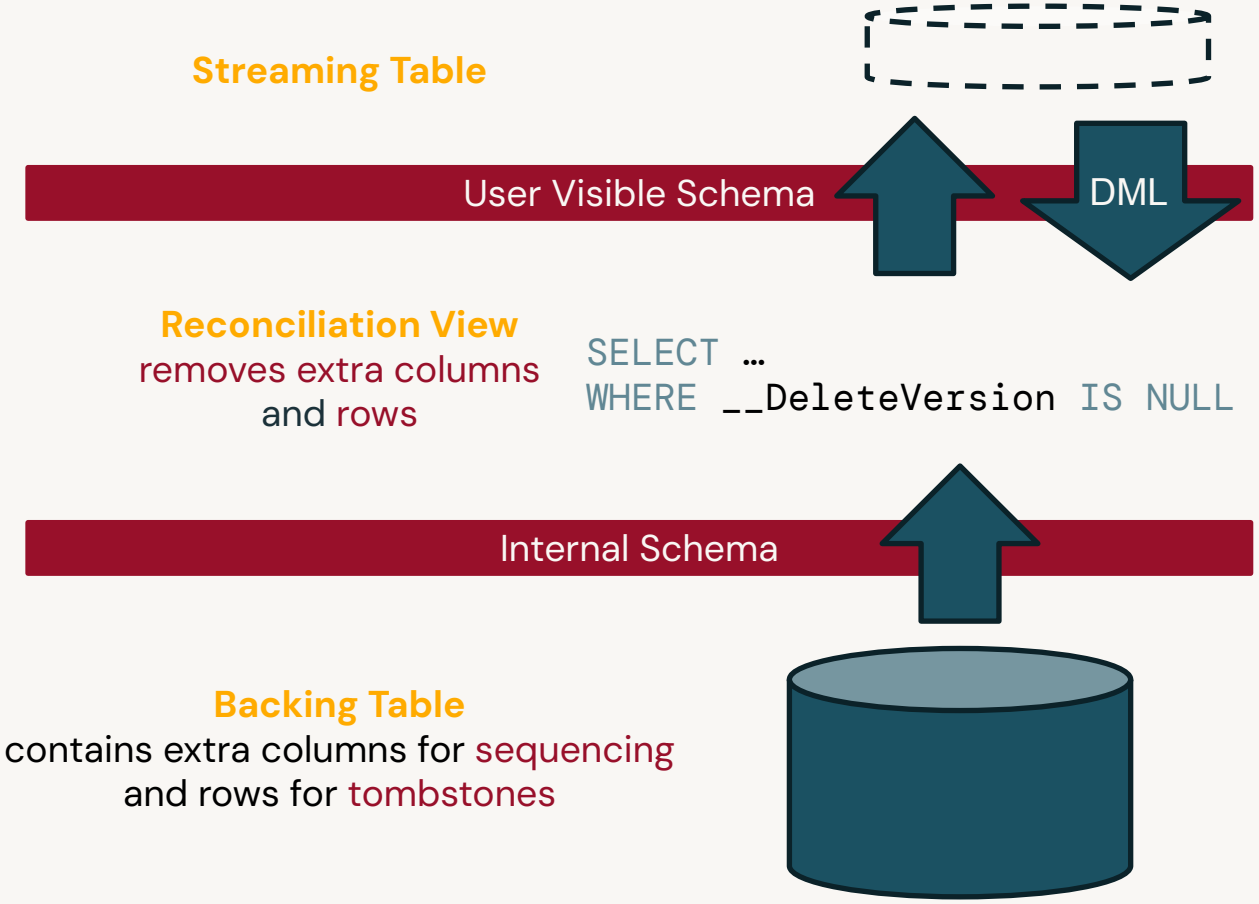
Id	timestamp	Operation
1	2	delete
1	1	update

Delete should happen after update

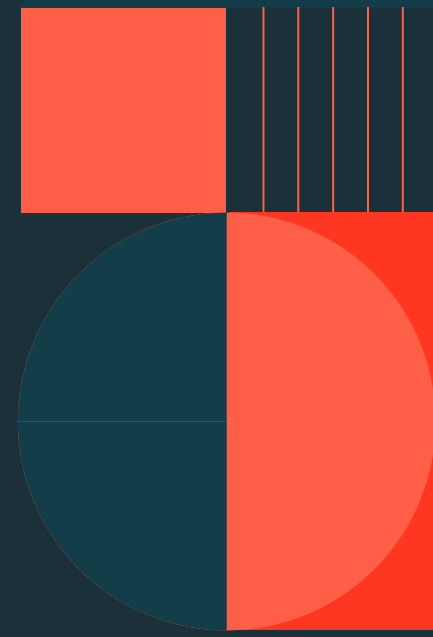
But the update arrived late

Need to remember the timestamp of the delete to prevent a permanent zombie row

Diagram depicts DLT+UC
In HMS we must fake it with a normal view



How can I track history?



Slowly Changing Dimensions Type 2

Keep a record of how values changed over time

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(LIVE.city_updates)
KEYS (id)
SEQUENCE BY ts
STORED AS SCD TYPE 2
```

`__starts_at` and `__ends_at` will have the type of the `SEQUENCE BY` field (`ts`).

city_updates

```
{"id": 1, "ts": 1, "city": "Bekerly, CA"}
{"id": 1, "ts": 2, "city": "Berkeley, CA"}
```

cities

id	city	__starts_at	__ends_at
1	Bekerly, CA	1	2
1	Berkeley, CA	2	null

When should I chain streaming?

Multi-hop streaming for cost and latency

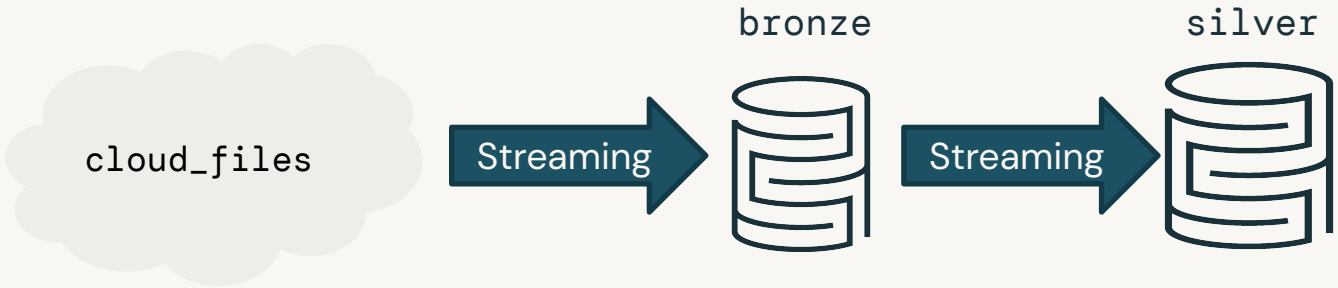
Append-only delta tables can be both a source and a sink

```
CREATE STREAMING TABLE bronze
AS SELECT * FROM cloud_files("/data/", "json”)

CREATE STREAMING LIVE TABLE silver
AS SELECT …
FROM STREAM(bronze)
```

Chain multiple streaming jobs for workloads with:

- Very large data
- Very low latency targets



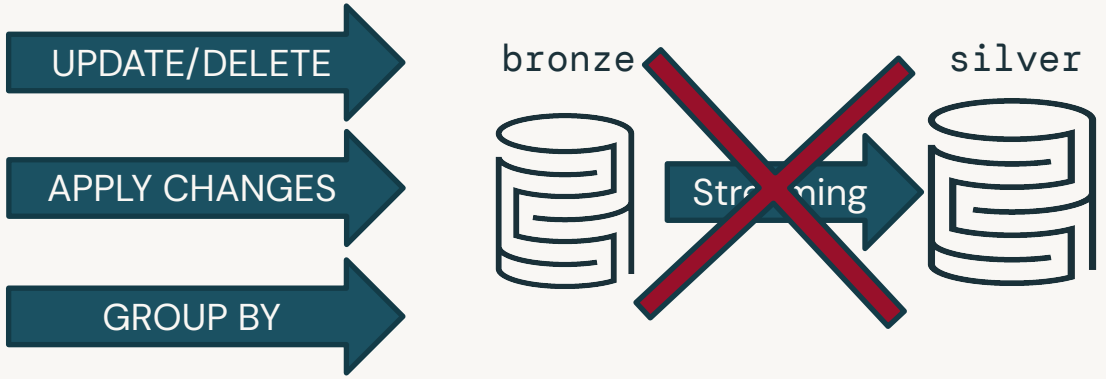
Pitfall: Updates in streaming pipelines

Structured streaming assumes an append-only input source

Updates to a streaming input table will break downstream computation

- Non-insert DML performed on streaming tables
- Streaming tables that compute aggregations without a watermark
- Streaming tables used with APPLY CHANGES INTO

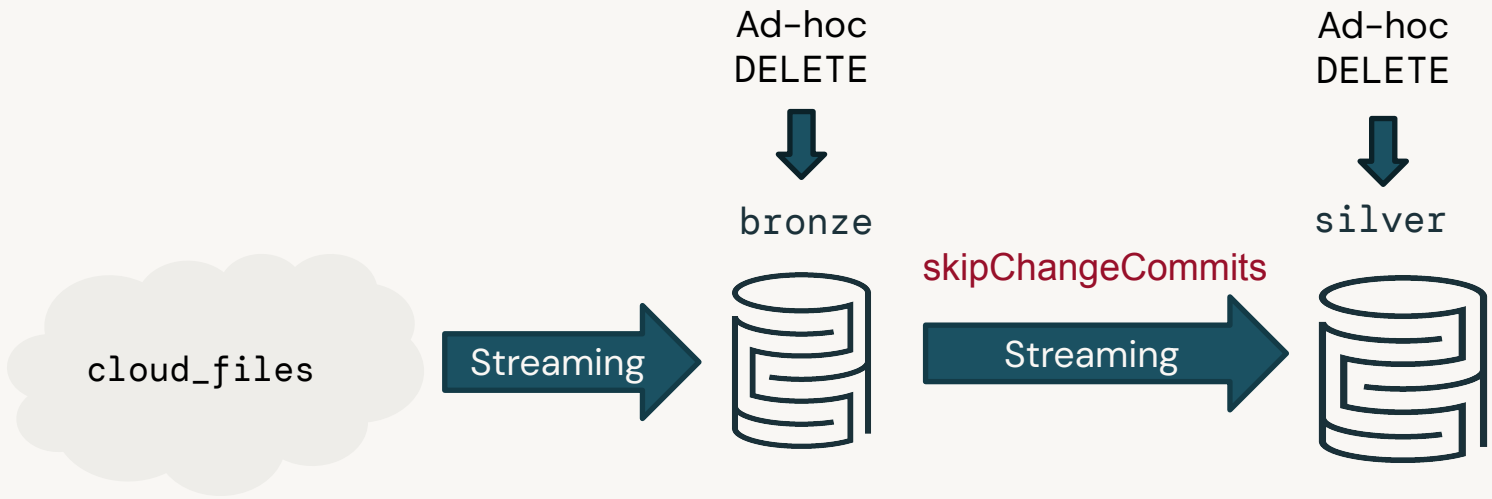
Repair the stream
after one-off DML
using a full-refresh



Manually handling updates with Streaming

skipChangeCommits will prevent a pipeline from failing after a change

- If changes are uncommon, skipChangeCommits will tell streaming to skip changes, rather than fail.
- DML must be used to manually propagate changes downstream.



Deep dive into Materialized Views

Materialized views simplify transformation

Materialized views always return the same results as their defining query

Compared to streaming tables, Materialized Views:

- Handle appends, updates, deletes to their inputs
- Support any aggregations / window functions
- Joins work the same as in ad-hoc queries
- Automatically handle changes to the query definition

Unlike streaming, MVs are allowed to
read the input more than once, and
recompute from scratch when necessary

MV Incremental refresh with Enzyme

Data is always changing

Incrementalization updates derived datasets without recomputing everything

date	city_id
2022-06-01	1
2022-06-01	2
2022-06-02	3

REPLACE TABLE

date	city_id	city_name
2022-06-01	1	Berkeley
2022-06-01	2	San Francisco
2022-06-01	3	Denver

date	city_id
2022-06-01	1
2022-06-01	2
2022-06-02	3

CREATE MATERIALIZED VIEW

date	city_id	city_name
2022-06-01	1	Berkeley
2022-06-01	2	San Francisco
2022-06-01	3	Denver

How can enzyme incrementally refresh different types of queries?

PREVIEW

Appending new data as it arrives

Works when new data is added and the query is monotonic

mon·o·ton·ic que·ry

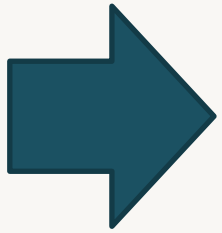
/ˌmänəˈtänik ˈkwirē/

noun

A query that does not lose any tuples it previously made output, with the addition of new tuples in the database

- Very efficient (like streaming!)
- Only works for select/project/inner join/etc
- Does not handle changes to the input

date	city_id
2022-06-01	1
2022-06-01	2
2022-06-02	3

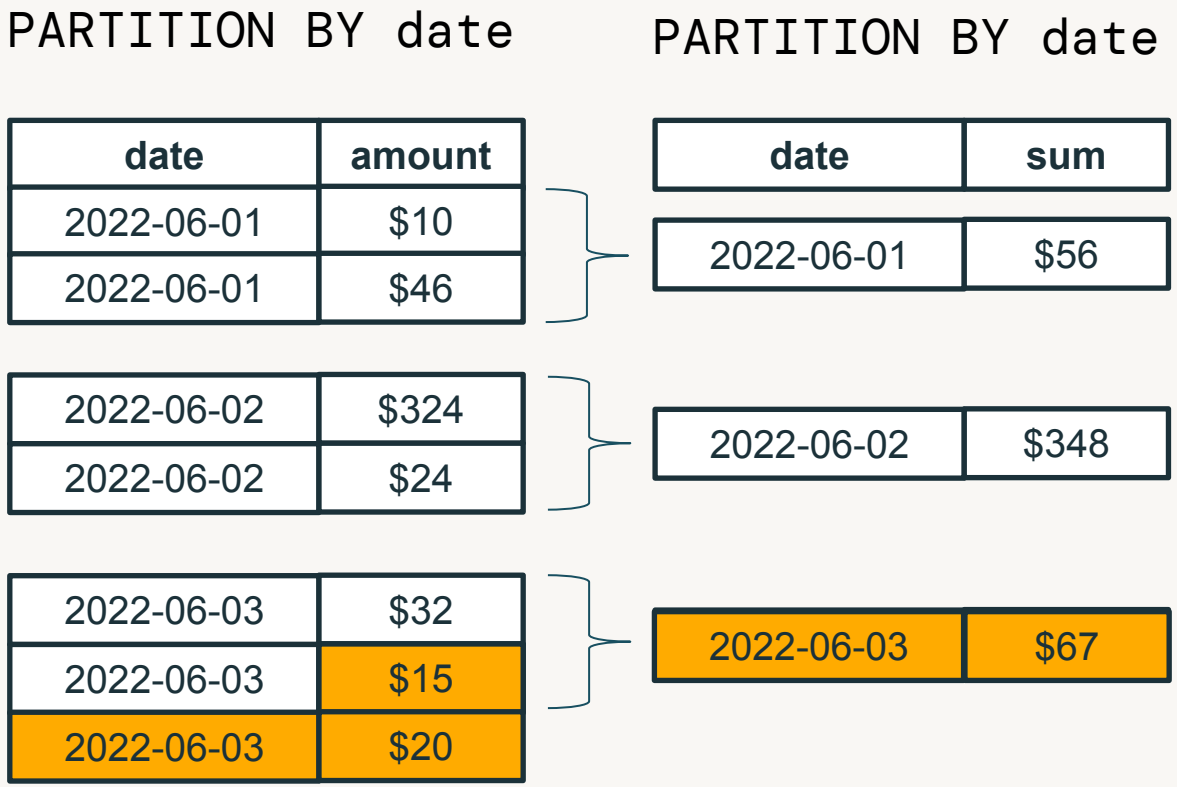


date	city_id	city_name
2022-06-01	1	Berkeley
2022-06-01	2	San Francisco
2022-06-02	3	Denver

PREVIEW

Partition recomputation

Split the table into partitions and only recompute ones that change



- Able to handle aggregations and updates
- Requires the input and output to have the same partitioning

PREVIEW

MERGE updates to specific rows

Use techniques from databases literature to compute changes to results

- Able to handle complicated queries
- Able to handle updates/inserts/deletes
- Merge is expensive
- Complicated to reason about

date	amount
2022-06-01	$10
2022-06-01	$46
2022-06-02	$324
2022-06-02	$24
2022-06-03	$32
2022-06-03	$15
2022-06-03	$20

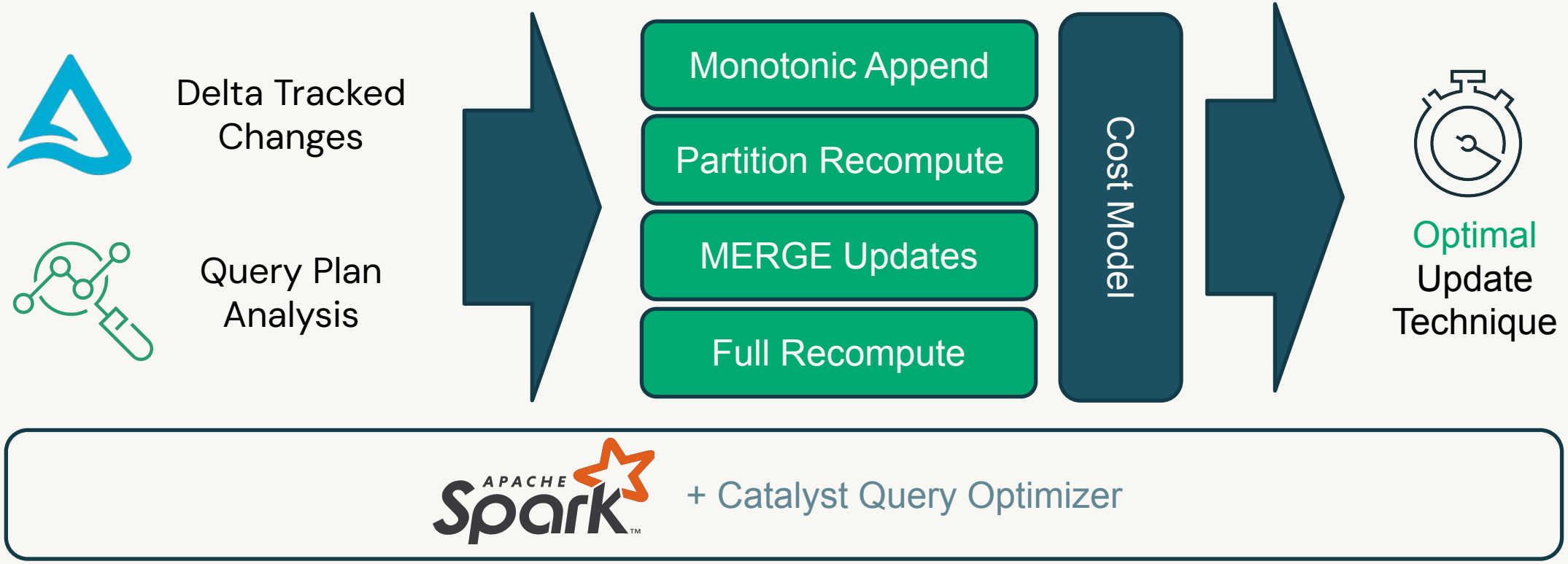
date	sum
2022-06-01	$56
2022-06-02	$348
2022-06-03	$67

How does enzyme pick the right technique?

PREVIEW

Introducing Enzyme

An automatic optimizer for incremental ETL

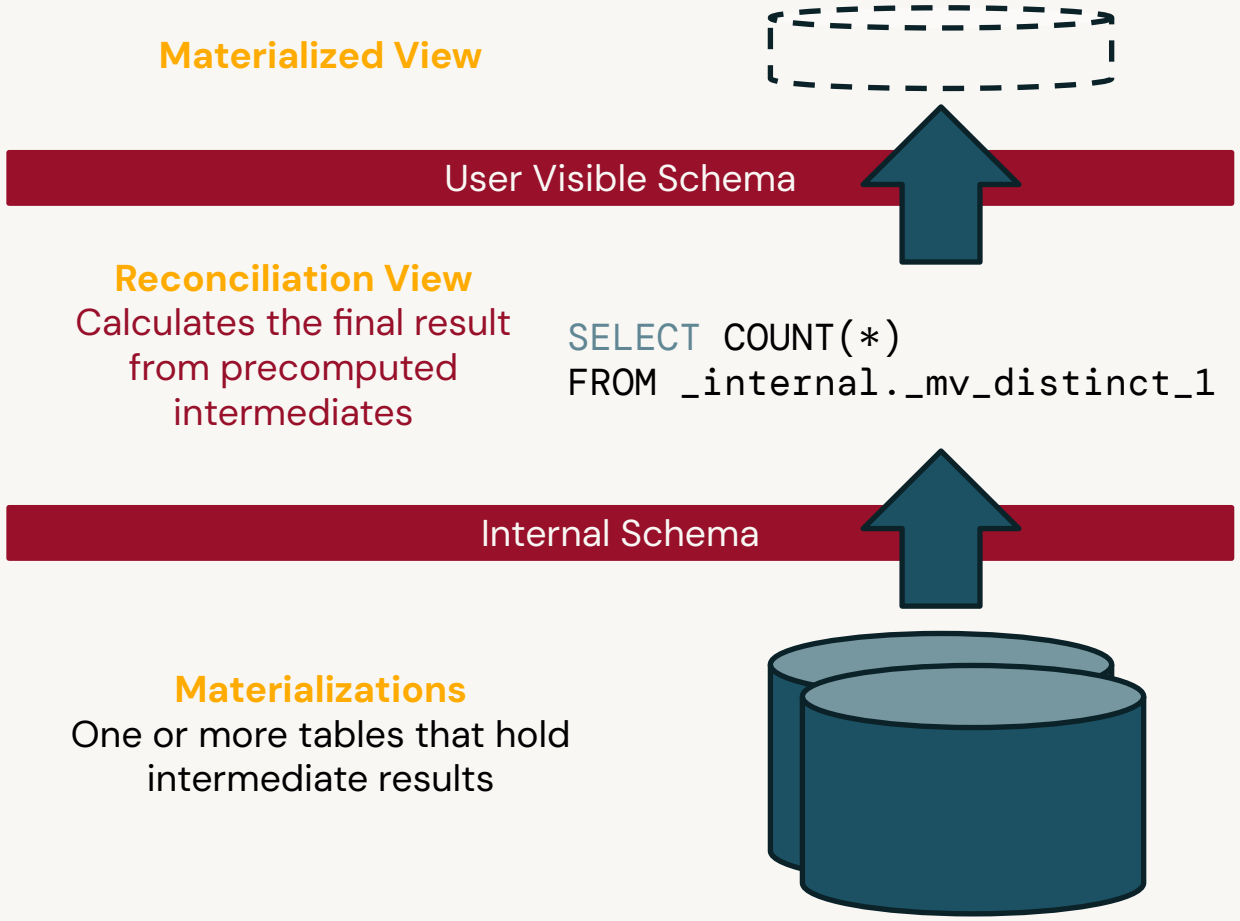


Enyme: How does it work?

Decomposition breaks up queries to enable incrementalization

```
CREATE MATERIALIZED VIEW total_unique_users
SELECT COUNT(DISTINCT customer_id)
FROM prod.clicks
```

```
CREATE MATERIALIZED VIEW _internal._mv_distinct_1
SELECT COUNT(*)
FROM prod.clicks
GROUP BY customer_id
```



Announcing: DLT Serverless

PREVIEW

DLT serverless has enzyme enabled by default

Enzyme is only available for:

- MVs created in DBSQL (public preview)
- MVs in DLT serverless pipelines (private preview)

Enzyme can incrementally refresh many queries:

- Co-partitioned
- Associative aggreagates
- Monotonic queries

Enzyme roadmap

- Inner/outer joins
- Semi-ring aggregates
- Distinct aggregates
- Window functions
- Predicates on current_date()

Demo

https://www.databricks.com/resources/demos/tutorials/lakehouse-platform/full-delta-live-table-pipeline

Thank you