databricks

Delta Live Tables

Concepts, Best Practices & Reference Architectures

May 2022

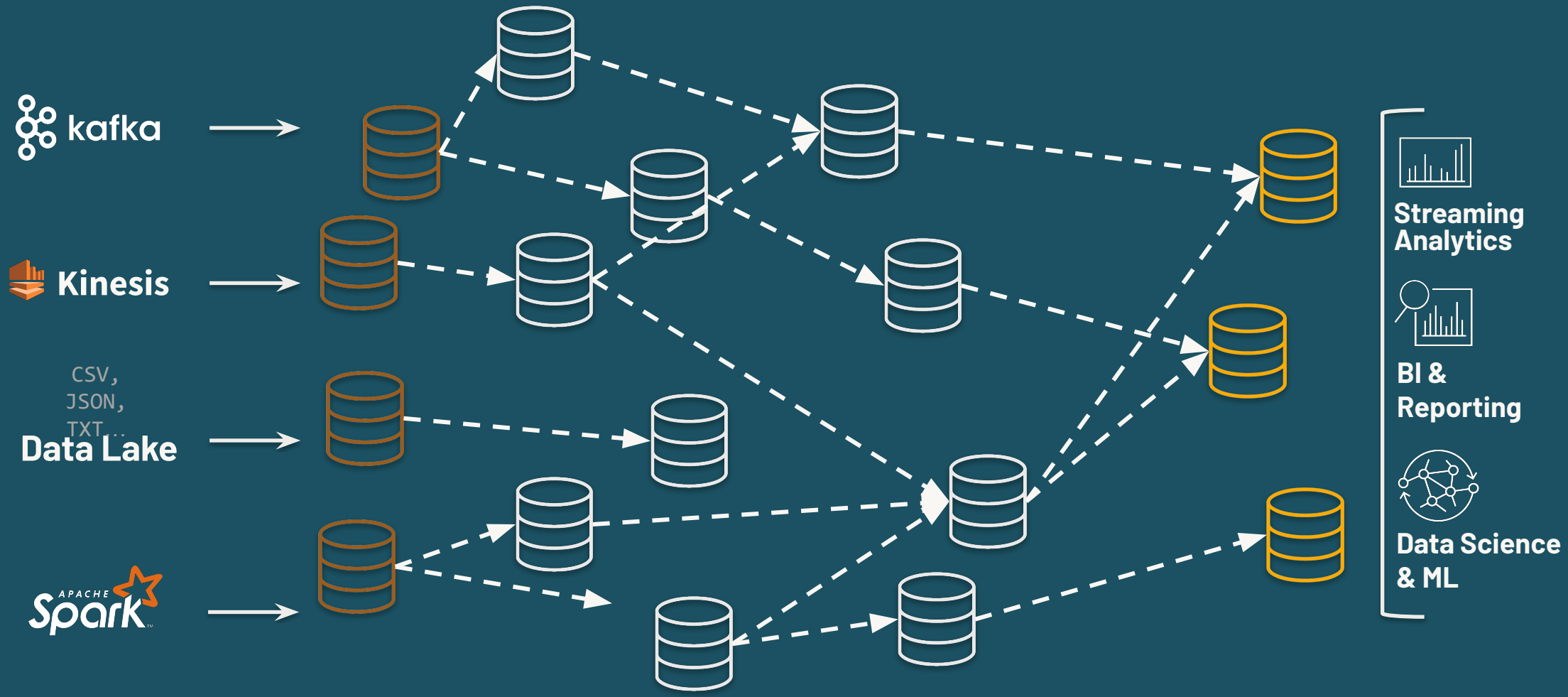
Michael Armbrust & Paul Lappas

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…

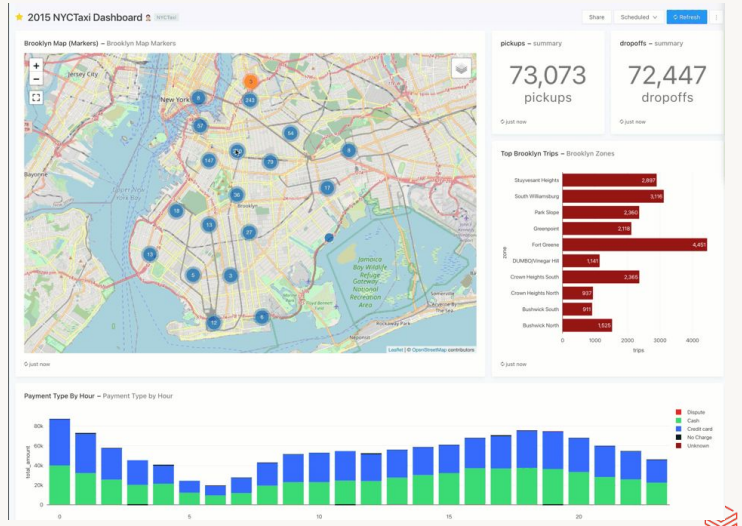
From: The CEO <ali@databricks.com>
Subject: Need an analysis ASAP!
To: Michael Armbrust
<michael@databrick.com>

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

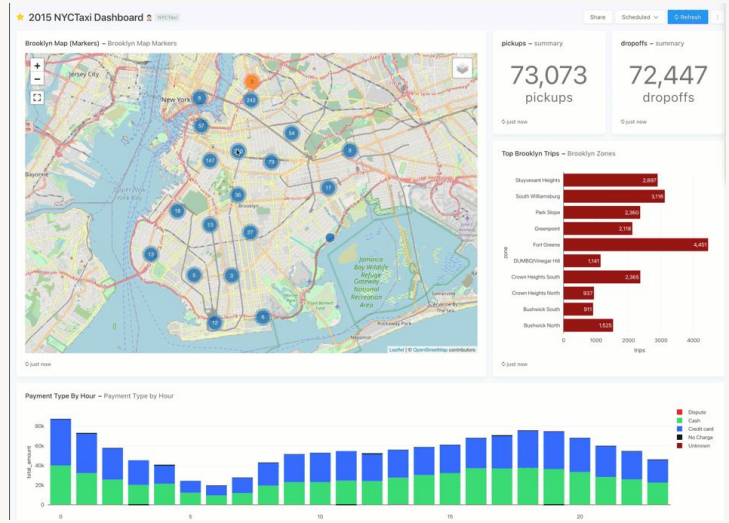
From: The CEO <ali@databricks.com>
Subject: Need an analysis ASAP!
To: Michael Armbrust <michael@databrick.com>

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
```

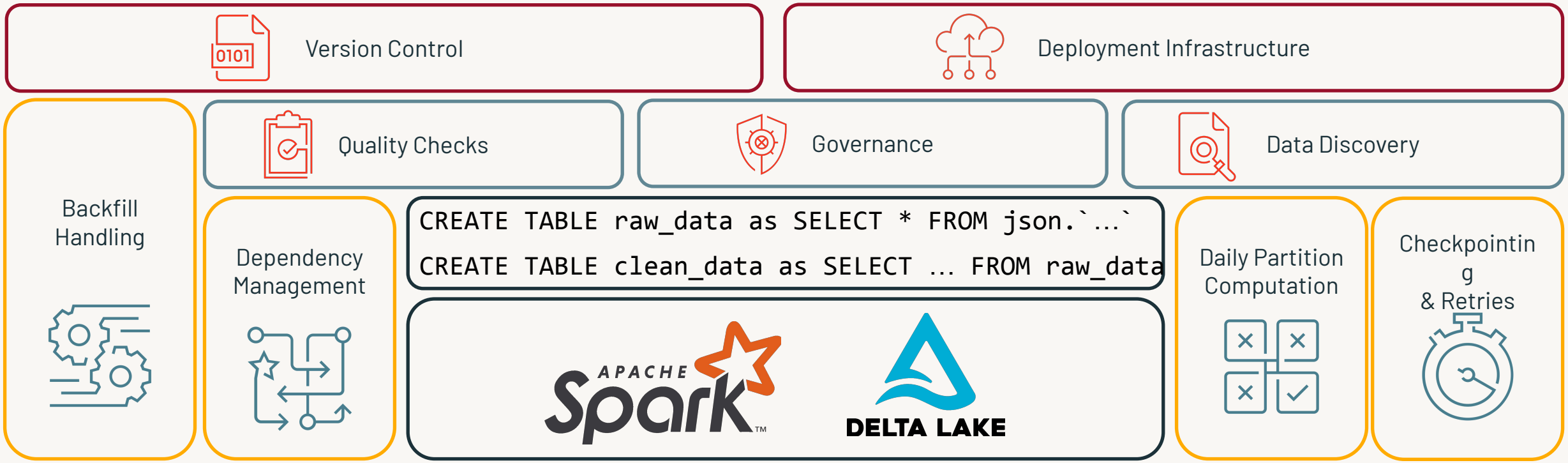
APACHE Spark™

DELTA LAKE



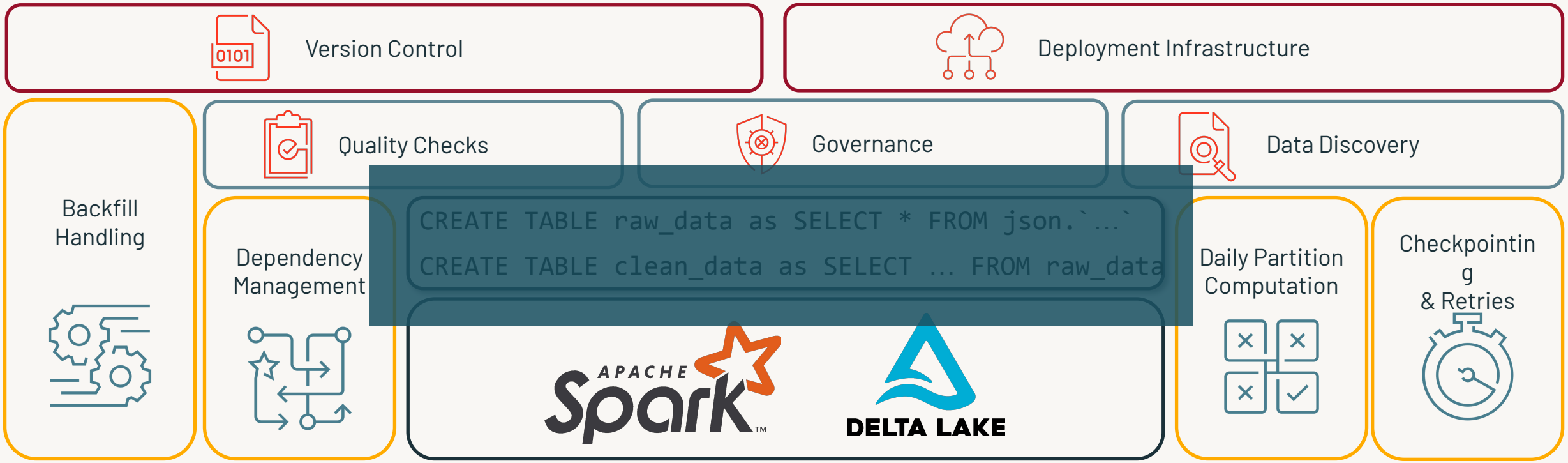
The slog from query to production

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



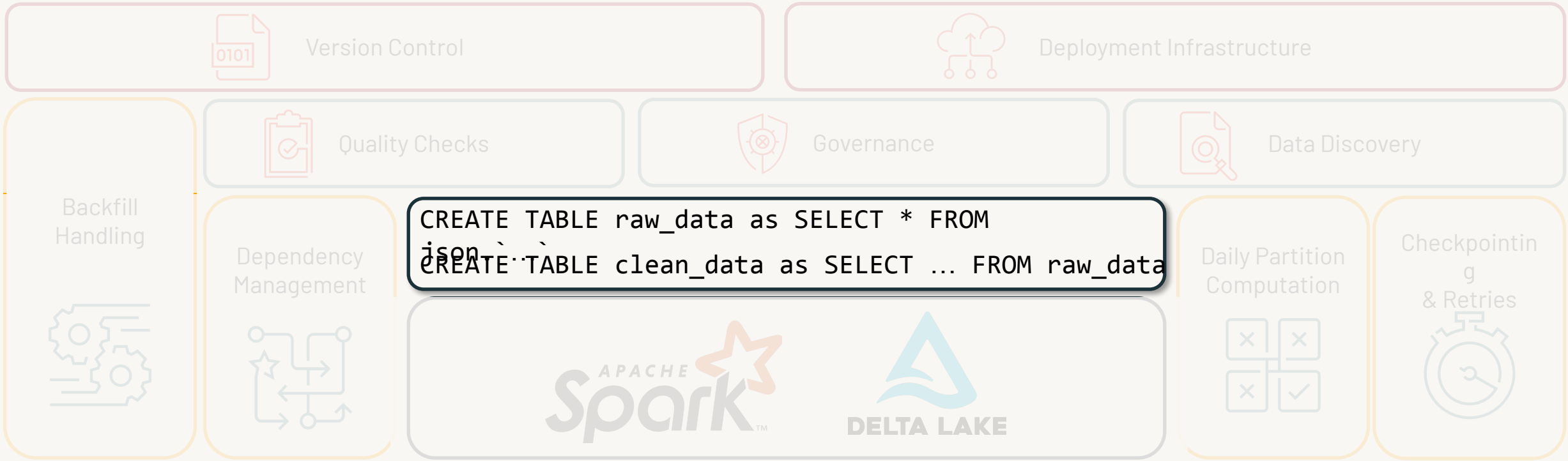
Operational complexity dominates

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

Repos

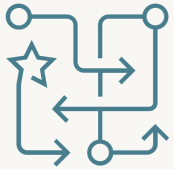
Unity Catalog*

Databricks Workflows

Delta Live Tables

Full Refresh

Dependency Management



Expectations

Incremental Computation*

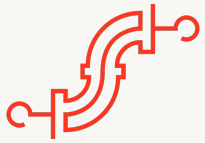
Checkpointing & Retries

* Coming Soon

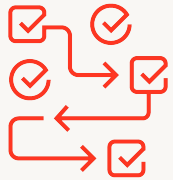
What is Delta Live Tables?

Modern software engineering for ETL processing

Delta Live Tables (DLT) is the first ETL framework that uses a simple, declarative approach to building reliable data pipelines. DLT automatically manages your infrastructure at scale so data analysts and engineers can spend less time on tooling and focus on getting value from data.

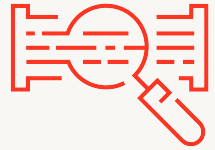


Accelerate ETL Development



Automatically manage your infrastructure

Have confidence in your data



Simplify batch and streaming

https://databricks.com/product/delta-live-tables

What is a LIVE TABLE?

What is a Live Table?

Live Tables are materialized views for the lakehouse.

A live table is:

- Defined by a SQL query
- Created and kept up-to-date by a pipeline

```
CREATE OR REPLACE LIVE TABLE report
AS SELECT sum(profit)
FROM prod.sales
```

Live tables provides tools to:

- Manage dependencies
- Control quality
- Automate operations
- Simplify collaboration
- Save costs
- Reduce latency

What is a Streaming Live Table?

Based on Spark™ Structured Streaming

A streaming live table is "stateful":

- Ensures exactly-once processing of input rows
- Inputs are only read once

```
CREATE STREAMING LIVE TABLE report
AS SELECT sum(profit)
FROM cloud_files(prod.sales)
```

- Streaming Live tables compute results over append-only streams such as Kafka, Kinesis, or Auto Loader (files on cloud storage)
- Streaming live tables allow you to reduce costs and latency by avoiding reprocessing of old data.

How do I use DLT?

Creating Your First Live Table Pipeline

SQL to DLT in three easy steps…

Write create live table

- Table definitions are written (but not run) in notebooks
- Databricks Repos allow you to version control your table definitions.

```
CREATE LIVE TABLE daily_stats
AS SELECT sum(rev) - sum(costs) AS profits
FROM prod_data.transactions
GROUP BY day
```

Create a pipeline

- A Pipeline picks one or more notebooks of table definitions, as well as any configuration required.

Click start

- DLT will create or update all the tables in the pipelines.

BEST PRACTICE

Development vs Production

Fast iteration or enterprise grade reliability

Development Mode

- Reuses a **long-running cluster** running for fast iteration.
- **No retries** on errors enabling faster debugging.

Production Mode

- Cuts costs by **turning off clusters** as soon as they are done (within 5 minutes)
- **Escalating retries**, including cluster restarts, ensure reliability in the face of transient issues.

In the Pipelines UI:

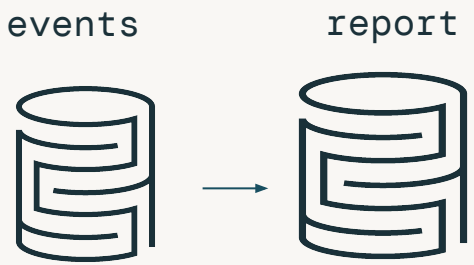
What if I have many tables?

Declare LIVE Dependencies

Using the LIVE virtual schema.

```
CREATE LIVE TABLE events
AS SELECT … FROM prod.raw_data

CREATE LIVE TABLE report
AS SELECT … FROM LIVE.events
```



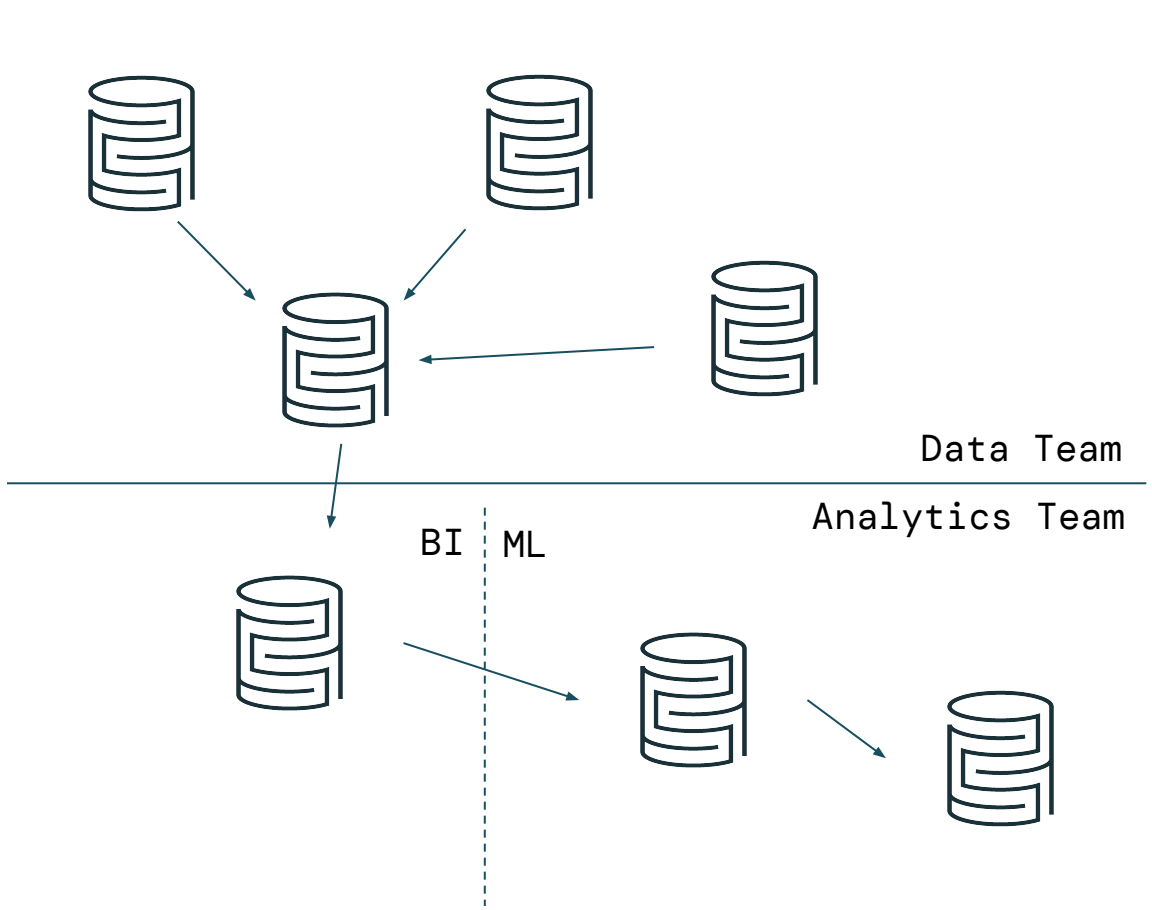
- Dependencies owned by other producers are just read from the catalog or spark data source as normal.
- LIVE dependencies, from the **same pipeline**, are read from the LIVE schema.
- DLT detects LIVE dependencies and executes all operations in **correct order**.
- DLT handles parallelism and captures the lineage of the data.

BEST PRACTICE

Choosing pipeline boundaries

Break up pipelines at natural external divisions.

- Larger pipelines can utilize cluster resources more efficiently reducing TCO.
- Use more than one pipeline:
  - At team boundaries
  - At helpful application-specific boundaries.
- Tables do not necessarily need to have inter-dependencies
- Advantages of co-locating Tables in a single pipeline:
  - Reduces overall Pipeline Count
  - Reduce Orchestration Complexity
  - Provides observability & lineage



BEST PRACTICE

Use autoscaling to reduce costs

Autoscaling lets you set a budget and leave the tuning to us

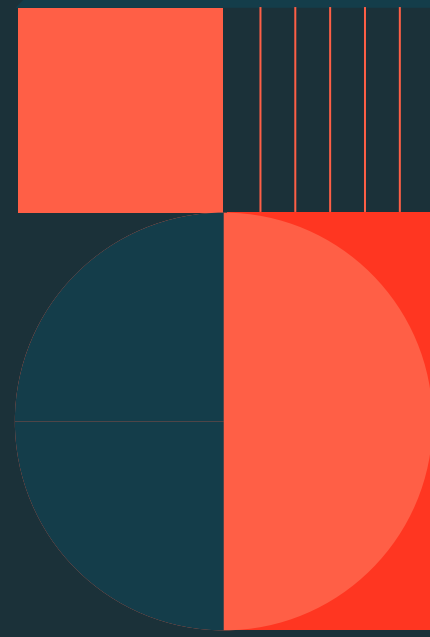
How to think about setting the various configurations in production:

- Min Nodes
  - Leave default
- Max Nodes
  - Maximum you would pay for a timely answer
- Instance Type
  - unset – leaves more freedom for DLT to pick the right instance type*

This strategy leverages cloud elasticity:

- Autoscaling will only add more resources if it thinks they will speed things up.
- Resources are freed as soon as they are no longer useful. Clusters are shut down as soon as all updates are complete.
- A larger maximum size makes you robust to changes in input volumes to avoid missing SLAs.

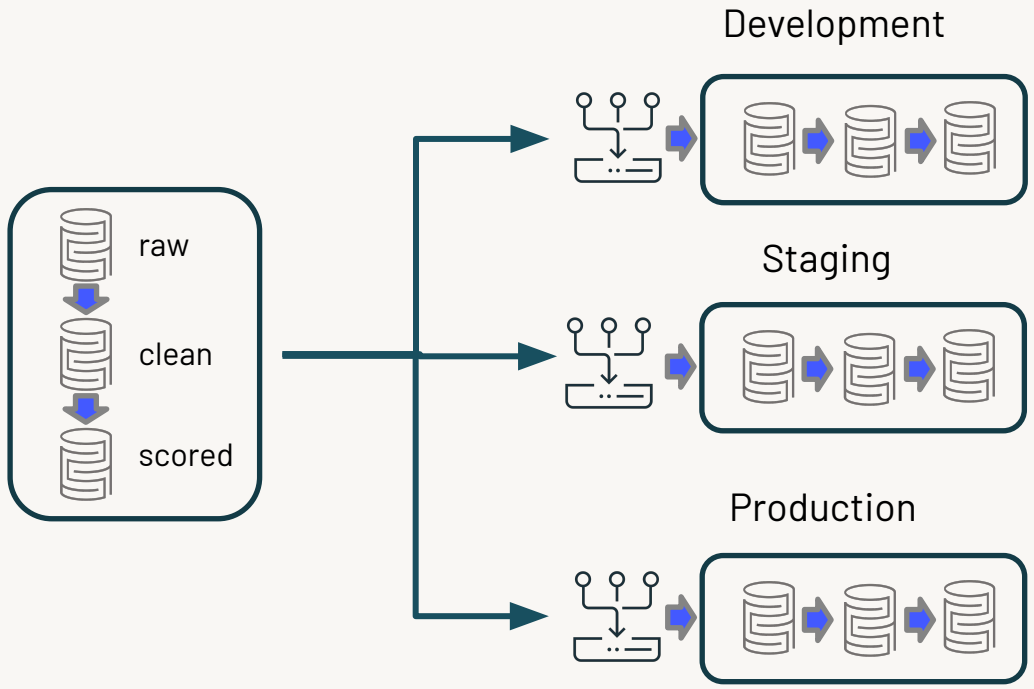
How can I develop safely with my coworkers?



Continuous Integration and Deployment (CICD) requires testing

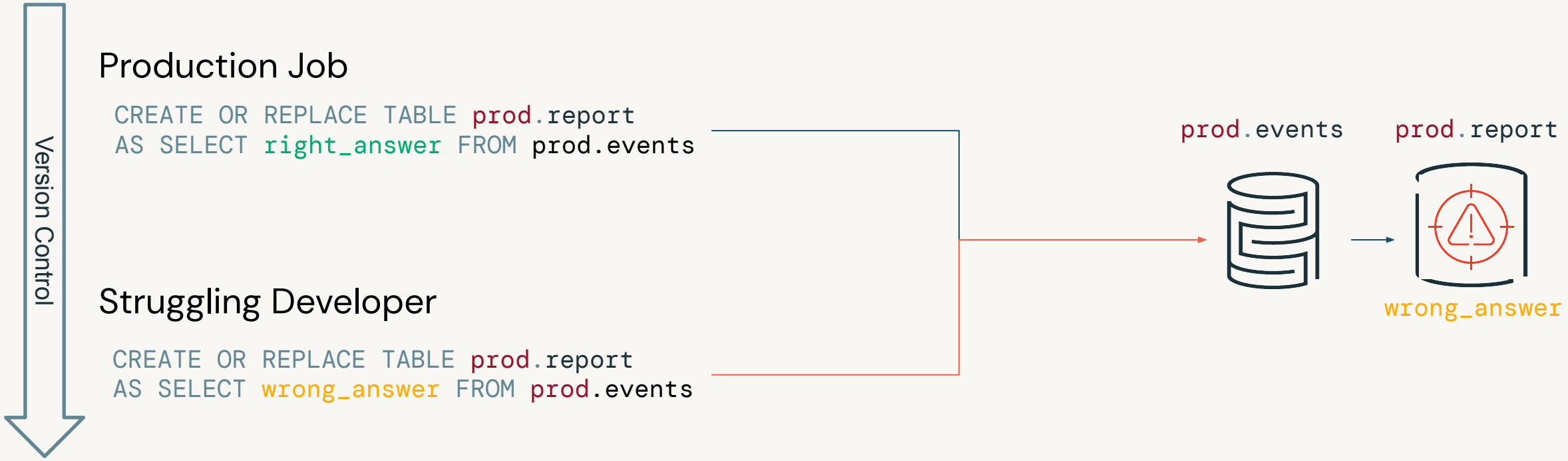
DLT allows the same version of your code to read and write to different, isolated environments

- DLT accomplishes this with **targets** and the **LIVE** schema.
- A **target** is a pipeline-level parameter that defines a schema to publish table(s) to
- The **LIVE** schema is a reserved keyword that locates your dependencies in the **target**.



Pitfall: hard-code sources & destinations

Problem: Hard coding the source & destination makes it impossible to test changes outside of production, breaking CI/CD



BEST PRACTICE

Best Practice: Keep environments isolated

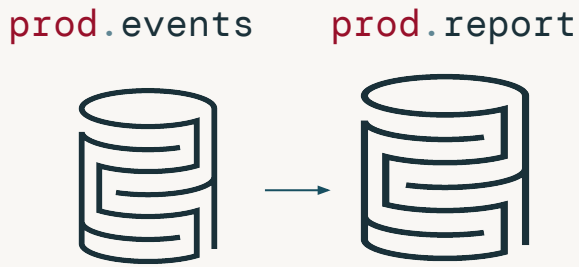
DLT automatically locates data using the pipeline's "target"

Production Job

```
CREATE LIVE TABLE report
AS SELECT right_answer FROM LIVE.events
```

Production Pipeline

```
{"target": "prod"}
```

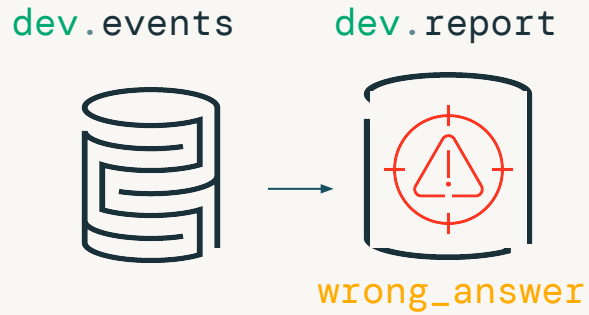


Struggling Developer

```
CREATE LIVE TABLE report
AS SELECT wrong_answer FROM LIVE.events
```

Development Pipeline

```
{"target": "dev"}
```



BEST PRACTICE

Best Practice: Use version control

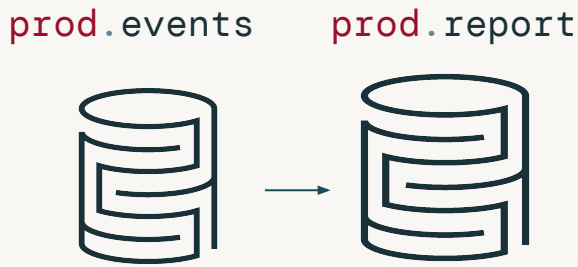
One code base that is promoted to production after validation

Production Job

```
CREATE LIVE TABLE report
AS SELECT right_answer FROM LIVE.events
```

Production Pipeline

```
{"target": "prod"}
```

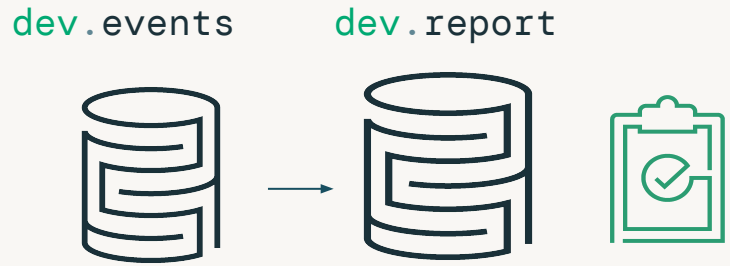


Successful Developer

```
CREATE LIVE TABLE report
AS SELECT better_answer FROM LIVE.events
```

Development Pipeline

```
{"target": "dev"}
```



BEST PRACTICE

Best Practice: Use version control

One code base that is promoted to production after validation

Version Control

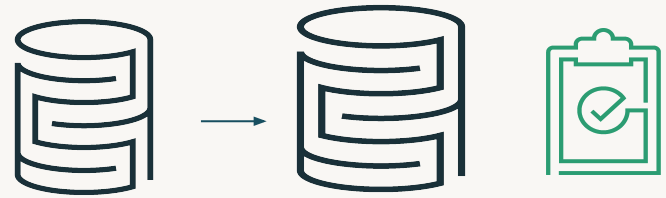
Production Job

```
CREATE LIVE TABLE report
AS SELECT better_answer FROM LIVE.events
```

Production Pipeline

```
{"target": "prod"}
```

prod.events → prod.report



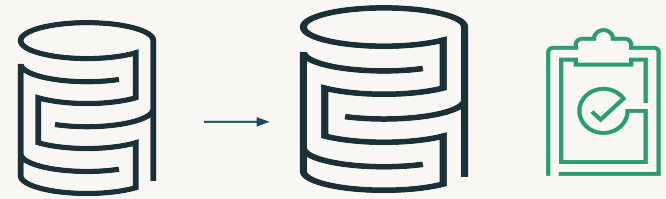
Successful Developer

```
CREATE LIVE TABLE report
AS SELECT better_answer FROM LIVE.events
```

Development Pipeline

```
{"target": "dev"}
```

dev.events → dev.report



How do I know my results are correct?

BEST PRACTICE

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

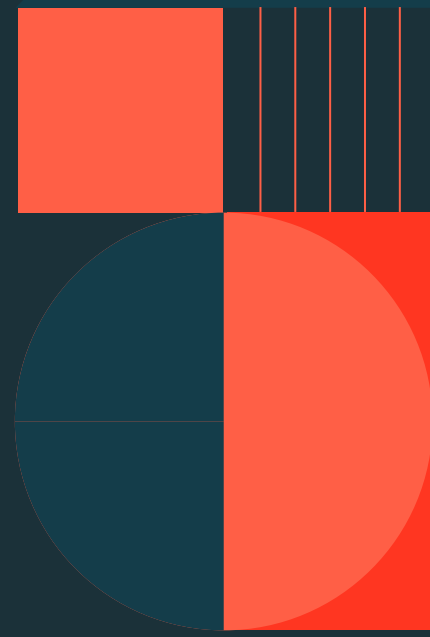
```
-- Make sure a primary key is always unique.
CREATE LIVE TABLE report_pk_tests(
  CONSTRAINT unique_pk EXPECT (num_entries = 1)
)
AS SELECT pk, count(*) as num_entries
FROM LIVE.report
GROUP BY pk
```

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 LIVE TABLE 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
```

What if I need more than SQL?



Using Python

Write advanced DataFrame code and UDFs

```
import dlt

@dlt.table
def report():
  df = spark.table("LIVE.events")
  return df.select(…)
```

```
spark.udf.register(
  "complex_function",
   lambda x: …)
```

- Add @dlt.table to any function that returns a DataFrame.
- Mix and match SQL/Python notebooks in a single pipeline.
- But a single notebook must be all Python or all SQL (cannot mix/match at notebook level)
- DataFrames can be constructed using PySpark, Koalas, or SQL strings.

Installing libraries with `pip`

`pip` is a package installer for python

- Access a wide range of python libraries using the `%pip` magic command.
- Python libraries installed in one notebook are available to all notebooks in the pipeline.
- Order of notebooks & dependencies does not matter.

- Compile external python files as `wheels` and %pip install:

```
%pip install
/path/to/my_package.whl
```

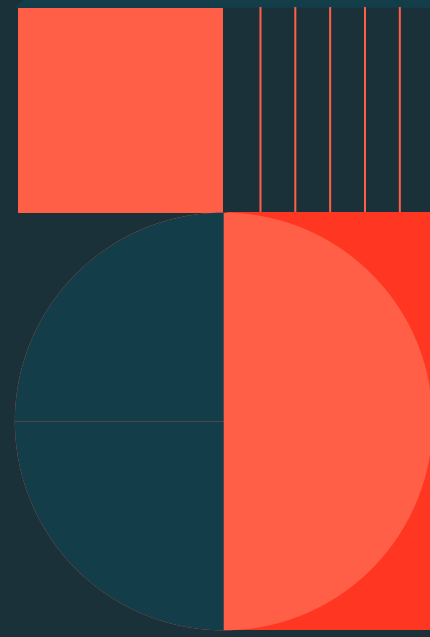
Metaprogramming in python

Create dynamic pipelines by programmatically creating tables

- Automate ingestion of many similar inputs
- Integrate with external metadata system like schema registries or other catalogs

```
for t in tables:
  @dlt.table(name=t)
  def report():
    df = spark.table(t)
    return df.select(…)
```

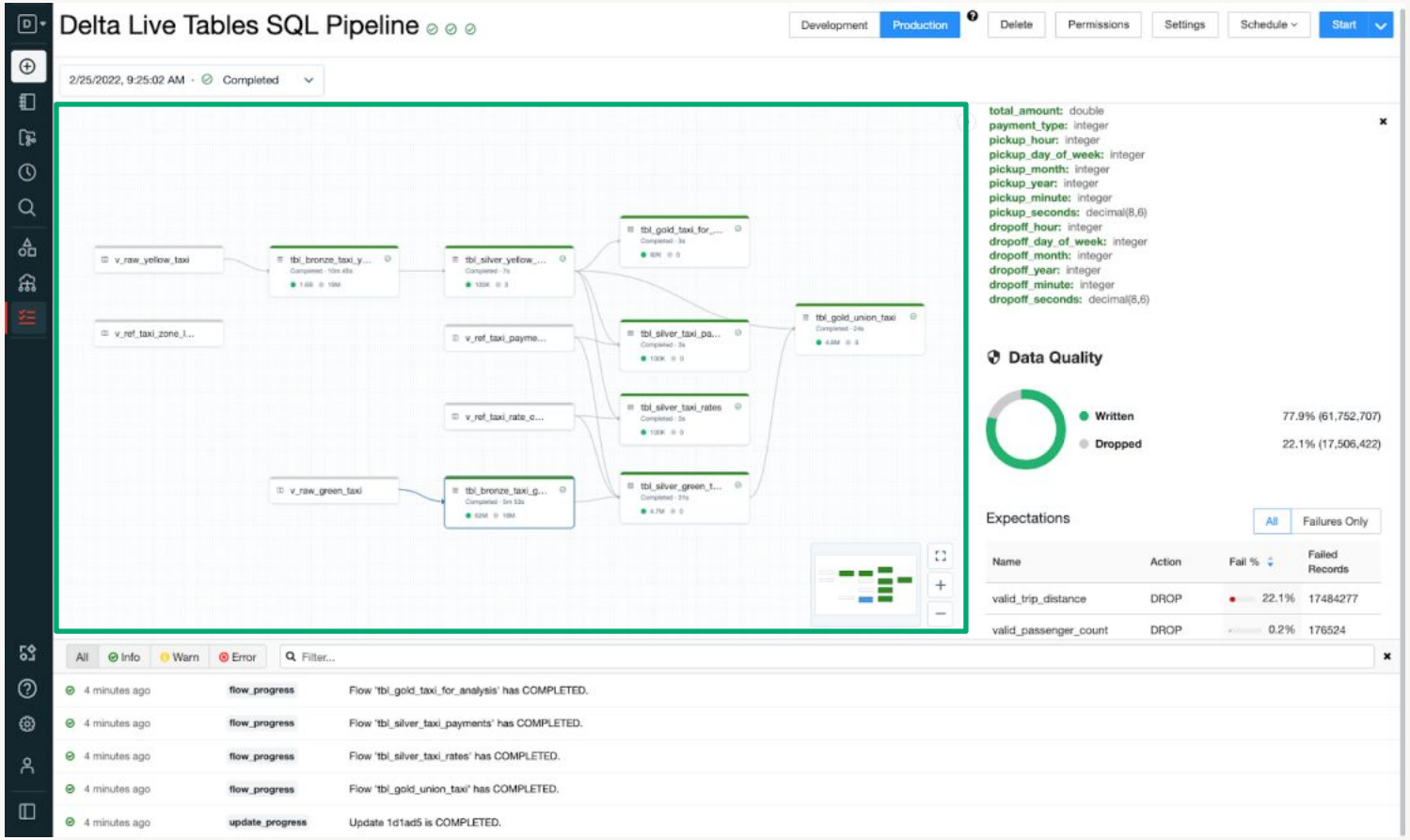
What about operations?



Pipelines UI

A one stop shop for ETL debugging and operations

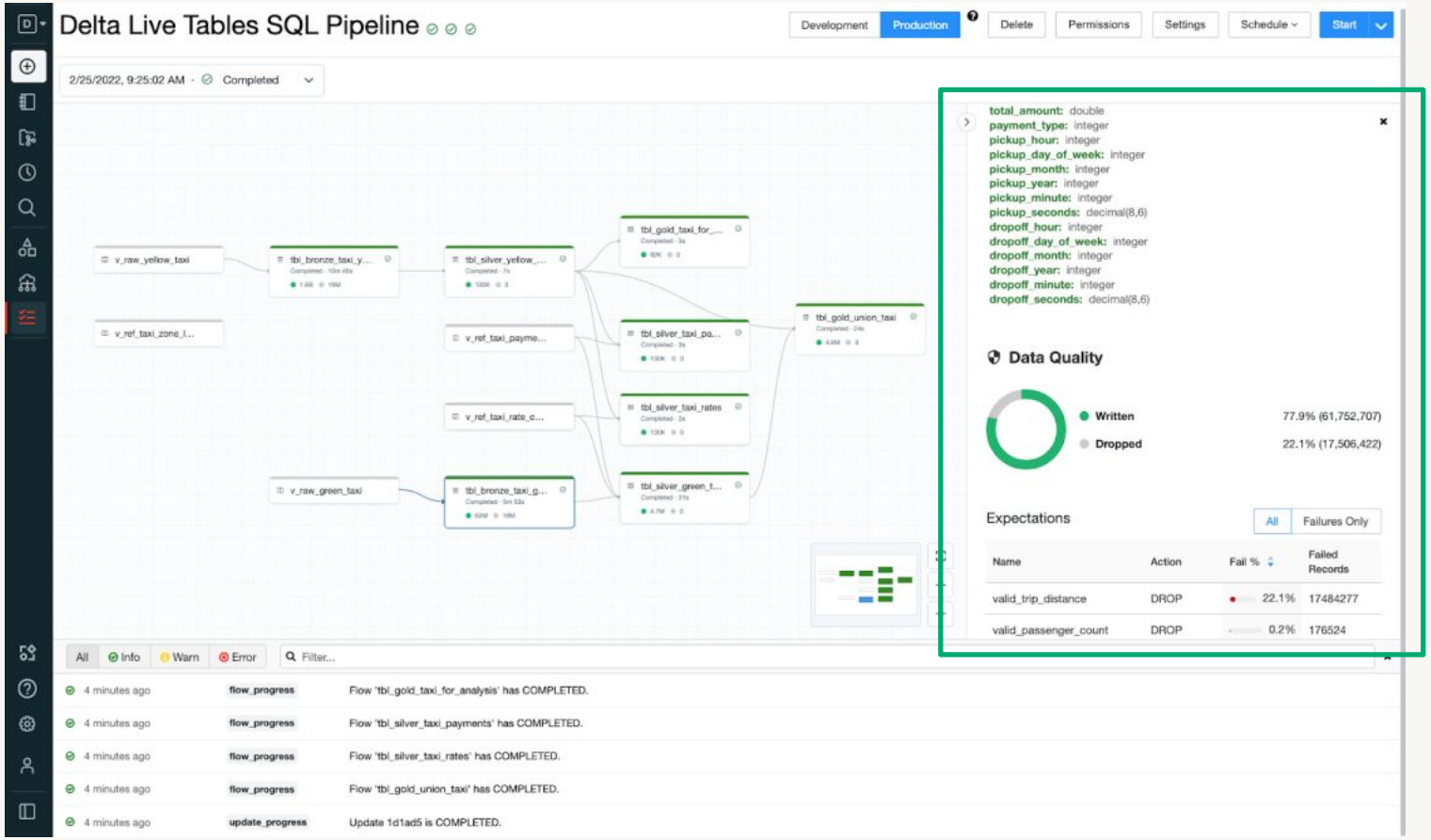
- Visualize data flows between tables



Pipelines UI

A one stop shop for ETL debugging and operations

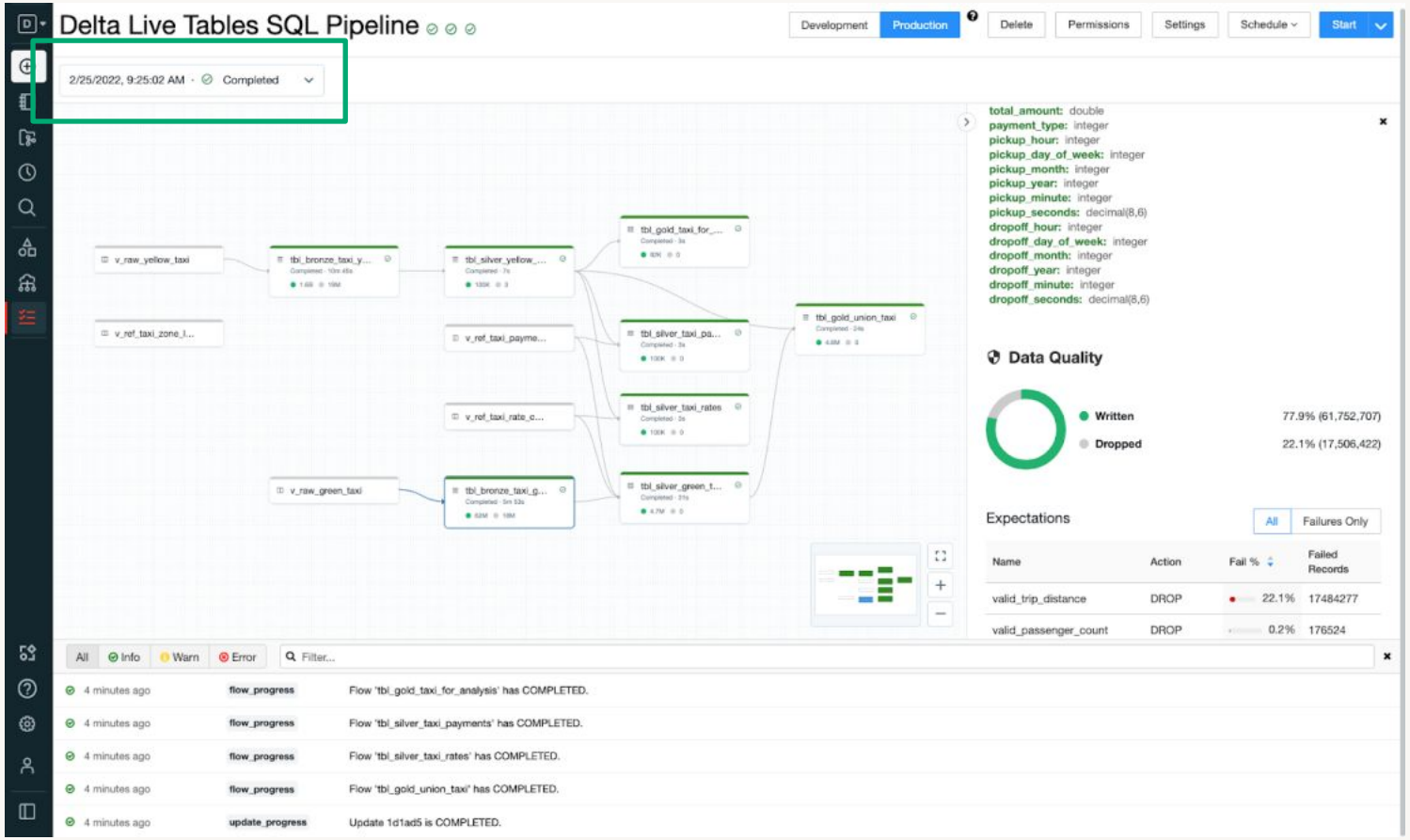
- Visualize data flows between tables
- Discover metadata and quality of each table



Pipelines UI

A one stop shop for ETL debugging and operations

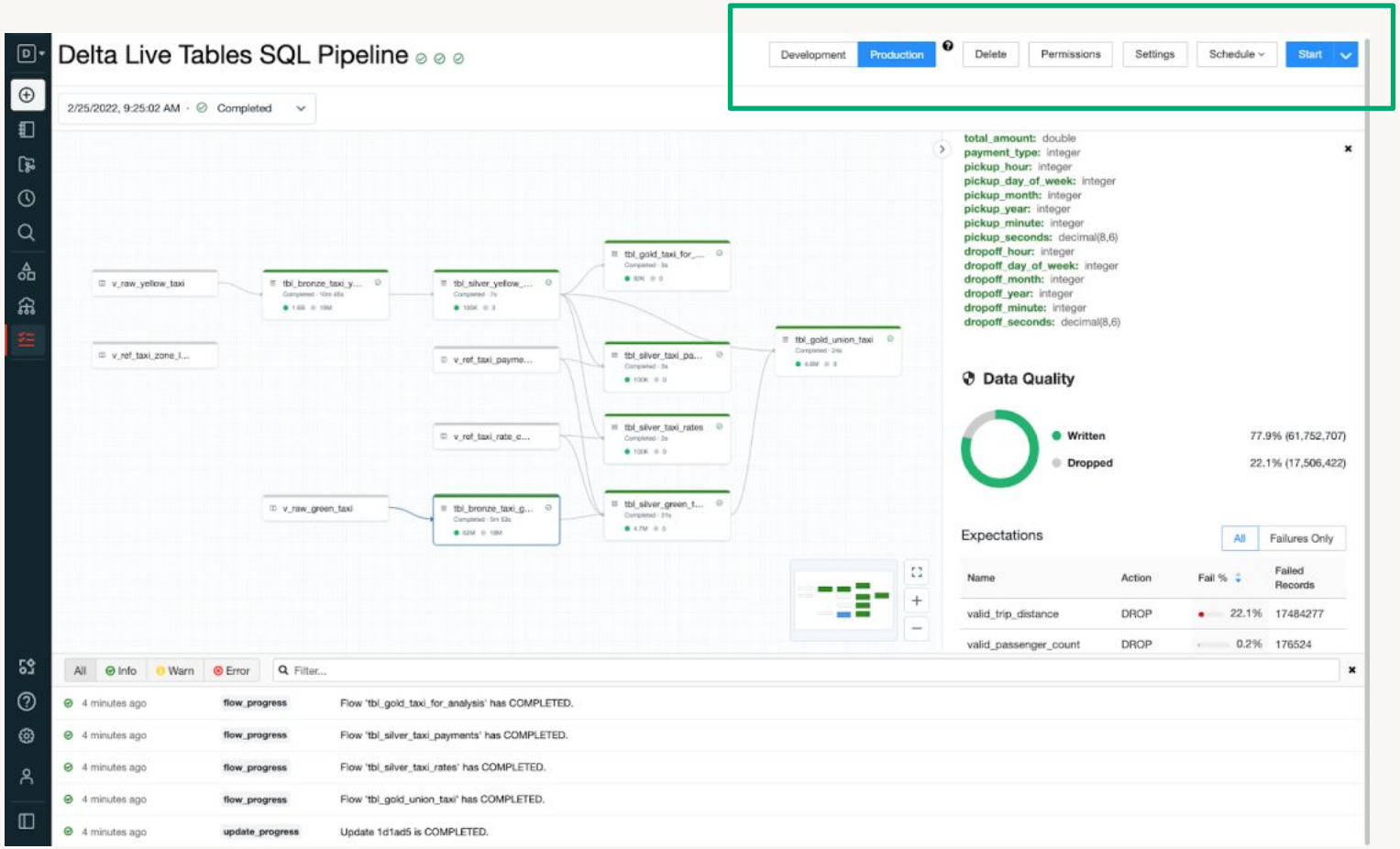
- Visualize data flows between tables
- Discover metadata and quality of each table
- Access to historical updates



Pipelines UI

A one stop shop for ETL debugging and operations

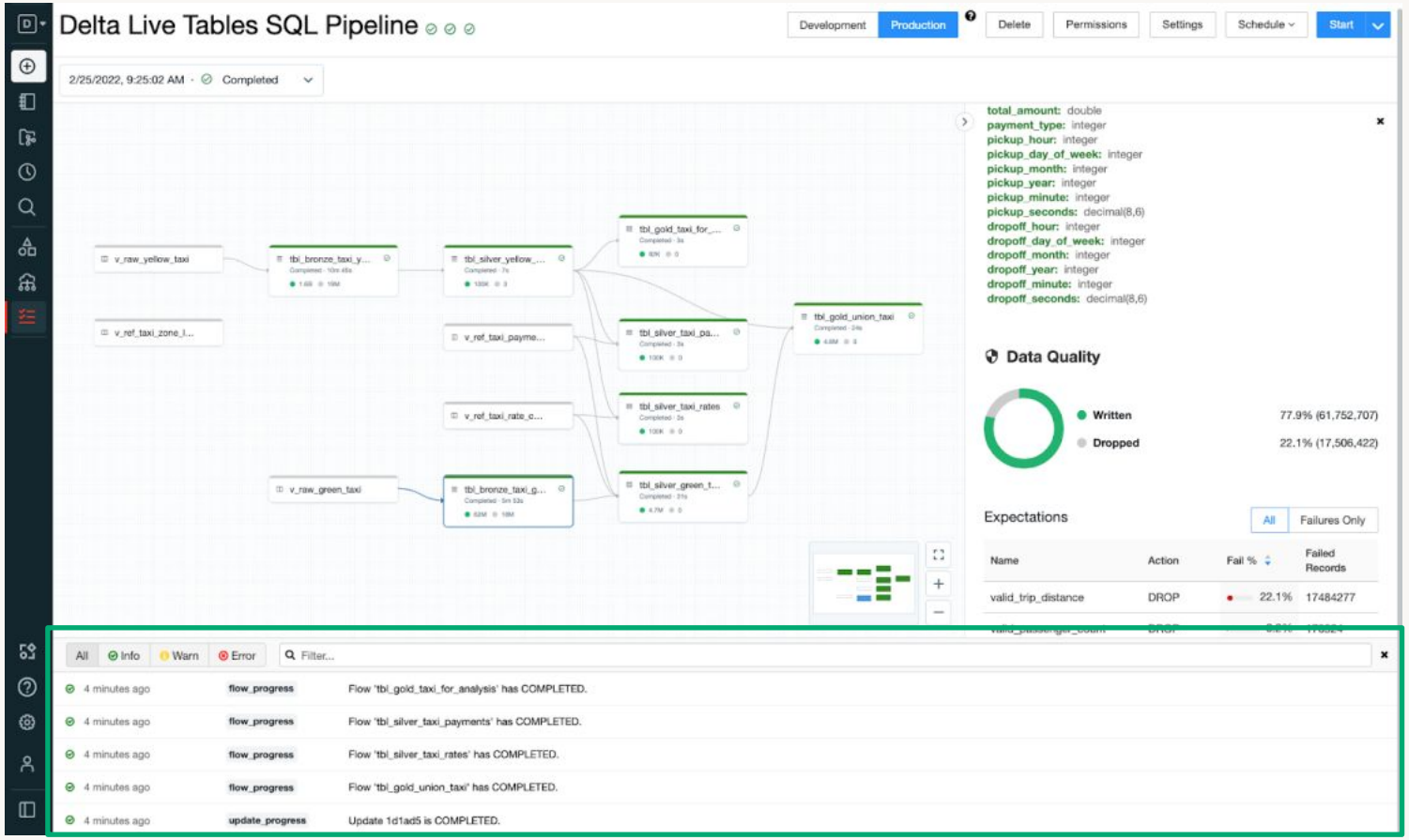
- Visualize data flows between tables
- Discover metadata and quality of each table
- Access to historical updates
- Control operations



Pipelines UI

A one stop shop for ETL debugging and operations

- Visualize data flows between tables
- Discover metadata and quality of each table
- Access to historical updates
- Control operations
- Dive deep into events



The Event Log

The event log automatically records all pipelines operations.

Operational Statistics

Time and current status, for all operations

Pipeline and cluster configurations

Row counts

Provenance

Table schemas, definitions, and declared properties

Table-level lineage

Query plans used to update tables

Data Quality

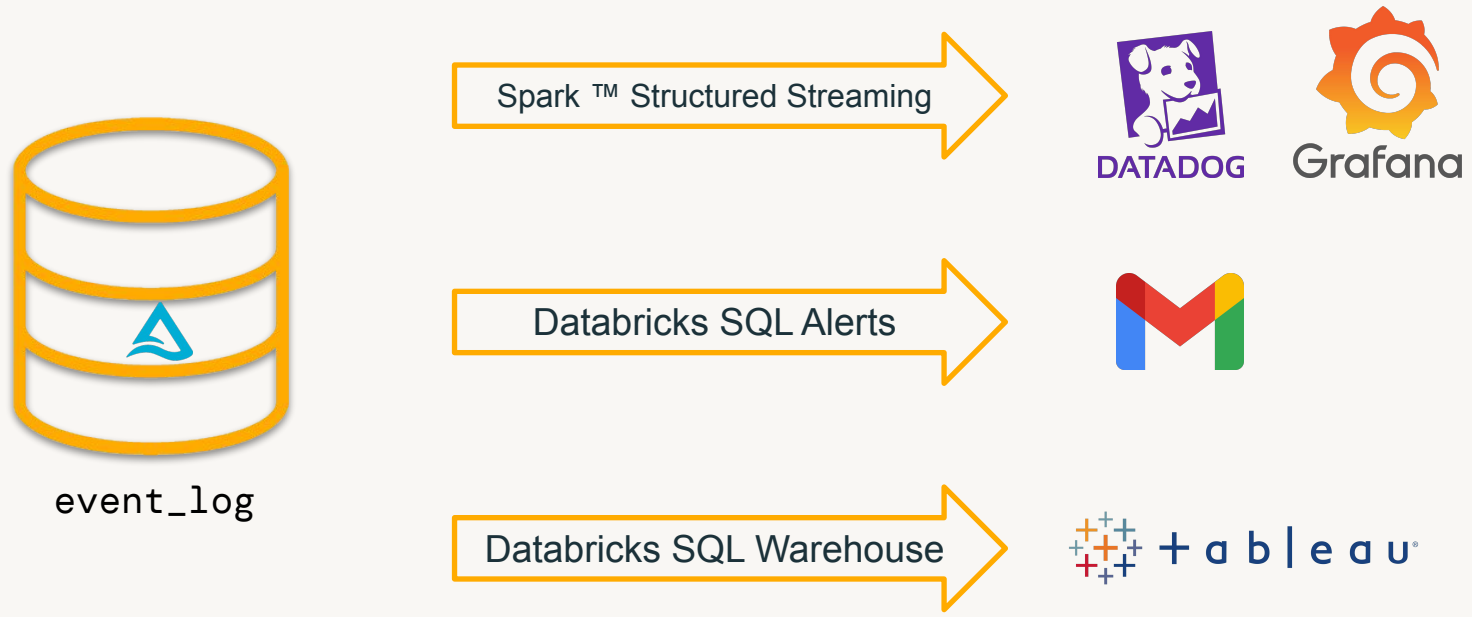
Expectation pass / failure / drop statistics

Input/Output rows that caused expectation failures

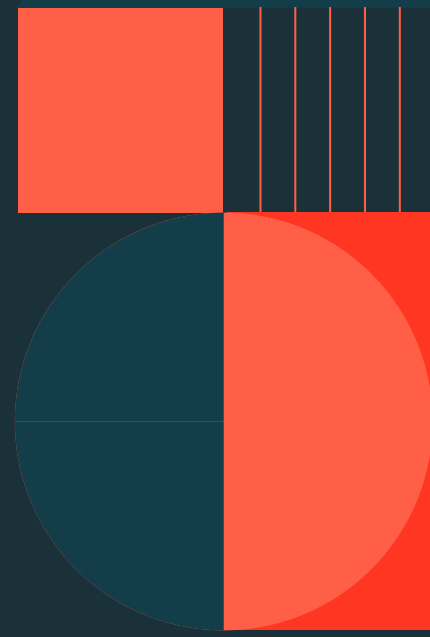
Best Practice: Integrate using the event log

Use the information in the event log with your existing operational tools.

The event log is just a delta table created for each pipeline.



How does DLT handle failures?



DLT Automates Failure Recovery

Transient issues are handled by built-in retry logic

- DLT uses escalating retries to balance speed with reliability
  - Retry the individual transaction
  - Next, restart the cluster in case its in a bad state
  - If DBR was upgraded since the last success, automatically try the old version to detect regressions.
- Transactionally
  - Operations on tables are atomic.
  - Updates to different tables in a pipeline are not atomic. Best effort to update as many as possible.

When should I use streaming?

What is Spark™ Structured Streaming?

Basis for Streaming Live Tables. Runs queries on continually arriving data.

Computation Model: Input is an ever-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

Using Spark™ Structured Streaming for ingestion

Easily ingest files from cloud storage as they are uploaded

```
CREATE STREAMING LIVE 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

```
import dlt


@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 LIVE TABLE mystream
  AS SELECT *
  FROM STREAM(my_table)
```

Pitfall: `my_table` must be an append-only source.

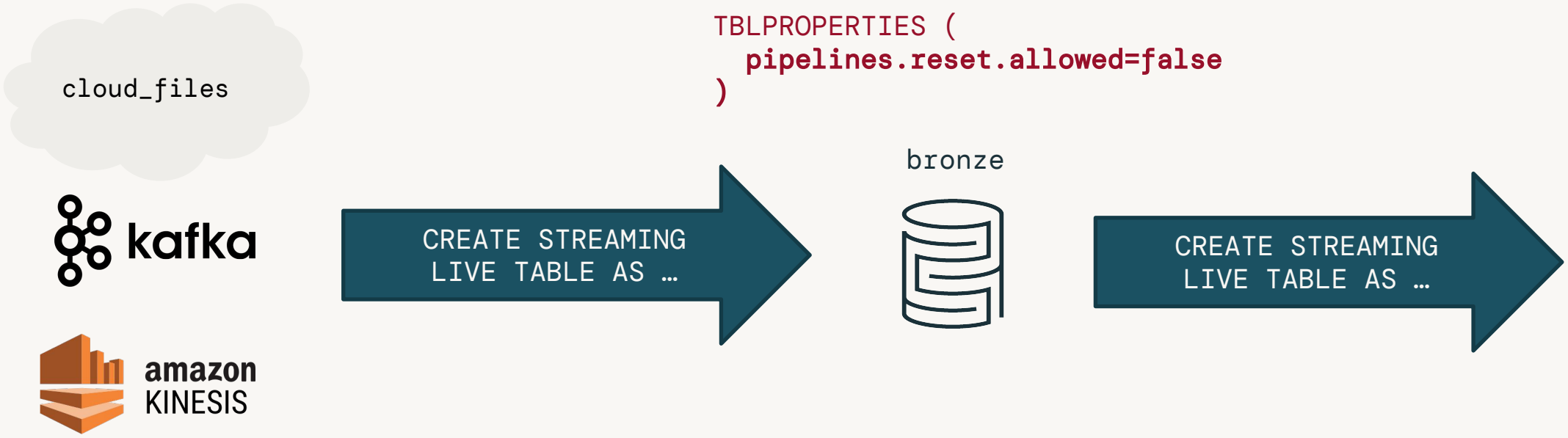
e.g. it may not:

- be the target of APPLY CHANGES INTO
- define an aggregate function
- be a table on which you've executed DML to delete/update a row (see GDPR section)

- `STREAM(my_table)` reads a stream of new records, instead of a snapshot
- `Streaming` tables must be an append-only table
- Any append-only delta table can be read as a stream (i.e. from the live schema, from the catalog, or just from a path).

Use Delta for infinite retention

Delta provides cheap, elastic and governable storage for transient sources



Use a short retention period to avoid compliance risks and reduce costs

Avoid complex transformations that could have bugs or drop important data

Retain history

Easy to perform GDPR and other compliance tasks

Setting `pipelines.reset.allowed=false` ensures that downstream computation can be full-refreshed without losing data

Streaming does not always mean expensive

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

Triggered: Manually

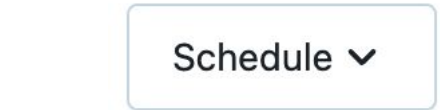
Costs: lowest

Latency: highest

Triggered: On a schedule using Databricks Jobs

Costs: depends on frequency

Latency: 10 minutes to months

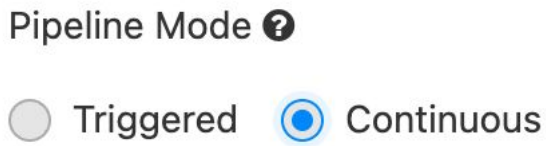


Continually

Costs: highest

Latency: minutes to seconds

(for some workloads)



When should I chain streaming?

Multi-hop streaming for cost and latency

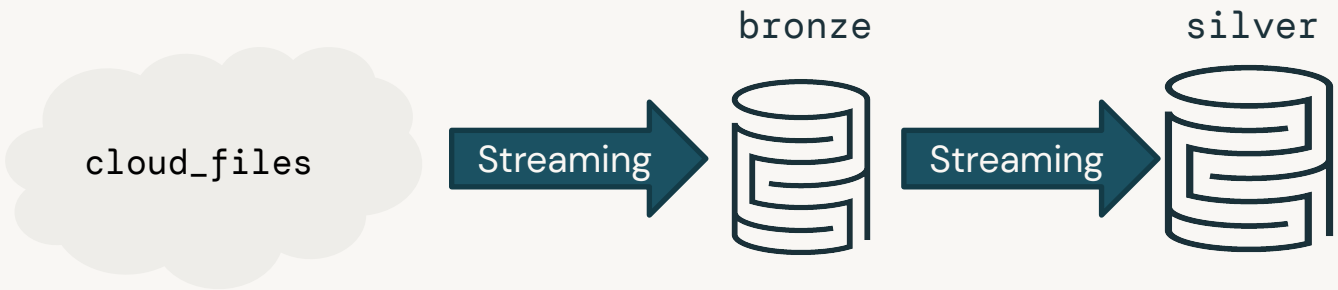
Append-only live tables can be both a source and a sink

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


CREATE STREAMING LIVE TABLE silver
AS SELECT
  CAST(ts AS TIMESTAMP) AS timestamp
  …
FROM STREAM(LIVE.bronze)
```

Chain multiple streaming jobs for workloads with:

- Very large data
- Very low latency targets



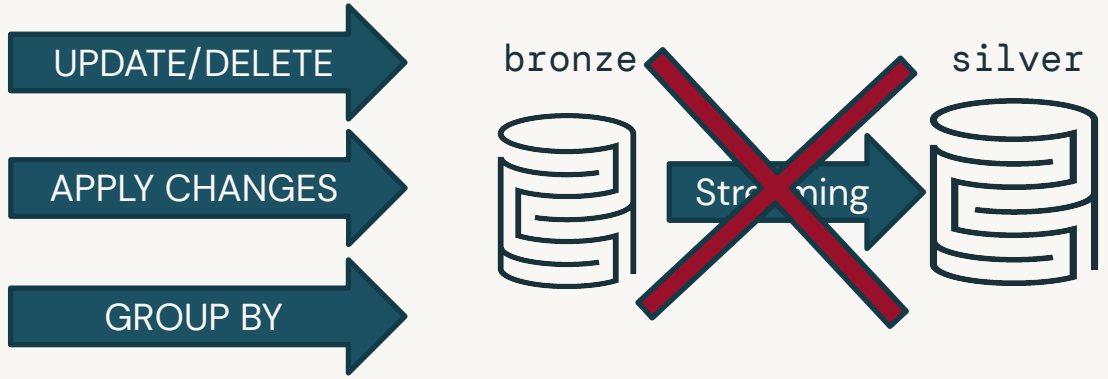
Pitfall: Updates in streaming pipelines

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



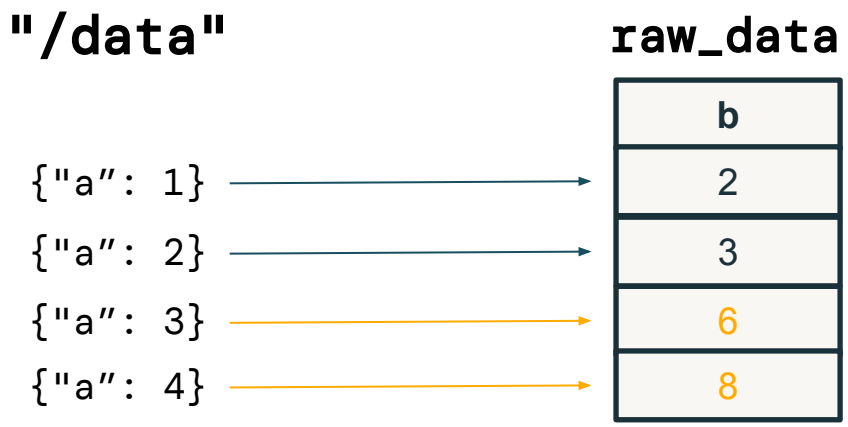
How do I manage streaming state?

Streaming queries are stateful

Each input row is processed only once.

A change to a streaming live table's definition does not recompute results by default:

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

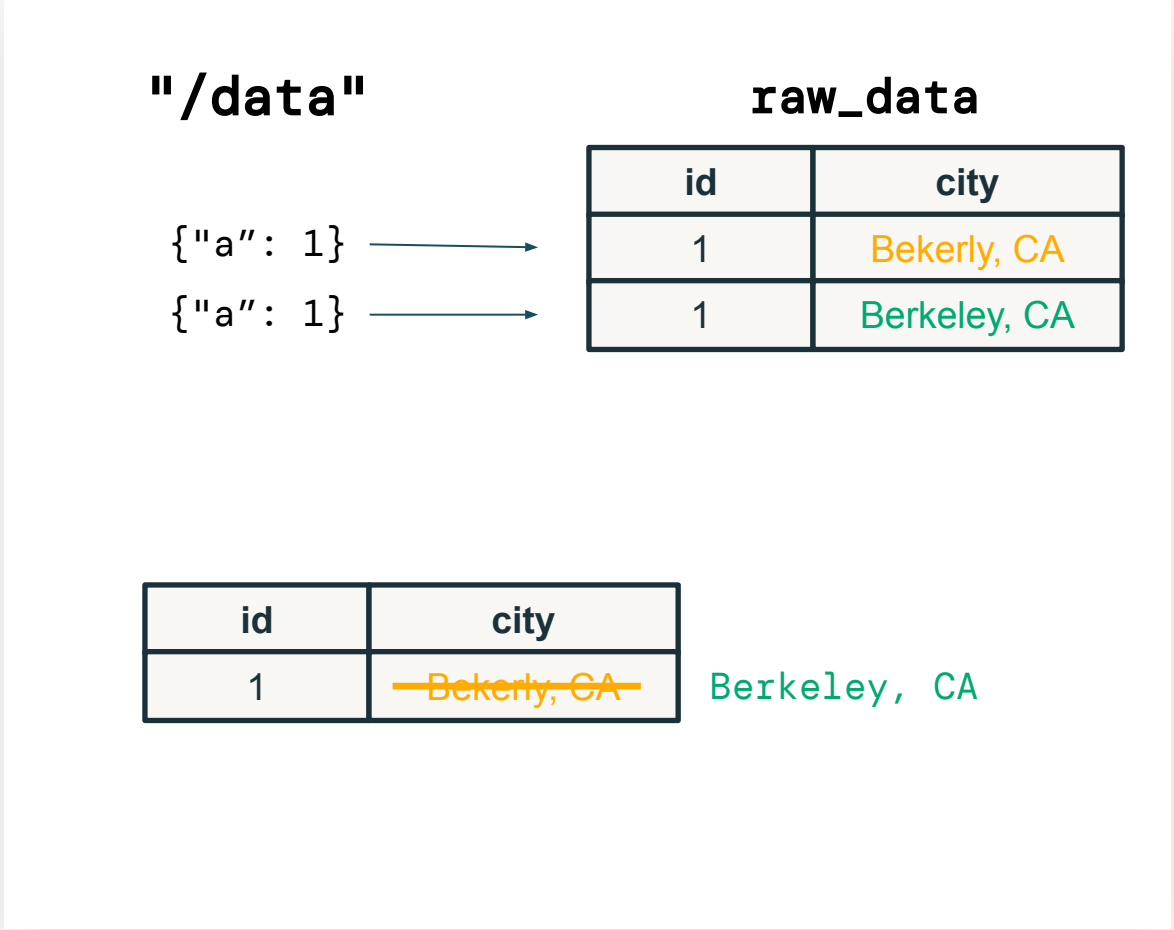


Streaming joins are stateful

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

A change to joined table snapshot does not recompute results by default:

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

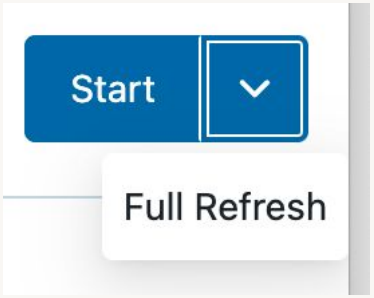


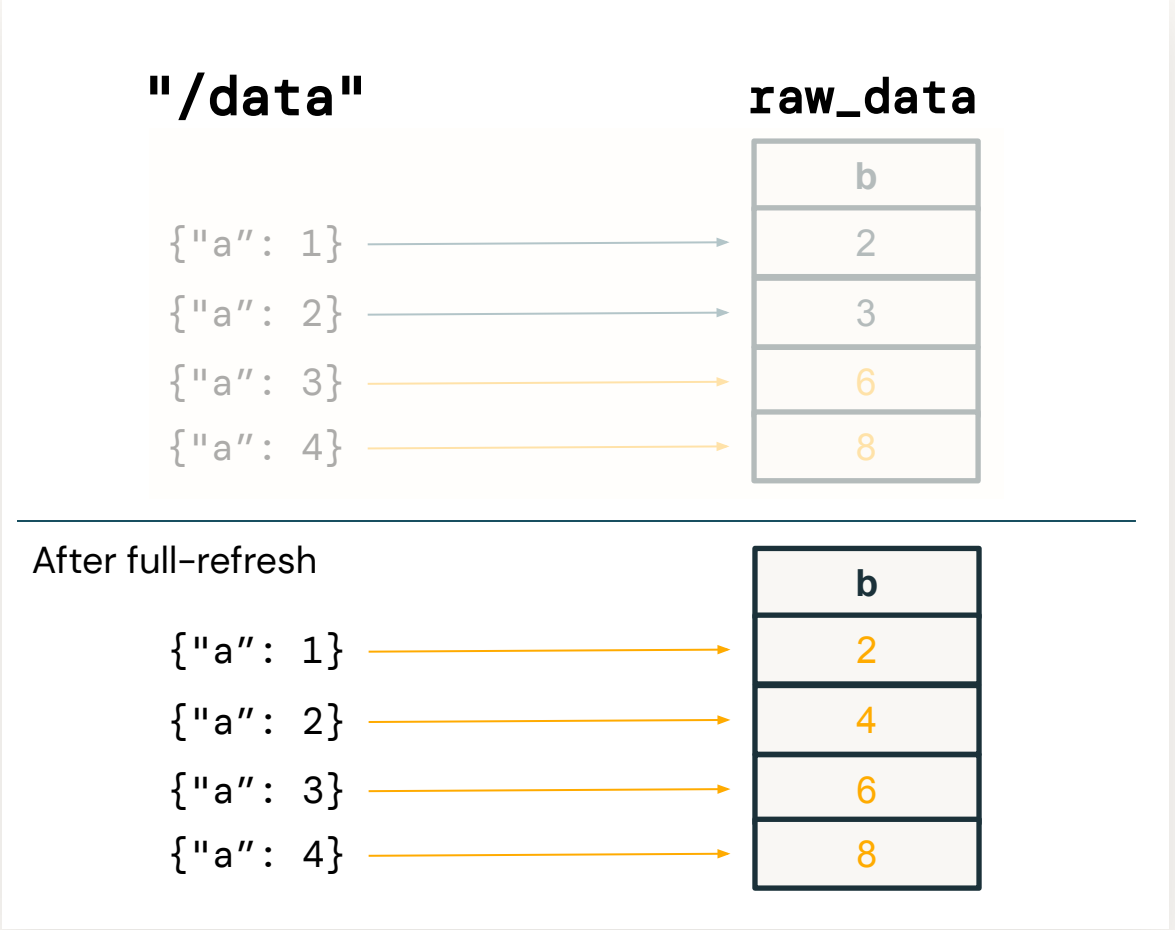
Clear state using full refresh

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





What about incrementalization?

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

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

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

How do data engineers incrementalize different types of queries?

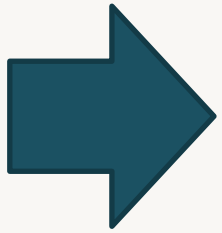
Appending new data as it arrives

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

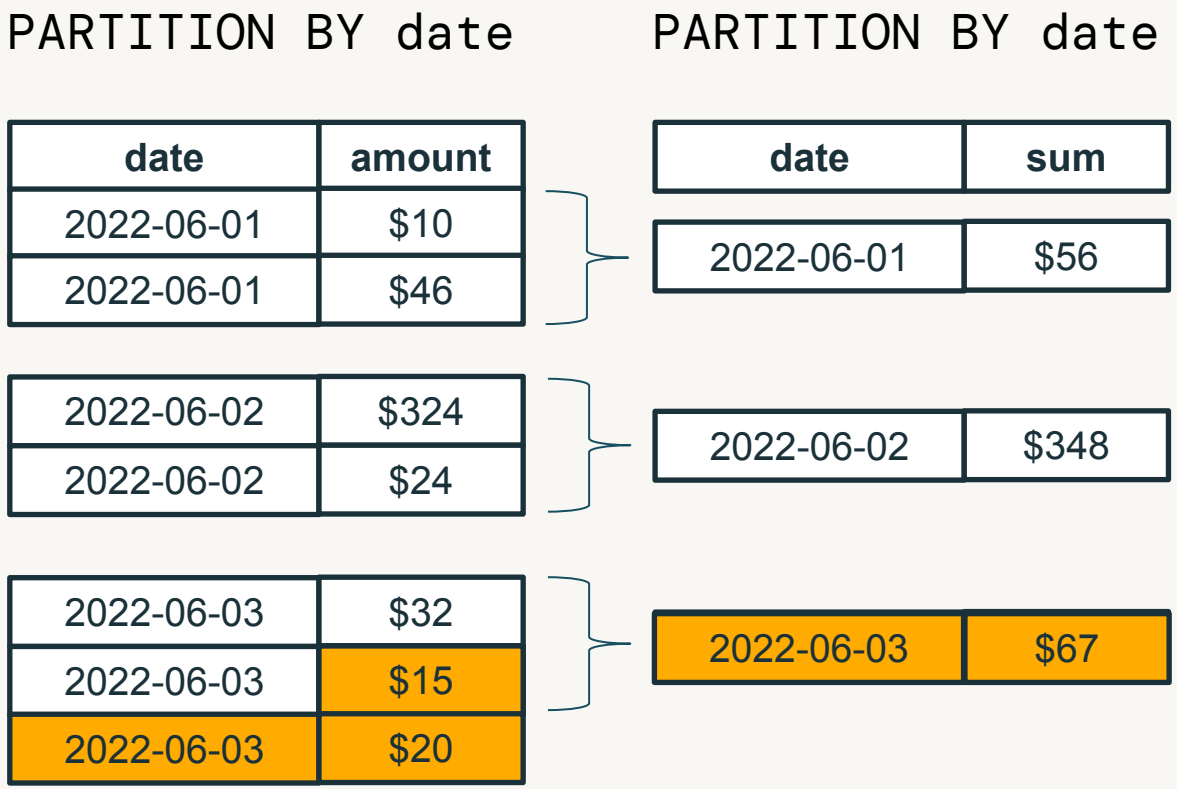
date	city_id
2022-06-01	1
2022-06-01	2
2022-06-02	3



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

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

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

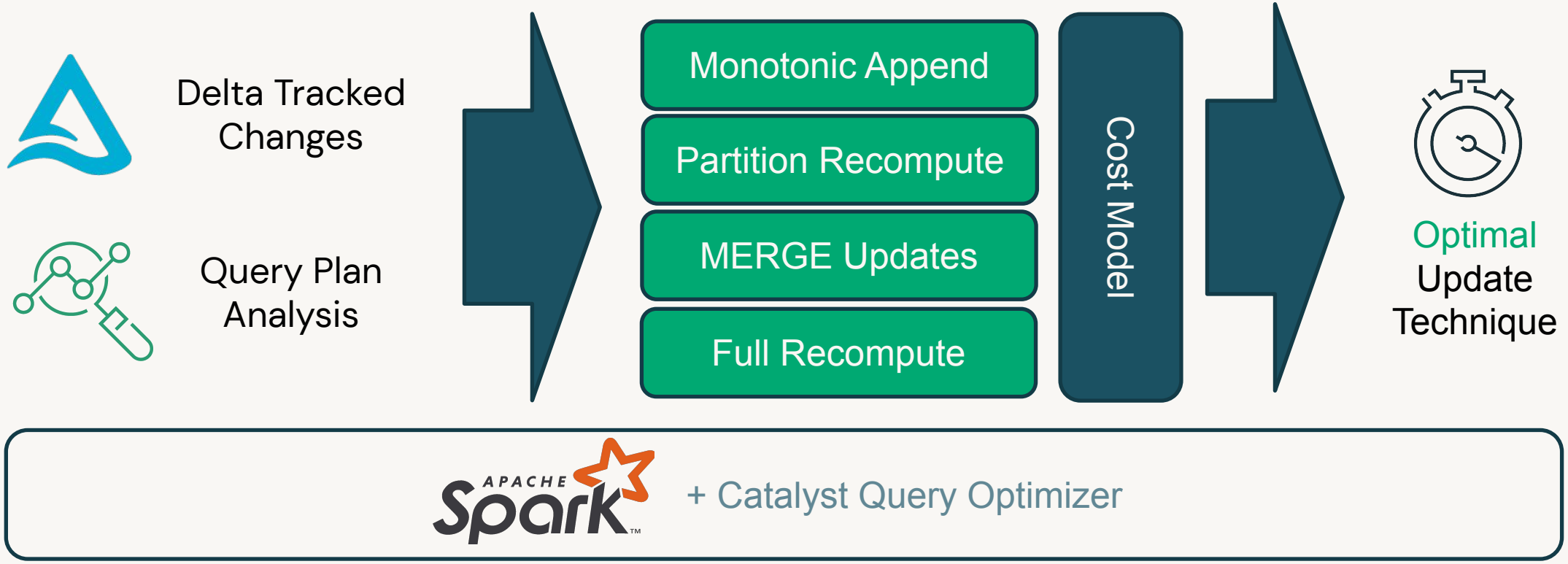
date	amount
2022-06-01	$10
2022-06-01	$46
2022-06-02	$324
2022-06-02	$24
2022-06-03	$32
2022-06-03	$15
2022-06-03	$20

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

How do you pick the right technique?

Introducing Enzyme

An automatic optimizer for incremental ETL

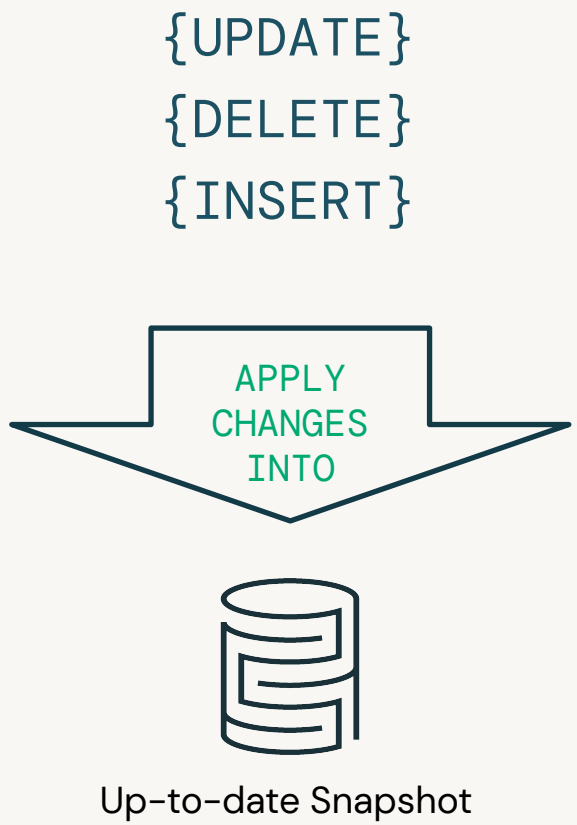


How can I do change data capture (CDC)?

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
```



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
```

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

city_updates

```
{"id": 1, "ts": 1, "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(LIVE.city_updates)
KEYS (id)
SEQUENCE BY ts
```

A target for the changes to be applied to.

city_updates

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

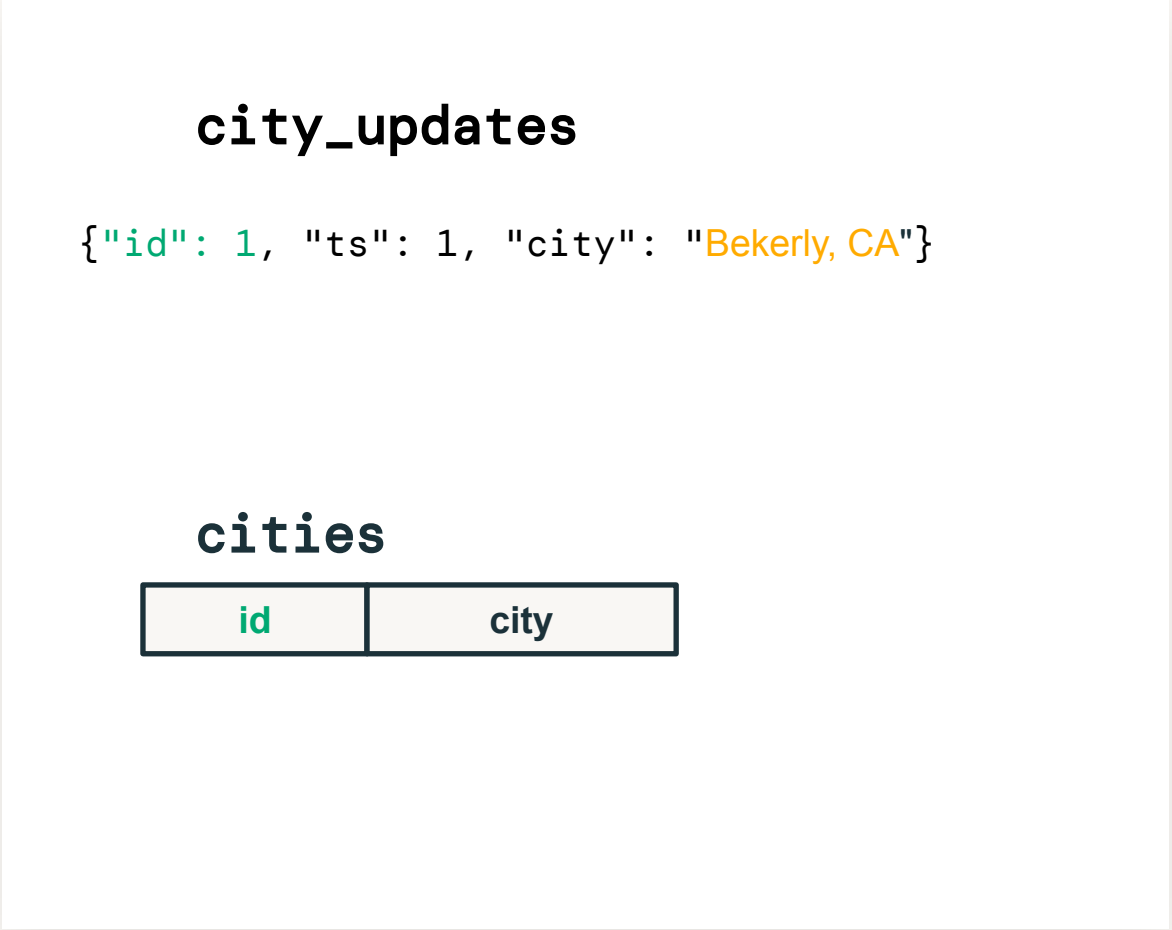
cities

APPLY CHANGES INTO for CDC

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

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

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



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
```

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

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
```

city_updates

```
{"id": 1, "ts": 100, "city": "Bekerly, CA"}
{"id": 1, "ts": 200, "city": "Berkeley, CA"}
```

cities

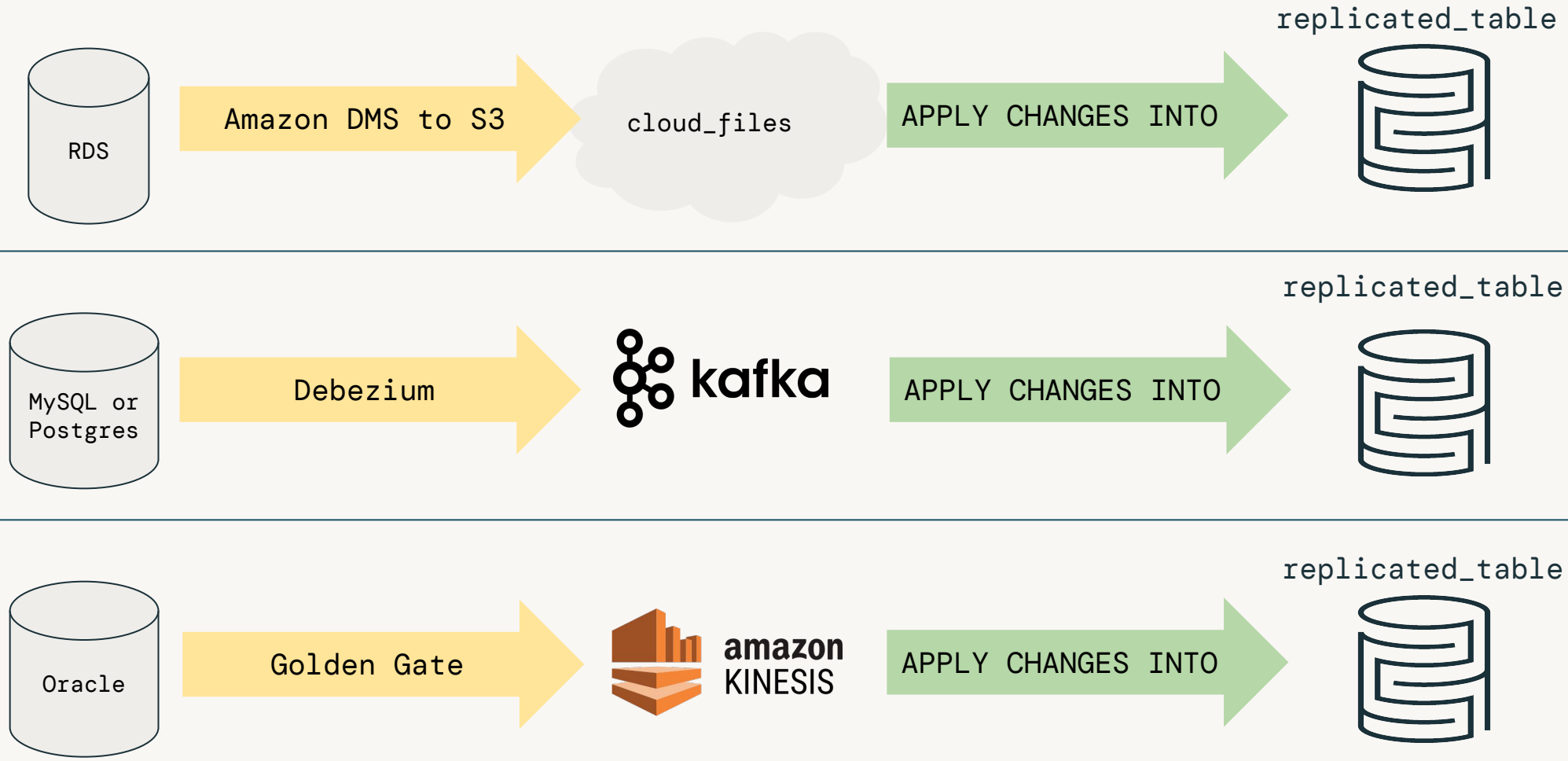
id	city
1	~~Bekerly, CA~~

Berkeley, CA

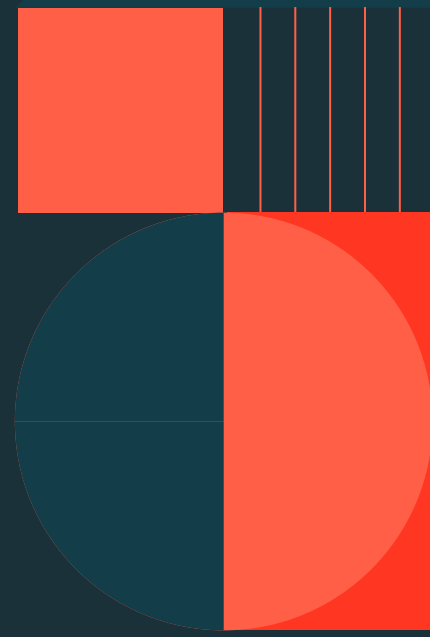
REFERENCE ARCHITECTURE

Change Data Capture (CDC) from RDBMS

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



What do I *no longer* need to manage with DLT?



Automated Data Management

DLT automatically optimizes data for performance & ease-of-use

Best Practices

What:

DLT encodes Delta best practices automatically when creating DLT tables.

How:

DLT sets the following properties:

- `optimizeWrite`
- `autoCompact`
- `tuneFileSizesForRewrites`

Physical Data

What:

DLT automatically manages your physical data to minimize cost and optimize performance.

How:

- runs vacuum daily
- runs optimize daily

You still can tell us how you want it organized (ie ZORDER)

Schema Evolution

What:

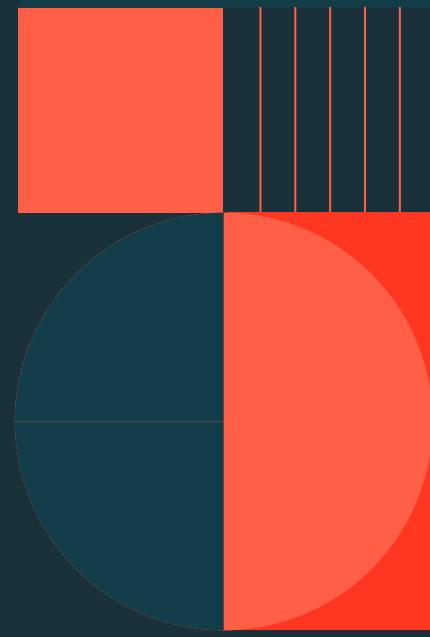
Schema evolution is handled for you

How:

Modifying a live table transformation to add/remove/rename a column will automatically do the right thing.

When removing a column in a streaming live table, old values are preserved.

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 take on 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

Can I perform DML on a Live Table? (i.e. GDPR)

How can you fix it?

Updates, deletes, inserts and merges on streaming live tables.

Ensure compliance for retention periods on a table.

```
DELETE FROM my_live_tables.records
WHERE date < current_time() - INTERVAL 3 years
```

Scrub PII from data in the lake.

```
UPDATE my_live_tables.records
SET email = hash(email, salt)
```

DML works on streaming live tables only

DML on live tables is undone by the next update

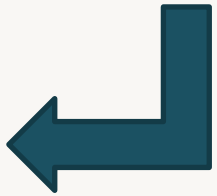
Live Table

```
UPDATE users
SET email = obfuscate(email)
```

```
CREATE OR REPLACE
users AS ….
```

Users

id	email
1	user@gmail.com



Streaming Live Table

```
UPDATE users
SET email = obfuscate(email)
```

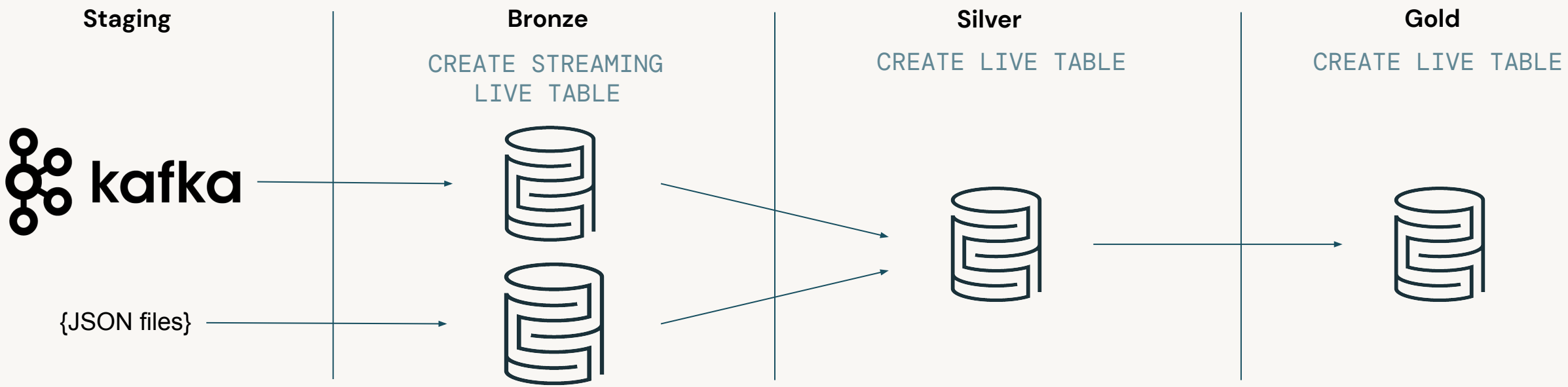
```
INSERT INTO users …
```

Users

id	email
1	****@gmail.com
2	****@hotmail...

Example GDPR use case

Using streaming live tables for ingestion and live tables after



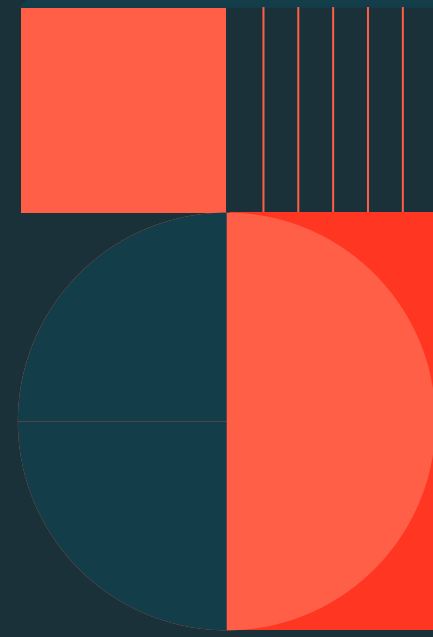
Limited Retention

Configurable Retention

Correction / GDPR

LIVE Tables automatically reflect changes made to inputs

When should I use streaming aggregation?



Incrementally compute simple aggregations

Reduce costs for aggregations with limited cardinality

Requirements

- One level of aggregation
- Associative aggregates (sum, count, min, max)
- Bounded cardinality
  - Naturally (group by state, region, etc)
  - Group by time with a watermark

Small cardinality aggregation: Compute incrementally, overwrite completely

```
@dlt.table()
def profit_by_region():
  return dlt.read_stream("sales") \
    .groupBy("region")
    .aggExpr("sum(profit) AS profit")
```

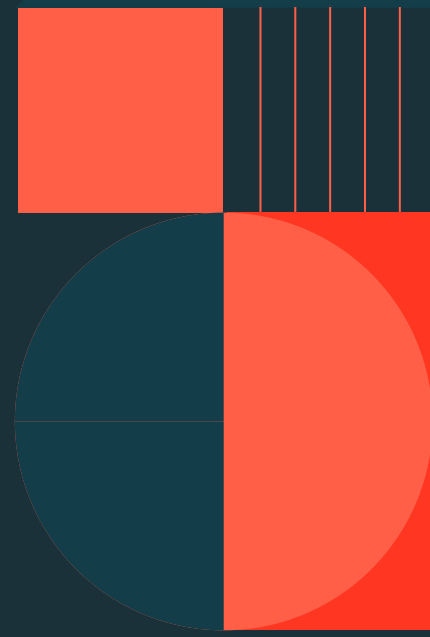
Time windows: Aggregate incrementally, append when window closes

```
@dlt.table()
def profit_by_hour():
  return dlt.read_stream("sales") \
    .withWatermark("timestamp", "1 hour")
    .groupBy(window("timestamp", "1 hour").alias("time"))
    .aggExpr("sum(profit) AS profit")
```

Deduplication, late data is dropped

```
@dlt.table()
def events():
  return dlt.read_stream("sales") \
    .withWatermark("timestamp", "1 hour")
    .dropDuplicates("timestamp", "event_id")
```

When should I use a stream-stream join?



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 an enrichment join, with static-snapshot, 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"""))
```

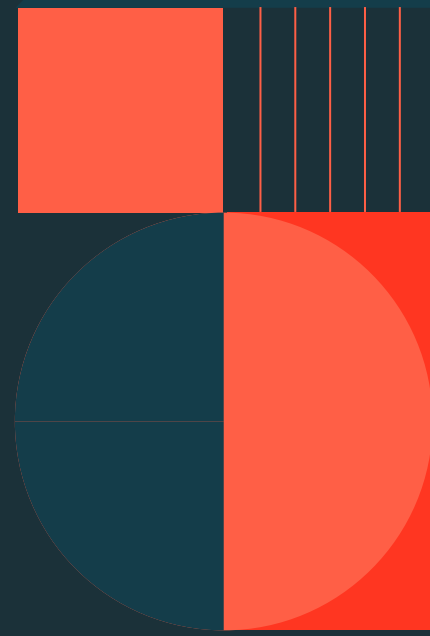
Two streams of facts, each with a watermark

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.

How can I use parameters?

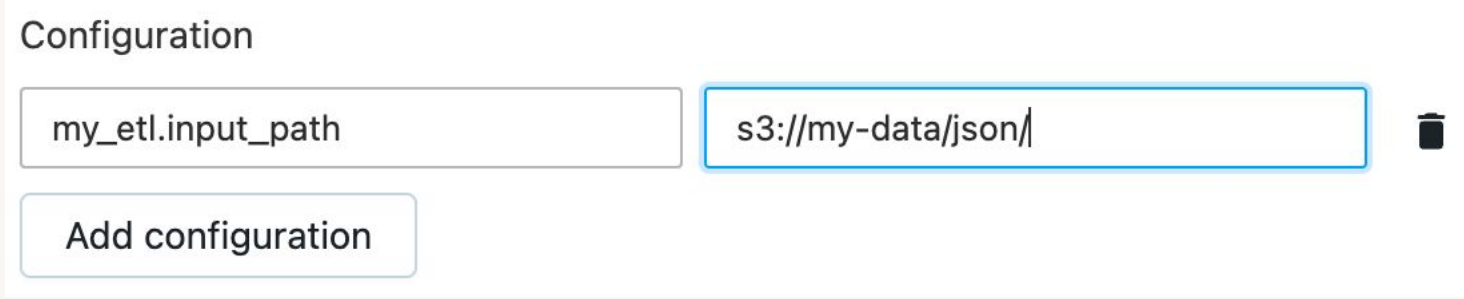


Modularize your code with configuration

Avoid hard coding paths, topic names, and other constants in your code.

A pipeline's configuration is a map of key value pairs that can be used to parameterize your code:

- Improve code readability/maintainability
- Reuse code in multiple pipelines for different data




```
CREATE STREAMING LIVE TABLE data AS
SELECT * FROM cloud_files("${my_etl.input_path}", "json")
```

```
@dlt.table
def data():
  input_path = spark.conf.get("my_etl.input_path")
  spark.readStream.format("cloud_files").load(input_path)
```

Pitfall: Accidental partial tables

Parameters in Settings cannot be used to control execution

When configuration is used to control what data is present in a table, changes will replace the table with a whole new result. Do not assume that it will append the new results to the old results!


```
INSERT INTO data
SELECT * FROM dat  HERE date = "${date_to_update}"
```

date
2022-04-17
2022-04-17
2022-04-18
2022-04-18

```
CREATE OR REPLACE
SELECT * FROM data WHERE date = "${date_to_update}"
```

date
~~2022-04-17~~
~~2022-04-17~~
2022-04-18
2022-04-18

BEST PRACTICE

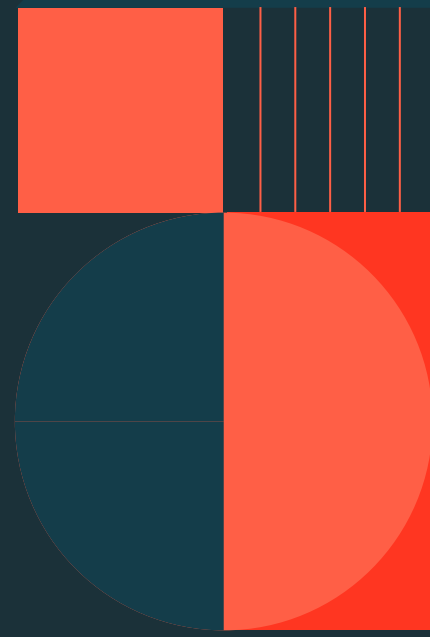
Configurable data volumes

Predicates can be used to reduce data volume

- Reduce data volumes in testing and staging using date range predicate.

```
CREATE LIVE TABLE data AS
SELECT *
FROM data
WHERE date > current_date() - INTERVAL "${my_etl.backfill_interval}"
```

What else can I manage with DLT?



BEST PRACTICE

All table attributes as code

One source of truth for governance, data discovery, and data layout.

```
CREATE LIVE TABLE report(
  customer_id LONG COMMENT "the customer id in salesforce", …
)
TBLPROPERTIES (
  delta.deletedFileRetentionDuration = "interval 30 days",
  pipelines.autoOptimize.zOrderCols = "timestamp,user_id",
  my_etl.quality = "gold",
  my_etl.has_pii = "false"
)
COMMENT "Weekly E-Staff report on key customer metrics, produced by growth team."
AS …
```

BEST PRACTICE

All table attributes as code

One source of truth for governance, data discovery, and data layout.

```
CREATE LIVE TABLE report(
  customer_id LONG COMMENT "the customer id in salesforce", …
)
TBLPROPERTIES (
  delta.deletedFileRetentionDuration = "interval 30 days",
  pipelines.autoOptimize.zOrderCols = "timestamp,user_id",
  my_etl.quality = "gold",
  my_etl.has_pii = "false"
)
COMMENT "Weekly E-Staff report on key customer metrics, produced by growth team."
AS …
```

Comments on both columns and whole tables can be used to document sources of data and details about what transformations are being performed

BEST PRACTICE

All table attributes as code

One source of truth for governance, data discovery, and data layout.

```
CREATE LIVE TABLE report(
  customer_id LONG COMMENT "the customer id in salesforce", …
)
TBLPROPERTIES (
  delta.deletedFileRetentionDuration = "interval 30 days",
  pipelines.autoOptimize.zOrderCols = "timestamp,user_id",
  my_etl.quality = "gold",
  my_etl.has_pii = "false"
)
COMMENT "Weekly E-Staff report on key customer metrics, produced by growth team."
AS …
```

delta.* properties are options supported by Delta Lake for retention and other physical aspects of the data being stored.

BEST PRACTICE

All table attributes as code

One source of truth for governance, data discovery, and data layout.

```
CREATE LIVE TABLE report(
  customer_id LONG COMMENT "the customer id in salesforce", …
)
TBLPROPERTIES (
  delta.deletedFileRetentionDuration = "interval 30 days",
  pipelines.autoOptimize.zOrderCols = "timestamp,user_id",
  my_etl.quality = "gold",
  my_etl.has_pii = "false"
)
COMMENT "Weekly E-Staff report on key customer metrics, produced by growth team."
AS …
```

pipelines.* properties are options that control how DLT operates including how data is organized.

BEST PRACTICE

All table attributes as code

One source of truth for governance, data discovery, and data layout.

```
CREATE LIVE TABLE report(
  customer_id LONG COMMENT "the customer id in salesforce", …
)
TBLPROPERTIES (
  delta.deletedFileRetentionDuration = "interval 30 days",
  pipelines.autoOptimize.zOrderCols = "timestamp,user_id",
  my_etl.quality = "gold",
  my_etl.has_pii = "false"
)
COMMENT "Weekly E-Staff report on key customer metrics, produced by growth team."
AS …
```

Application specific properties stored as key/value pairs.

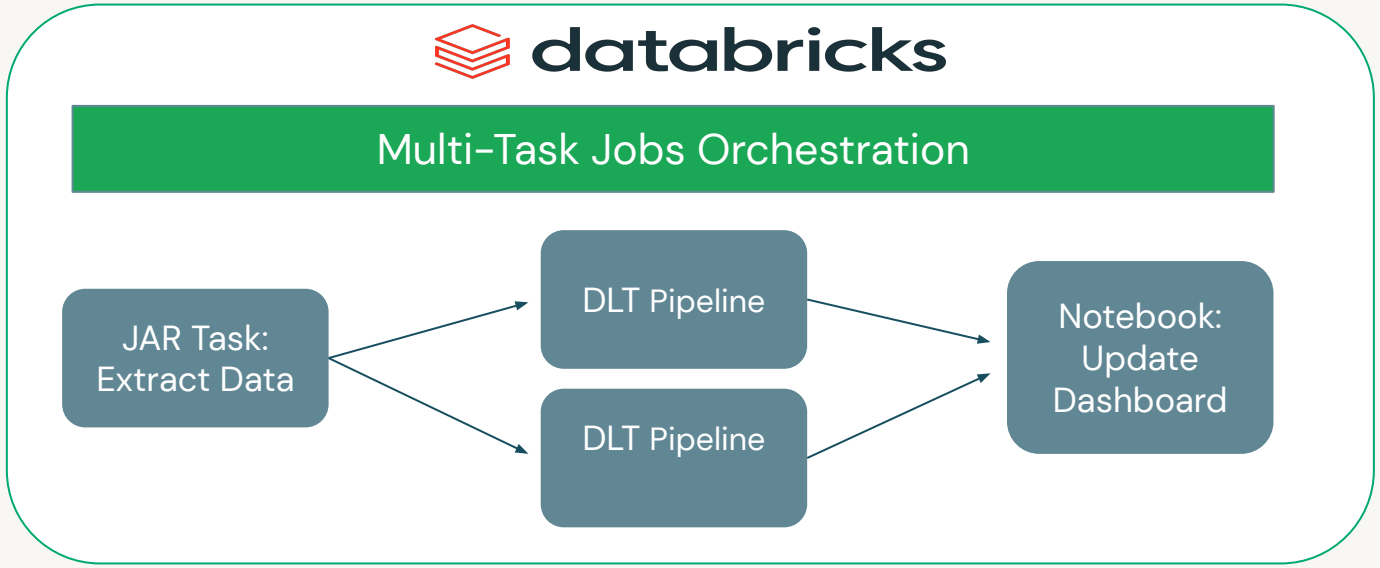
A best practice is to use a common prefix to identify these properties.

Orchestration

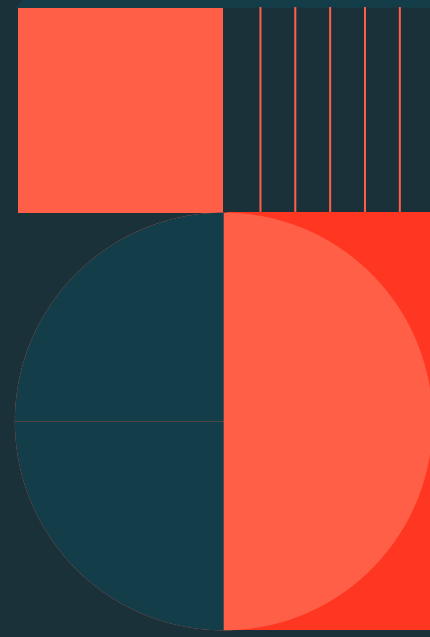
Workflow Orchestration

For Triggered DLT Pipelines

- Use Databricks Jobs to easily orchestrate DLT Pipelines and any other task in the same Multi-task Job.
- Fully integrated in Databricks platform, making inspecting results, debugging faster
- Orchestrate and manage workloads in multi-cloud environments
- You can also run a Delta Live Tables pipeline as part of a data processing workflow in Apache Airflow or Azure Data Factory.



How do I modularize my SQL/Dataframe Code?



BEST PRACTICE

When to use Tables vs Views?

Tables are materialized, views are virtual

TABLES

A table is **a materialized copy** of the results of a query

- Creating and updating a tables costs storage and compute
- The results can be reused by multiple downstream computations

VIEWS

Views are virtual, **a name for a query**

- Use views to break up large/complex queries
- Expectations on views validate correctness of intermediate results
- Views are recomputed every time they are queried

Limits & Scalability Guidelines

System Limits

Resource	Limit
Notebooks per DLT Pipeline	25 limit, can be raised on request
Concurrent DLT Pipelines per Workspace	20 limit, can be raised on request

Scalability Guidelines

Resource	Guidance on Table Count
# of live tables in a triggered pipeline	< 100
# of streaming live tables in a triggered pipeline	< 50
# of live + streaming tables in a continuous pipeline	< 25

These guidelines are based on actual, real-world DLT usage.
We recommend doing a PoC to validate these limits for your own environment.