databricks

Streaming ETL Patterns with DLT

Agenda

Streaming ETL Patterns with DLT

Lesson Name	Lesson Name
Lecture: Data Ingestion Patterns	Lecture: Data Modeling
Data Ingestion Patterns ADE 2.1 – Follow Along Demo – Auto Load to Bronze ADE 2.2 – Follow Along Demo – Stream from Multiplex Bronze	ADE 2.5 – Follow Along Demo – Data Modeling – SCD Type 2
Lecture: Data Quality Enforcement Patterns	Lecture: Streaming Joins and Statefulness
ADE 2.3 – Follow Along Demo – Data Quality Enforcement	ADE 2.6 – Follow Along Demo – Streaming Joins
ADE 2.4L – Streaming ETL Lab	

Data Ingestion Patterns

Why Do We Need These Patterns?

Limitations at Data Ingestion Stage

- Streaming sources like Kinesis, Kafka and EventHubs only retain data for a **limited** amount of time
- **Need for retention** – full history of data
    - Reprocessing raw data
    - Perform GDPR and compliance tasks
    - Recover data
- Need for a simple, **maintainable and scalable** architecture
- Keeping full history in the streaming source is **expensive**

Pattern 1: Use Delta for **Infinite Retention**

Delta provides cheap, elastic and governable storage for transient sources

cloud_files

kafka

amazon KINESIS

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

```
CREATE STREAMING
LIVE TABLE AS …
```

Avoid complex transformations that could have bugs or drop important data

```
TBLPROPERTIES (
  pipelines.reset.allowed=false
)
```

bronze

Retain history
Easy to perform GDPR and other compliance tasks

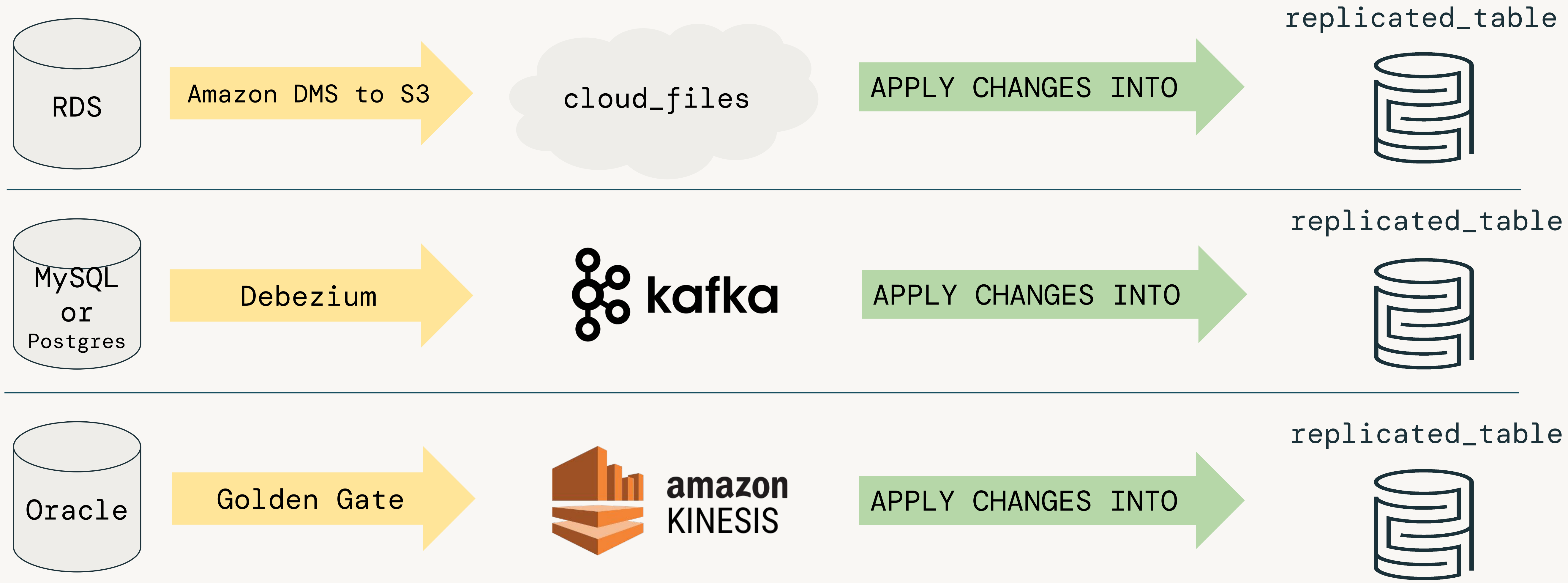
```
CREATE STREAMING
LIVE TABLE AS …
```

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

Pattern 2: Up-to-date **Replica with CDC**

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

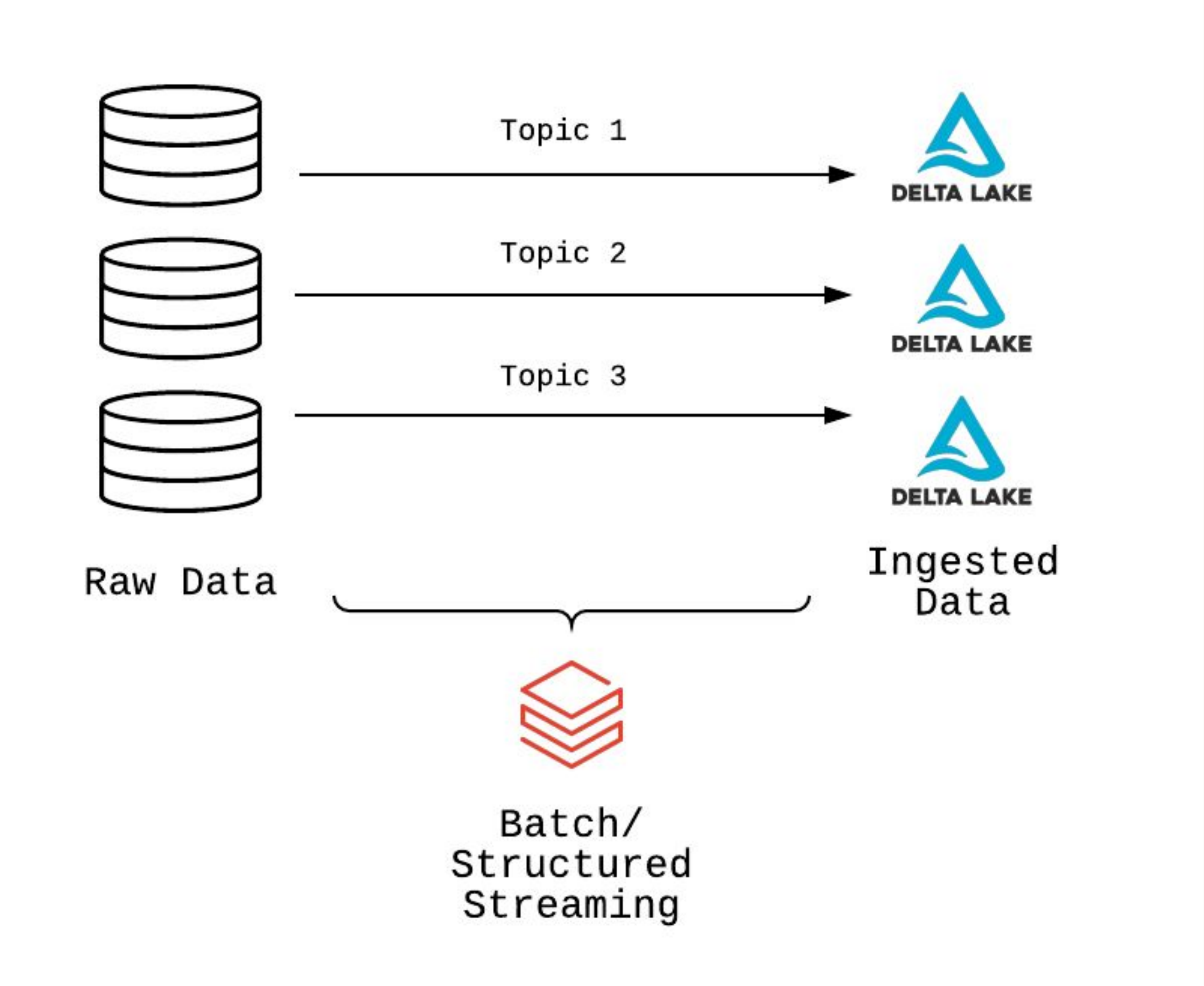
- Use Change Data Capture (CDC) from RDMS and create replica as Delta
- A variety of 3rd party tools can provide a streaming change feed



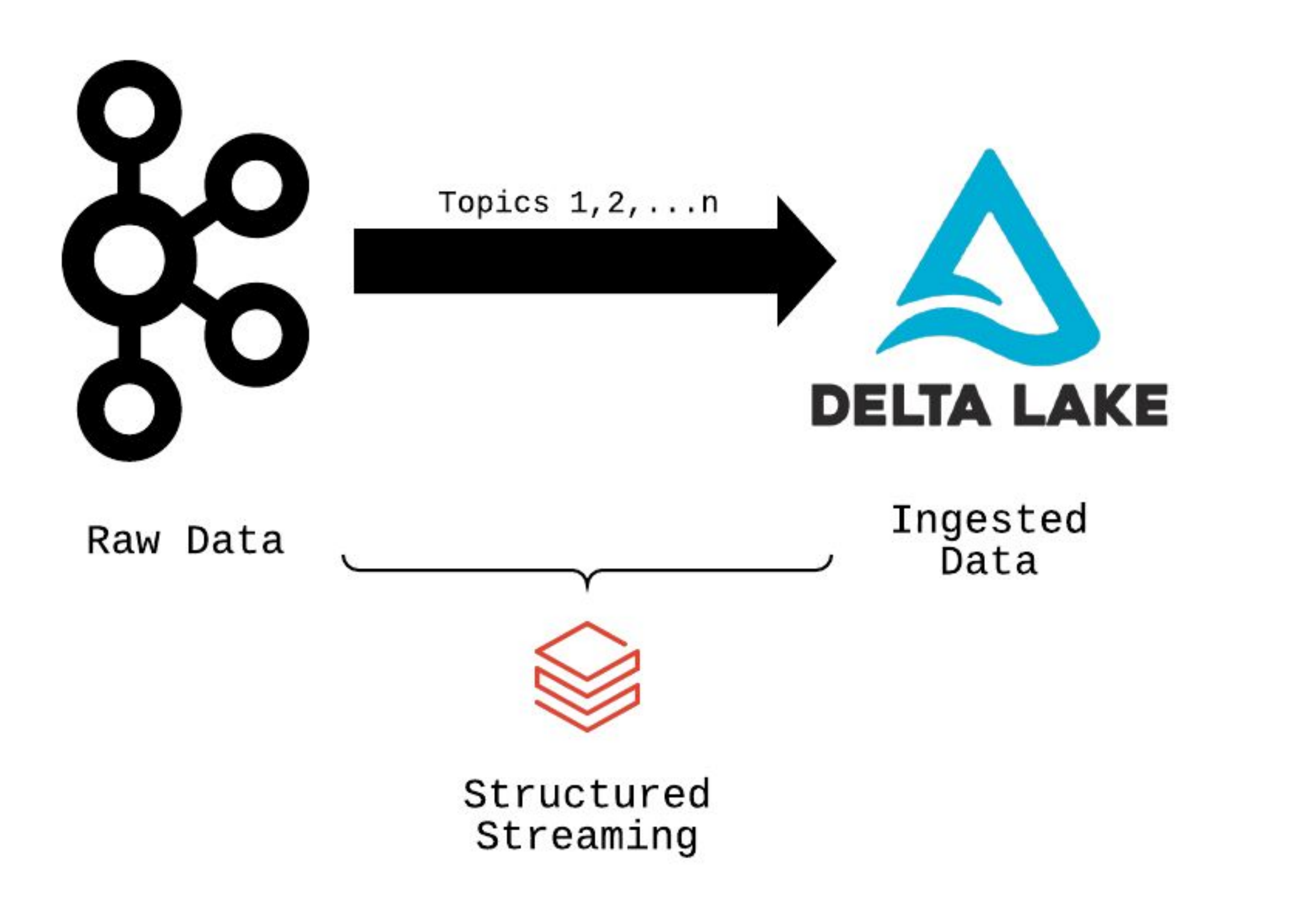
Pattern 3: Multiplex Ingestion

Multiplexing is used when a set of independent streams all share the same source

Simplex



Multiplex

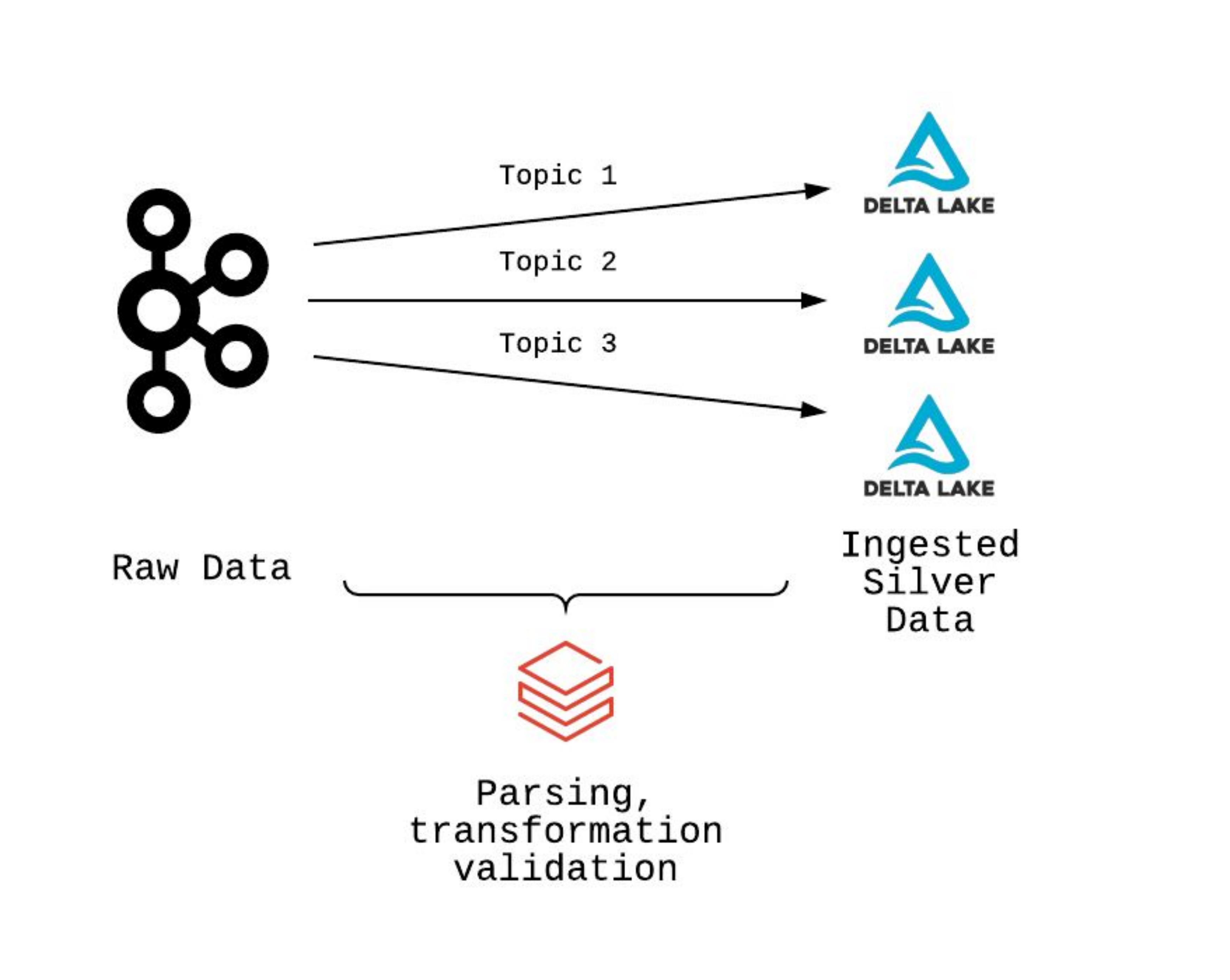


Pattern 3: Multiplex Ingestion

Anti-Pattern: Using Kafka as Bronze Table

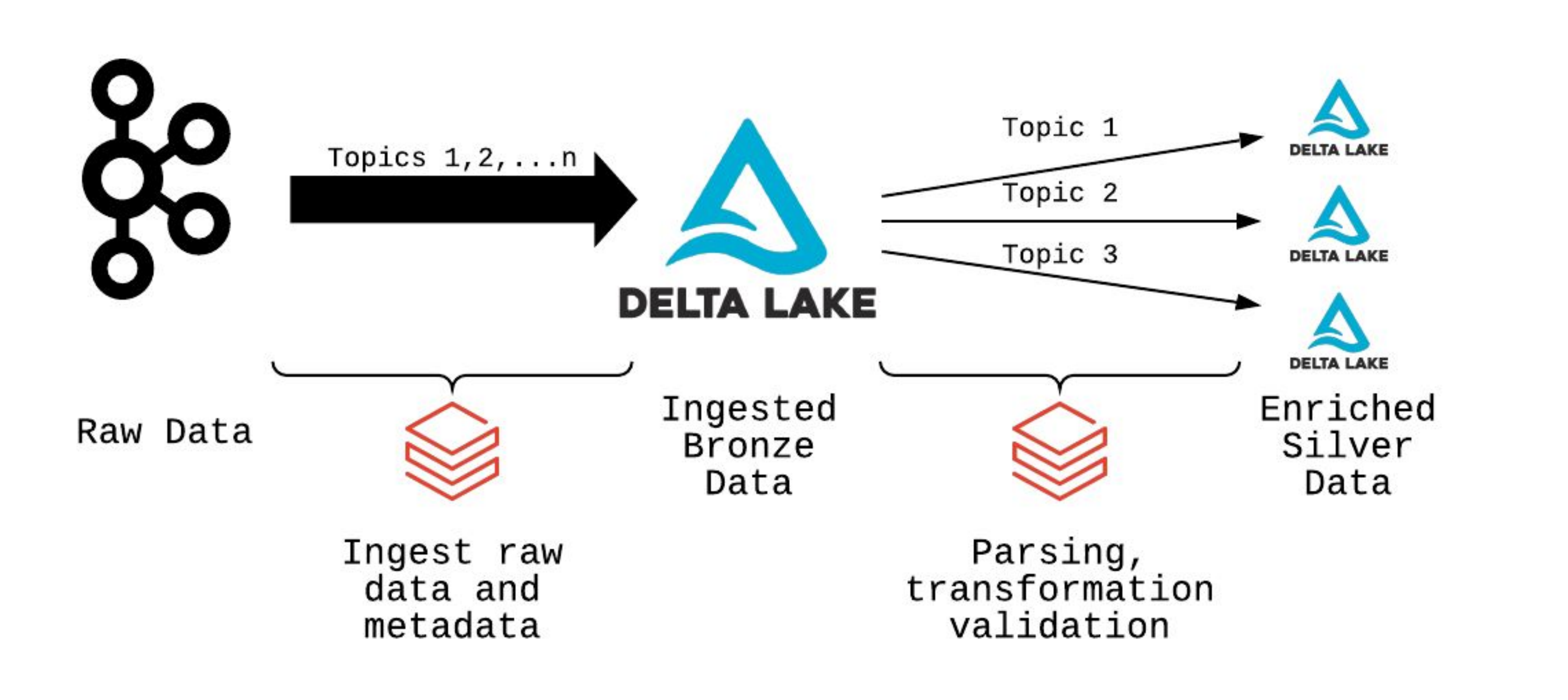
Don't use Kafka as Bronze Table:

- Data retention limited by Kafka; expensive to keep full history
- All processing happens on ingest
- If stream gets too far behind, data is lost
- Cannot recover data (no history to replay)



Pattern 3: Multiplex Ingestion Pattern

Multiplexing is used when a set of independent streams all share the same source



Demo: Auto Load to Bronze

Demo: Stream from Multiplex Bronze

Data Quality Enforcement Patterns

Silver Layer for Quality Enforcement

Silver Layer Objectives

- Validate data quality and schema
- Enrich and transform data
- Optimize data layout and storage for downstream queries
- Provide single source of truth for analytics

Schema Enforcement & Evolution

- Enforcement prevents bad records from entering table
  - Mismatch in type or field name
- Evolution allows new fields to be added
  - Useful when schema changes in production/new fields added to nested data
  - Cannot use evolution to remove fields
  - All previous records will show newly added field as Null
    - For previously written records, the underlying file isn't modified.
    - The additional field is simply defined in the metadata and dynamically read as null

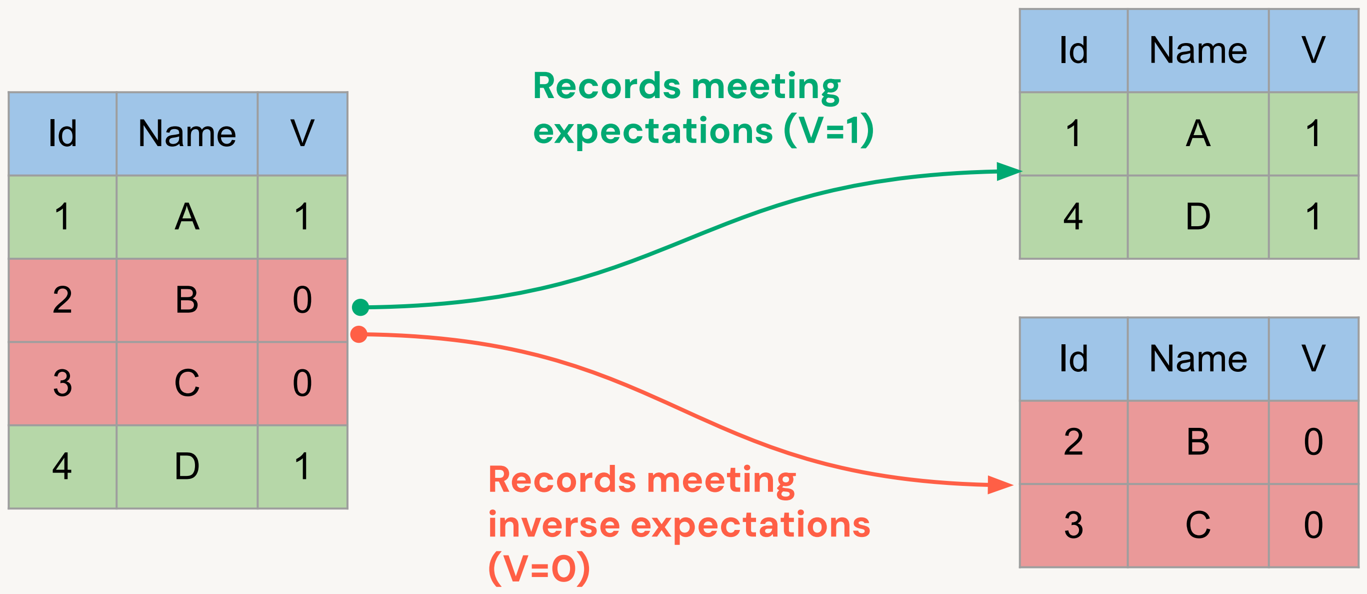
Alternative Quality Check Approaches

- Add a “validation” field that captures any validation errors and a null value means validation passed.
- Quarantine data by filtering non-compliant data to alternate location
- Warn without failing by writing additional fields with constraint check results to Delta tables

Pattern: Quarantine Invalid Records

What if we want to save the records that violate data quality constraints for analysis?

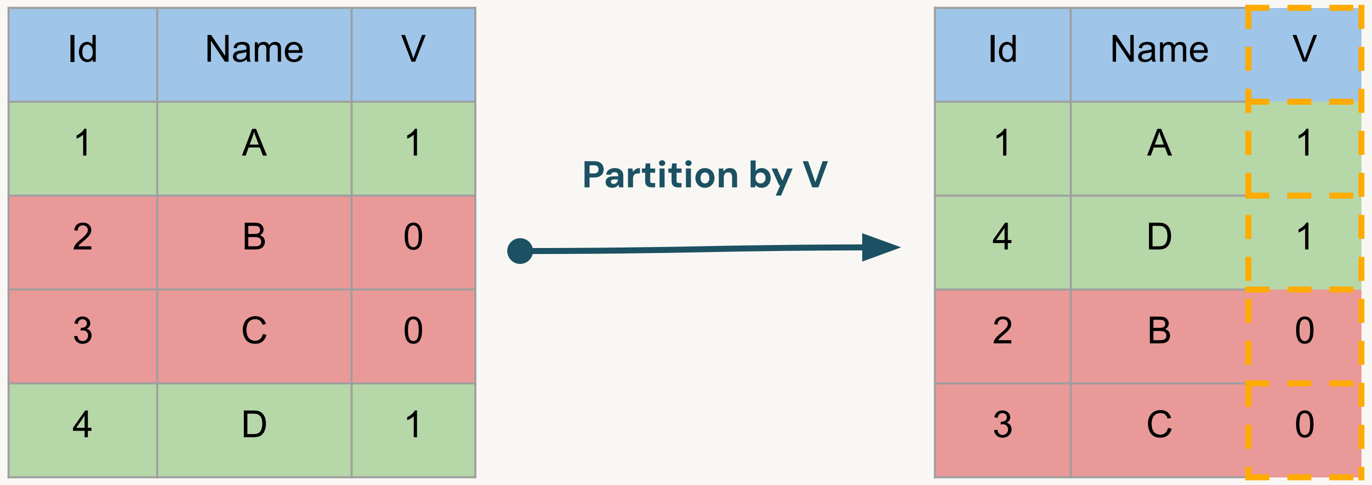
Method 1: Create Inverse Expectation Rules



Limitations:

- Processes the data twice

Method 2: Add a validation status column and use for partitioning



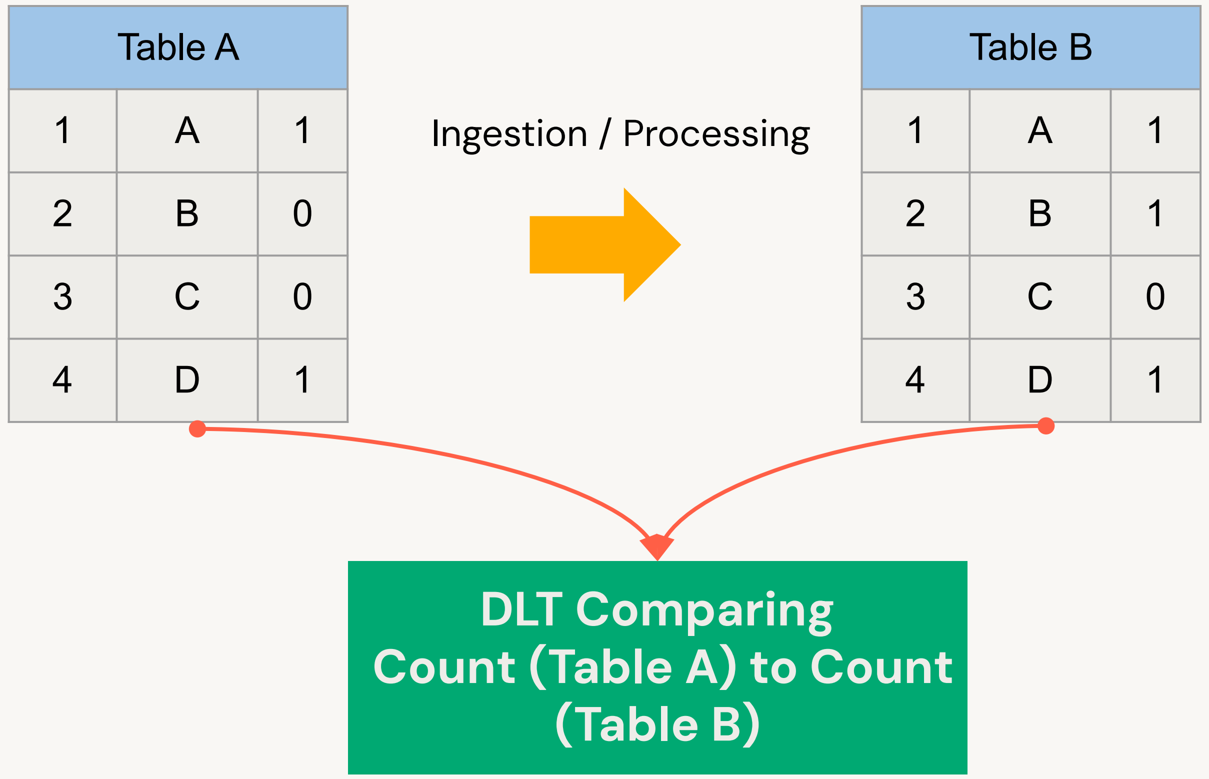
Limitations:

- Doesn't use expectations; data quality metrics are not available in the event logs or the pipelines UI.

Pattern: Verify Data with Row Comparison

Validate row counts across tables to verify that data was processed successfully without dropping rows.

- Solution:
  - Add an additional table to your pipeline that defines an expectation to perform the comparison.
  - The results of this expectation appear in the event log and the Delta Live Tables UI.



DLT.sql

```sql
CREATE OR REFRESH LIVE TABLE count_verification(
  CONSTRAINT no_rows_dropped EXPECT (a_count == b_count)
) AS SELECT * FROM
  (SELECT COUNT(*) AS a_count FROM LIVE.tbla),
  (SELECT COUNT(*) AS b_count FROM LIVE.tblb)
```

Pattern: Define Tables for Adv. Validation

Perform advanced data validation with DLT expectations

- Complex data quality checks examples;
  - A derived table contains all records from the source table
  - Guaranteeing the equality of a numeric column across tables
- Solution:
  - Define DLT using aggregate and join queries and use the results of those queries as part of your expectation checking.

```sql
-- Validates all expected records are present
in the "report" table

CREATE LIVE TABLE compare_tests(
  CONSTRAINT no_missing_records
  EXPECT (r.key IS NOT NULL)
)
AS SELECT * FROM LIVE.validation_copy v
LEFT OUTER JOIN LIVE.report r ON v.key = r.key
```

ADE 2.3 - Data Quality Enforcement

ADE 2.4L - Streaming ETL Lab

Data Modeling

Learning Objectives

By the end of this lesson, you should be able to:

1. Describe main concepts of dimensional modeling
2. Describe SCD tables and implementation with Delta Live Tables
3. Explain a common pipeline wherein a streaming data source joins to a static table.

Slowly Changing Dimensions in the Lakehouse

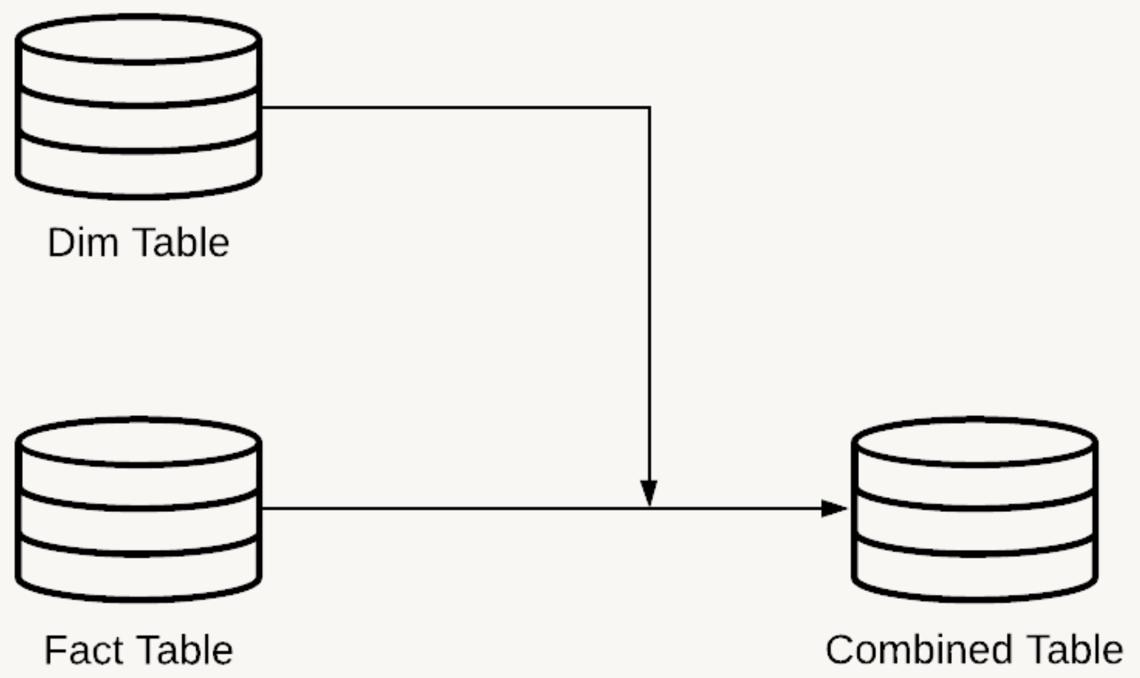
Dimensional Modeling

Fact Tables vs. Dimension Tables

- **Fact Tables:** Often contain a granular record of activities
- **Dimension Tables:** Often contain data may be updated or modified over time.

Modeling Guidelines:

- Denormalize dimension and fact tables
- Use conformed dimensions
- Use slowly changing dimensions as necessary



Dimensional Modeling

Fact Tables as Incremental Data

- Often is a time series
- No intermediate aggregations
- No overwrite/update/delete operations
- Often append-only operations

Slowly Changing Dimensions (SCD)

3 types of dimension tables

Type 0

- No changes allowed
- Tables are either static or append only
- Examples: static lookup tables, append-only fact tables

Type 1

- Overwrite but no history is maintained
- May contain recording of when record was entered, but not previous values
- Example: valid customer mailing address

Type 2

- Add a new row; mark old row as obsolete
- Strong history is maintained
- Example: tracking product price changes over time

Slowly Changing Dimensions (SCD)

3 types of dimension tables

Type 0 / Type 1

user_id	street	name
1	123 Oak Ave	Sam
2	99 Jump St	Abhi
3	1000 Rodeo Dr	Kasey

Type 2

user_id	street	name	valid_from	current
1	123 Oak Ave	Sam	2020-01-01	true
2	99 Jump St	Abhi	2020-01-01	false
3	1000 Rodeo Dr	Kasey	2020-01-01	false
2	**430 River Rd**	Abhi	**2021-10-10**	**true**
3	1000 Rodeo Dr	**Casey**	2021-10-10	**true**

SCD Type 2 with DLT

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

Applying SCD Principles to Facts

- Fact table usually append-only (Type 0)
- Can leverage event and processing times for append-only history

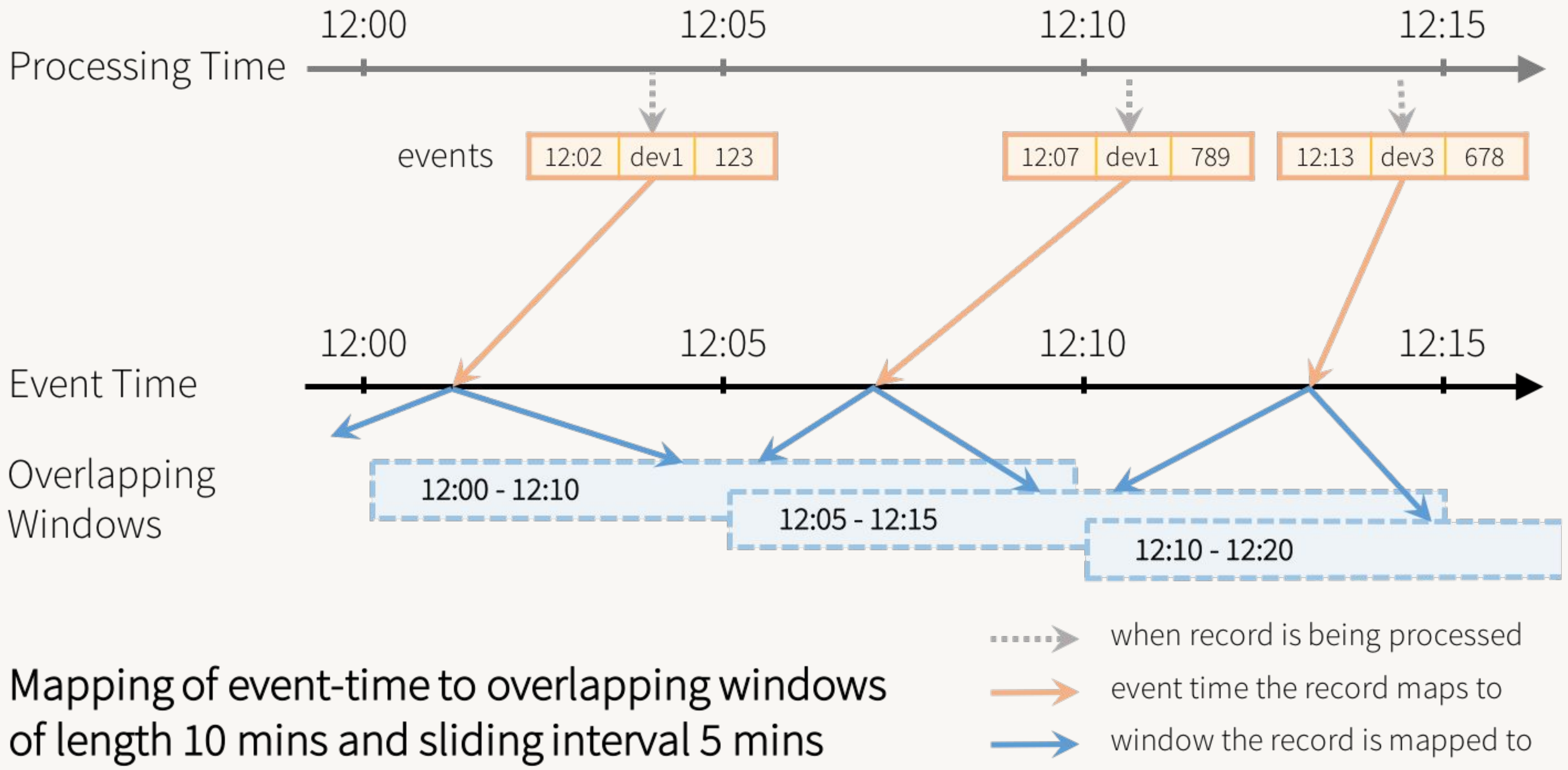
order_id	user_id	occurred_at	action	processed_time
123	1	2021-10-01 10:05:00	ORDER_CANCELLED	2021-10-01 10:05:30
123	1	2021-10-01 10:00:00	ORDER_PLACED	2021-10-01 10:06:30

ADE 2.5 – Data Modeling

Streaming Joins and Statefulness

The Components of a Stateful Stream

```
windowedDF =
  (eventsDF
   .groupBy(window("eventTime",
      "10 minutes",
      "5 minutes"))
   .count()
   .writeStream
   .trigger(processingTime="5 minutes")
  )
```



Mapping of event-time to overlapping windows of length 10 mins and sliding interval 5 mins

Statefulness vs. Query Progress

- Some operations are specifically **stateful** in that the results of processing earlier records from the stream affect the processing of later records.
    - Examples include deduplication, aggregation, and stream-stream joins
- Other transformations just need to store incremental query progress and are **not stateful**.
    - Examples include simple transformations and stream-static joins
- Progress and state are stored in **checkpoints** and managed by the driver during query processing.

Stream-Static Joins

Using **Dimension Tables** in Incremental Updates

- Delta Lake enables dynamic stream-static joins
- Each micro-batch captures the most recent state of the Delta table that is the static side of the join
  - This does not occur if the static side of the join is another format such as Parquet
- Allows modification of dimension while maintaining downstream composability

Note: Because Delta Lake does not enforce foreign key constraints, it is possible that joined data will go unmatched.

Streaming Queries are **Not 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")

"/data"

{"a": 1}
{"a": 2}
{"a": 3}
{"a": 4}

raw_data

b
2
3
6
8

Streaming Joins are Not **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
```

"/data"

{"a”: 1}

{"a”: 1}

raw_data

id	city
1	Bekerly, CA
1	Berkeley, CA

id	city
1	~~Bekerly, CA~~

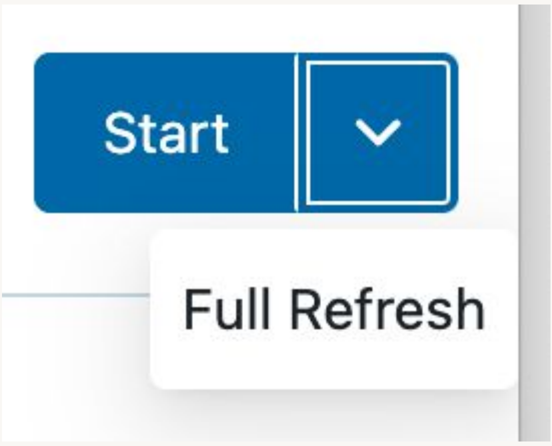
Berkeley, CA

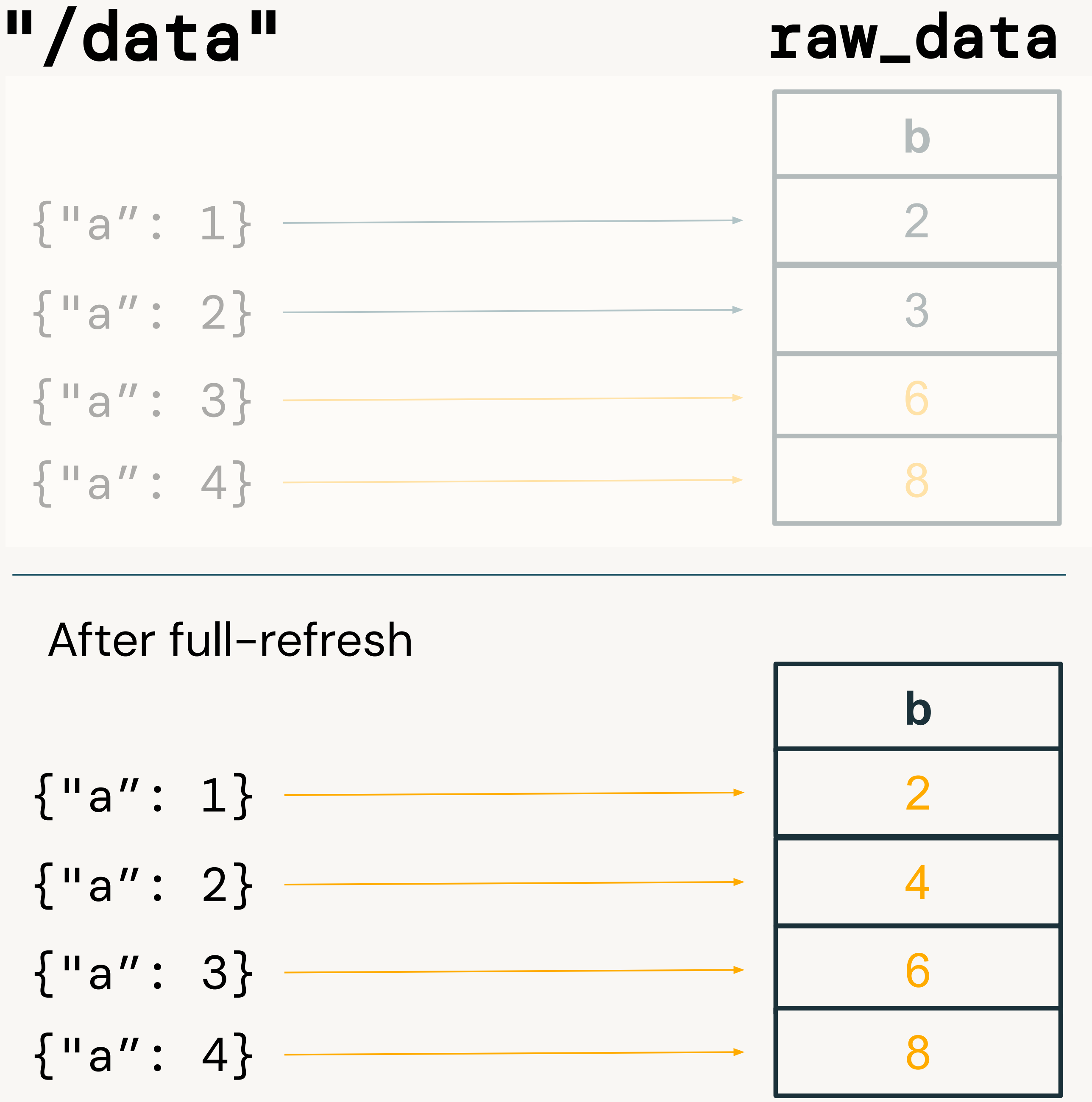
Clear State in DLT

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")
```





Stream–Static Join & Merge

- Join driven by streaming data
- Join triggers shuffle
- Join itself is stateless
- Control state information with predicate
- Goal is to broadcast static table to streaming data
- Broadcasting puts all data on each node

1. Main input stream

```
salesSDF = (
  spark
    .readStream
    .format("delta")
    .table("sales")
)
```

2. Join item category lookup

```
itemSalesSDF = (
  salesSDF
    .join(
      spark.table("items")
       .filter("category='Food'), #
Predicate
      on=["item_id"]
    )
)
```

ADE 2.6 - Streaming Joins

Knowledge Check

Which of the following is considered a recommended best practice for ingesting streaming data?

Select one response.

A. Use streaming live tables for raw data and streaming tables for bronze, silver, and gold quality data.
B. Use streaming tables for bronze quality data and streaming live tables for silver and gold quality data.
C. Use streaming live tables for bronze quality data and streaming tables for silver and gold quality data.
D. Use streaming tables for raw data and streaming live tables for bronze, silver, and gold quality data.

A data engineer has data that needs to be updated. However, they need to have access to a recorded history of the information previously stored in the dataset before the update. Which of the following table types should the data engineer use for their data?

Select one response.

A. Type 0
B. Type 1
C. Type 2
D. Type 1 or Type 2

Which of the following operations can be performed on stateless tables to limit the state dimension?

Select one response.

A. Stream-stream join
B. Stream-static join
C. Stateful aggregation
D. Drop duplicates

Which of the following statements about fact tables and dimension tables are true?

Select two responses.

A. Transactional guarantees and Delta Lake ensure that the newest version of a dimension table will be referenced each time a query is processed for incremental workloads.
B. Joined data cannot go unmatched because of Delta Lake's foreign key constraint.
C. Dimension tables contain a granular record of activities, while fact tables contain data that is updated or modified over time.
D. Modern guidelines suggest denormalizing dimension and fact tables.

The following line of code is supposed to create a set of inverted rules for a quarantine table.

quarantine_rules = _____

Which of the following correctly fills in the blank?

Select one response.

A. `{"invalid_record": f"NOT({' AND '.join(rules.values())})"}`
B. `{"invalid_record": f"&&({' ! '.join(rules.values())})"}`
C. `{"invalid_record": f"NOT({' OR '.join(rules.values())})"}`
D. `{"invalid_record": f"IF({' NULL '.join(rules.values())})"}`